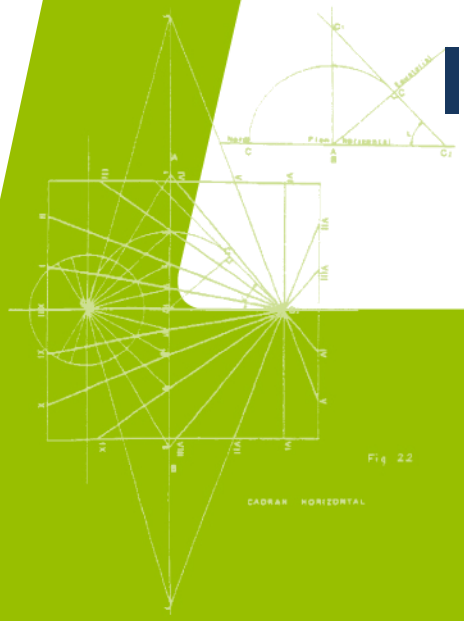
INF3055
2021-2022

Université de Yaoundé I

INF3055 : Conception Orientée Objet

Introduction Générale



Octobre 2021

Valéry MONTHE

valery.monthe@facsciences-uy1.cm

Bureau R114, Bloc pédagogique 1

Présentation du cours

- **Pré-requis:**
  - Les compétences de base en structure de données
  - Les compétences en programmation orienté-objet
  - Les notions de base en génie logiciel et système d'information

- **Matériels:**
  - Cahier de cours et TD
  - Ordinateur personnel
  - Outils de modélisation : *StarUML, Visual Paragdigme, ArgoUML, Smart Drawer, Enterprise Architect, etc*.

Présentation du cours

- **Objectifs:**
  - Présenter les principes et concepts de bases de la conception orientée-objet
  - Présenter le processus de développement et la place de la conception dans ce processus
  - Introduire la notion de modèle, de modélisation, et la modélisation orientée objet
  - Étudier un langage de conception et de modélisation orientée-objet : UML
- **Compétences visées:**

  A la fin de ce cours, l'étudiant doit être capable de :
  - Élaborer des solutions réutilisables et extensibles
  - Conduire la modélisation d'un logiciel à l'aide du langage UML
  - Construire différents diagrammes UML selon la phase du cycle de développement
  - identifier, bien structurer les différentes classes et produire le code source correspondant à ce diagramme.
  - Définir le code d'une méthode à partir d'un diagramme de collaboration

Présentation du cours

Contenu

1. Rappels sur le processus de développement
2. La conception dans le processus de développement
3. L'approche fonctionnelle et l'approche objet
4. Les principes et concepts de base de l'orientée objet
5. Etude du langage UML
6. Analyse orientée-objet
7. Conception orientée-objet
8. Traduction dans les langages de programmation
9. Etude d'une méthode de modélisation basée sur UML : le processus Unifié

Déroulement du cours

- **CM** :
  - Des séances de 3h
  - Lieu : **En ligne** et en **salle de cours**
- **TD/TP :**
  - Des séances de 2 h
  - Lieu : **En ligne** et en **salle de cours**
- **TPE:**
  - Lectures
- **Outils:**
  - Supports de cours
  - Cahier de cours et TD
  - Ordinateurs personnels

Règles du Cours

- **Accès:**
  - Plus d'entrées possibles après 20 min
  - Cas des séances en ligne : renseignez vos noms et prénoms
- **Discipline:**
  - Respect, courtoisie, ponctualité et assiduité
  - Cours en ligne :
    - activer son micro sur autorisation,
    - lieu calme, pas de vidéo, etc.
- **Mail:**
  - près de 50 mails reçu par jour
  - Objets de vos mails : [INFO305] au début de l'objet
  - **Exemple : [INF3055] : remise du TP N°1**
- **Remise de travaux :** pénalité de 10% par jour de retard

Evaluations

- **Présences :** Prise en compte
- **TD :**
  - Contrôle des exercices faits à domicile
  - Petites évaluations basées sur des exercices de la fiche de TD
- **CC:**
  - Contrôle continu écrit
- **Examen:**
  - Examen écrit en salle
- **TP:**
  - TP individuels
  - Projets en groupes

Cycle de développement

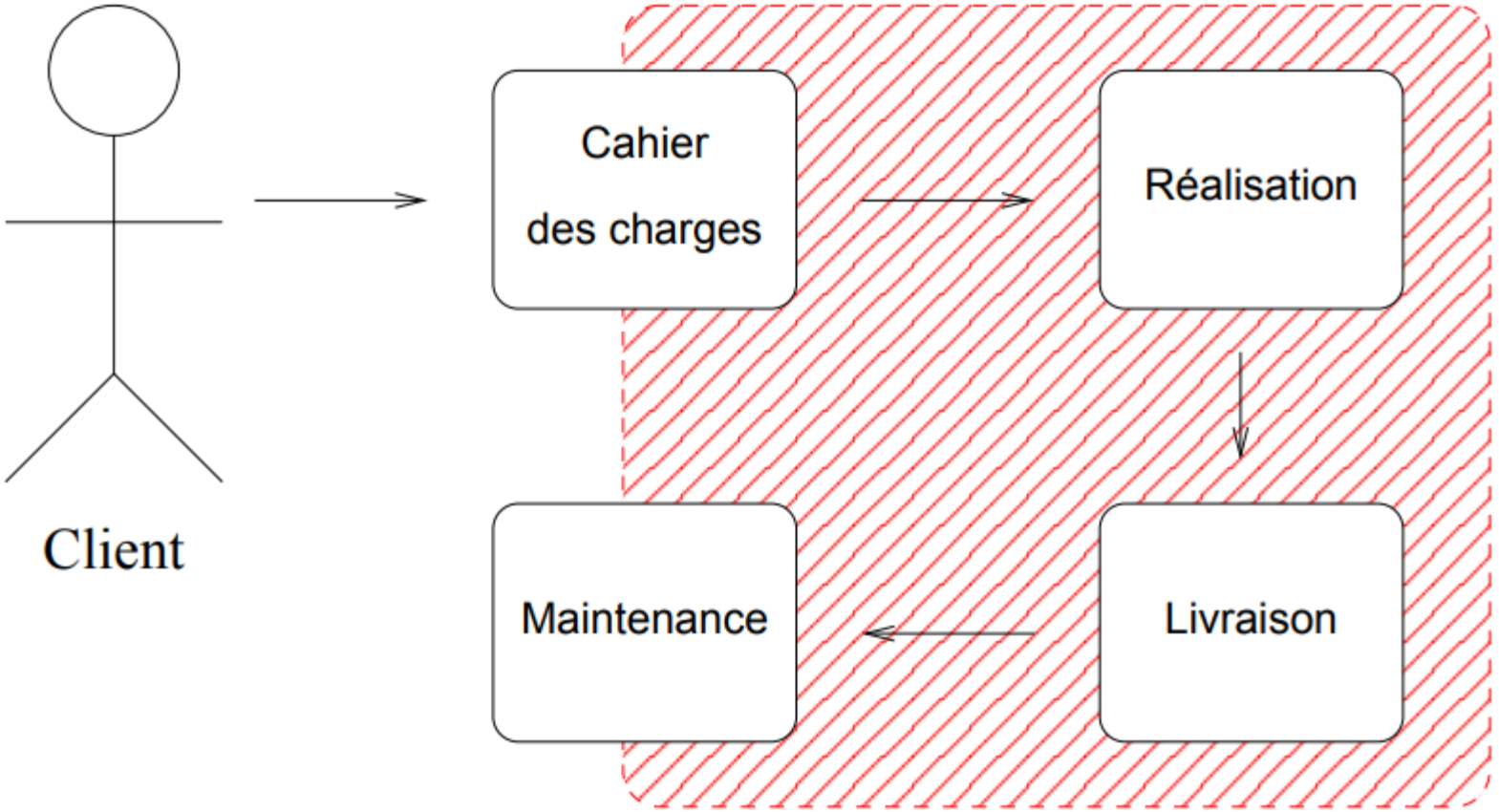
Rappels

Cycle de vie du logiciel

Le cycle de vie du logiciel se décompose en :

- **Pré-développement**
  - Précède le développement proprement dit
  - Pourquoi faut-il réaliser le logiciel?
  - Ressources nécessaires : budget, personnel, matériel, etc,
  - Question de délai et planification du développement
  - => *cahier de charges du projet ou spécification du projet*
- **Développement**
  - Contient les différentes activités du développement, de l'analyse des besoins au test d'acceptation
- **Post-développement**
  - Installation, exploitation, maintenance et support, retrait
- **Autres activités**
  - V & V, gestion de la configuration, documentation, formation

Processus de développement



Source : support de Emmanuel Polonowski

Processus de développement

1. Recueil des besoins
2. **Analyse**
    1. **Analyse des besoins**
    2. **Spécification du système**
3. **Conception**
    1. **Conception architecturale / générale**
    2. **Conception détaillée**
4. Implémentation
    1. Coder / programmer
    2. Tester

Processus de développement

1. Recueil de besoins

- Comprendre clairement ce que veut le client : besoin brut ou besoin client
- Qu'est ce que le logiciel devra faire?
- Comment fonctionnera t-elle? Quel est son périmètre?
- Le MOA s'intéresse au seul comportement externe : boîte noire
- Le MOE essaye de comprendre ce que fera cette boite noir
- Recueillir les exigences fonctionnelles et techniques du logiciel
- Les exigences qualitatives souhaitées

2. Analyse des besoins

- Décrire les services que l'application doit rendre
- Définir précisément quelles sont les fonctionnalités que l'application devra avoir et visibles des utilisateurs
- Définir les contraintes sous lesquelles l'application doit être développée et s'exécuter
- Principales questions à se poser :
  - A qui l'application est-elle destinée?
  - Quels sont les services à rendre à chacun des utilisateurs?
  - Comment sera-t-elle utiliser : organisation des écrans, interactions, etc, => Maquettage des IHM

=> Sorties : *document de spécification des besoins ou cahier de charge fonctionnelle, plan de contrôle et d'assurance qualité, plan de recette.*

3. Spécification du système

- Décrire la structure du système
- Identifier et décrire les éléments qui interviennent dans le système d'information, leurs structures et relation
  - Savoir – faire des entités (objets) : *axe fonctionnel*
  - Structure des entités : a*xe statique*
  - Comportement/cycle de vie des entités : *axe dynamique*
- Modéliser le système d'information existant
  - Modélisation des processus métiers : qui fait quoi, quand et où dans le métier
  - Modélisation du domaine : spécifier les concepts fondamentaux du domaine cible de l'application
  - Modélisation de l'application : spécifier les aspects informatique de l'application

=> Sorties : *document de spécification/d'analyse, plan de validation.*

4. Conception architecturale et conception détaillée

- **Conception architecturale**
  - Apporter une solution au problème défini lors de l'analyse
  - Apporter des solutions techniques : architecture, performance, optimisation
  - Définir la manière dont l'application doit être construite pour répondre au problème défini lors de l'analyse des besoins
  - Définir les composants, relations entres ces composants, relations avec son environnement (logiciels, BD, flux de message, éléments physiques, serveurs, intergiciels, etc.)

4. Conception architecturale et conception détaillée

- **Conception détaillée**
  - Détailler les composants(modules) : réalisation des fonctionnalités par les composants
  - Structuration des données
  - Organisation précise des IHM
  - Écriture des algorithmes

=> Sorties : *dossier de conception + plan de test global et par module*

5. Implémentation et test

- Coder dans le(s) langage(s) cible(s) l'ensemble des fonctionnalités
- Tester les différents composants isolément, puis tester progressivement leur assemblage, jusqu'au test de livraison complet
  - ***Test unitaire*** : chaque composant
  - ***Test d'intégration*** : tester progressivement leur assemblage
  - ***Test système*** : test en grandeur nature du système complet par l'équipe : *test alpha*
  - ***Test de validation*** : tester chez le client par les utilisateurs sélectionnés. *Test beta*

=> Sorties : *application + dossier d'implémentation + manuels d'utilisateurs + guide d'installation + etc.*

Post-développement

6. Déploiement

- Livraison sur les sites concernés
- Installation (manuelle ou automatisée) des composants
- Configuration
- Mise en pré-production, puis en production
- Formation des utilisateurs

7. Maintenance

Commence dès que l'application est déployée

- **Objectif** : gérer le flux des remontées d'information en provenance des utilisateurs. Il peut s'agir de :
  - Rapports d'anomalie (bug reports),
  - Demande de modification(change requests) ,
  - Demande d'extension (feature requests).
- Analyser, trier puis planifier et traiter les demandes quand elles le méritent.

Trois formes principales :

- ***Maintenance corrective*** : identifier et corriger les erreurs détectées ;
- ***Maintenance perfective*** : améliorer les performances ou ajouter des fonctionnalités;
- ***Maintenance adaptative*** : adaptation de l'application aux changements de son environnement (environnement d'exécution, formats de données, etc.).

Conception

- Solution conceptuelle au problème posé
- Comment satisfaire les besoins
- Processus créatif qui consiste à représenter les diverses fonctions du système
- Une « bonne » conception se définit en termes de la satisfaction des besoins et des spécifications
- Une bonne conception participe largement à la production d'un logiciel qui répond aux critères de qualité
- Une bonne conception se base sur la modularité : couplage, cohérence, compréhensibilité, adaptabilité, etc,

Conception : critères de qualité

- **Cohésion** : le caractère de ce qui forme un tout, dont les parties sont difficilement séparables. degré avec le quel
- **Couplage** : exprime le degré d'interconnexion des différents composants d'un système.
- **Compréhensibilité** : la compréhensibilité d'un module dépend de
  - Sa forte cohésion
  - Son nom : nom significatif
  - La documentation
  - La complexité
- **Adaptabilité** : dépend du couplage et de la documentation. Un logiciel adaptable doit avoir un haut degré de lisibilité

Approches de conception

Approches de Conception

On distingue principalement trois approches de conception :

- Approche systémique
- Approche fonctionnelle
- Approche orientée objet

Approche systémique

- Elle est influencée par les systèmes de gestion de bases de données.

- Propose une double démarche de modélisation :
  - la modélisation des données
  - La modélisation des traitements

- **Points forts:**
  - Approche globale prenant en compte la modélisation des données et des traitements
  - Bonne adaptation à la modélisation des données et à la conception des BD

- **Points faibles**
  - Double démarche de conception : données et traitements
  - Pas de fusion possible des deux aspect

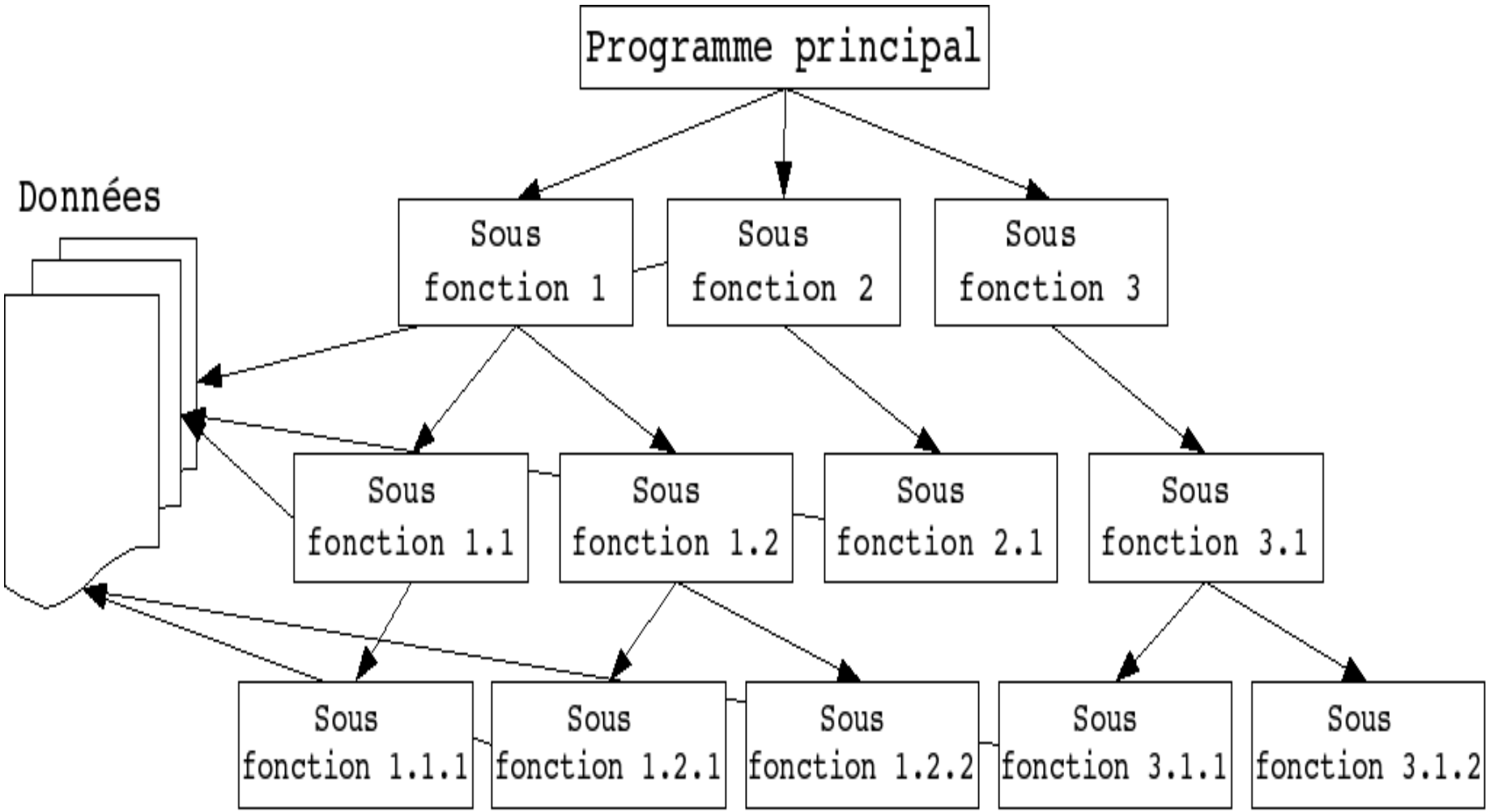
Approche fonctionnelle

- Consiste à décomposer hiérarchiquement une application en un ensemble de sous applications
- Approche hiérarchique descendante et modulaire
- Utilise les raffinements successifs pour produire des applications
- met en évidence les fonctions à assurer
- Dissocie le problème de la représentation des données de celui de leur traitement

- **Points forts:**
  - Simplicité du processus de conception
  - Capacité à répondre rapidement aux besoins ponctuels des utilisateurs

- **Points faibles**
  - Fixer les limites pour les décompositions hiérarchiques
  - Redondances(éventuelle) des données

Approche fonctionnelle

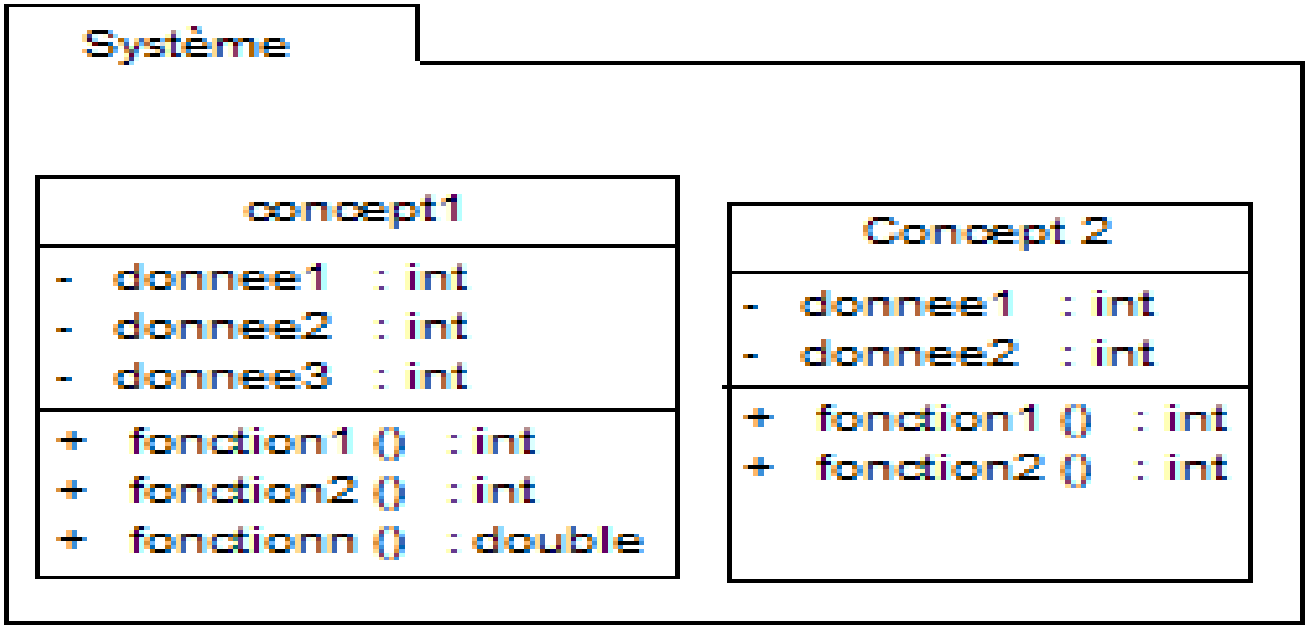


Approche orientée-objet

- Le logiciel est considéré comme une collection d'objets dissociés définis par un identifiant et des caractéristiques (attributs, fonction).
- Rapproche les données et leurs traitements associés au sein d'une unique entité : une structure de données et une collection d'opérations
- Définit des entités autonomes, avec ensemble d'attributs et traitements associés
- Contrairement aux approches fonctionnelle et systémique, l'approche OO est ascendantes

Approche orientée-objet

- **Avantages**
  - Offre un ensemble de concepts et principes pour faciliter la conception
  - Abstraction, modularité, encapsulation, polymorphisme, héritage
  - Permet de développer du code facile à maintenir
  - Favorise la réutilisation des composants



Approche fonctionnelle vs. objet

- **Approche fonctionnelle**
  - Dirigée par les traitements
  - Privilégie la fonction comme moyen d'organisation du logiciel
  - Architecture du système dictée par la réponse au problème (ie la fonction du système)
- **Approche orientée objet**
  - Dirigée par le type de données
  - Considère le logiciel comme une collection d'objets
  - Architecture du système dictée par la structure du problème
  - Conduit à des architectures logicielles fondées sur les objets du système, plutôt que sur les fonctions qu'il est censé réaliser.

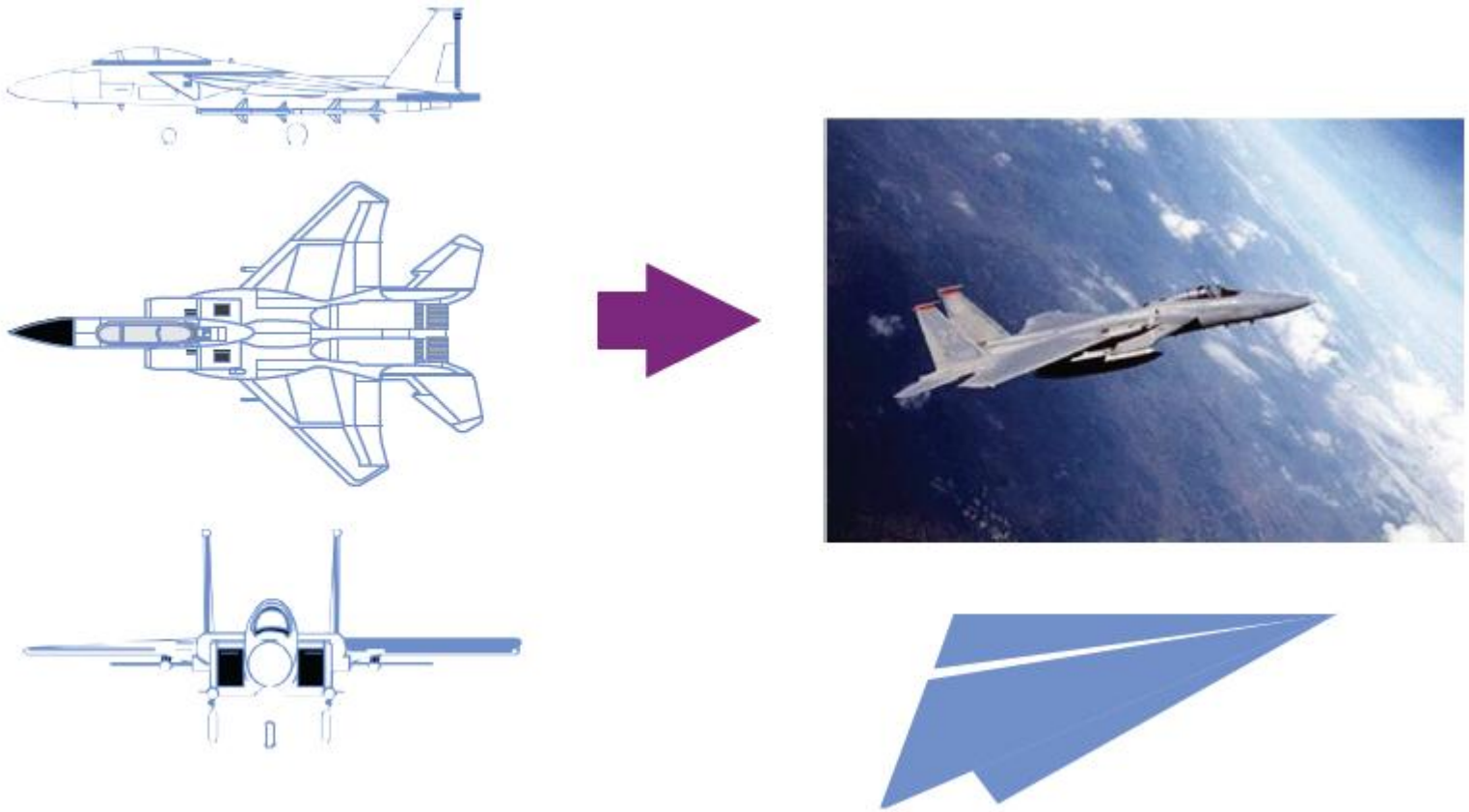
Concepts et principes de base de la modélisation OO

Objectifs des technologies à objets

- Utiliser le langage du domaine
  - *Modèle et vocabulaire métier*
- Construire des modèles faciles à étendre
  - *Etendre, modifier, valider, vérifier*
- Faciliter l'implantation
  - *Génération facilitée vers les langage à objets*

Modèle

- Une simplification de la réalité



Modèle

- **Modèle** : représentation abstraite et simplifiée d'une entité/objet (phénomène, processus, système) du monde réel.
- **Objectif de sa construction**
  - Aider à **visualiser** le système
  - **Spécifier** la structure et le comportement du système
- **Avantages d'un modèle**
  - **Abstrait** : il fait ressortir les points importants tout en enlevant des détails non nécessaires
  - **Compréhensible** : il permet d'exprimer une chose complexe dans une forme plus facile à comprendre par l'observateur
  - **Précis** : il représente fidèlement le système modélisé
  - **Prédictif** : il permet de faire des prévisions correctes sur le système modélisé
  - **Peu coûteux** : il est bien moins coûteux à construire et étudier que le système lui-même.

- **Pourquoi modéliser ?**
  - Pour décrire, expliquer ou prévoir ces entités
  - Réduire la complexité d'un phénomène en éliminant les détails.
  - Accent sur la recherche et la description des objets ou concepts du domaine étudié.
  - Permet de mieux comprendre le fonctionnement du système avant sa réalisation
  - langage commun, précis, qui est connu par tous les membres de l'équipe (vecteur de communication)
  - Mieux répartir les tâches et automatiser certaines d'entre elles

- **Résumé**
  - le modèle, un facteur de réduction des coûts et des détails
  - Le choix du modèle a une influence capitale sur les solution obtenues.

Quelques concepts et principes de l'OO

1. Objet
2. Classe
3. Encapsulation
4. Héritage
5. Généralisation/Spécialisation
6. Polymorphisme
7. Message
8. Agrégation
9. Composition

Objet

- Structure ayant une identité, des propriétés et des comportements.

- Entité autonome, regroupant un ensemble de propriétés cohérentes et de traitements associés

- Objet=identifiant + données + traitements agissant sur ces données
- Objet = identité + état + comportement

- Instance d'une classe

Objet : Exemple

Un compte bancaire

Identifiant de l'objet compte

- **Données :**
  - numéro compte : **325420145245**
  - Nom propriétaire : **Monthé**
  - Solde : **50F**

- **Traitement :**
  - Dépôt
  - Retrait
  - Afficher solde
  - Changer propriétaire

Classe

- Représentation abstraite d'une catégorie (=type) d'objets
- Collection d'objets partageant les mêmes caractéristiques(attributs et comportements).

- Type de données abstrait précisant des caractéristiques communes à toute une famille d'objets

- Permet de créer (instancier) des objets possédant les mêmes caractéristiques

- Est vue comme le *type* des objets qui seront créés à partir d'elle

Objet/Classe

Analogie avec le monde réel :

- Un chien
- Qui s'appelle Charlie
- Qui est de couleur blanche
- Qui a faim
- Qui va chercher la balle
- Qui aboie

Objet/Classe

Analogie avec le monde réel :

- Un chien → Objet
- Qui s'appelle Charlie
- Qui est de couleur blanche → Propriétés/état
- Qui a faim
- Qui va chercher la balle → Comportement/ méthode
- Qui aboie

Objet/Classe

Classe(d'objets) : représentation abstraite d'une catégorie (=type)d'objets.

Chien

nom : couleur : affamé :
aboyer : chercher (bale) :

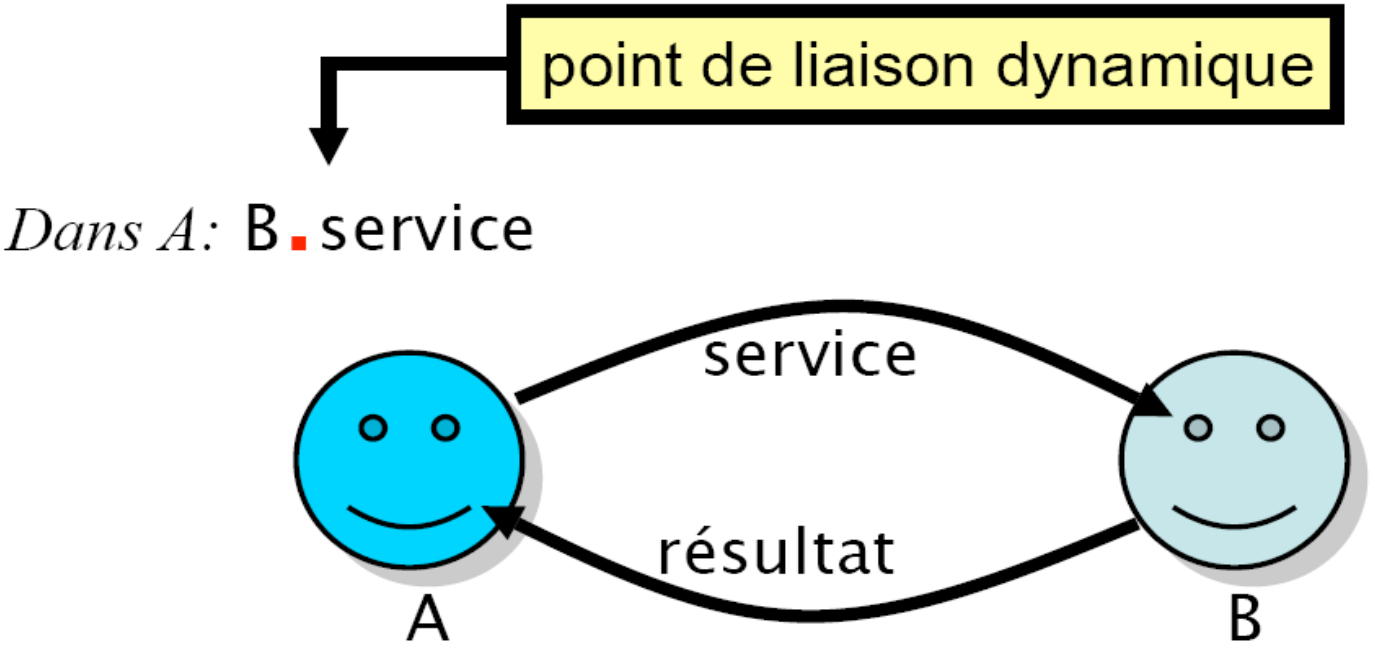
Objet/Classe

Objet : représentation en mémoire d'une entité physique (appartenant à une classe).

Chien : monChien

nom : Charlie couleur : Blanc affamé : oui
aboyer : chercher (bale) :

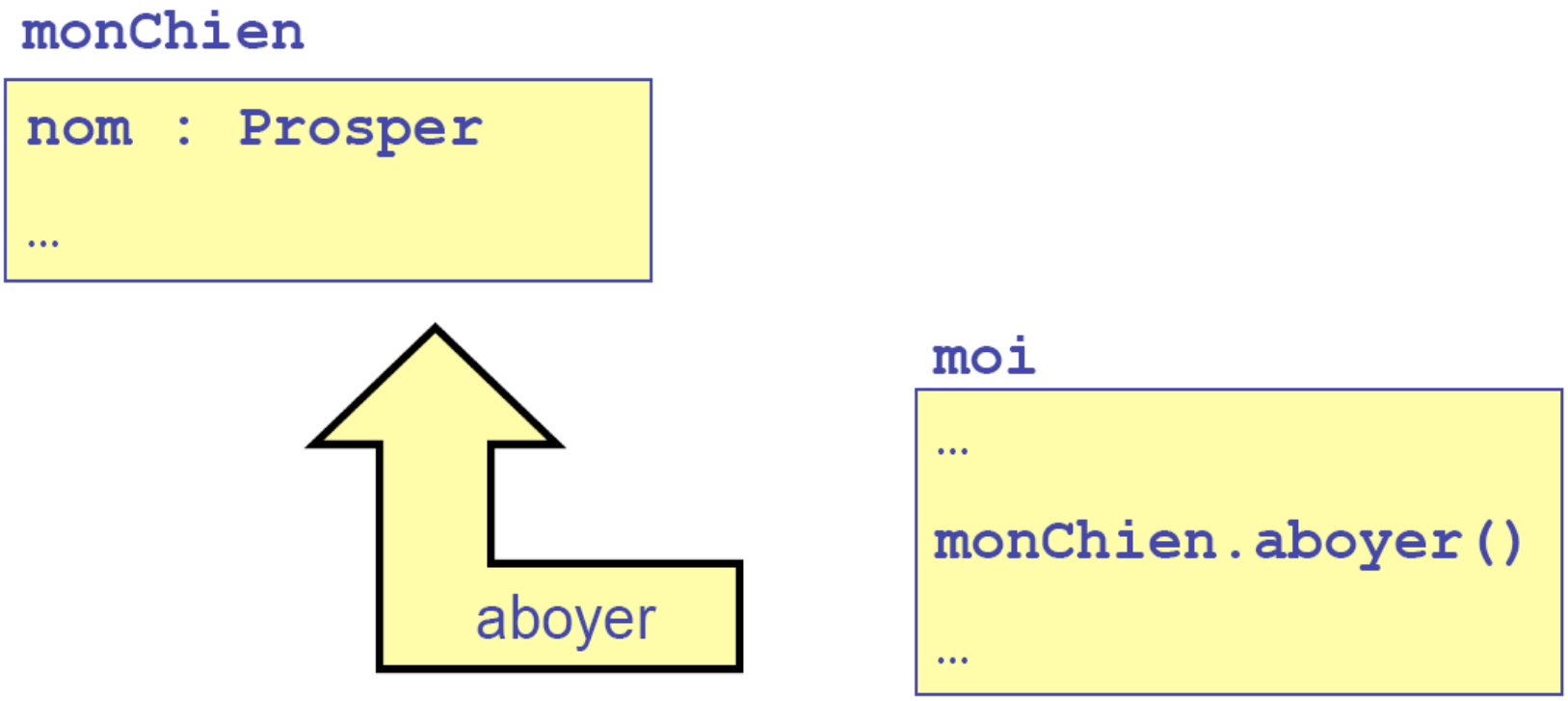
Communication inter - objets



Service: donnée ou une méthode.
Objet: données + méthodes/fonctions

Communication inter - objets

Communication : Invocation de message en un point de liaison dynamique



Encapsulation

- Masquer les détails d'implémentation d'un objet, en définissant une **interface.**

- **L'interface :** vue externe d'un objet, elle définit les services accessibles (offerts) aux utilisateurs de l'objet.

- L'objet **encapsule les variables** d'instance, non manipulable directement par l'utilisateur mais via les méthodes.

Encapsulation : Intérêt

- **Facilite l'évolution d'une application,** car stabilise l'utilisation des objets : on peut modifier l'implémentation des attributs d'un objet sans modifier son interface.

- L'encapsulation **garantit l'intégrité des données,** car permet d'interdire l'accès direct aux attributs des objets (utilisation d'accesseurs : getteur, setteur).

Notion d' Héritage

- Mécanisme de transmission des caractéristiques d'une classe (ses attributs et méthodes) vers une sous-classe.

- Possibilité de créer une nouvelle classe en enrichissant ou en raffinant une classe déjà existante.

Héritage : caractéristiques affectées

- L' enrichissement/raffinement des classes peut porter sur :
  - Les données : ajout de nouvelles variables d'instances
  - Les méthodes :
    - ajout de nouvelles méthodes
    - redéfinition de méthodes déjà présentes

Héritage : super-classe / sous-classe

- Super-classe (classe mère) : classe dont on souhaite hériter.

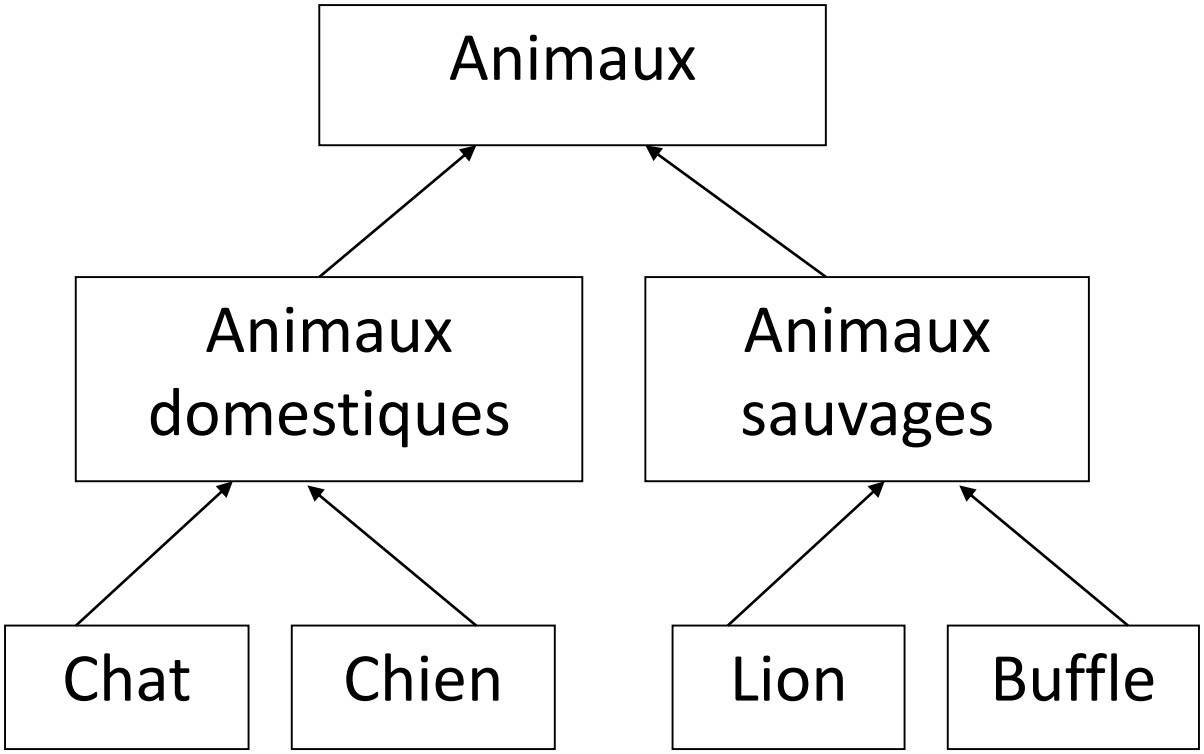
- Sous-classe (classe fille) : classe qui hérite.

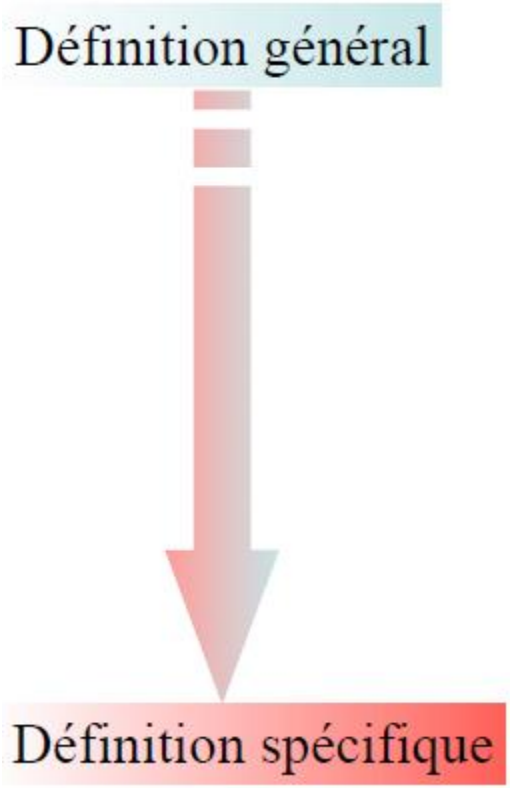
Héritage simple – héritage multiple

- **Simple** : une classe n'hérite que d'une seule autre classe

- **Multiple** : une classe hérite de plusieurs autres classes
  - **Java ne le permet pas**
  - **C++ le permet**

Héritage : Schématisation





Généralisation/Spécialisation

- Généralisation de plusieurs classes en une classe qui les factorise, afin de regrouper les **caractéristiques communes d'un ensemble de classes**.

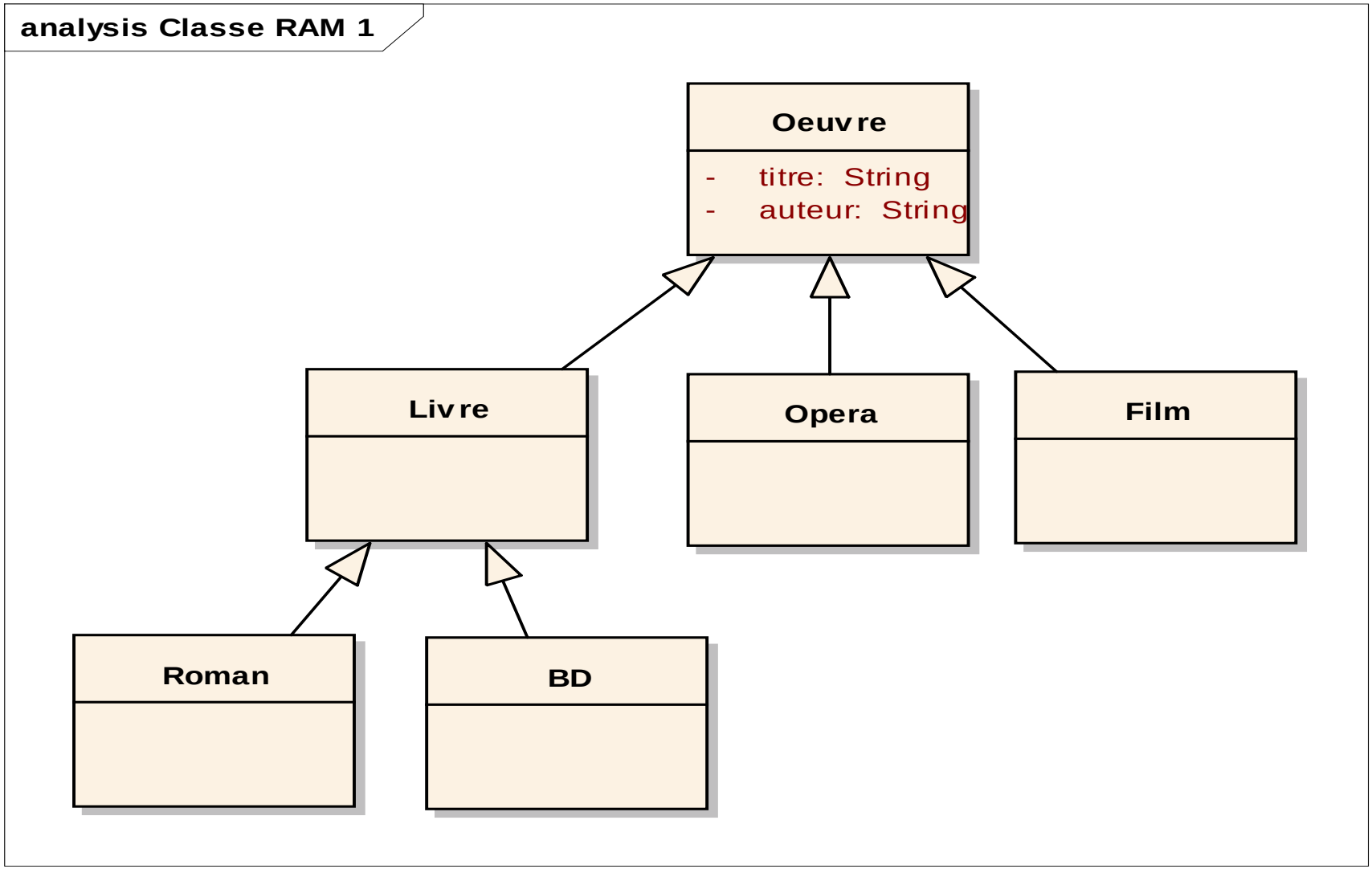
- Spécialisation d'une classe en d'autres classes, afin d'y **ajouter** des caractéristiques spécifiques ou d'en **adapter** certaines (redéfinition)

Généralisation/Spécialisation

- Permettent de construire des hiérarchies de classes.

- L'héritage évite la duplication et encourage la **réutilisation**.

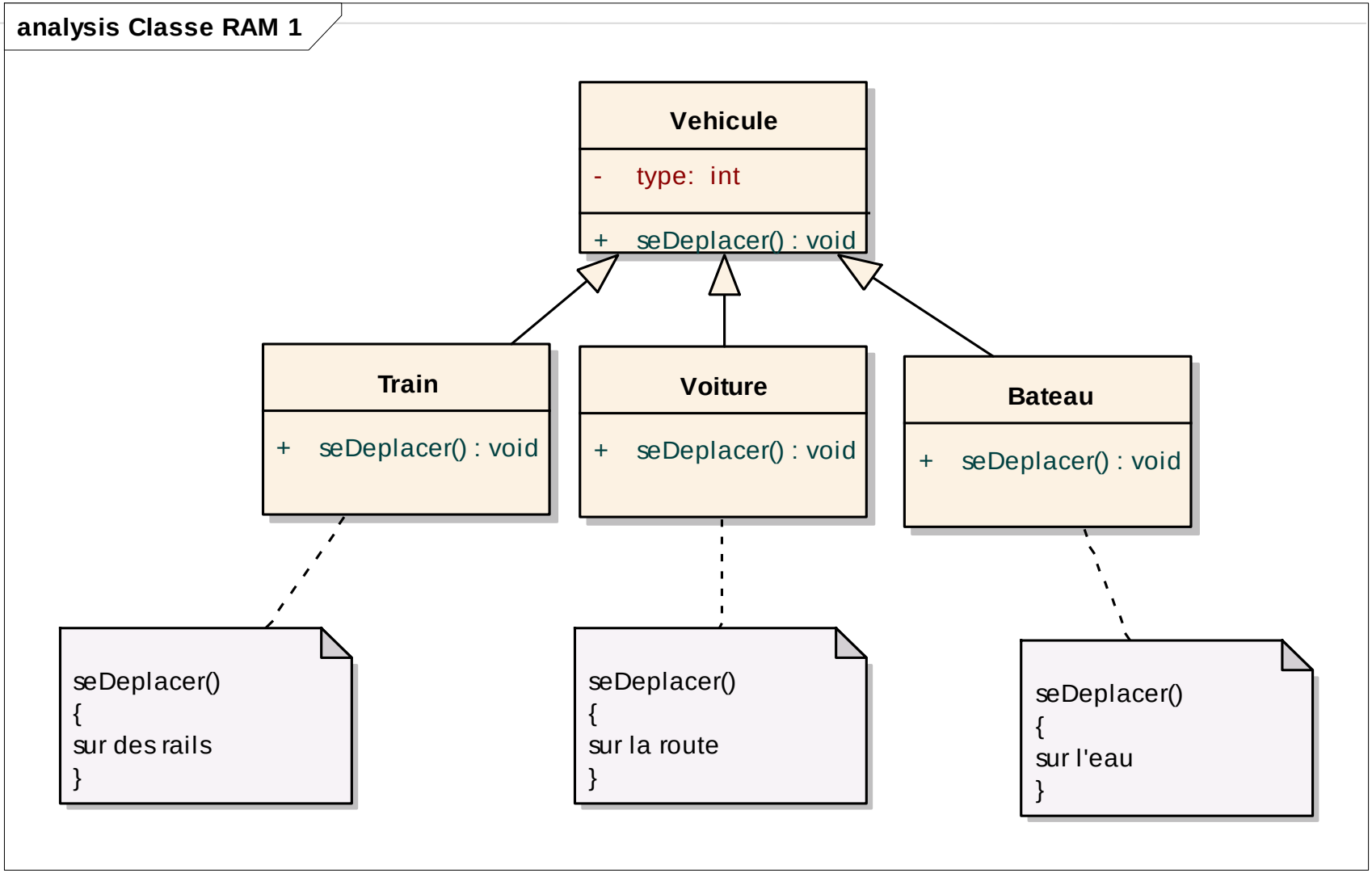
Héritage : Schématisation



Polymorphisme

- Faculté d'une méthode à pouvoir s'appliquer à des objets de classes différentes.
- Les objets de classes différentes répondent aux mêmes messages différemment.
- Autorise plusieurs opérations différentes a porter le même nom.
- on utilise chaque objet comme s'il était de la classe de base, mais son comportement effectif dépend de sa classe effective
- Augmente la généricité du code
  - Et donc sa qualité.

Polymorphisme



Agrégation

- Relation entre deux classes, spécifiant que les objets d'une classe sont des composants de l'autre classe.

- Une relation d'agrégation permet donc de définir des objets composés d'autres objets.

- Permet d'assembler des objets de base, afin de construire des objets plus complexes.

Classe abstraite

- Classe qui ne permet pas d'instancier des objets.
- Elle ne peut servir que de classe de base pour une dérivation.
- On peut y trouver classiquement des méthodes et des champs, dont héritera toute classe dérivée.
- On peut aussi y trouver des méthodes dites abstraites (i.e dont on ne fournit que la signature et le type de la valeur de retour).

```
abstract class A
{
  public void f() { ..... }   // f est definie dans A
  public abstract void g(int n) ; // g n'est pas definie dans A ; on ne fourni que l'en-tete
}
```

- On peut dériver de A une classe B qui défini g.
- NB : une classe qui définie au moins une méthode abstraite est elle-même abstraite.

Les interfaces

- Classe abstraite n'implantant aucune méthode et aucun champ (hormis des constantes).

- Une interface définit les en-têtes des méthodes, ainsi que des constantes.

- une classe pourra implémenter plusieurs interfaces (alors qu'une classe ne pouvait dériver que d'une seule classe abstraite)

- Une même classe peut implémenter plusieurs interfaces :

```
public interface I1
{ void f() ;
}
public interface I2
{ int h() ;
}
class A implements I1, I2
{ // A doit obligatoirement definir les methodes f et h prevues dans I1 et I2
}
```

TP 1 : Classe simple en JAVA

1. Créer une classe ***Point*** pour manipuler les points du plan. Un point a des coordonnées (abscisse, ordonnée). Cette classe doit disposer des trois méthodes suivantes :
   - ***initialiser***: *pour attribuer des valeurs aux coordonnées d'un point;*
   - ***deplacer***: *pour modifier les coordonnées d'un point;*
   - ***afficher***: *pour afficher un point ; par souci de simplicité, nous nous contenterons ici d'afficher* les coordonnées du point (cette méthode affichera : «je suis un point de coordonnées X et Y»).

TP 2 : l'Héritage en JAVA

1. Créer une classe ***Pointcol*** pour manipuler les points colores du plan. Elle doit avoir un attribut ***couleur*** de type byte et une opération ***colorer***

2. Modifier la classe Pointcol, en y ajoutant une méthode ***afficheCol***, qui en plus des coordonnées du point colore, affiche sa couleur.

3. Modifier la classe ***Point***, y ajouter un constructeur. Puis ajouter également un constructeur dans la classe ***Pointcol,*** qui permet de construire les objets de point colore, directement avec leur couleur.

TP 3 : Polymorphisme en JAVA

- Note : Java permet d'affecter à une variable objet non seulement la référence à un objet du type correspondant, mais aussi une référence à un objet d'un type dérivé.

```
Point p = new Point (3, 5) ;
p.affiche () ; // appelle la methode affiche de la classe Point
p = new Poincol (4, 8, 2) ;
p.affiche () ; // appelle la methode affiche de la classe Pointcol
```

TP 4 : Polymorphisme en JAVA

1. Modifier la classe ***Pointcol***, pour que sa méthode ***affichec*** s'appelle ***affiche***.

2. Ecrire un exemple de programme qui exploite les possibilités de polymorphisme pour créer un tableau "hétérogène" d'objets, c'est-à-dire dans lequel les éléments peuvent être de type différent (***Point***, ***Pointcol***). Parcourir le tableau et l'afficher.

TP 5 : Polymorphisme en JAVA

1. Modifier Les classes ***Point*** et ***Pointcol***, pour ne garder la méthode ***affiche*** que dans la classe ***Point***.

1. Définir une méthode ***identifie*** () qui affiche pour un :
   - Point : « **Je suis un point** »
   - Pointcol : « **Je suis un point colore de couleur couleur** »

TP 6 : Classe abstraite en Java

- Créer une classe abstraite ***Affichable,*** ayant juste la signature d'une méthode ***affiche***().

- Dériver deux classes, qui construisent respectivement les entiers et les flottants et les affichent:
  - Je suis un entier de valeur 25
  - Je suis un flottant de valeur 1.25

- NB : Utiliser un tableau d'objets de type différents

TP 7 : Les interfaces en Java

- Transformer la classe abstraite ***Affichable,*** en interface.

- Modifier les classes Entier et Flottant, pour obtenir le même résultat qu'au TP précédent.

TP 8 : l'Héritage en JAVA

1. Créer la classe compte suivante.

Compte
Numero : Chaine Proprietaire : Chaine Solde : Entier
Depot (montant : entier) Retrait (montant : entier)

2. Créer une classe compte d'epargne qui permet d'augmenter le solde suivant un certain taux.

3. On veut sécuriser la méthode retrait de manière à n'autoriser un retrait uniquement que si le solde est suffisant. Créer un compte sécurisé qui le fait.

Bibliographie et ressources

Ouvrages recommandés

- Pascal Roques, UML 2 par la pratique: Etudes de cas et exercices corrigés, 5ème édition, Eyrolles
- Sommerville Ian (2000), “Software Engineering (6th Edition)”. Addison-Wesley, Boston USA
- Larman C. (2004), “Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development”, (3rd Edition) 3rd Edition, Prentice Hall; 3 edition (October 30, 2004), ISBN-13: 978-0131489066
- Ariadne Training (2001), “UML Applied Object Oriented Analysis and Design Using the UML”, Ariadne Training Limited

Autres ressources utilisées

- Liu Z., (2001), “Object-Oriented Software Development Using UML”, The United Nations University, UNU-IIST International Institute for Software Technology, Tech Report 229
- Pélagie Houngue, “l’analyse et la conception orientée-objet”, Université Virtuelle Africaine
- Amosse Edouard, “Conception orientée objet”, MIAGE, Université de Nice