UMR IRISA

Eléments de **Génie Logiciel**

Cours 1-3

Pr. Jean-Marc Jézéquel

IRISA – Université de Rennes

Campus de Beaulieu

F-35042 Rennes Cedex

e-mail : jezequel@irisa.fr

http://people.irisa.fr/Jean-Marc.Jezequel

X @jmjezequel

istic Informatique Électronique

Université de Rennes

Université de Rennes

1 - Introduction

Université de Rennes

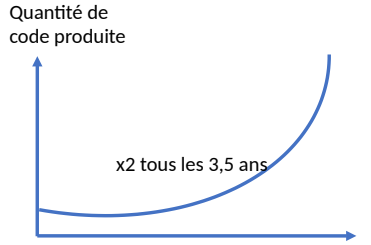
Software is eating the world (Marc Andreessen)

- Le logiciel forme le tissu de notre société
  - Tel l'oxygène qu'on respire, force motrice essentielle, bien qu'invisible, du monde actuel
  - pratiquement aucun aspect de la société qui ne soit pas facilité ou médiatisé par le logiciel
- *Toute (grande) entreprise est désormais une entreprise de logiciels (S. Nadela)*
  - Le logiciel est une technologie clé de transformation et d'habilitation qui impacte de nombreux domaines de l'économie moderne
- Les logiciels stimulent le progrès scientifique dans de nombreux domaines
  - IA, la science des données, la médecine, l'ingénierie, etc.
- Les logiciels sont essentiels pour nos infrastructures critiques
  - énergie, les télécommunications, l'aérospatiale, l'automobile, la finance, la santé
- Le volume total de logiciels dans le monde croît à un rythme exponentiel
  - Double tous les 3,5 ans

Université de Rennes

Taille de quelques logiciels (lignes de code)

- MS-DOS: 4 k
- WhatsApp: 30 k
- Telegram: 50 k
- Zoom: 60 k
- TikTok: 80 k
- Space Shuttle: 400 k
- Minecraft: 500 k
- Instagram: 1 M
- US Military Drone: 3.5 M
- YouTube: 5.4 M
- World of Warcraft: 5.5 M
- Boeing 787: 6.5 M
- Google Chrome: 6.7 M
- Twitter: 10 M
- Android: 12 M
- iOS: 12 M
- Mozilla Firefox: 21 M
- Windows XP: 45 M
- Large Hadron Collider: 50 M
- Ubuntu: 50 M
- Facebook: 62 M
- MacOS X: 84 M
- Tesla: 100 M
- Google: 2 milliards



Ce n'est pas demain que les (bons) développeurs seront au chômage!

Même avec Copilot/ChatGPT…

Université de Rennes

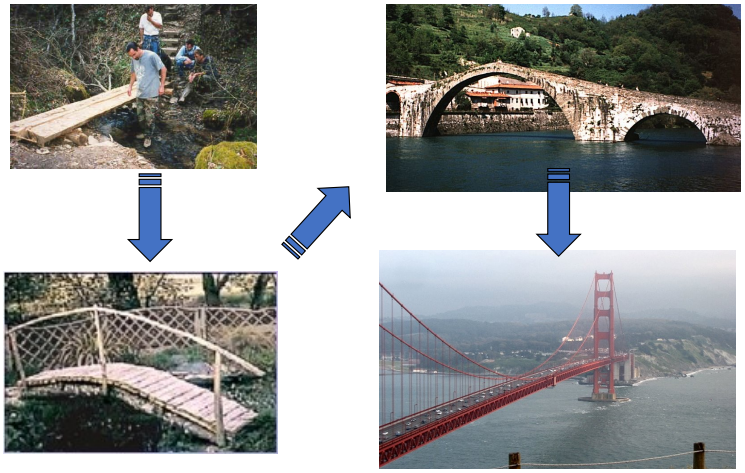
Des logiciels souvent complexes...

- Logiciels de grande taille
  - des millions de lignes de code
  - des équipes nombreuses
  - durée de vie importante
  - des lignes de produits
  - plateformes technologiques complexes
  - évolution continue
  - Problèmes de fiabilité, sécurité, efficacité, utilisabilité, etc.
- Besoin d'ingénierie pour les réaliser
  - Génie logiciel

Université de Rennes

Ingénierie : Gérer la complexité due à la taille

Traverser un cours d'eau



Université de Rennes

Qu'est-ce que le génie logiciel ? (Wikipédia)

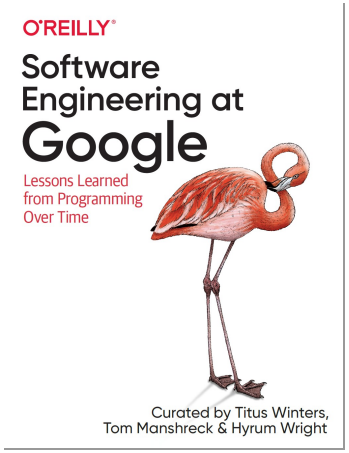
- Procédures systématiques qui permettent d'arriver à ce que des logiciels de grande taille
  - correspondent aux attentes du client,
  - soient fiables,
  - aient un coût d'entretien réduit
  - et de bonnes performances
  - tout en respectant les délais et les coûts de construction
- Qui utilise ces procédures systématiques du génie logiciel ?
  - ***Tous*** les développeurs de logiciels (en entreprise ou dans le logiciel libre)

Université de Rennes

Qu'est-ce que le génie logiciel ?

- La définition de Google :
  - *programming integrated over time*

The Google codebase includes approximately one billion files and has a history of approximately 35 million commits spanning Google's entire existence. The repository contains 86TB[a] of data, including approximately two billion lines of code in nine million unique source files.



D'où je vous parle...

- Professeur Université de Rennes
  - Membre senior de l'Institut Universitaire de France (IUF)
  - Président de Informatics Europe
  - Directeur de l'IRISA (2012-2020)
- Chercheur en Sciences du Logiciel
  - Médaille d'Argent du CNRS
  - Principes, méthodes et outils utilisés dans le développement de logiciels
    - Vise à mieux comprendre et améliorer les processus de fabrication : qualité, productivité
  - Nombreuses collaborations industrielles
    - avec Airbus, Thalès, Orange, PME et start-ups.

https://fr.wikipedia.org/wiki/Jean-Marc_Jézéquel

2 – Présentation du module *Eléments de Génie Logiciel* (GEN)

Page du cours :

http://people.irisa.fr/Jean-Marc.Jezequel/enseignement/L2GEN

Qu'allez vous faire pendant cette UE ?

- Vous ne pourrez plus dire :
  - ce qu'on voit en cours ne nous servira jamais
  - les cours ne parlent pas de choses utilisées dans la "vraie vie"
- Cette UE va vous confronter à de vrais problèmes de développement
  - travailler en équipe *(seul on va plus vite, en équipe on va plus loin)*
  - communiquer ! !
  - organiser l'équipe
  - organiser son travail dans l'équipe
  - partager son code
  - tester son code
  - rendre son code lisible, utilisable et modifiable par d'autres
  - documenter son code
  - Modifier le code des autres
  - Modifier son code
  - . . . trouver des solutions algorithmiques, programmer

Importance décroissante

Objectifs généraux du cours

- Confrontation à la réalité du développement collaboratif de logiciels de grande taille, en présentant quelques principes et outils pour parvenir à le maîtriser.
- À l'issue de ce cours, les étudiants sont conscients de problèmes posés par :
  - Le développement de logiciels de taille importante
    - Organisés en modules/classes/interfaces (Architecture logicielle)
  - Le développement en équipe (de ~ 10 personnes)
  - La maintenance évolutive du logiciel
  - Test logiciel (test unitaire, test d'intégration)
  - Gestion de version
  - Méthodes agiles
- Ils ont renforcé leur compréhension de
  - La programmation objet
  - L'utilisation d'IDE de type industriel (VSCode)
  - L'utilisation de LLMs pour la programmation (ChatGPT, Copilot…)

Ce n'est pas un cours de Java, même si Java est le langage qui sera utilisé en TP

En PO vous avez appris à coder l'intérieur des classes. En GEN on va mettre le focus sur les relations entre les classes aka *Architecture Logicielle*

Université de Rennes

Pourquoi enseigner le génie logiciel dès la L2?

- Parce que c'est nécessaire dans la formation de tout développeur
- Pour que vous puissiez, dès maintenant, vous impliquer dans des projets de développement collaboratifs (par ex. logiciel libre)
- Pour que vous soyez plus rapidement autonomes en stage
- Pour que vous soyez confronté au problème suffisamment tôt
- Permet de comprendre l'importance de beaucoup de cours de L3/M1
- Parce que c'est nécessaire au bon déroulement des projets de L3/M1
- Parce que c'est un prérequis de nombreux cours de L3/M1/M2

Université de Rennes

Programme des CM

- Concepts de base des langages à objets
  - objets, classes, encapsulation, masquage d'information, interface, modularité
- Concepts fondamentaux des langages à objets
  - héritage, polymorphisme, typage
- Introduction à l'architecture logicielle
  - diagrammes de classes, associations, diagrammes de séquence
- Les 3 dimensions de la gestion de configuration
  - variants, versions, et concurrence, principes de Git
- Introduction au test de logiciel
  - test unitaires, JUnit
- Outils pour la qualité du logiciel
  - documentation , typage statique, programmation par contrats, qualité des tests
- Introduction au développement agile
  - processus de développement, principes de l'agile, Use cases/user stories
  - utilisation de LLMs type ChatGPT ou Copilot

Université de Rennes

Et les Travaux dirigés ?

- Pas de Travaux dirigés !
  - 8 CMs et 20 TPs + 4 TPs non encadrés
  - Vous pouvez poser des questions pendant les cours (ou à la fin)
  - Vous devez lire la documentation mise à votre disposition
  - **Vous devez poser des questions en TP**
- Des quiz notés (sans documents) en CM
  - La mauvaise nouvelle : vous devez connaitre et comprendre le cours
  - La bonne nouvelle : le cours est réduit ! (8 CMs)

Université de Rennes

Travaux pratiques

- Conception de l'architecture logicielle du projet
  - Développement collaboratif du projet
  - Génération de la documentation
  - Test systématique et automatique du logiciel
  - Validation par des tiers (enseignants, tests automatisés)
- Outils utilisés en TP :
  - VS Code (avec plugins Java)
  - Java
  - JUnit 4
  - Git
  - LLMs type ChatGPT / Copilot
- Si vous n'avez pas fini un TP pendant le temps imparti, vous **devez** le terminer sur votre temps de travail personnel (sauf indication contraire)
  - Les TP ne sont pas « faciles » et demandent un ***vrai travail***

Evaluation : Contrôle Continu intégral

- Modalités du CC
  - Durant le semestre, plusieurs quiz de 10 min sans documents (1/5 de la note)
    - en amphi au début ou à la fin de certains CM
    - Absence au CM => 0 au CC (sauf ABJ : note neutralisée)
      - Les PAEH peuvent me demander à être dispensés (par e-mail)
  - Durant le semestre, deux notes de travaux pratiques avec pour chaque :
    - Une note d'équipe
    - Une note individualisée après entretien individuel avec le ou la chargé de TP (2/5 de la note)
      - Explication du code produit
      - Compréhension de l'architecture globale et de l'intégralité du code
      - Implication dans le projet : êtes-vous un élément moteur ? Un *team player*? (2/5 de la note)
  - À la fin du semestre 4, un CC final d'1h30, tous documents autorisés.

Politique en matière d'IA pour le cours **GEN**

Déjà utilisé *jamais utilisé ChatGPT*

Je vous encourage et je m'attends à ce que vous **utilisiez** l'IA (ChatGPT et/ou Copilot) dans ce cours
Apprendre à utiliser l'IA est une compétence émergente et il est important de savoir l'utiliser en génie logiciel

Soyez toutefois conscient des **limites** de ChatGPT/Copilot/… : https://youtu.be/R2fjRbc9Sa0

- Si vous fournissez des questions (*prompts*) en fournissant peu d'effort, vous obtiendrez des résultats de faible qualité. Affinez vos *prompts* afin d'obtenir de bons résultats. **Cela demande du travail.**
- Ne vous fiez jamais à ce qu'il dit. **Gardez un esprit critique. Vous serez responsable de toute erreur** ou omission fournie par l'outil. Il vaut mieux utiliser l'IA sur des sujets que vous comprenez.
- L'IA est un outil que vous **devez reconnaître avoir utilisé.** Veuillez utiliser les commentaires de votre code pour expliquer à quelles fins vous avez utilisé l'IA et quels prompts vous avez utilisés pour obtenir les résultats. Ne pas le faire constitue une violation des politiques d'honnêteté académique.
- Vous devez **comprendre** le code que propose l'IA. Vous pouvez d'ailleurs lui demander de vous l'expliquer. Lors de votre évaluation individuelle **tout code que vous n'êtes pas capable d'expliquer entrainera une note de 0**.
- Réfléchissez au moment où cet outil est utile. Ne l'utilisez pas s'il n'est pas approprié au contexte du travail à fournir, **au minimum ayez un usage raisonné en lien avec l'impact environnemental associé**.

Questions?

3 – **Concepts de base des langages à objets**

Objets et Classes

Université de Rennes

"Objet"
(Définition)

- Formellement : la fermeture transitive d'une fonction
- Concrètement : encapsulation d'un état avec un ensemble d'opérations travaillant sur cet état
  - abstraction d'une entité du monde réel
  - existence temporelle :
    - création, évolution, destruction
  - identité propre à chaque objet
  - peut être vu comme une machine
    - ayant une mémoire privée et une unité de traitement,
    - et rendant un ensemble de services

Université de Rennes

Objet : identité vs. égalité

P1 = 

P2 = P1 => P2 == P1 // *identité*

P2 = 

⇒ P2 != P1 // *pas identité*
⇒ P2.equal(P1) == true // *Egalité*

Université de Rennes

"Classe" : Définition en temps que type

- Description des propriétés et des comportements communs à un ensemble d'objets
  - Implantation d'un type de donné abstrait
  - un objet est une instance d'une classe (eq. variable vs. type)
- Chaque classe a un nom, et un corps qui défini :
  - les attributs possédés par ses instances
  - les opérations définies sur ses instances,
    - et permettant d'accéder, de manipuler, et de modifier leurs attributs
  - une classe peut elle même être un objet (*reflexivité*)
    - Par ex. en Java : Class<?> maClasse = monObjet.getClass();
      - Et alors …. maClasse.getClass() ???

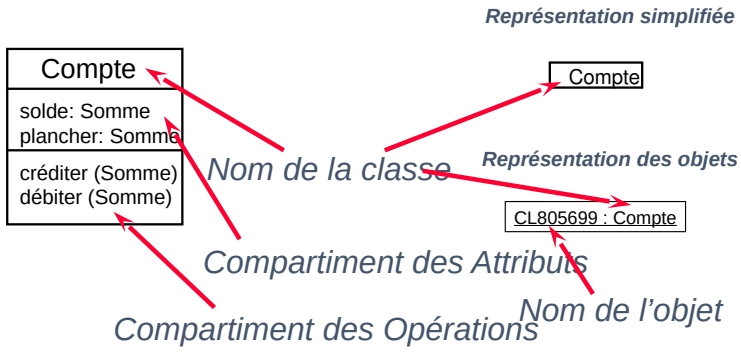
Université de Rennes

Représentation des Objets et des Classes

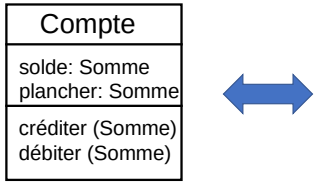
- UML : Langage de modélisation pour le logiciel *(Unified Modeling Language)*
  - Réfléchir
  - Définir la structure « gros grain » : Architecture Logicielle
  - Documenter
  - Guider le développement
  - Développer, Tester, Auditer

- UML permet de modéliser de nombreuses autres choses…
  - Mais ca, c'est pour la L3

- Représentation d'une classe



Correspondance UML 🔔 Java




```
class Compte {
public Compte(int plancher) {...}
private int solde;
private int plancher;
public int getSolde(){
   return solde;
 }
public void crediter(int montant) {
 }
public void debiter(int montant) {
 }
}
```

Conventions de nommage :
- Le nom d'une **classe** est un **nom**
- Les **attributs** sont des **noms**
- Les **méthodes** sont des **verbes**
  - *(à l'infinitif)*

Quelques langages « orientés objets »

Déjà utilisé *jamais utilisé*

(par ordre de popularité selon IEEE Spectrum)

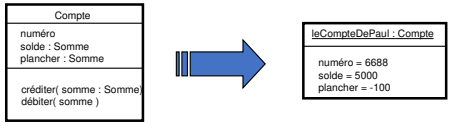
- **Python** (Académique)
  - -> scripts, ML
- **Java** (Oracle)
  - -> Serveurs etc.
- **C++** (Bell Labs)
  - -> infra/telecom/images
- **C#** (Microsoft)
  - -> .NET
- **Swift** (Apple)
  - -> IOS
- **Dart** (Google)
  - -> IOS + Android (Flutter)
- **Kotlin** (JetBrains+ Google)
  - -> Android

+ Quelques langages OO plus ésotériques:
- JavaScript/TypeScript
- Objective-C
- Go
- Scala
- Eiffel
- Julia

A distinguer de langages comme:
- SQL, HTML
- C, Fortran, Cobol
- Haskell, Rust, OCamel, LISP…

Vous devriez être capable de passer d'un langage orienté objet à l'autre avec un **minimum d'efforts**

Instanciation



- Java / C# / C++
  - `Compte leCompteDePaul = new Compte(-100);`
- Python
  - `leCompteDePaul = Compte(-100)`
- Kotlin
  - `leCompteDePaul = Compte(-100)`
- Dart
  - `var leCompteDePaul = Compte(-100)`
- Swift
  - `let leCompteDePaul = Compte(plancher:-100)`

```
class Compte:
   def __init__(self, plancher):
      self.plancher = plancher
```

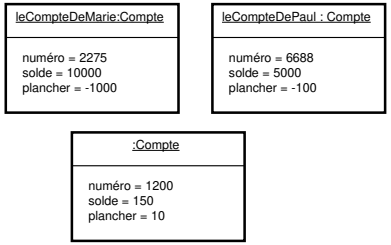
```
class Compte(val plancher: Int)
```

```
class Compte {  int plancher;
 Compte(this.plancher);}
```

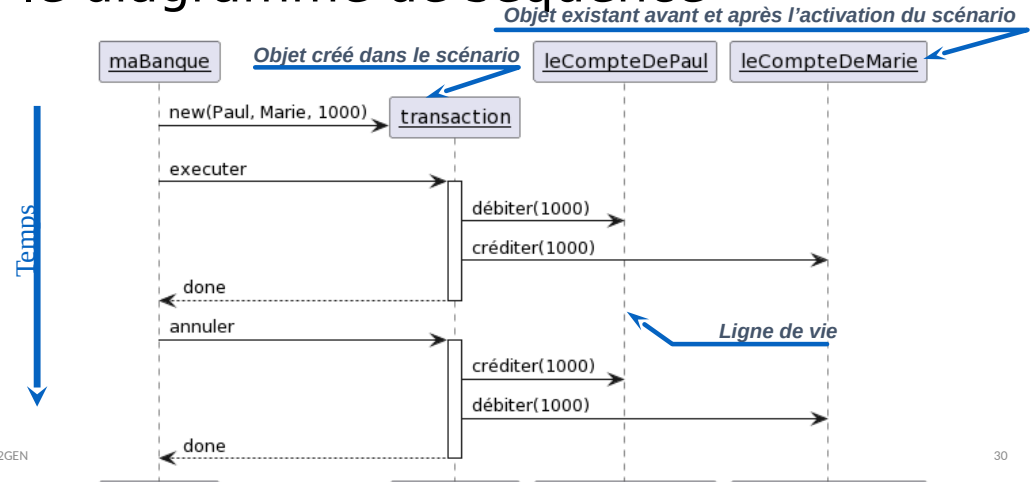
```
class Compte {
   var plancher: Int
   init(plancher : Int) {
      self. plancher = plancher  }
}
```

Diagramme d’objets

- Des **objets** peuvent être ajoutés ou détruits pendant l ’exécution
- La valeur des attributs des objets peut changer



Exemple de diagramme d’objets : le diagramme de séquence



Classe vs. Objets

Une classe *un objet*

Une **classe** spécifie la structure et le comportement d'un ensemble d'objets de même nature

- La structure d'une classe est constante

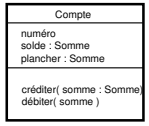


Diagramme de classes

M1

M0

- Des **objets** peuvent être ajoutés ou détruits pendant l ’exécution
- La valeur des attributs des objets peut changer

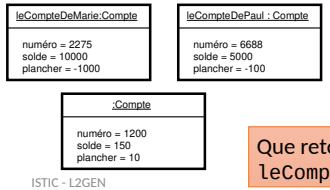


Diagramme d ’objets

Que retourne : `leCompteDeMarie.getClass()`?

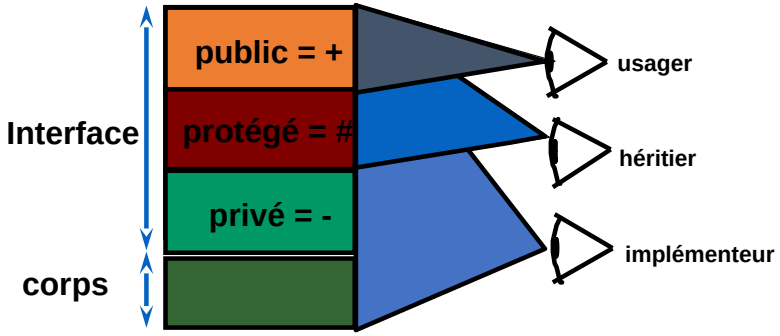
4 – Modularité

La “Classe” en temps que module

Principe de l'approche objet

- Structurer les systèmes autour des objets
  - Plutôt qu'autour des fonctions
- Obtenir des systèmes modulaires et maintenables
  - Par assemblage de briques de base
  - Composition d'éléments simples pour obtenir des systèmes sophistiqués
- Favorise :
  - masquage d'information (abstraction)
  - encapsulation (facilite les modifications à portée locale)
  - dissociation interface/implantation=> composant réutilisab

- Différentes visibilités des membres d'une classe



Contre exemple de modularité : le bug de l'an 2000

Encapsulation/masquage d'information

date
year: String[2]

equal (date)
isLess(date)
delta(date)

- Un choix de format de stockage pour l'année
  - 3 opérations principales (et leurs dérivées) + print
- Correction bug an 2000?
  - modification du format de stockage
- Sans encapsulation/masquage d'information:
  - Cout de la modification proportionnel à la taille du programme
    - Car il faut passer sur toutes les lignes pour reporter l'éventuelle modification
- Avec encapsulation/masquage d'information:
  - Cout de la modification indépendant de la taille du programme
    - Le reste du code est protégé de mes choix

Notion d'API « *Application Programming Interface* »

- **Interface de Programmation Applicative**
  - ensemble de classes, de méthodes, de fonctions et de constantes
  - qui sert de *façade* par laquelle un logiciel offre des services à d'autres logiciels
- Le programmeur n'a pas besoin de connaître les détails de la logique interne du logiciel qui se cache derrière la façade de son API
  - Seule l'API est réellement nécessaire pour utiliser le système
- Convention de nommage
  - Noms pour les classes (Compte)
  - Verbes infinitif pour action (créditer)
  - Noms pour attributs (solde)
- En anglais idem
  - + getter & setter

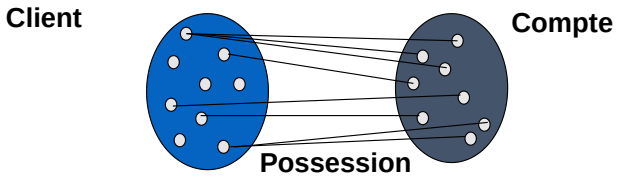
5 – Liens entre Objets

Et associations entre classes

Les objets ne sont pas définis isolément

- 2 catégories
  - Liens entre objets
    - Marie possède les comptes en banque n°12345 et n° 67890
    - Cette voiture est composée du châssis n° XX564 et des 4 roues AvG, AvD, ArG, ArD
    - => Associations entre classes
      - Une Personne possède un ou plusieurs Comptes en banque
      - Une Voiture est composée de 1 Châssis et de 4 Roues
  - Classification
    - Médor est un Chien, il est donc aussi un Mammifère
      - Médor appartient à l'ensemble des Chiens, qui est contenu dans l'ensemble des Mammifères
    - => Héritage entre classes
      - Un Chien est une sorte de Mammifère, qui est une sorte d'Animal

Vue ensembliste d'une relation : Graphe de la relation

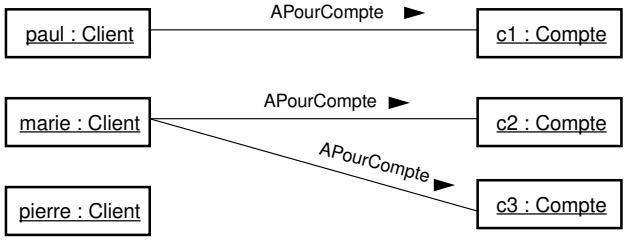


Une relation met en correspondance des éléments d'ensembles

Université de Rennes

Liens (entre objets)

Un **lien** indique une connexion entre deux objets



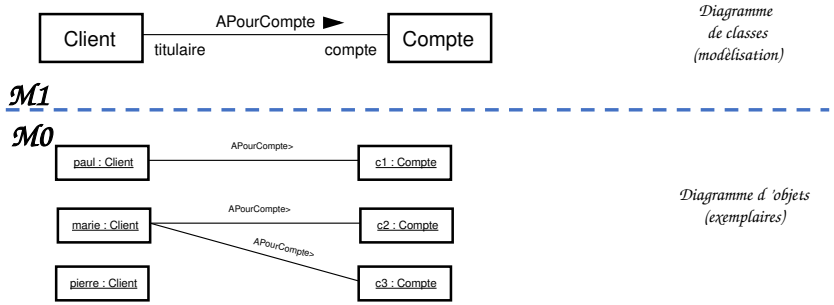
Note de style :

- les noms des liens sont des formes verbales et commencent par une majuscule
- ► indique le sens de la lecture (ex: « paul APourCompte c1 »)

Université de Rennes

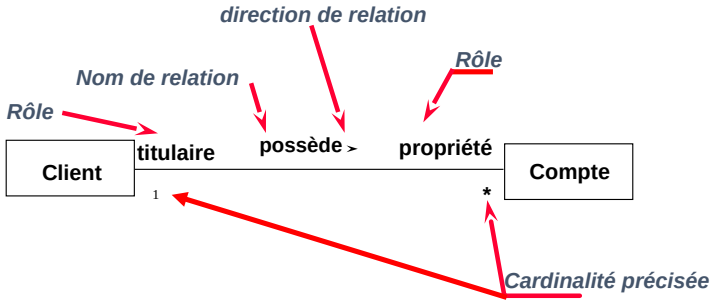
Associations (entre classes)

Une **association** décrit un ensemble de liens de même "sémantique"



Université de Rennes

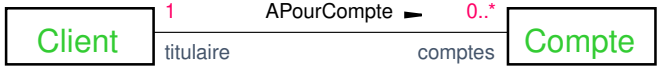
Représentation des associations : direction, rôle, cardinalité



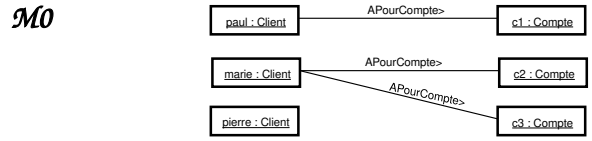
Université de Rennes

Cardinalités d'une association

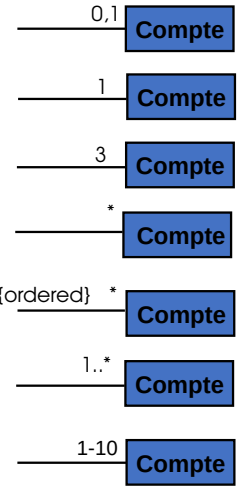
- Précise combien d'objets peuvent être liés à un seul objet source
- Cardinalité minimale et cardinalité maximale ($C_{min}..C_{max}$)
- Doivent être des constantes



« Tout client a toujours 0 ou plusieurs comptes »
« Tout compte a toujours 1 et 1 seul titulaire »



Cardinalité d'une association



Optionnelle (0 ou 1)

Exactement un

Exactement 3

Plusieurs, non ordonnés

Plusieurs, ordonnés

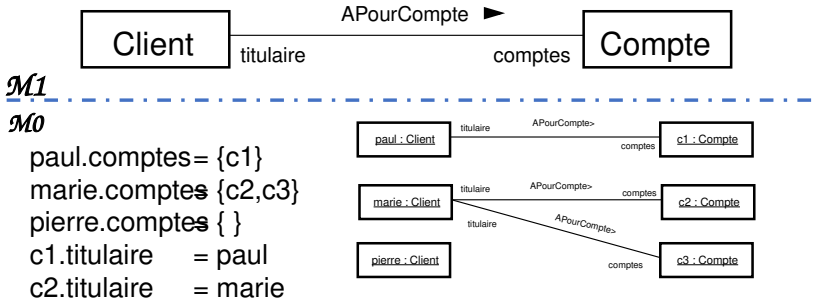
Au moins un

Intervalle

En Java par exemple :

```
Optional<Compte> c;
Compte c; //or @NotNull
Compte c1, C2, c3;
Set<Compte> c;
List<Compte> c;
Set<Compte> c; // @NotEmpty
Compte[] c = new Compte[10];
```


Utiliser les rôles pour «naviguer»

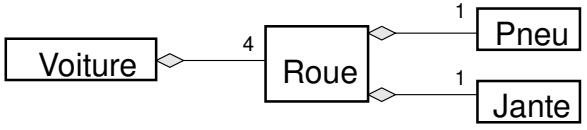


paul.comptes = {c1}
marie.comptes = {c2,c3}
pierre.comptes = { }
c1.titulaire = paul
c2.titulaire = marie
c3.titulaire = marie

Nommer en priorité les rôles

Composition

Notion intuitive de "composants" et de "composites"

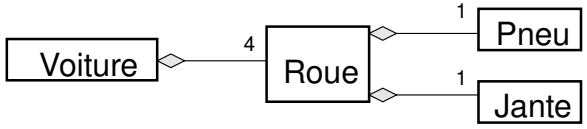


composition =
cas particulier d'association
+ contraintes décrivant la notion de "composant"...

Composition

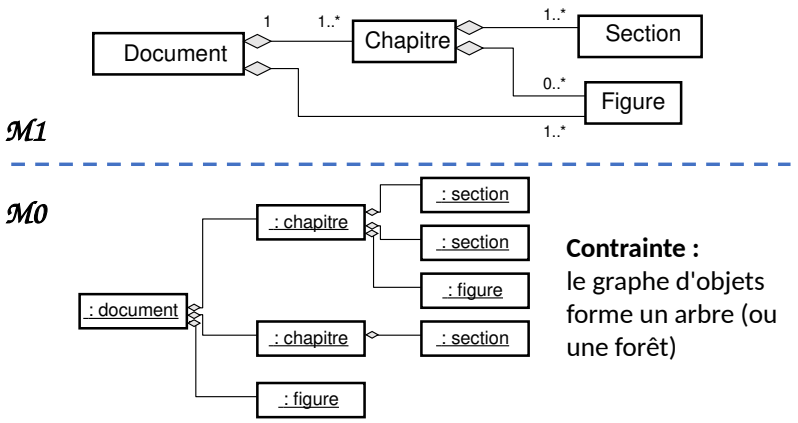
Contraintes liées à la composition :

1. Un objet *composant* ne peut être que dans 1 seul objet *composite*
2. Un objet *composant* n'existe pas sans son objet *composite*
3. Sun objet composite est détruit, ses composants aussi



Dépend de la situation modélisée !
(Ex: vente de voitures vs. casse)

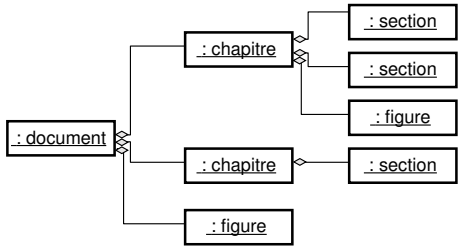
Composition



Contrainte :
le graphe d'objets forme un arbre (ou une forêt)

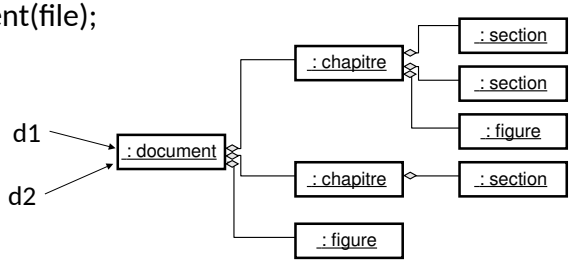
Copie d'objets composites

- Document d1 = new Document(file);



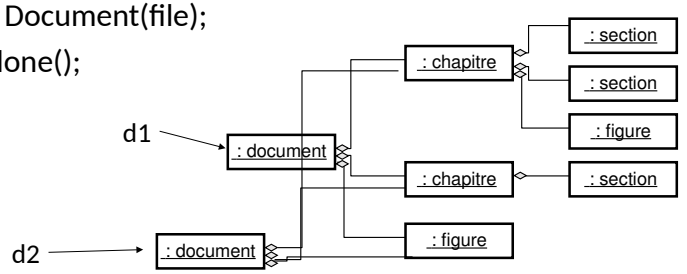
Affectation

- Document d1 = new Document(file);
- Document d2 = d1;



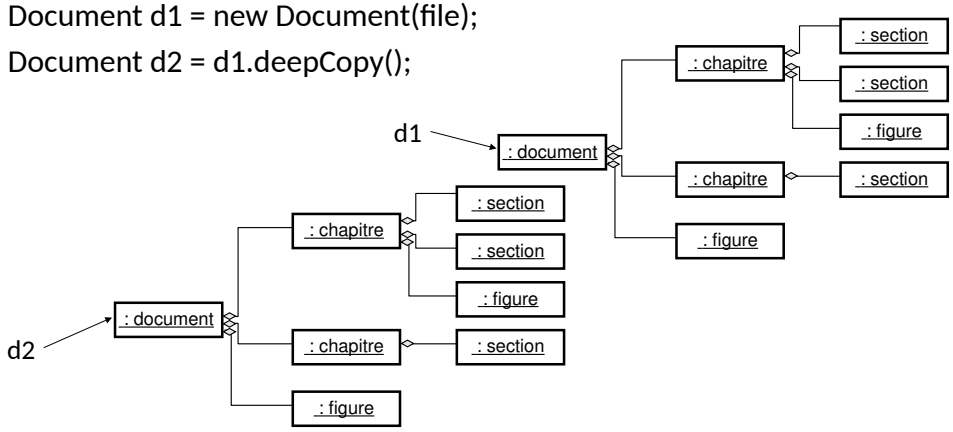
Copie simple

- Document d1 = new Document(file);
- Document d2 = d1.clone();



Copie profonde

- Document d1 = new Document(file);
- Document d2 = d1.deepCopy();



Copie d'objets complexes / Identité vs Egalité

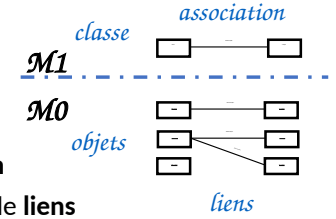
1. Affectation
   - d1 = d2;
2. Copie simple
   - d1 = d2.clone();
3. Copie profonde
   - d1 = d2.deepClone();

1. Test d'identité
   - d1 == d2
2. Test d'égalité
   - d1.equals(d2);
3. Test d'égalité profonde
   - d1.deepEquals(d2);

Support plus ou moins direct en fonction du langage
- Java : seulement 1 et ~2
- Python : 1, 2 (copy) et 3 (deepcopy)

Association vs. Liens

- Un **lien** lie deux **objets**
- Une **association** lie deux **classes**



- Un **lien** est une instance d'**association**
- Une **association** décrit un ensemble de **liens**

- Des **liens** peuvent être ajoutés ou détruits pendant l 'exécution, (ce n 'est pas le cas des associations)

6 – Classification

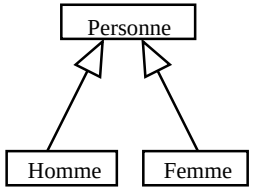
Héritage entre classes

Les objets ne sont pas définis isolément

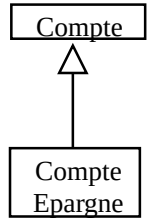
- 2 catégories
  - Liens entre objets
    - Marie possède les comptes en banque n°12345 et n° 67890
    - Cette voiture est composée du châssis n° XX564 et des 4 roues AvG, AvD, ArG, ArD
    - => Associations entre classes
      - Une Personne possède un ou plusieurs Comptes en banque
      - Une Voiture est composée de 1 Châssis et de 4 Roues
  - Classification
    - Médor est un Chien, il est donc aussi un Mammifère
      - Médor appartient à l'ensemble des Chiens, qui est contenu dans l'ensemble des Mammifères
    - => Héritage entre classes
      - Un Chien est une sorte de Mammifère, qui est une sorte d'Animal

Héritage et Polymorphisme : Généralisation / Spécialisation

Une classe peut être la généralisation d'une ou plusieurs autres classes.
Ces classes sont alors des spécialisations de cette classe.



"Super classe"
Cas général



"Sous classes"
Cas spécifique

On dit que CompteEpargne **hérite** *de Compte*

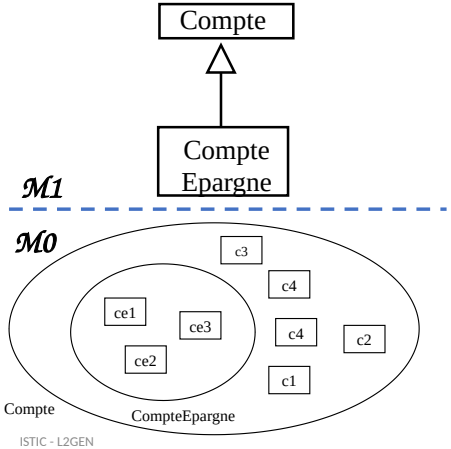
Deux points de vue complémentaires

- Mécanisme de classification : sous-typage
  - relation X est-une-sorte-de Y (est substituable à)
  - organisation des systèmes complexes (cf. Linnaeus)
  - réutilisation d'interface

- Mécanisme d'extension de module
  - ajout de fonctionnalités dans la sous-classe
  - "customisation" et combinaison de composants logiciels
  - réutilisation de code

Héritage : Vision ensembliste

Tout objet instance d'une sous-classe appartient également à l'extension de la superclasse



Liskov's Substitution Principle (LSP)
Partout où un Compte est attendu, on peut toujours utiliser un CompteEpargne à la place

Héritage dans les langages à objets

- Java / Dart

```
class CompteEpargne extends Compte {
 void ajouterIntérêts() {…}
}
```

- Python

```
class CompteEpargne (Compte):
 def ajouterIntérêts(self):
```

- C++/ C#

```
class CompteEpargne : Compte {
 void ajouterIntérêts() {…}
}
```

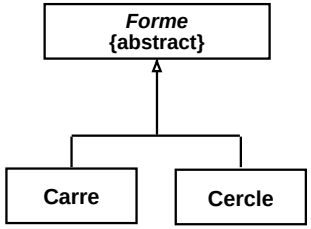
- Swift

```
class CompteEpargne : Compte {
    func ajouterIntérêts() {…}
}
```

- Kotlin

```
class CompteEpargne : Compte(){
    fun ajouterIntérêts() {..}
}
```

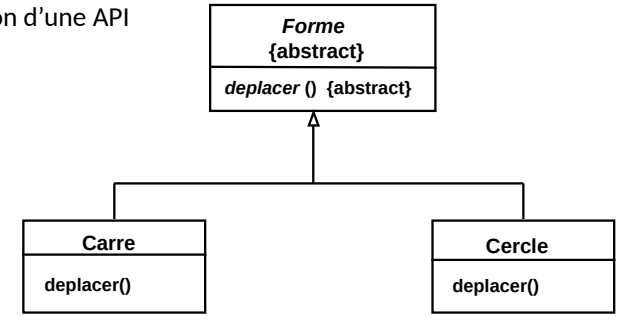

- Capturent des comportements communs
- Ne peuvent donc pas être instanciées
- Servent à structurer un système
- Peuvent avoir des opérations dont l'implantation est absente
  - *pure virtual* en C++
  - *abstract* en Java, C#, Kotlin etc.



- Classes sans instances immédiates

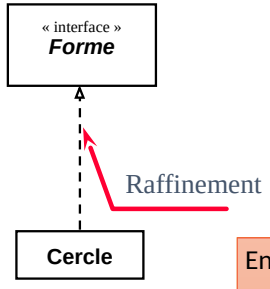
Une instance de «**Forme**» est obligatoirement une instance de la classe **Carre** ou de la classe **Cercle**

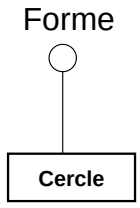
- Opération sans corps d'une classe abstraite
  - Définition d'une API



Interfaces et « lollipop »

- Interface = classe « totalement » abstraite
  - Dont toutes les méthodes sont abstraites



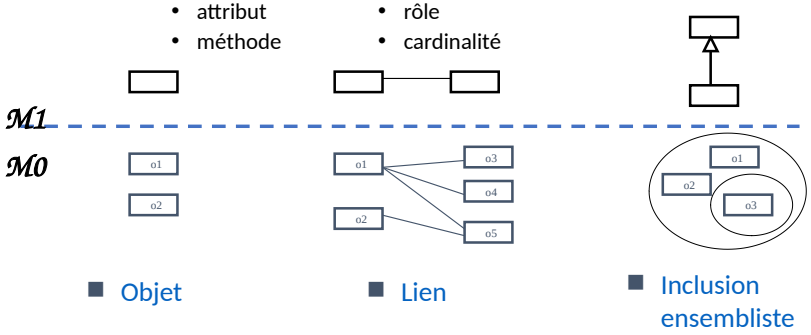


En Java :

```
class Cercle implements Forme {
    …
}
```

Synthèse des concepts de base

- Classe
  - attribut
  - méthode
- Association
  - rôle
  - cardinalité
- Héritage



- Objet
- Lien
- Inclusion ensembliste

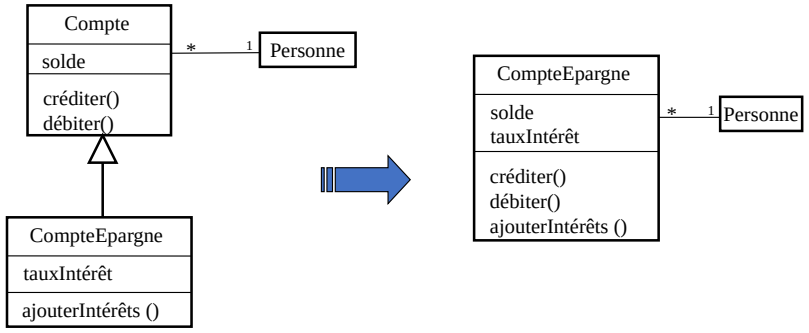
Deux points de vue complémentaires

- Mécanisme de classification : sous-typage
  - relation X est-une-sorte-de Y (est substituable à)
  - organisation des systèmes complexes (cf. Linnaeus)
  - réutilisation d'interface

- Mécanisme d'extension de module
  - ajout de fonctionnalités dans la sous-classe
  - "customisation" et combinaison de composants logiciels
  - réutilisation de code

Relation d'héritage

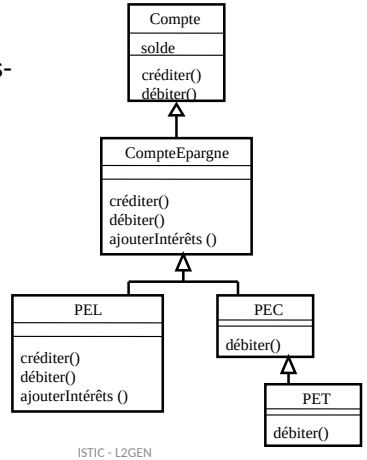
Les sous-classes « *héritent* » des propriétés des super-classes (attributs, méthodes, associations, etc.)



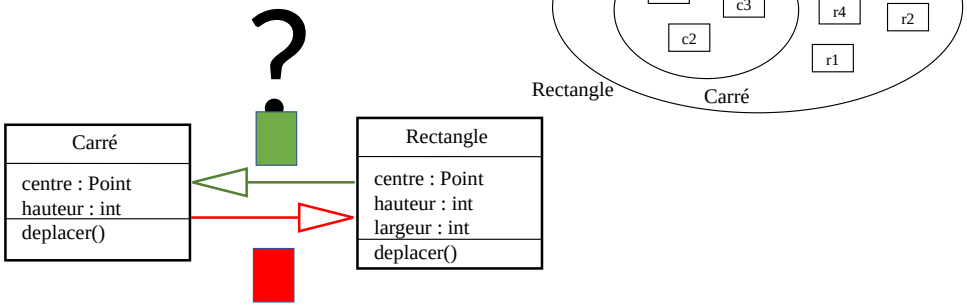
Relation d'héritage et redéfinitions

Une opération peut être "redéfinie" dans les sous-classes

Permet d'associer des méthodes spécifiques à chaque pour réaliser une même opération



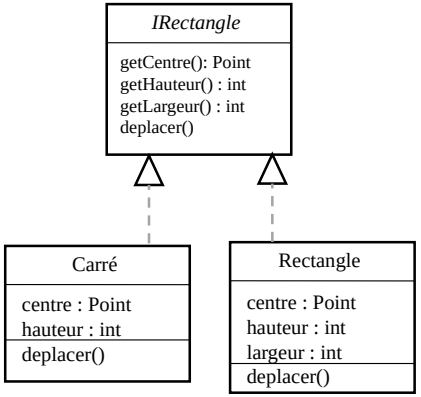
Carré vs. Rectangle



Parfois besoin de dissocier sous-typage et héritage d'implantation!

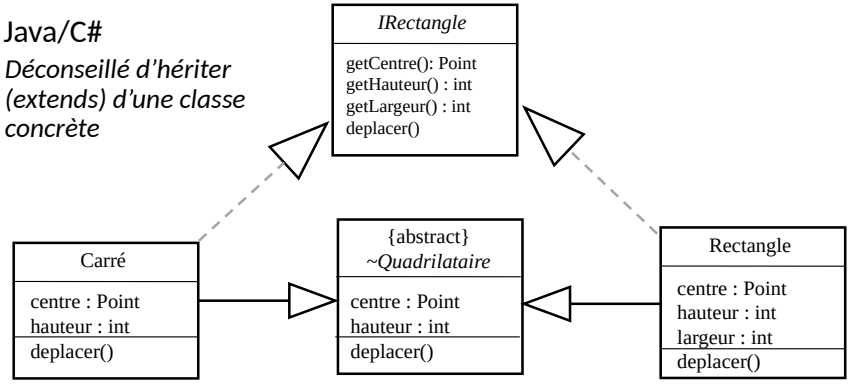
Distinguer Sous-Type de Sous-Classe

- En Java/C#



Distinguer Sous-Type de Sous-Classe

- En Java/C#
  - *Déconseillé d'hériter (extends) d'une classe concrète*



7 – Polymorphisme

Et liaison dynamique

Polymorphisme et liaison dynamique

- Polymorphisme : possibilité de changer de forme

  Forme f; c = new Cercle(); r = new Rectangle();

  f = c;

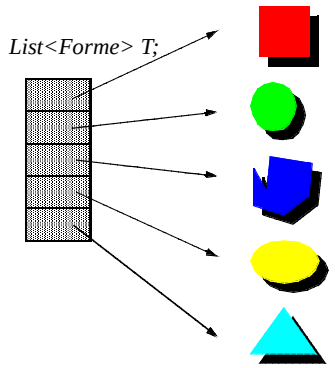
  f = r;
- Liaison dynamique : l'effet de l'appel d'une opération d'un objet dépend de sa forme effective à l'exécution
  - f.imprimer(); -- différent selon que f est Cercle ou Rectangle
- Espace de nommage réduit et uniforme
  - Mise en facteur des parties communes
  - Possibilité de "conteneurs" hétérogènes

Polymorphisme : exemple

- T contient des Formes
  - en fait des instances de sous-types de Forme
- Programmes de type :

```
T[0] = new Carré();
T[1] = new Cercle();
...
for (Forme f : T)  f.imprimer();
```

- Si ajouts ultérieurs:
  - e.g. triangle
- Pas de modifications globales
  - case-less programming



Parcours d'arbre orienté objet

- Un Groupe contient
  - Des cercles, des rectangles…
  - …et d'autres groupes
- Ce qui forme une structure d'arbre

```
void deplacer(double dx, double
dy){
  for (Forme forme : formes) {
     forme.deplacer(dx, dy);
  }
}
```

```
void deplacer(double dx, double dy){
  formes.forEach(f-
>f.deplacer(dx,dy));
}
```

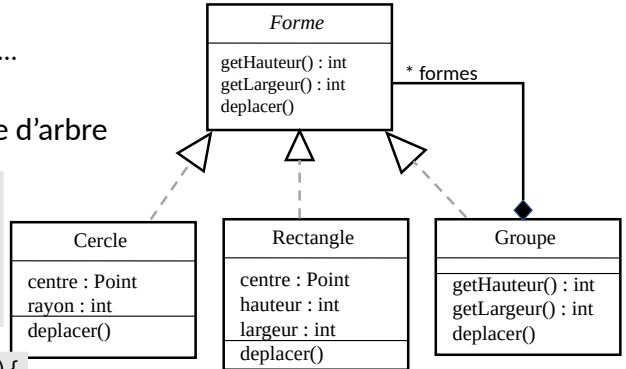
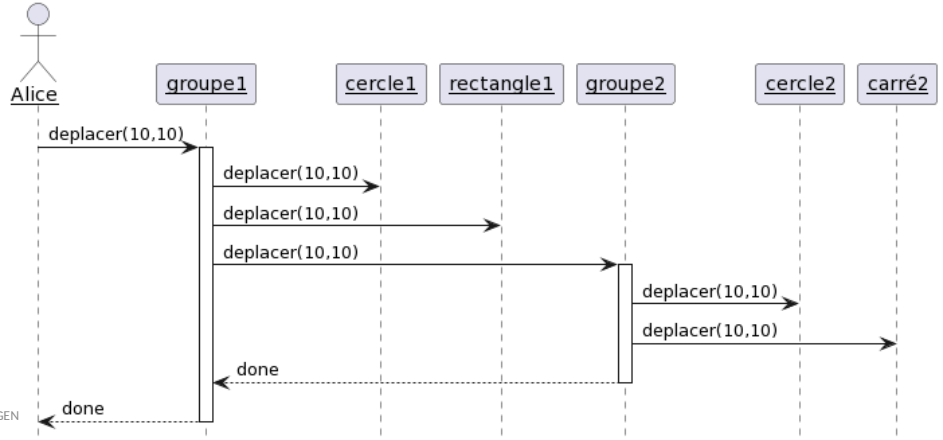


Illustration de la dynamique avec un diagramme de séquence

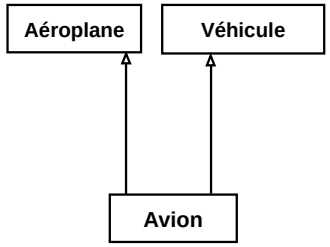


Université de Rennes

8 – Héritage multiple

Héritage entre classes

- Héritage multiple, exemple :
  - Un avion est *à la fois*
    - Un Aéroplane
    - Un Véhicule de transport
  - L'ensemble des avions est donc *l'intersection* entre
    - l'ensemble des aéroplanes et
    - l'ensemble des véhicules de transport

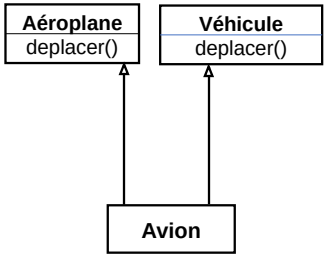


- Réalisation de ce concept très différente selon les langages de programmation
  - Seuls Python, C++ et Eiffel le supportent directement
  - Les autres (dont Java) le supportent indirectement

```
#Python
class
Avion(Aéroplane,Véhicule):
```


Problème avec l'héritage multiple

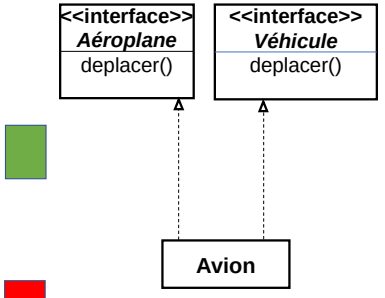
- Avion hérite de quelle version de déplacer()?
  - Aéroplane::déplacer()
  - Véhicule::déplacer()
- Des algorithmes complexes pour fonctionner dans le cas général (Python)
- Des constructions ad hoc pour aider le développeur à résoudre ces ambiguïtés (C++, Eiffel)
- Java : seule l'héritage multiple d'interfaces est autorisé



Héritage multiple d'interfaces en Java

```
// Java?
class Avion implements Aéroplane,
Véhicule {
//…
}
```

```
// Java?
class Avion extends Aéroplane, Véhicule {
//…
}
```



Université de Rennes

Avec la notion de Trait (Java 8)

```
public interface Aéroplane {
  public default void deplacer() {
    System.out.println("deplacer un aéroplane");
  } }
```

```
public interface Véhicule {
  public default void deplacer() {
    System.out.println("deplacer un véhicule");
  } }
```

```
public class Avion implements Aéroplane, Véhicule {
  public void deplacer() {
    Véhicule.super.deplacer();
  } }
```


Université de Rennes

9 – Typage statique

Détecter des erreurs dès la compilation

Université de Rennes

Typage dans les langages objets

- Type de chaque objet est déterminé par la classe dont il est instance
  - Défini quelles opérations peuvent être appelées sur l'objet, et avec quels paramètres
  - x.f() légal <=> l'objet désigné par x dispose d'une méthode f()
- liaison dynamique :
  - la *bonne* interprétation de f est choisie, selon le type de x

Université de Rennes

Typage dynamique

- Type de chaque objet est déterminé par la classe dont il est instance
  - Défini quelles opérations peuvent être appelées sur l'objet, et avec quels paramètres
- **Typage dynamique :** les variables peuvent désigner n'importe quel objet
  - Exemple : JavaScript ou Python
    - c = Cercle() ; c = Compte()
  - Les erreurs de type sont **détectées pendant l'exécution** du programme
    - c.crediter(100) // OK
    - c.deplacer(10,20) // Erreur

Université de Rennes

Typage statique

- Type de chaque objet est déterminé par la classe dont il est instance
  - Défini quelles opérations peuvent être appelées sur l'objet, et avec quels paramètres
- **Typage statique :** les variables doivent être déclarées avec un type spécifique
  - Compte c = new Compte();
  - Ce type ne peut généralement pas être modifié par la suite.
  - Les opérations sur les variables sont vérifiées par le compilateur pour s'assurer qu'elles respectent les règles du type déclaré
  - Les erreurs de type sont **détectées avant l'exécution** du programme
    - ce qui permet d'identifier et de corriger les problèmes liés aux types dès le stade de développement

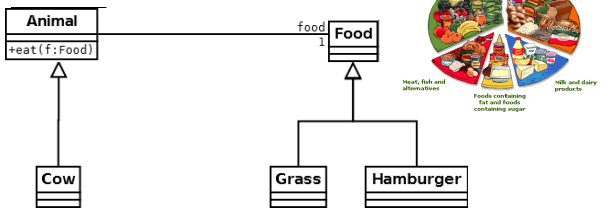
Université de Rennes

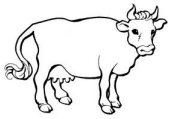
Héritage et typage

- Typage statique :
  - x.f() légal <=> vérification avant l'exécution que tout objet potentiellement désigné par x dispose d'une méthode f()
- liaison dynamique :
  - la bonne interprétation de f est choisie
- Combinaison de ces notions
  - Python, JavaScript : typage dynamique, liaison dynamique
  - Java, C#, Swift, Dart, Kotlin : typage statique, liaison dynamique
  - C++ : typage statique, liaison dynamique
    - pour les fonctions "virtuelles"
  - UML : au choix (!)

Université de Rennes

Co-variance et contra-variance



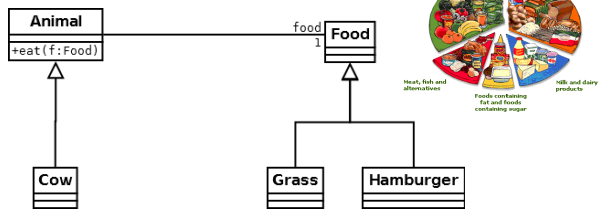


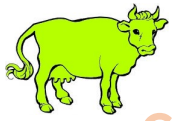
```
Animal a = new Animal()
Food f = new Food()
a.eat(f)
```

```
Animal a = new Cow()
Food f = new Food()
a.eat(f)
```

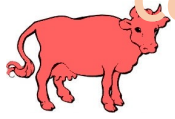

Université de Rennes

Co-variance et contra-variance





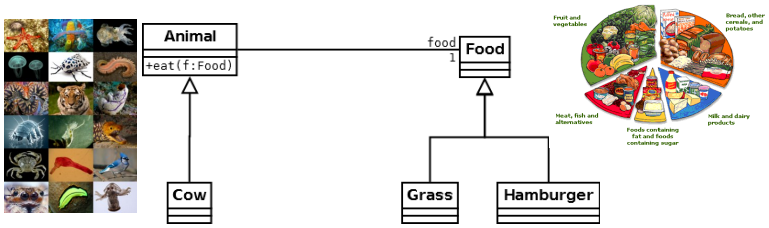
```
Cow a = new Cow()
Grass f = new Grass()
a.eat(f)
```



Comment interdire ça ?

```
Cow a = new Cow()
Food f = new Hamburger()
a.eat(f)
```

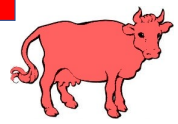
Co-variance et contra-variance



Redéfinition ***covariante***

Légal en Java ?

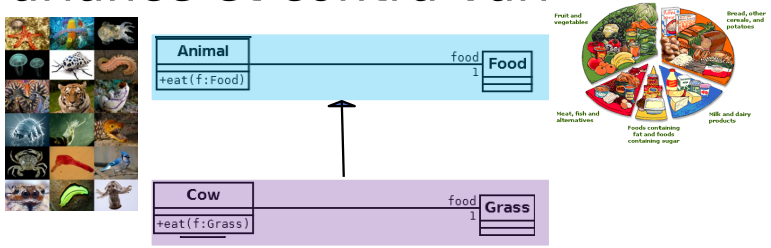
Illégal?



Ne règle pas le problème !

```
Animal a = new Cow()
Food f = new Hamburger()
a.eat(f)
```

Co-variance et contra-variance



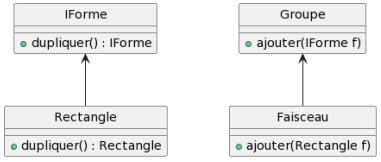
- En Java (& autres langages à typage statique) : généricité paramétrique
  - **Class** Animal<T **extends** Food> {void eat(f:T){}}
  - **Class** Cow **extends** Animal<Grass>{} // *devient implicitement* eat(f:Grass)
- Erreur *cow.eat(new Hamburger())* détectée à la compilation

Co-variance et contra-variance

- **Covariance**
  - **Définition :** Autorise la substitution d'un type dérivé (sous-type) à la place d'un type de base (super-type)
- **Contravariance**
  - Autorise la substitution d'un type de base à la place d'un type dérivé
- Règle en Java lors de la redéfinition de méthodes
  - Covariance pour les types de retours
  - Contravariance pour les paramètres
    - Exemples :

Légal

Illégal



10 – Notion de package

Modularisation à l'échelle supérieure

Université de Rennes

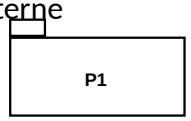
Notion de package

- Élément structurant les classes
  - Modularisation à l'échelle supérieure
  - Un package partitionne l'application :
    - Il référence ou se compose des classes de l'application
    - Il référence ou se compose d'autres packages
  - Un package réglemente la visibilité des classes et des packages qu'il référence ou le compose
  - Les packages sont liés entre eux par des liens d'utilisation, de composition et de généralisation
  - Un package est la représentation informatique du contexte de définition d'une classe

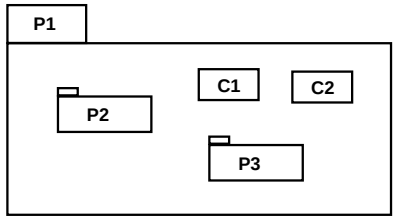
Université de Rennes

Représentation d'un package

- Vue graphique externe



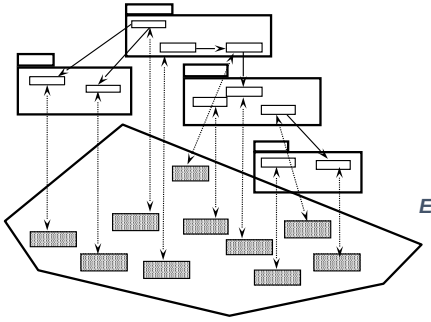
- Vue graphique externe et interne



Université de Rennes

Partitionnement d'une application

- Définition de vues partielles d'une application



SYNTHÈSE EN PACKAGES

ENSEMBLE DES CLASSES DE L'APPLICATION

N.B.: ***une classe appartient à un et un seul package***

Université de Rennes

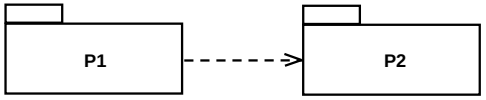
Visibilité dans un package

- Réglementation de la visibilité des classes
  - **Classes de visibilité publique** :
    - classes utilisables par des classes d'autres packages
  - **Classes de visibilité privée** :
    - classes utilisables seulement au sein d'un package
- Représentation graphique

CLASSE D'INTERFACE *CLASSE DE CORPS* *CLASSE EXTERNE*

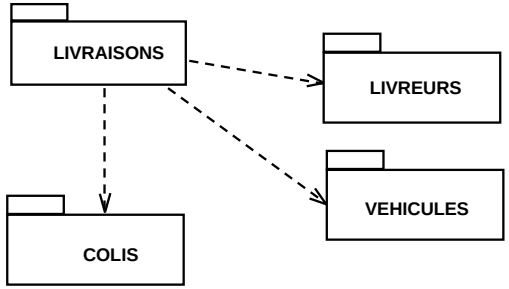
Utilisation entre packages

- Définition
  - Il y a utilisation entre packages si des classes du package utilisateur accèdent à des classes du package utilisé
  - Pour qu'une classe d'un package p1 puisse utiliser une classe d'un package p2, il doit y avoir au préalable une déclaration **explicite** de l'utilisation du package p2 par le package p1
- Représentation graphique



Vue externe du package P1

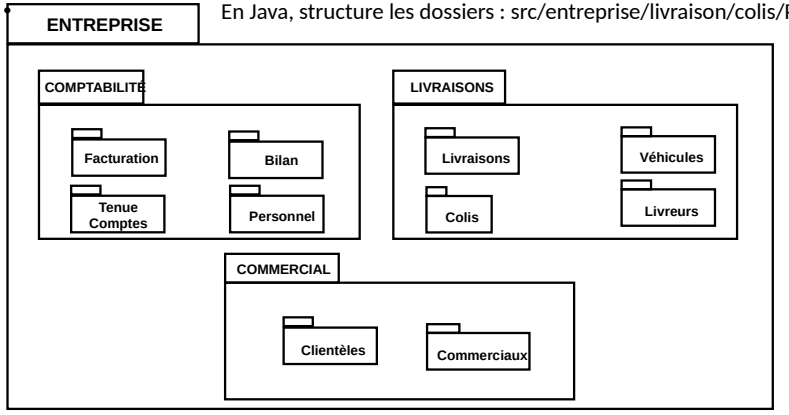
- Exemple (vue externe du package livraisons)



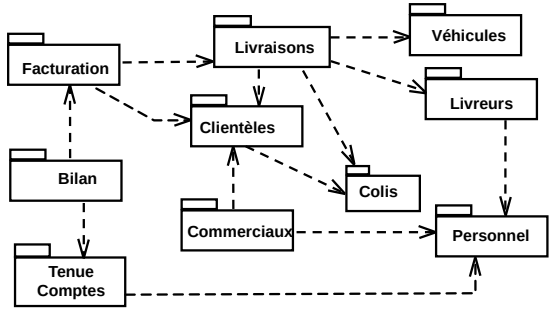
Architecture logicielle : vue hiérarchique

- Composition des packages en sous-packages

En Java, structure les dossiers : src/entreprise/livraison/colis/Produit.java



Architecture logicielle : dépendances

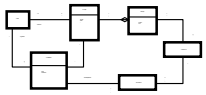


Utilité des packages

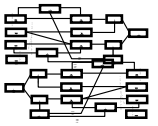
- Réponses au besoin
  - Contexte de définition d'une classe
  - Unité de structuration
  - Unité d'encapsulation
  - Unité d'intégration
  - Unité de réutilisation
  - Unité de configuration
  - Unité de production

Synthèse diagrammes de classes et d'objets

- Un diagramme de classes
  - Défini l'ensemble de tous les états possibles
  - les contraintes doivent toujours être vérifiées



- Un diagramme d'objets
  - décrit un état possible à un instant t, un cas particulier
  - doit être conforme au modèle de classes



- Les diagrammes d'objets peuvent être utilisés pour
  - expliquer un diagramme de classe (donner un exemple)
  - valider un diagramme de classe (le "tester")