Chapitre IV

Approche orientée objet

Sommaire

1 Méthodologie **134**
1.1 Approche fonctionnelle 135
1.2 Approche objet 135
1.3 UML 136
1.4 Concept d'objet 138
2 Classe **138**
2.1 Instance de classe 138
2.2 Encapsulation 139
2.3 Persistance 140
2.4 Cohésion 140
2.5 Couplage 141
3 Relations entre classes **141**
3.1 Relations 143
3.2 Association 143
3.3 Agrégation 144
3.4 Composition 144
3.5 Navigabilité 145
3.6 Héritage 147
3.7 Héritage multiple et hiérarchie d'héritage 148
3.8 Interface 150
3.9 Abstraction 150
3.10 Polymorphisme 151
3.11 Délégation d'héritage 151
3.12 Exemple UML 153
4 Erreurs classiques **153**
4.1 Erreurs de conception 154
4.2 Erreurs architecturales 154

5 Comparaison des langages de POO **155**
5.1 Smalltalk 156
5.2 C++ 156
5.3 Java 157
5.4 C # 157
5.5 Eiffel 157
5.6 Ruby 158
6 Stratégie de codage en C++ **159**
6.1 Programmation par contrat 159
6.2 Outils du compilateur 160
7 Critique de l'approche objet **162**
7.1 Essence de l'argumentation 163
7.2 Echec de l'acceptation de l'échec 163
7.3 Le mythe de la réutilisation de code 164
7.4 Échec de l'encapsulation 165
7.5 Problème de conception 166
7.6 Problème de l'apprentissage 166

Introduction

Nous effectuons dans cette partie une introduction à la conception de systèmes à partir de concept objets.

A savoir :

- la notion d'objet
- l'approche orientée objet
- la notion de classe
- les différentes relations entre les classes,
- l'utilisation d'UML comme outil de représentation,
- la comparaison de différents langages orientés objets,
- les erreurs de conception classique,
- une critique de l'approche objet.

1 Méthodologie

Trois phases dans un projet informatique :

- **Analyse :** qu'est-ce-qui doit être fait ?
  besoins, problèmes posés.
- **Conception :** trouver comment concevoir un système qui répond aux objectifs fixés par l'analyse.
- **Réalisation :** réalisation pratique du système (implémentation).

Deux grands types méthodes :

- L'approche fonctionnelle
- L'approche "orientée objet"

1.1 Approche fonctionnelle

Concepts utilisés pour une approche fonctionnelle :

- L'analyse et la conception sont guidées par les **fonctions** que le système doit posséder.
  Exemple de fonctions : matérielles, programmes, services, ...
- Nécessite d'identifier les fonctions du système, puis de découper chacune en sous-fonctions, ...
- **Types de fonction :**
  - ◇ **fonction principale** (= celle qui satisfait le besoin),
  - ◇ **fonction contrainte** (= participent à définir le besoin sous les conditions définies par le produit),
  - ◇ **fonction complémentaire** (= facilite, améliorer ou complète le service rendu).

Inconvénients :

- La modification des données a possiblement un impact sur tous les blocs fonctionnels qui l'utilisent,
- souvent difficile à faire évoluer,
- maintenance parfois problématique (ne doit pas faire dériver la fonction),
- en terme informatique cette approche à pour conséquence :
  - ◇ séparation des données et des traitements,
  - ◇ absence de niveau d'abstraction sur les données.

Bon résultat pour des problèmes où les fonctions sont clairement identifiées et stables dans le temps (cahier des charges pas ou peu évolutif).

1.2 Approche objet

Idée de l'approche objet : décrire le système comme étant composé d'un ensemble d'objets communiquant entre eux.

Objet : un objet est une entité contenant :

- des données propres (propriétés), intrinsèque ou dépendant d'éléments extérieurs.
- des méthodes manipulant ces données (comportements), intrinsèque ou dépendant d'éléments extérieurs.
- des échanges de messages avec d'autres objets.

L'architecture est ainsi constituée :

- d'un ensemble d'objets clairement identifiés,
- des relations et des communications possibles entre ces objets,

Le comportement général du système (= aspect dynamique) obtenu à travers l'envoi de messages entre les objets, engendrant les traitements permettant de réaliser la fonction du système.

1.3 UML

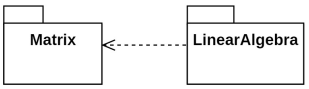
UML (acronyme de Unified Modeling Language) est un langage de modélisation graphique utilisé notamment lors de la conception de programme.

UML est une norme de représentation (et non une méthode) : il permet de représenter des conceptions sous une forme standardisée.

Il existe trois catégories de diagrammes UML :

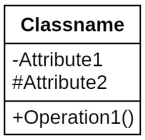
- **Les diagrammes de structure** : modélise l'architecture du système.
  diagrammes de paquetages, de classes, d'objets, de composants, ...
- **Les diagrammes de comportement** : modélisent le comportement du système.
  diagrammes d'états/transition, des cas d'utilisations, d'activité.
- **Les diagrammes d'inter-actions** (dynamiques) : modélisent les interactions entre les différents actions/composants du système.
  diagramme de séquence, de communication, de temps.

- **paquetage :**



  un paquetage UML (=bibliothèque) est un espace de nommage en C++.

- **classe :**



  Le classe est réprésentée par une boite rectangulaire à plusieurs compartiments :
  - ◇ le premier est le nom de la classe,
  - ◇ le second contient les attributs,
  - ◇ le troisième contient les méthodes.

Modificateurs d'attributs/méthodes :

- **Accès aux classes :** + = public, #=protected, -=private, italique=virtuel.
- :type (pour spécifier le type),
- =val (pour donner la valeur par défaut),
- <<`constant`>> = attribut constant,
- <<`property`>> = attribut qui dispose d'une setter et d'un getter,
- `{query}` = la méthode est constante.
- {ordered} indique que le stockage d'un attribut est ordonné,
- les contraintes peuvent également être indiquées,
- les méthodes statiques sont soulignées.
- <<`create`>>/<<`destroy`>> peuvent être utilisés pour marquer les constructeurs/destructeur.

Exemple :

ExampleClass
-<<constant>>pi=3.14159 -<<property>>age #empty:bool #counter=0 -listofvalues:list{ordered} -teeth:int < 99
+getValue():int {query}

La modélisation des relations entre classes sera présentée ultérieurement.

1.4 Concept d'objet

DÉFINITION 1 : objet
Un objet est une unité atomique possédant :
- une identité (peut être nommé ou identifié),
- une état (défini par ses attributs ou ses données membres),
- un comportement (défini par l'ensemble de ses méthodes)

Un objet peut être utilisé pour représenter :
- un objet concret avec une réalité physique (une personne, un voiture, un moteur, un outil, ...),
- un concept abstrait (un compte bancaire, une accélération, une contrainte sur un système, ...),
- une activité produisant des effets observables (un calcul, un affichage, ...).

En particulier, pour les méthodologies de l'approche objet, on citera :
- UML comme outil d'analyse et de conception.
- RUP (Rational Unified Process), XP (Extreme Programming) comme groupes de pratiques de projets de développement,
- les design patterns, programmation par contrat comme technique de conception.

2 Classe

DÉFINITION 2 : classe
Une classe est un modèle représentant une famille d'objets ayant :
- les mêmes attributs
- les mêmes méthodes,
- les mêmes relations

Une classe ne contient pas la valeur des attributs : elle définit seulement le rôle qu'ils jouent dans la classe et comment ils sont représentés et stockés (i.e. leurs types).

La notion de **classe** fournit le mécanisme de base avec lequel les attributs et les méthodes communs sont groupés ensemble.
- défini les attributs (champs, propriétés, méthodes) et le comportement
- une méthode contient le code pour l'exécution

La notion d'**interface** définit un ensemble d'opération (méthodes et propriétés vides) dont l'implémentation est laissée pour plus tard.

2.1 Instance de classe

Avec les définitions précédentes :

DÉFINITION 3 : instanciation
La création d'un objet comme exemplaire concret d'une classe s'appelle une instanciation.

L'instanciation a pour but de définir une valeur concrète pour chacun des attributs de la classe.

Exemple :
- la classe `Genius` permet de définir les attributs des génies scientifiques.
- l'objet `LeonardDeVinci` est un exemple concret de `Genius`.

- l'objet `AlanTuring` est un autre exemple concret de `Genius`,

En C++, une nom de classe est un nouveau type du langage, cela permet de :
- définir facilement un objet (= une instance de la classe) qui utilise ce modèle,
- redéfinir les opérateurs classiques (+,*,...) si ces opérations ont un sens entre deux objets.

2.2 Encapsulation

Encapsulation :
- Cache les détails de l'implémentation interne (éléments,méthodes) d'une classe
- Les méthodes accessibles constituent l'interface publique (= aucun membre de l'interface caché),
  Les autres méthodes sont privées sauf constructeurs et destructeur qui restent publics
- Tous les champs de la classe sont cachés (privé), et éventuellement accessible à travers des propriétés (=méthodes).

Intérêt de l'encapsulation :
- assure que les changements structurels restent locaux :
  - ◇ le changement de la structure interne de la classe n'affecte aucun des codes qui l'utilise,
  - ◇ même chose pour le changement de l'implémentation des méthodes
- permet d'assurer la cohérence des données (en s'assurant que les données internes ne sont jamais modifiés par d'autres moyens que l'interface).
- occulter les détails de l'implémentation réduit la complexité, et donc facilite la maintenance.

L'encapsulation permet à l'objet sous deux vues différentes :

Définition 4 : vue externe d'un objet
la vue externe de l'objet est la vue du l'utilisateur de l'objet.
- elle définit l'interface de l'objet (=comment on utilise l'objet),
- elle fournit la liste des services accessibles aux utilisateurs de l'objet,
- aucun accès n'est fourni aux mécanismes internes de l'objet (données et méthodes).

Note : en C++, les parties `public` d'un objet.

Définition 5 : vue interne d'un objet
la vue interne d'un objet est la vue du concepteur de l'objet.

Elle définit l'ensemble des détails de constitution interne de l'objet, à savoir :
- comment il est structuré,
- comment son état est défini à partir de ses attributs,
- quelle implémentation est utilisée pour ses comportements.

Note : en C++, les parties `private` ou `protected` d'un objet.

Conséquences :
- l'encapsulation permet de dissimuler à l'utilisateur l'implémentation interne,
  favorise l'évolution du modèle : les détails internes peuvent évoluer au cours du temps, toute en conservant ou ajoutant les fonctionnalités externes de l'objet.
- l'interface externe est donc la seule partie de l'objet qui doit être pérenne dans le temps.
- l'encapsulation favorise l'intégrité des données :
  - ◇ interdiction d'accès depuis l'extérieur aux détails internes,

- ◇ contrôle et garantie la cohérence des données (i.e. les méthodes externes doivent assurer que cette cohérence est maintenue).

L'implémentation d'un objet est ainsi découplée de son comportement.

2.3 Persistance

DÉFINITION 6 : persistance
La persistance est la possibilité de sauvegarder les attributs d'un objet sur un support externe afin de pouvoir les restaurer ultérieurement.

Conséquences :

permet, après l'arrêt d'une machine, de retrouver les objets dans l'état dans lesquels ils étaient au moment de la sauvegarde.

Essentiel pour l'approche objet :

- la fonction du système est engendrée par l'architecture (=relations entre les classes) **et** l'instanciation des classes (les objets instanciés).
- **conserver la fonction du système** = préserver l'état et les relations entre objets.

2.4 Cohésion

La cohésion mesure à quel point les membres et le code de ces membres agisse dans le sens de l'objectif central attribué à la classe.

- La cohésion d'une classe doit être forte (en général, une abstraction bien définie permet de conserver un cohésion forte, car elle permet de se concentrer sur le but essentiel assigné à la classe).
- Une classe doit contenir des fonctionnalités fortement liées, et n'avoir pour but qu'un seul objectif.
- La cohésion est utile pour gérer la complexité (au sens où elle aide à décomposer un problème sous une forme sous laquelle chacun des composants a un rôle limité et clairement défini).

En revanche, si la cohésion d'une classe est faible :

- elle ressemblera à un fourre-tout,
- son rôle ne sera pas clairement défini.

Si l'architecture a une forte cohésion :

- **la clarté :** la classe joue un rôle bien spécifique,
  $\Rightarrow$ plus lisible, moins de ligne de code, code clair et facilement compréhensible.
- **la maintenabilité :** si une modification doit être faite,
  - ◇ chaque classe ayant un rôle bien spécifique, on sait exactement quelles parties du code (=classes) modifier,
  - ◇ lors d'un travail collaboratif, les tâches de modification affectées à des parties différentes du code ont toutes les chances de ne pas entre en conflits lors des merges.
- **l'extensibilité :** un traitement ajouté à une classe peut être déléguée à une nouvelle classe spécialisée.
- **la testabilité :** l'écriture des tests unitaires est simplifiée (plus l'objectif d'une classe est simple, plus il est facile de tester si ses méthodes réalisent leurs objectifs).

- **la réutilisabilité :** chaque classe ayant un rôle unique, elle peut être utilisée dans un autre contexte.

2.5 Couplage

Le couplage est une mesure de la dépendance entre les classes ou les modules.

Le couplage décrit à quel point une classe ou une fonction est liée à une autre classe ou une autre routine.

Règle : Le couplage doit être faible.

- Les modules doivent **peu** dépendre les uns des autres,
- Toutes les classes ou les fonctions doivent avoir des relations **petites**, **directes** et **visibles** envers les autres classes ou fonctions.
- Un module doit pouvoir être **facilement** utilisé par un autre module.

Conséquence :

un couplage faible signifie que le code est plus facilement maintenable, évolutif et réutilisable.

La notion de couplage est donc essentielle pour comprendre les relations entre classe.

3 Relations entre classes

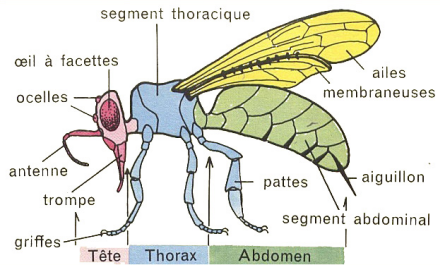
L'analyse de systèmes réels ou conceptuels s'effectue souvent suivant :

soit par **une approche structurelle :**

- par composition d'éléments simples pour construire un objet plus complexe,
- par décomposition d'un élément complexe en éléments plus simples.

Exemple :

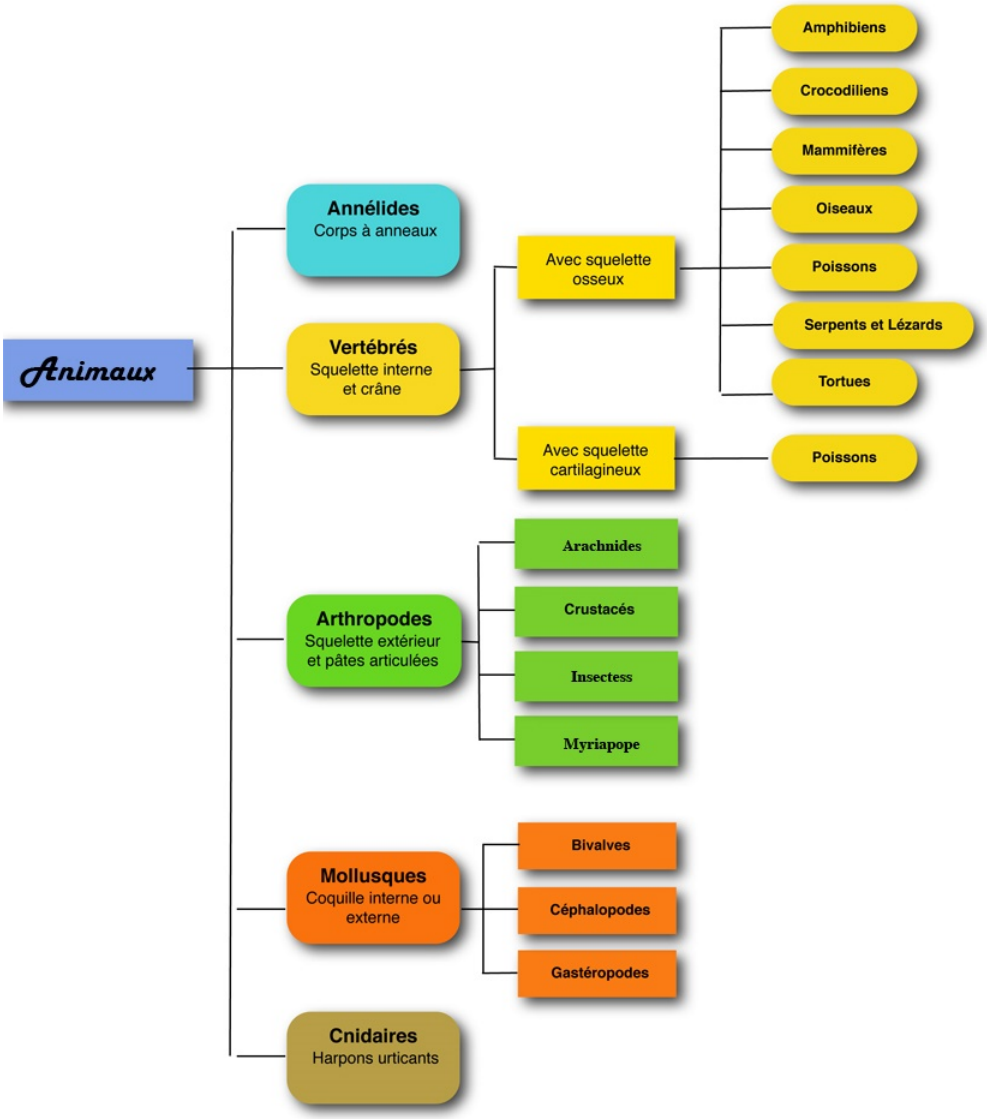
un insecte est composé d'une tête, d'un thorax, d'un abdomen, d'antennes et de pattes.



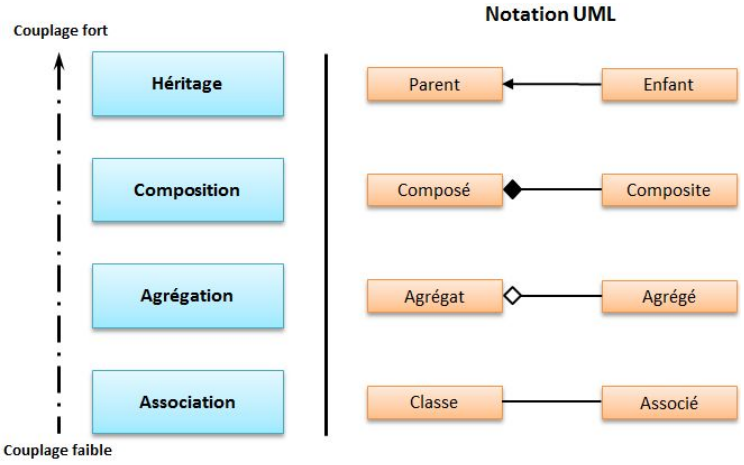
soit par **une approche hiérarchique** (par classification)

Exemple :

les animaux sont classifiés dans le règle animal sous forme d'une hiérarchie d'espèces regroupées par caractéristiques biologiques communes.



Les principales relations entre classes sont :



- chaque type de relation induit un niveau de couplage,
- à droite, la façon dont cette relation est représentée en conception UML.

L'approche objet permet plusieurs types de relations entre classes :

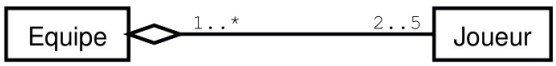
- l'**héritage** (traduit une classification hiérarchique),
- l'**agrégation** ou la **composition** : le contenant contient des agrégats ou des composants (décomposition structurelle possible),
- l'**association simple** : permet à deux classes de se connaître et d'échanger des messages,
- l'**utilisation** ou la **dépendance** : traduit le fait qu'une classe se sert d'une autre.

Les relations entre les classes modélisent les interactions et les liens qui seront possibles entre les objets.

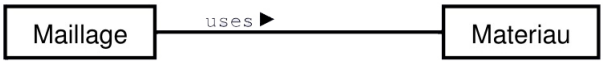
3.1 Relations

Une relation est caractérisée par :

- son **type** (association simple, agrégation, composition)
- sa **cardinalité** (nombre d'instances en jeu dans la relation),
- son **nom** (identifiant de l'instance dans la relation)

Cardinalité : 

- Dans l'exemple ci-dessus, une équipe peut avoir de 2 à 5 joueurs, et chaque joueur participer à une équipe au moins,
- Peut être superflue.

Nom : 

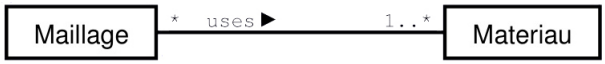
- Le nom identifie le rôle de l'instance, vue de l'autre instance.
- Peut être superflu.

3.2 Association

DÉFINITION 7 : association simple

L'association simple est un lien qui indique qu'une instance de classe peut envoyer un message à une instance d'une autre classe.

Notation UML :



L'association simple est le lien le plus simple entre deux classes.

Exemple d'association simple en C++ :

l'instance d'un objet utilise l'instance d'un autre objet à travers l'appel d'une fonction ou d'une méthode.

3.3 Agrégation

DÉFINITION 8 : agrégation

L'agrégation est un lien indiquant une relation de contenance entre un conteneur et un contenu.

Notation UML :

Le losange vide est du côté du conteneur.

L'agrégation : traduit la relation "possède", "est constitué de" ou "est fait de".

Exemple :

- Un cours est constitué d'un professeur, d'une salle, d'une matière, et d'un groupe d'étudiants.
- Un groupe d'étudiants est constitué d'une liste d'étudiants, d'une formation, et d'un niveau de formation.

Noter que chaque élément qui compose cette agrégation a une existence propre à l'extérieur du conteneur .

En C++, l'agrégation de l'instance d'un objet avec :

- l'instance d'un autre objet s'exprime à travers un pointeur vers l'instance (cardinalité 1),
- un tableau d'instances d'autres objets s'exprime à travers une collection (tableau dynamique, pile, map, ...) de pointeurs vers ces instances.

Notes : la cardinalité 0 est ajoutée sur la relation si le pointeur peut être nul.

Durée de vie :

- les instances pointées n'ont *a priori* pas besoin d'être libérée dans le destructeur, ni d'être copiée lors d'une construction ou assignations par copie, car ces instances ont leurs existences propres.
- néanmoins, la conception de l'application doit permettre de faire en sorte que ces instances sont effectivement libérées.
- la durée de vie de l'instance pointée n'a donc *a priori* pas de lien avec la durée de vie de l'instance possédant le pointeur.
- si cela n'est pas le cas, c'est que le lien est plus sûrement une **composition** et non une agrégation.

3.4 Composition

DÉFINITION 9 : composition

La composition est un lien indiquant :

- une relation de contenance entre un conteneur et un contenu,
- un lien de durée de vie (le contenu ne peut exister sans le contenu).

Notation UML :

Le losange plein est du côté du conteneur.

L'agrégation traduit aussi la relation "possède", "est constitué de" ou "est fait de", mais avec l'idée d'une possession exclusive.

Exemple :
- Une voiture est composée de 4 roues, d'un moteur, ...
- Une étudiant est constitué d'une identité propre, d'un numéro d'inscription, ...

Noter que chaque instance agrégée est propre à l'instance du conteneur : chaque voiture a son propre moteur (i.e. le moteur n'est pas partagé entre plusieurs voiture), le numéro d'inscription d'un étudiant est unique, ...

En C++, la composition de l'instance d'un objet avec l'instance d'un autre objet s'effectue avec :
- l'instance contenue est une valeur dans l'instance du conteneur (*i.e.* l'instance contenue est un champ du conteneur).
  Exemple : `struct B { A a; };`
- idem agrégation (i.e. la relation s'exprime à travers un pointeur ou un conteneur à pointeur).

Durée de vie : la durée de vie du contenu est égale à la durée de vie du conteneur.

Conséquences : (**obligatoire**)
- le destructeur du conteneur doit détruire tout contenu composé pointé.
- lors d'un constructeur ou d'une assignation par copie, tout contenu composé doit être copié afin que la copie (elle-aussi composée) dispose de ses instances propres.

Remarques :
- Relation transitive (si un chat a des pattes, et que ces pattes ont des griffes, alors le chat a des griffes), réflexive, non symétrique.
- L'utilisation des champs est la manière la plus évidente (et facile) de créer un relation de composition.
  Permet une gestion automatique de la destruction et des copies si elles sont écrites correctement (cf deep copy).

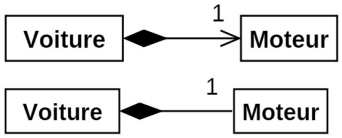
Néanmoins, l'utilisation d'un pointeur est **obligatoire** si :
- le contenu est polymorphe (i.e. la taille de l'objet à composer est potentiellement variable),
  Rappel : il est dangereux de faire du polymorphisme autrement (cf object slicing).
- le contenu est créé après la création du contenant
  (sauf à utiliser la destruction explicite/construction placée, potentiellement dangereux dans les types dérivés).

3.5 Navigabilité

Définition 10 : navigabilité
La navigabilité d'un lien est la possibilité d'utiliser ce lien dans un sens et/ou dans l'autre.

En notation UML, l'absence de flèche signifie une absence de navigabilité.

Exemple :



lien navigable

lien non navigable

La moteur peut envoyer un message à destination du moteur (par exemple démarrer).

Si le lien n'est pas navigable, une instance du moteur n'a aucune connaissance du véhicule sur lequel il est monté.

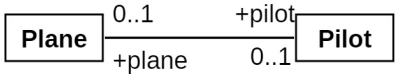
Remarques :

- Une navigabilité à sens unique est plus simple à implémenter,
- La navigabilité à double sens pose des problèmes potentiels : connaissance mutuelle des classes, messages en cas de mise à jour de l'une des deux (risques de référence circulaire, d'incohérence de lien, ...).

Précision de l'association à double sens :

- une seule des deux extrémités peut être une composition ou une association vis-à-vis de l'autre,
- la mise à jour doit se faire dans les deux sens.

Exemple :




```
class Plane { Pilot *pilot; public: void setPilot(Pilot *p); };
class Pilot { Plane *plane; public: void setPlane(Plane *p); };
```

Évidemment, la mise à jour ne saurait être :

```
void Plane::setPilot(Pilot *p) { pilot=p; }
void Pilot::setPlane(Plane *p) { plane=p; }
```

Cette solution peut lier un pilote à un avion, alors que ce pilote est déjà affecté à un avion (et vice-versa). Un code qui semble plus adapté serait le suivant (`setPilot` symétrique) :

```
void Pilot::setPlane(Plane *p) {
    if (plane) plane->setPilot(nullptr);
    plane = p;
    if (plane) plane->setPilot(this);
}
```

Malheureusement, ce code génère un appel infini (`setPlane` fait appel à `setPilot`, et `setPilot` à `setPlane`).

Deux solutions sont possibles à ce problème :

- **choisir un sens privilégié et interdire l'autre :**
  définir un accesseur privé avec l'autre classe déclarée amie.

```
class Pilot {
private: Plane *plane;
public:
   void setPlane(Plane *p);
};
```

```
class Plane {
private: Pilot *pilot;
   void setPilot(Pilot *p);
public: friend class Pilot;
};
```

```
void Pilot::setPlane(Plane *p); // idem précédent
void Plane::setPilot(Pilot *p) { pilot = p; }  // privé
```

Ainsi, un pilote peut être affecté à un avion, mais pas un avion à un pilote.

- **ajouter un test empêchant les aller-retours entre les deux classes :**

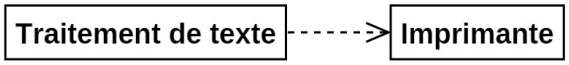
```
void Pilot::setPlane(Plane *p) {
    // brise le lien pilot-plane courant
    if (plane && p) plane->setPilot(nullptr);
    plane = p;  // nouveau pilot-plane
    // fixe en retour le lien plane-pilot
    if (plane && p) plane->setPilot(this);
}
```

Inconvénient, l'accesseur est plus lent.

Définition 11 : auto-association
Un lien est auto-associatif lorsqu'une classe est associée à elle-même.

Remarque : pas de spécificité par rapport à une association normale, excepté que cette association est nécessairement par pointeur.

Définition 12 : dépendance
Une classe est dépendante d'une autre si l'une de ses instance a **temporairement** besoin de connaitre l'autre classe pour effectuer l'une de ses opération.

Notation UML : 

origine=dépendant de

Exemples :
- si l'une des méthodes appelées par une instance utilise une variable locale de l'autre classe (= rôle temporaire).
- induit que le fichier d'en-tête de la classe dépendante doive être inclus dans la définition de l'autre classe.

3.6 Héritage

Définition 13 : Héritage
L'héritage d'une classe (dite mère) par une autre classe (dite fille) est une relation qui associe la classe mère à la classe fille en permettant à cette dernière de récupérer les attributs et les méthodes de la classe mère.

L'héritage traduit la **relation "est un"**.

Exemples : une voiture est un véhicule, un chat est un félin, un roman est un livre, ...

L'héritage est le mécanisme fondamental pour faire évoluer un modèle :
- si une classe fille est une spécialisation de sa classe mère, l'héritage permet à la classe fille de réutiliser directement des fonctionnalités de sa classe mère,
- on évite ainsi la duplication de code sur la classe fille si elle n'est pas nécessaire.

L'héritage est LE mécanisme par lequel on transmet les propriétés (attributs et méthodes) d'une classe à une autre :

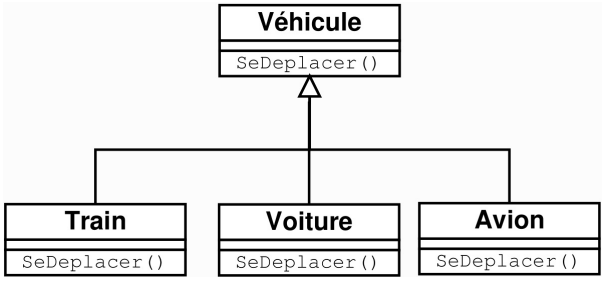
La classe de base (=classe mère, parent) transmet toutes ses propriétés transmissibles (protected ou public) aux classes dérivées (=classe fille, enfant).

Conséquence : la classe de base peut donc posséder des attributs et des méthodes que ses classes filles ne connaissent pas.

Une classe dérivée :

- ne peut pas accéder aux membres privés de la classe de base,
- possède ses propres attributs et méthodes que la classe de base ne connaît pas,
- peut redéfinir (améliorer, spécialiser, ...) les méthodes transmises (=définies par défaut) par la classe de base/

Exemple de spécialisation :



Une classe dérivée sait faire tout ce que sait faire la classe de base, (puisqu'une classe dérivée hérite des méthodes publiques de la classe de base).

Elle le fait souvent mieux ou différemment (en pouvant adapter les méthodes).

Propriétés de l'héritage :

- l'héritage peut être :
  - ◇ **simple** : si une classe dérivée hérite (directement) d'une seule classe mère,
  - ◇ **multiple** : si une classe dérivée hérite (directement) de plusieurs classes de bases.
- arborescence d'héritage : lorsque qu'une classe dérivée est à son tour utilisée comme classe base pour une autre dérivation.
  on forme ainsi des hiérarchies de classes.
- un héritage ne peut pas être réflexif (impossibilité pour une classe d'hériter d'elle-même).

Le principe de substitution de Liskov

Si $q(x)$ est une propriété démontrable pour tout objet `x` de type `T`, alors $q(y)$ est vraie pour tout objet `y` de type `S` tel que `S` est un sous-type de `T`.

Autrement dit :

- Les sous-classes doivent pouvoir remplacer leur classe de base,
- Les méthodes qui utilisent des objets d'une classe doivent pouvoir utiliser des objets dérivés de cette classe sans s'en rendre compte.

3.7 Héritage multiple et hiérarchie d'héritage

Plus formellement, on peut voir l'héritage de deux façons :

L'héritage par spécialisation (descendant)

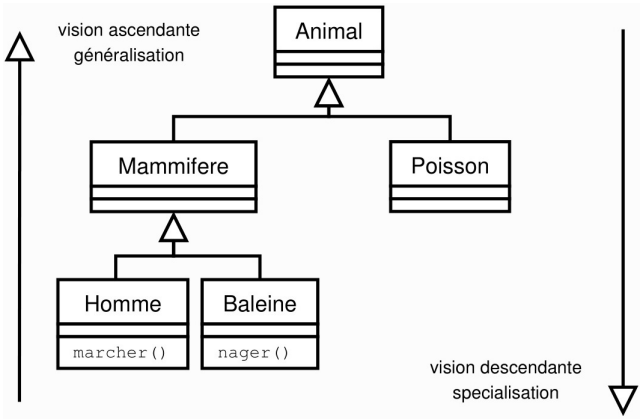
- La classe de base est dérivée pour produire une (ou plusieurs) classe dérivée spécialisée (spécialisation fonctionnelle ou avec des attributs particuliers/supplémentaires).
- Démarche utilisée dans la phase initiale de conception pour structurer les classes
  - ◇ pour la réutilisation de tout ou d'une partie des attributs/méthodes d'une classe de base,

◇ pour ne rajouter dans la classe dérivées que les attributs/méthodes qui spécialise la classe de base.

L'héritage par généralisation (ascendant)

- Factorisation des attributs/méthodes commun dans une classe de base.
- Démarche pour imposer un modèle commun au sein d'une hiérarchie.
- Démarche utilisée en fin de phase de conception lorsque l'on s'aperçoit que certaines classes possèdes des attributs/comportements communs factorisable dans une classe de base (soit dans une branche de la hiérarchie, soit à travers une hiérarchie).

Exemple : approche par spécialisation/généralisation



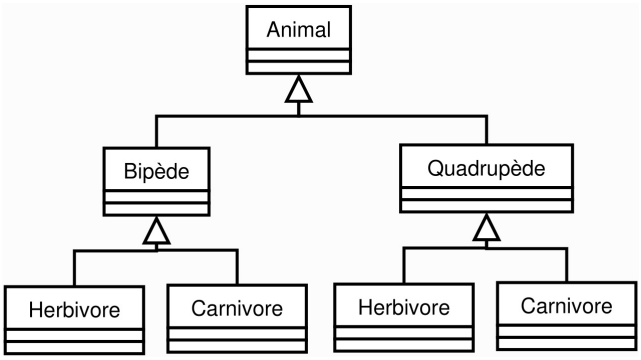
La classification par hiérarchie n'est pas toujours triviale.

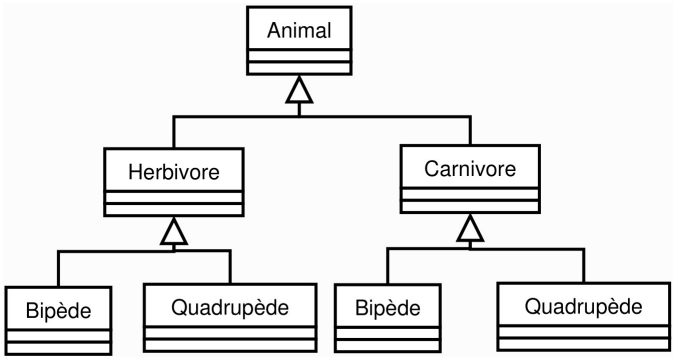
Exemple : hiérarchie d'animaux

soit les classes associées :

- aux régimes alimentaires = { Herbivore, Carnivore, ... }.
- aux modes de locomotion = { Bipède, Quadrupède, ... }.

Deux hiérarchies différentes peuvent être construites :





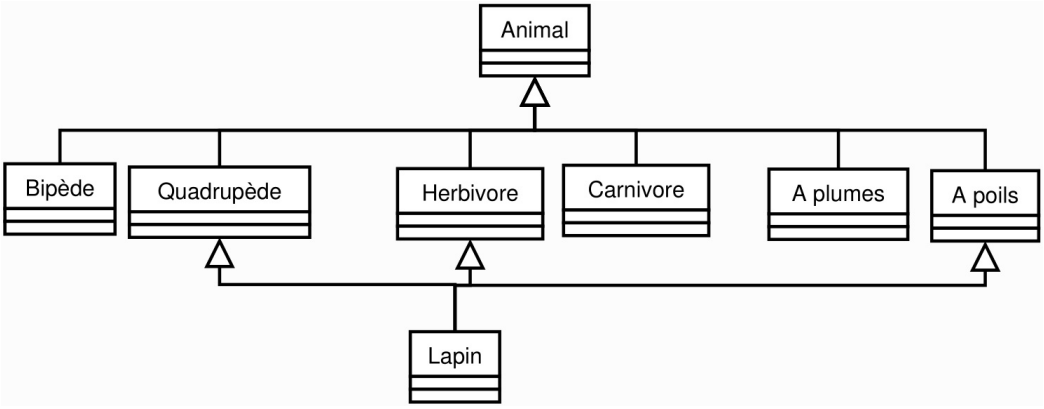
Quelle construction préférer ?

Ces deux hiérarchies posent problèmes :

Elles posent le même problème de redondance (doublement des héritages au second niveau de la hiérarchie)

Donc, doublement du code à écrire (constructeur) dans chaque branche.

Solution élégante :



L'héritage multiple consiste à hériter en même temps de l'ensemble des propriétés.

Permet de limiter le nombre d'héritage, et de ne construire que les objets issus des combinaisons les plus appropriées.

3.8 Interface

Définition 14 : interface
une interface est l'ensemble des méthodes qui peuvent appliquée sur un objet.

Remarques :
- façon dont l'objet communique avec l'extérieur,
- définit un comportement abstrait (pas besoin de connaitre l'implémentation pour l'utiliser) ⇒ forme d'abstraction

Permet d'arriver à la notion de type de donnée abstrait (TDA) : type de données défini par un ensemble d'opération (=interface).

Exemple : une liste (TDA) est définie par l'interface d'accès à la liste, indépendamment de l'implémentation concrète (chainée, tableau, ...).

Définition 15 : interface abstraite
une interface est une classe dont les membres sont des fonctionnalités (=méthodes) abstraites (= non implémentées).

L'implémentation de l'interface abstraite est définie lors de son héritage sur une classe concrète.

3.9 Abstraction

Définition 16 : classe abstraite
une classe abstraite est une classe dont une partie de son interface est abstraite.

Propriétés :
- une classe abstraite définie et exécute des actions abstraites.
- abstraire signifie ignorer les fonctionnalités, propriétés ou fonctions non pertinentes (à la compréhension), et accentuer (par contraste) celles qui le sont.

- façon de gérer la complexité en représentant une réalité complexe sous la forme d'un modèle simplifié (abstrait).

Conséquences :

- les fonctionnalités sont éventuellement partiellement implémentée (ou fournies dans une implémentation par défaut),
- les fonctionnalités non implémentées sont abstraites,
- une classe abstraite ne peut pas être instanciée : un objet abstrait a toujours une réalisation concrète.
- une classe abstraite est destinée à être héritée : charge à la classe fille d'implémenter toutes les méthodes (sauf si elle est elle-aussi abstraite).

3.10 Polymorphisme

DÉFINITION 17 : polymorphisme

Le polymorphisme est la capacité pour un objet d'être vu comme membre de l'une des classes dont il hérite, et en conséquence, à prendre plus d'une forme (=type).

Exemple : si un chat est un félin, et un félin est un animal, alors on peut considérer l'instance d'un chat comme l'instance d'un félin ou d'un animal.

Si une méthode de la classe mère est spécialisée dans la classe fille, alors la méthode spécialisée est toujours utilisée qu'importe la forme que prend l'instance de l'objet (elle utilise toujours sa méthode la plus appropriée).

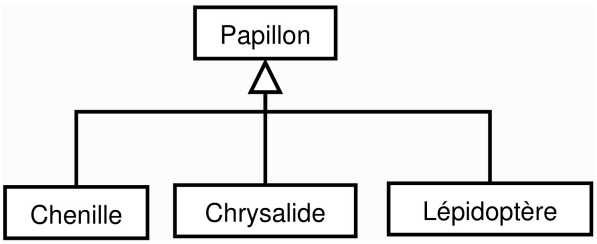
Pourquoi manipuler un objet d'un type donné comme un objet de son type de base ?

- permet l'appel aux fonctions abstraites (en utilisant le type sous-jacent),
- permet de construire des collections d'objets de type différent, mais ayant le même type de base,
- permet de passer des objets plus spécialisés comme paramètre d'une méthode qui demande un type plus général.

3.11 Délégation d'héritage

La relation d'héritage introduit un couplage statique non adaptée à une mutation (changement dynamique de type).

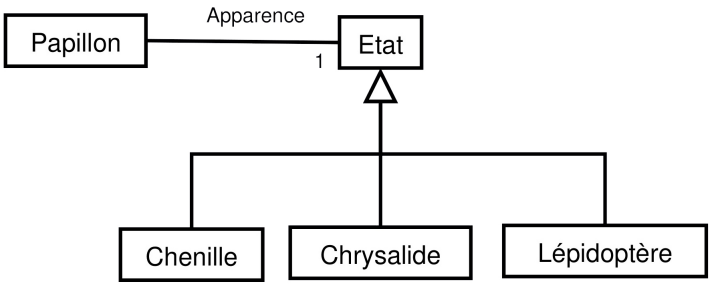
Exemple : le papillon



La solution consiste à effectuer de la délégation d'héritage =

- sortir le caractère mutable de la classe pour le déléguer à une classe T qui devient la base de la hiérarchie des types mutables.
- la classe déléguée T est ensuite associée à la classe qui possède le caractère mutable.

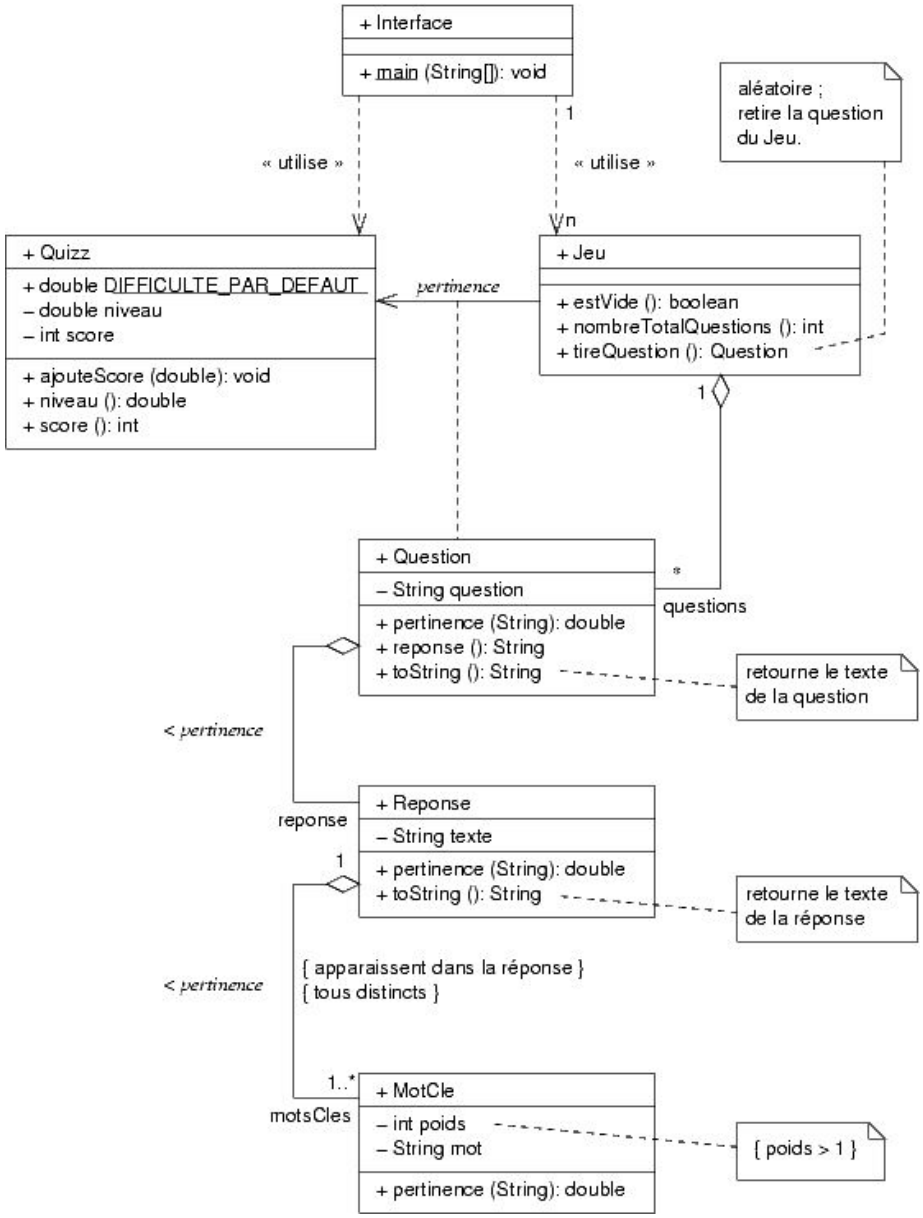
Exemple : correction de l'exemple du papillon




```
// classe polymorphe modélisant le concept
class Etat {};
class Chenille : public Etat { ... };
class Chrysalide : public Etat { ... };
class Lepidoptere : public Etat { ... };

// classe pour le papillon
class Papillon {
protected: Etat *state;
  ...
};
```

3.12 Exemple UML



4 Erreurs classiques

Lors du développement, il est souvent plus facile de se rendre compte que la conception ou les différents cycles de développement ont conduit à des problèmes qui n'avaient pas été anticipés.

Les principes généraux ayant déjà été édictés (forte cohésion, faible couplage), les erreurs présentées sont de deux ordres :

- Les **erreurs de conception** sont les erreurs présentes généralement dès la conception mais qui peuvent émerger au fur et à mesure de l'évolution du projet au cours du développement.
- Les **erreurs architecturales** sont les erreurs liés à l'architecture sous-jacente utilisée lors de la conception.

Ce type d'erreurs s'appelle un anti-modèle (anti-pattern) par opposition aux modèles de conception (pattern). Un ensemble de modèles de conception de base pour les classes sera présenté dans un chapitre dédié.

Dans les fait, il s'avère souvent qu'un anti-pattern est plus facile à identifier qu'un pattern, ce qui peut générer de bons réflexes critiques lorsque l'on s'aperçoit que l'on est en train d'en construire un.

4.1 Erreurs de conception

Les erreurs courantes de développement sont les suivantes :

- **Abstraction inverse :** fait d'utiliser une interface complexe pour effectuer des actions simples.
- **Action à distance :** emploi immodéré de variables globales ou des interdépendances accrues entre objets.
- **Ancre de bateau :** composant inutilisé mais qui est gardé en pensant que ce code servira plus tard.
- **Erreur de copier/coller :** duplication de code sans vérification, au lieu de factoriser les parties communes.
- **Code spaghetti :** programmation impliquant l'impossibilité de modifier une petite partie du logiciel sans altérer le fonctionnement de tous les autres composants.
- **Code ravioli :** programmation utilisant des centaines de petites classes, finissant par produire un code où l'on ne sait plus vraiment où les choses se passent.
- **Réinventer la roue (carrée) :** réécriture d'un code (par exemple) déjà existant dans une bibliothèque (carrée = idem, mais mal).
- **Objet divin :** objet assurant trop de fonctions essentielles.
- **Coulée de lave :** partie de code encore immature mis en production, le forçant à rester tel qu'il est en empêchant sa modification.
- **Couteau suisse :** utilisation d'un outil qui fait beaucoup de choses de manière acceptable, mais rien de manière vraiment efficace.
- **Marteau doré :** utilisation de manière obsessionnelle d'une technologie familière (avec un bon marteau, tous les problèmes ressemblent à des clous).
- **Poulet vaudou :** lorsque quelque chose n'est pas complètement compris dans un problème de programmation (exemple : code obtenu par copié-collé d'autres codes, et qui fonctionne, mais personne ne comprend pourquoi).
- **YAGNI** (You Aren't Gonna Need It) : implémenter un composant dont on pense qu'il sera inutile plus tard.

4.2 Erreurs architecturales

Les erreurs courantes de conception sont les suivantes :

- **Le Blob :** système contenant peu de classes (blobs) qui ont des centaines de méthodes et de variables et qui encapsulent la logique de tout le programme ; le reste étant composé de nombreux objets sans queue ni tête.

- **Architecture comme préalable :** fait d'utiliser une nouvelle architecture par simple préférence (nouveauté, nouvelle technologie) alors qu'elle n'est pas nécessaire et n'a fait l'objet d'aucune demande par le client.
- **Architecture comme conséquence :** absence de spécification d'architecture dans un projet en développement, risquant de faire émerger des problèmes qui auraient pu être anticipé lors de la phase d'étude.
- **Effondrement de l'analogie :** la conception manque de précision est fait en permanence référence à une analogie qui au fil du temps se révèle inadaptée.
- **Mariage sumo :** mariage de deux gros composants logiciels trop rigides pour permettre une nouvelle extension.
- **Tuer deux oiseaux avec une pierre :** fait de prendre des décisions à long terme sur l'architecture en se basant sur un couteau suisse.
- **Se tenir sur les épaules d'un nain :** fait d'imposer l'utilisation d'une bibliothèque pour résoudre un problème, et que celle-ci se révèle être truffée d'erreurs de conception ou de hack (voir qu'elle ne fasse pas la moitié de ce que vous souhaiteriez qu'elle fasse).

Pour l'ensemble des anti-pattern (developpement et architecture), voir `http://c2.com/cgi/wiki?AntiPatternsCatalog`.

5 Comparaison des langages de POO

Définition 18 : Langage Orienté Objet Pur

Un langage OO est pur si il ne contient que des objets et des classes, à savoir :

• tous les types de données (prédéfinis et définis par l'utilisateur) sont des objets.

• le langage est basé sur les concepts suivants : encapsulation, occultation des données, héritage, polymorphisme, abstraction.

Conséquence : un langage OO qui a des types de données primitifs (int, float, ...) ou une classe enveloppe (wrapper class) qui encapsule les types n'est pas pur.

Définition 19 : Typage dynamique et statique

- un typage est dynamique si toute variable est attaché à un seul objet.
- un typage est statique si toute variable est attaché à un type (à la compilation) et un objet.

Conséquence : dans un langage OO à typage dynamique, le type associé à une variable peut changer au cours de l'exécution.

Analyse

- **Orientation objet pure :** important car revient à mettre des exceptions sur la façon de penser et considérer les objets et leurs relations.
- **Typage dynamique :** permet de penser un objet comme pouvant avoir un type évolutif.
- **Héritage multiple :** à savoir l'héritage **à la fois** de plusieurs classes. L'héritage simple d'une classe et éventuellement d'interfaces devrait être privilégié. L'héritage multiple peut poser des problèmes conceptuels (héritages en diamant), mais est une solution élégante à certains types de lien.
- **Surcharge de méthode :** permet de spécialiser des méthodes issues d'une classe mère dans une classe fille.

- **Surcharge d'opérateur :** permet de définir le sens des opérateurs sur les objets.
- **Classe générique :** permet de définir un modèle de classe paramétrable par un type (cette notion n'a pas de sens pour un typage dynamique). Problème : comment définir les types acceptables par une classe générique.
- **Lien tardif :** (dynamic binding) mécanisme utilisé pour gérer le polymorphisme (*i.e.* la capacité de regarder un objet sous ses multiples formes).

Table de comparaison

	Small -talk	Java	C++	C#	Eiffel	Ruby
Orientation Objet	Pure	Hyb.	Hyb.	Hyb.	Pure	Pure
Typage	Dyn.	Stat.	Stat.	Stat.	Stat.	Dyn.
Héritage	Simp.	Mult.	Mult.	Mult.	Mult.	Mult.
Surcharge de méthode	non	oui	oui	oui	non	non
Surcharge d'opérateur	oui	non	oui	oui	oui	oui
Classes génériques	-	oui	oui	oui	oui	-
Lien tardif	oui	oui	oui	oui	oui	oui

5.1 Smalltalk

Smalltalk est le premier langage orienté objet mis à disposition (1980), en partie à but éducatif, dans le but de mettre un langage de programmation "naturel".

- langage orienté objet dynamiquement typé.
- modèle uniforme d'objet : tout est objet (types primitives, types définis par l'utilisateur).
- toutes les opérations sont effectuées en envoyant des messages.
  un objet Smalltalk fait essentiellement trois choses : conserver son état ; recevoir un message pour lui-même ou un autre objet ; pendant qu'il traite un message, envoyer un message à lui-même ou à un autre objet.
- héritage complet : tous les aspects de la classe parent sont accessibles.
- pas d'héritage multiple.
- gestion mémoire automatique (compteur de référence)
- système totalement réflexif (reflexion = capacité pour un programme d'analyser sa propre structure).
  possibilité de demander à un objet comment il est construit, de demander d'où vient un message, ...

5.2 C++

C++ a été créé à partir de 1983 (toujours en cours d'évolution C^{++}_{15}, C^{++}_{17}, ...)

- conçu comme une version orientée objet du C : ajoute la programmation orientée objet (typage statique, héritage multiple), les méthodes virtuelles et la programmation générique,
- permet une approche procédurale ou orientée objet (= POO non pure),
- typage statique à la compilation (i.e. un type est affecté à l'objet à la compilation, mais n'empêche pas l'objet d'être polymorphe).

- pas de ramasse-miettes (garbage collecting), pas de vérification de bornes mémoires
- permet les manipulations d'emplacement mémoire à travers des pointeurs,
- permet des opérations sur des composants système bas-niveau.

Cette approche permet d'obtenir :
- une efficacité d'exécution très importante,
- tout en conservant une empreinte mémoire faible.

Ceci en fait un langage les plus utilisés pour les applications haute-performances ou temps réel.

5.3 Java

Crée à partir de 1990 par Sun Microsystem, puis par Oracle.
- syntaxe similaire au C++ (bien qu'incompatible), mais plus facile à apprendre.
- ne permet pas de programmation bas niveau, en vue d'assurer la sécurité de manipulation et d'accès au données,
  donc pas de pointeur.
- ramasse-miettes (pour désallocation, et possible rélocalisation d'objets).
- hiérarchie de classe avec la classe Object à la racine,
- héritage simple, interfaces pour l'héritage multiple.
- POO non pure en raison de la présence de BIT et des opérations arithmétiques de base sur ces BITs.

Java a été créé pour être un langage portable :
- langage interprété s'exécutant sur une machine Java virtuelle protégée,
- donc, presque toujours plus lent et moins performant qu'un exécutable natif.

5.4 C

C# est le langage de POO du framework .NET
- permet la programmation fonctionnelle, orientée objet et impérative (donc pas POO pure),
- syntaxe similaire à C++ et Java.
- conception des classes, instances, l'héritage, le polymorphisme sont assez standard.
- ramasse-miettes
- comme Java, compilé dans un langage intermédiaire et exécuté sur une machine virtuelle (composant nommé Common Language Runtime),
- introduit le concept de propriété (façon d'encapsuler l'accès à un champs dans une classe) et de delegate (similaire à un pointeur sur une fonction),

C# est conçu pour concurrencer Java sur les plateformes Windows.

5.5 Eiffel

Crée en 1986 par Bertrand Meyer (auteur de "Conception et Programmation Orientées Objet).
- la conception est basée sur des classes.
- une classe a la capacité d'exporter certains de ses attributs visibles pour ses clients (tout en gardant les autres cachés). Permet de contrôler la visibilité de ses fonctionnalités.

- l'héritage multiple est supporté (les conflits de nom sont résolus en permettant de renommer le nom des méthodes héritées ; noms dupliqués illégaux).
- les assertions définis dans une classe parent sont héritées (permet de définir des contraintes qui assure que les sous-classes conservent la sémantique attendue).

Introduit des concepts novateurs :

- utilise des assertions pour exprimer les propriétés formelles des méthodes sous forme de pré-condition, post-condition et l'invariant de classe (= programmation par contrat),
- séparation commande-requête (CQS) : toute méthode doit être exclusivement soit une commande (qui effectue une action), soit une requête (qui retourne des données à l'appelant). poser une question ne doit pas changer la réponse.
- principe de l'accès uniforme (voir le cours C++),
- principe ouvert-fermé : toute entité (classe, modules, ...) doit être ouvert aux extensions mais fermé aux modifications (*i.e.* peut être étendue sans changer le code source).
- principe séparation option-opérande : les arguments d'une opération doit contenir uniquement des opérandes. Les options doivent être fixées dans des opérations séparées.
- les assertions définis dans une classe parent sont héritées (permet de définir des contraintes qui assure que les sous-classes conservent la sémantique attendue).

Initialement conçu pour améliorer la fiabilité du développement de logiciels commerciaux.

- la conception est basée sur des classes.
- une classe a la capacité d'exporter certains de ses attributs visibles pour ses clients (tout en gardant les autres cachés). Permet de contrôler la visibilité de ses fonctionnalités.
- l'héritage multiple est supporté (les conflits de nom sont résolus en permettant de renommer le nom des méthodes héritées ; noms dupliqués illégaux).
- une classe a la capacité d'exporter certains de ses attributs visibles pour ses clients (tout en gardant les autres cachés). Permet de contrôler la visibilité de ses fonctionnalités.
- l'héritage multiple est supporté (les conflits de nom sont résolus en permettant de renommer le nom des méthodes héritées ; noms dupliqués illégaux).
- les assertions définis dans une classe parent sont héritées (permet de définir des contraintes qui assure que les sous-classes conservent la sémantique attendue).

Initialement conçu pour améliorer la fiabilité du développement de logiciels commerciaux.

5.6 Ruby

Langage de scripting OO développé en 1993 par Matsumiko Yukihiro avec un but similaire à Python ou Perl.

- influencé par Smalltalk et Eiffel,
- POLA (Principe Of the Least Astonishment) = syntaxe qui minimise la surprise sur la manière d'écrire quelque chose et son résultat,
- héritage simple, mais mixin (héritage multiple) par inclusion de modules dans les classes.
- métaclasses (métaprogrammation).
- tout est expression et tout est exécuté de manière impérative.
- réflexion dynamique et altération dynamique (changement d'un type en cours d'exécution).
- intégration des expressions régulières dans la syntaxe du langage.

- utilisation de blocs comme paramètre (type de lambda-fonction),
- ramasse-miettes.

Le but est d'avoir un langage OO facile et agréable à utiliser.

Il est vivement conseillé d'apprendre ce langage.

6 Stratégie de codage en C++

6.1 Programmation par contrat

Basé sur la constatation que lorsque l'on appelle une fonction,

- les paramètres d'entrée doivent respecter des contraintes (domaine de validité de la valeur, taille d'un tableau, ...)
  pré-conditions = conditions nécessaires à vérifier afin que la fonction s'exécute correctement.
- les structures de données sont cohérentes à l'entrée et à la sortie.
  à savoir, une structure qui entre dans une fonction est supposée cohérente.
  La cohérence d'une structure n'est pas modifiée par la fonction (on parle alors d'**invariance**).
- avant le retour de la fonction,
  post-conditions = conditions permettant de vérifier que la fonction s'est exécutée correctement.

Autrement dit, une fonction doit commencer par vérifier que les pré-conditions sont remplies, se terminer en vérifiant que les post-conditions le sont aussi, et que l'invariance des structures de données sont conservées.

Conséquences : la programmation par contrat dans une méthode permet :

- de savoir si la méthode est utilisée dans le cadre de sa définition.
- de s'assurer que le résultat de la méthode est conforme à ce qui est attendu.

En terme de recherche d'erreur, permet d'identifier rapidement :

- qu'une méthode ne fonctionne pas correctement (échec des post-conditions).
- que l'objet modifié n'est plus cohérent (échec de l'invariant).
- qu'une méthode n'est pas utilisée correctement (échec des pré-condition).

Mais comment l'implémenter facilement ?

En utilisant `assert` :

- inclure assert.h (`#include <cassert>`)
- pour toutes les fonctions, partout où cela se justifie,

```
TYPE fun(...) {
  // éventuellement, plusieurs asserts
  assert( préconditions );
  // code de la fonction
  ...
  assert( postconditions );
  // éventuellement, plusieurs asserts
  assert( vérifier invariance );
  return value;
}
```

Lorsqu'une assertion **n'est pas vérifiée**, elle affiche l'assertion ayant échouée, l'endroit où l'échec a eu lieu, produit un arrêt non récupérable du programme, avec la génération d'un fichier core (utilisable par un débogueur).

Exemple :

```
float fun(const float x, const int n, const float *tableau) {
   assert( (0.f<=x) \&\& (x<=10.f) );
   assert( n > 0 );
   assert( tableau != NULL);
   ...
   assert( 0.f<=y);
   return y;
}
```

Exemple d'échec à l'exécution :

```
assertion "n>0" failed: file "toto.cpp", line 12,
    function: float fun(...) Aborted (core dumped)
```

Attention :

- les `assert` ne sont effectués que lorsque le programme est compilé en mode debug (lorsque le symbole `_DEBUG` est défini).
  Manuellement, pour activer `assert` :
  - ◇ **globalement :** ajouter `-D_DEBUG` aux `CPPFLAGS` du `makefile`.
  - ◇ **localement** dans un module, ajouter `#define _DEBUG` dans le source du module permet d'activer `assert` dans ce module.
- autrement dit, les `asserts` ne doivent être utilisés que pour vérifier les conditions d'usage et de fonctionnement de la fonction pendant la mise au point, et ne doivent **en aucun cas** effectuer un traitement d'erreurs nécessaire au bon déroulement du programme.

6.2 Outils du compilateur

a) Compilation

Un compilateur peut être un outil précieux pour trouver des erreurs autres que des erreurs syntaxiques :

- le compilateur C/C++ avec l'option `-Wall` affiche les avertissements associés à des comportements à risque (problème de portabilité, ...).
- l'utilisation d'un outil de vérification statique (outil effectuant une analyse plus poussé du code qu'un compilateur et signalant des erreurs d'ordre sémantique) est vivement conseillé.
  Exemple : `lint`, `splint`, `cppcheck`

Dans tous les cas,

- **aucun warning ne doit rester incompris**.
- **aucun warning ne doit rester** : le code doit être corrigé afin que tous les warnings disparaissent.

b) Précompilateur

Le précompilateur est une phase préparatoire de la compilation et dont le but est d'appliquer toutes les commandes de précompilation (= celles qui commencent par # dans le code).

Comme nous l'avons en partie déjà vu, elles permettent :

- `#define SYMB VAL` ou `#define SYMB` : définir des symboles (ayant une valeur ou non).
- `#define MAC(...) ...` : définir des macros.
- `#ifdef SYMB ... #endif` ou `#ifdef SYMB ... #else ... #endif` : tester la définition d'un symbole.
- `#undef SYMB` : annuler la définition du symbole `SYMB`.

Par ailleurs, les symboles suivants sont en général définis :

- `_FILE_` : nom du fichier qui contient le code courant.
- `_FUNCTION_` : nom de la fonction courante.
- `_LINE_` : numéro de la ligne courante.
- `_DEBUG` : si le code est compilé en mode débogue.

Exemple d'utilisation (1) : portion de code exécuté seulement en mode debug

On pourrait entourer tout code devant être exécuté en mode débogue de `#ifdef _DEBUG ... #endif`, mais cela est un peu verbeux.

Une solution plus ingénieuse consiste à définir :

```
#ifdef _DEBUG
#define DEBUG(x) x
#else
#define DEBUG(x)
#endif
```

Et écrire ensuite :

```
DEBUG( code );
```

pour exécuter chaque ligne (ou bloc) de *code* uniquement en mode débogue.

Exemple d'utilisation (2) : activer des logs (= traces)

Définir :

```
#ifdef _LOG
#define LOG(s,...) printf( s, __VA_ARGS__ )
#else
#define LOG(s,...)
#endif
```

Puis utiliser `LOG` comme un `printf` :

```
LOG("i=%d␣a=%f$\n",i,a);
```

Pour activer les logs :

- **localement** dans un module : mettre `#define _LOG`
- **globalement** : compiler avec l'option `-D_LOG` (à mettre dans les `CPPFLAGS` du `makefile`).

Notes :

- Plusieurs niveaux de log peuvent être créés (correspondant à la quantité de traces que l'on souhaite observer). On peut également les sauvegarder dans un fichier (fichier de log) pour des analyses futures.
- les macros variadiques ne sont pas implémentés sur tous les compilateurs. Ok pour `g++ v4.9.2` et VC14.

Exemple d'utilisation (3) : simplifier l'écriture d'erreurs et de warnings

Définir :

```
#define FATAL(s,...) { printf("Fatal␣[%s:%s:%d]␣" s "\n",
    __FILE__, __FUNCTION__, __LINE__, __VA_ARGS__); exit(-1); }
#define ERROR(s,...) { printf("Error␣[%s:%s:%d]␣" s "\n",
    __FILE__, __FUNCTION__, __LINE__, __VA_ARGS__); }
#define ERROR_RET(v,s,...) { printf("Error␣[%s:%s:%d]␣" s "\n",
    __FILE__, __FUNCTION__, __LINE__, __VA_ARGS__); return v; }
#define WARNING(s,...) { printf("Warning␣[%s:%s:%d]␣" s "\n",
    __FILE__, __FUNCTION__, __LINE__, __VA_ARGS__); }
```

Note : un `#define` doit être défini sur une seule ligne.

`FATAL`, `ERROR` et `WARNING` s'utilisent comme `printf`. `ERROR_RET` a en premier argument la valeur de retour de la fonction. `FATAL` provoque l'arrêt du code.

Exemple :

```
if (tab == NULL) FATAL("pas␣assez␣de␣mémoire");
if (eps > 1e-1) WARNING("précision␣insuffisante:␣eps=%f",eps);
if (val < 0.f) ERROR_RET(errcode,"résultat␣invalide.");
```

7 Critique de l'approche objet

Avis de pionners à l'origine des langages objets ou du C++ :

- «*Actually I made up the term 'object-oriented', and I can tell you I did not have C++ in mind.*» (**Alan Kay**)
- «*There are only two things wrong with C++ : The initial concept and the implementation.*» (**Bertrand Meyer**)
- «*C makes it easy to shoot yourself in the foot ; C++ makes it harder, but when you do, it blows away your whole leg.*» (**Bjarne Stroustrup**)
- «*Within C++, there is a much smaller and cleaner language struggling to get out.*» (**Bjarne Stroustrup**)

Le Java n'est pas mieux considéré :

- «*C++ is history repeated as tragedy. Java is history repeated as farce.*» (**Scott McKay**)
- «*Java, the best argument for Smalltalk since C++.*» (**Frank Winkler**)
- «*If Java had true garbage collection, most programs would delete themselves upon execution.*» (**Robert Sewell**)

«*There are only two kinds of languages : the ones people complain about and the ones nobody uses.*» (**Bjarne Stroustrup**)

7.1 Essence de l'argumentation

L'approche objet s'attache à perfectionner chaque objet en essayant d'en faire des entités parfaites, efficaces, mathématiquement démontrables, ...

Problème : cette approche a tendance à négliger :

- la vision globale sur les interactions entre objets,
- le but qui est de créer un système complet, fiable et flexible,
- le travail avec ceux qui essayent d'accomplir quelque chose dans le monde réel.

Il y a donc une dichotomie entre :

- l'approche objet consistant à encapsuler et isoler tout problème dans un monde statique,
- le monde réel : en évolution constante et chaotique, basé sur des humains.

Exemple : (analogie célèbre d'Alan Kay)

- l'approche objet consiste à dire que pour construire une niche, j'ai besoin de quelques montants, des planches et de shingle pour le toit (= les objets). Puis, avec un marteau et des clous (relations entre objets), je peux assembler tout cela pour obtenir ma niche (le programme).
- fort de cette expérience, je veux maintenant utiliser la même approche pour construire un lieu de culte monumental (100 fois plus grand).
  après tout, quelle est la différence avec un grande niche ?
  problème : 100 fois plus grand $\rightarrow$ 100^3 = 1.000.000 de fois plus lourd.
- **conséquence :**
  - ◇ les contraintes mécaniques sur les structures de cette taille ne sont pas du tout les mêmes que pour une petite structure (vent, poids, résistance des matériaux, ...)
  - ◇ tout cela va finir par s'effondrer, et faire une pyramide de gravats.
    ajouter de la structure supplémentaire = ajouter de la rigidité, et n'empêche pas l'effondrement.
- l'approche objet consiste à faire ainsi, et une fois que tout s'est écroulé, mettre un toit sur la pyramide, et affirmer que c'est ce que l'on voulait faire depuis le début.
  c'est peut-être ainsi que les pyramides ont été construites (sans rire ...)

7.2 Echec de l'acceptation de l'échec

En 1994, Peter Deutsch énonce 7 idées fausses de l'informatique distribuée : le réseau est fiable, la latence est zéro, la bande passante est infinie, la topologie ne change pas, le coût de transport est nul, ...

Des principes similaires peuvent être énoncés pour un ordinateur simple :

- l'ordinateur est fiable : lors de l'exécution, l'appel d'une fonction et d'une méthode peut échouer (bugs, appels incorrects, données sémantiquement mauvaises, ...)
- la latence est nulle : le retour d'un appel peut être long (échec, un thread peut mettre longtemps à s'arrêter, ...).
- la bande passante est infinie : temps de copie long, boucle infinie basée sur des données fausses, ...
- l'ordinateur est sécurisé : si n'importe quel composant logiciel a été écrit par un autre que vous, ...
- la topologie ne change pas : programmation basée sur les entités, appel dynamique de méthodes, ...
- le coût de transport est nul : un disque dur n'est pas différent d'un réseau, ...
- ...

Problème : l'informatique passe son temps à faire en sorte que les échecs ne se produisent pas :

- notions de plus en plus statiques (= dont le champs se restreint),
- langages de programmation sont de plus en plus strict,
- méthodologies de plus en plus lourdes.

Un échec est appelé une **exception** !
dans le monde réel, c'est la vie ! (`sh** happens!`)

Rend difficile de rendre un code résilient, qui peut se réparer lui-même quand quelque chose ne va pas (objets autonomes, avec une identité, avec un instinct de survie, ...).

Remarque :

la résilience est nécessaire pour les applications et les systèmes futurs afin de survivre aux erreurs issues de l'utilisateur, du système ou du réseau.

7.3 Le mythe de la réutilisation de code

La réutilisation de code est généralement un échec :

- faire en sorte qu'un classe soit réutilisable demande du travail supplémentaire, car nécessite :
  - ◇ un dépôt de code centralisé,
  - ◇ avoir un moyen de localiser le morceau de code réutilisable
    bonne méthode de classification, "bibliothécaire".
  - ◇ chaque code doit être documenté
- donc, pour beaucoup de codes, il est plus rapide de le réécrire que de le retrouver et le comprendre.
- un code ne peut être considéré comme réutilisable que s'il l'a été au moins 3 fois
  en général, on s'aperçoit que le code que l'on veut réutiliser n'a pas toutes les fonctionnalités requises, et il est remodifié (et complexifié).
  le code n'est réutilisable tel quel qu'au bout de plusieurs de ces itérations.
- souvent plus rentable économiquement de résoudre rapidement un problème spécifique, que de résoudre un problème plus complexe (analyse du domaine) dont on ne sait s'il sera réutilisé.
  coûte du temps et de l'argent sur le projet courant.

Conséquence : nécessite un investissement de la part des codeurs (méthode) et du management (coût financier).

Autres raisons de fond :

- **Granularité :** la granularité d'un système est la mesure de l'importance avec laquelle il peut être découpé en parties.
  - ◇ **gros grain :** plus facile à réutiliser car plus de fonctionnalités
  - ◇ **petit grain :** plus réutilisable.

  maximiser la possibilité de réutilisation d'un code consiste à construire les composants à gros grains à partir de composants à petit grain.
- **Poids :** le poids d'un composant est la mesure de l'importance avec laquelle il dépend de son environnement.
  - ◇ **composant léger :** plus difficile à concevoir (pour minimiser les dépendances), plus facilement réutilisable.
  - ◇ **composant lourd :** plus facile à utiliser (encapsule tout ce dont il a besoin).
- une hiérarchie a tendance à ne pas rester stable lors de sa vie,
  à moins d'avoir beaucoup d'utilisateurs et être utilisée depuis longtemps.

Le spectre des codes réutilisables est donc souvent très étroit.

Donc, la réutilisabilité du code est un mythe.

7.4 Échec de l'encapsulation

L'encapsulation marche parfaitement là où l'on en a besoin, mais devient problématique là où elle n'est plus nécessaire.

Comme en POO, l'encapsulation est imposée lors de la conception, violer ces règles amène à écrire du code interdépendant (friend, transfert d'accès).

Mais le plus problématique est que :

- un des aspects fondamentaux du génie logiciel est l'évolution du code,
- droits d'accès et accesseurs constituent la base de l'encapsulation.
- dans le cycle de vie du code, lorsqu'une classe doit être réusinée (refactoring), généralisée, paramétrisée en fontion des nouvelles exigences.
  - champs déplacé ⇒ déplacement d'accesseurs
  - champs ajouté ⇒ création d'accesseurs

En conséquence, l'encapsulation semble :

- engendrer beaucoup de travail (sans que rien de vraiment intéressant ne soit produit),
- être un frein à l'évolution du code,
- ne protéger les développeurs que d'eux-mêmes.

Exemple : amène à développer en C et exposer une interface publique C++.

7.5 Problème de conception

Les patrons de conception sont des façons de structurer des classes ou des relations entre objets permettant de résoudre des problèmes de conception courants dans une application.

Question : pourquoi ces conceptions ne découle-t-elle pas naturellement de la syntaxe du langage ?

Problème sur ce point avec les langages de POO les plus populaires (C++, Java, C#, Python).

Peter Norvig a montré que, pour 16 des 23 patrons de conception du GoF (Gamma et al.), il existe une manière naturelle de formuler ces patrons en `Lisp`.

Cela révèle un manque pour les langages précités :

- de puissance de programmation,
- de puissance d'abstraction.

Ces constructions devraient relever de l'abstraction du langage et non de l'écriture de code.

7.6 Problème de l'apprentissage

Le niveau de compétence requis pour programmer monte continument.
limite le public de qui peut être un développeur

L'approche tout objet est monolithique :

la complexité exige des niveaux d'expression multiples.

problèmes sont plus facilement décrits par certains concepts (complexité).

«*OOP is about taming complexity through modeling, but we have not mastered this yet, possibly because we have difficulty distinguishing real and accidental complexity.*» (**Oscar Nierstrasz**)

Conséquence : pour un langage objet comme le C++ :

- il est illusoire de croire que vous pourrez apprendre le C++ rapidement,
- il est faux de considérer que vous maîtriserez le C++ à la fin de ce cours,
- vous devez programmer très régulièrement en C++,
- essayer d'approfondir ses fonctionnalités (template, lambda, ...),
- revenir régulièrement à la norme et à la documentation (cppreference),
- lire et suivre les conférences des gourous C++ (Meyer, Alexandrescu, Sutter, ...)

Il y a toujours quelques choses à apprendre et/ou comprendre.

Conclusion

Dans ce chapitre, nous avons vu :

- Les bases de l'approche objet (concepts d'objet, de classe, d'instance, d'encapsulation, de persistance, de cohésion et de couplage).
- les bases de la représentation UML,
- les relations entre objets qui permettent de modéliser des systèmes complexes,
- les erreurs classiques de conception ou architecturales,
- la comparaison des principaux langages orientés objets,
- le principe de la programmation par contrat,

- une critique de l'approche objet

Cette partie sera complétée par une partie plus avancée sur les modèles de conception.