Compléments en Programmation Orientée Objet

Aldric Degorre

Version `2020.11.00` du 27 novembre 2020

Chargés de TP en 2020 :
Isabelle Fagnot [1] (lundi et mardi), Yan Jurski [2] (jeudi) et Aldric Degorre [3] (vendredi).

En remerciant mes collaborateurs des années passées, qui ont aidé à élaborer ce cours et à le faire évoluer.

---

1. `fagnot@irif.fr`
2. `jurski@irif.fr`
3. `adegorre@irif.fr`

Autres “codes”

Marquage du texte

- “**Blah :**” : titre de paragraphe
- “important” : mot ou passage important
- “**très important**” : mot ou passage très important
- “**concept**” : concept clé (en général défini explicitement ou implicitement dans le transparent... sinon, il faudra s’assurer de connaître ou trouver la définition)
- “*foreign words*” : passage en langue étrangère
- “`void duCodeJavaEnLigne( ){}`” : code java intégré au texte
-

```
    void duCodeJavaNonIntegre( ) {
        System.out.println("Java c'est cool !");
    }
```

Code non intégré au texte.

La programmation orientée objet

Principes

- **Principe de la POO :** des messages [1] s'échangent entre des objets qui les traitent pour faire progresser le programme.
- → **POO = paradigme centré sur la description de la communication entre objets.**
- Pour faire communiquer un objet a avec un objet b, il est nécessaire et suffisant de connaître les messages que b accepte : l'**interface** de b.
- Ainsi objets de même interface interchangeables → **polymorphisme**.
- Fonctionnement interne d'un objet [2] caché au monde extérieur → **encapsulation**.

Pour résumer : la POO permet de raisonner sur des **abstractions** des composants réutilisés, en ignorant leurs détails d'implémentation.

1. appels de **méthodes**
2. Notamment son état, représenté par des **attributs**.

La programmation orientée objet

Avantages et raisons du succès de la POO

La POO permet de découper un programme en composants

- peu dépendants les uns des autres (faible **couplage**)
  → code **robuste et évolutif**
  (composants testables et déboguables indépendamment et aisément remplaçables) ;
- réutilisables, au sein du même programme, mais aussi dans d'autres ;
  → facilite la création de logiciels de grande taille.

POO → (discutable) façon de penser naturelle pour un cerveau humain "normal" [1] : chaque entité définie représente de façon abstraite un concept du problème réel modélisé.

1. Non « déformé » par des connaissances mathématiques pointues comme la théorie des catégories (cf. programmation fonctionelle).

- **1992 :** langage Oak chez *Sun Microsystems* [1], dont le nom deviendra Java ;
- **1996 :** JDK 1.0 (première version de Java) ;
- **2009 :** rachat de *Sun* par la société *Oracle* ;

En 2020, Java est donc « dans la force de l'âge » (24 ans) [2], à comparer avec :

- même génération : Haskell : 30 ans, Python : 29 ans, JavaScript, OCaml : 24 ans
- anciens : C++ : 35 ans, C : 42 ans, COBOL : 61 ans, Lisp : 63 ans, Fortran : 66 ans...
- modernes : Scala : 16 ans, Go : 11 ans, Rust : 10 ans, Swift : 6 ans, Kotlin : 4 ans, ...

Depuis plusieurs années, Java est le 1er ou 2ème langage le plus populaire. [3]

1. Auteurs principaux : James Gosling et Patrick Naughton.
2. Je considère la version 1.0 de chaque langage. Mais sur le web, Java a déjà fêté ses 25 ans cet été !
3. Dans la plupart des classements principauxs : TIOBE, RedMonk, PYPL, ...
Selon les métriques, l'autre langage le plus populaire est C, Javascript ou Python.

Versions « récentes » de Java :

- **03/2014 :** Java SE 8, la version long terme précédente ;[1]
- **09/2018 :** Java SE 11, la version long terme actuelle ;[2]
- **03/2019 :** Java SE 12, la dernière version
- **15/09/2019 :** Java SE 15, la prochaine version, imminente !
- **09/2021 :** Java SE 17, la prochaine version long terme.

Ce cours utilise Java 11, mais :

- la plupart de ce qui y est dit vaut aussi pour les versions antérieures ;
- les nouveautés de Java 12 à 15 ne sont pas censurées.

1. Utilisée pour POOIG et CPOO5 jusque l'an passé.
2. Depuis Java 9, une version "normale" sort tous les 6 mois et une à "support long terme", tous les 3 ans.

Java

Caractérisitiques principales

« Java » (Java SE) est en réalité une plateforme de programmation caractérisée par :

- le langage de programmation Java
  - orienté objet à classes,
  - à la syntaxe inspirée de celle du langage C [1],
  - au typage statique,
  - à gestion automatique de la mémoire, via son **ramasse-miettes** (*garbage collector*).
- sa machine virtuelle (**JVM** [2]), permettant aux programmes Java d'être multi-plateforme (le **code source** se compile en **code-octet** pour JVM, laquelle est implémentée pour nombreux types de machines physiques).
- les bibliothèques officielles du JDK (fournissant l'**API** [3] Java), très nombreuses et bien documentées (+ nombreuses bibliothèques de tierces parties).

1. C sans pointeurs et **struct** $\simeq$ Java sans objet
2. *Java Virtual Machine*
3. *Application Programming Interface*

Domaines d'utilisation :

- applications de grande taille [1] ;
- partie serveur « *backend* » des applications web (technologies *servlet* et JSP) [2] ;
- applications « desktop » [3] (via Swing et JavaFX, notamment) multiplateformes (grâce à l'exécution dans une machine virtuelle) ;
- applications mobiles (années 2000 : J2ME/MIDP ; années 2010 : Android) ;
- cartes à puces (via spécification Java Card).

1. Facilité à diviser un projet en petits modules, grâce à l'approche OO. Pour les petits programmes, la complexité de Java est, en revanche, souvent rebutante.
2. En concurrence avec PHP, Ruby, Javascript (Node.js) et, plus récemment, Go.
3. Appelées aujourd'hui "**clients lourds**", par opposition à ce qui tourne dans un navigateur web.

Mais :

- Java ne s'illustre plus pour les clients « légers » (dans le navigateur web) : les *applets* Java ont été éclipsées [1] par Flash [2] puis Javascript.
- Java n'est pas adapté à la programmation système [3].
  $\rightarrow$ C, C++ et Rust plus adaptés.
- Idem pour le temps réel [4].
- Idem pour l'embarqué [5] (quoique... cf. Java Card).

---

1. Le plugin Java pour le navigateur a même été supprimé dans Java 10.
2. ... technologie aussi en voie de disparition
3. APIs trop haut niveau, trop loin des spécificités de chaque plateforme matérielle.
Pas de gestion explicite de la mémoire.
4. Ramasse-miette qui rend impossible de donner des garanties temps réel. Abstractions coûteuses.
5. Grosse empreinte mémoire (JVM) + défauts précédents.

Compilation

Compilation : traduction code source (lisible par l'humain) $\rightarrow$ code « machine »

Une seule fois pour une version donnée du code source.

Cas classique : machine = architecture physique (ex : processeur x86-64, ARM, etc.)

Cas de Java : machine = JVM, la **machine virtuelle** [1] de Java.

Dans les deux cas : transformation d'un langage évolué, plus ou moins haut-niveau, vers une séquence d'instructions (appartenant à un **jeu d'insructions** de petite taille).

1. programme qui exécute des programmes écrits en **code-octet**. Ce programme est lui-même exécuté sur une **machine hôte** (machine physique ou bien autre machine virtuelle...).

Compilation

Autour de la JVM

- « JVM » est le standard de machine virtuelle pour Java publié [1] par *Oracle*.
- L'implémentation par *Oracle* est appelée *HotSpot* (plusieurs distributions [2]).
- Implémentation tierce compatible avec HotSpot : *OpenJ9* (fondation *Eclipse*).
- VMs incompatibles : notamment *Dalvik* (sous Android).
- On peut même compiler Java vers d'autres 'cibles' : code natif (cf GCJ, cf ART dans Android ≥ 5, jaotc/Graal dans Java ≥ 9 , ...), voire vers autre langage (ex : Javascript, via GWT).
- Vice-versa, d'autres langages que Java sont compilés pour la JVM : Scala, Groovy, Clojure, Gosu, Ceylon, Kotlin...

1. Spécifié par la "JVMS" : `https://docs.oracle.com/javase/specs/jvms/se11/html/index.html`
2. L'officielle d'Oracle, ainsi que toutes les dérivées d'OpenJDK : AdoptOpenJDK, Amazon Corretto, Microsoft Azul Zulu, BellSoft Liberica, SAP SapMachine, ...

Exécution

Exécution : à chaque fois qu'on veut utiliser le programme. La machine traite une à une les opérations élémentaires dictées par le code compilé.

Dans le cas classique : le processeur physique (CPU), lit les instructions une par une et active à chaque fois le circuit dédié correspondant.

Dans le cas de Java : la JVM traduit le code-octet à la volée en instructions natives pour le CPU, qui les exécute immédiatement.

code source Java $\xrightarrow[\textit{compilation}]{\texttt{javac}}$ *code-octet* $\xrightarrow[\textit{exécution}]{\texttt{java (JVM)}}$ *instructions CPU "réelles"*

Compilation ?

Est-ce que le bytecode est vraiment juste interprété par la JVM ?

En réalité, plusieurs stratégies :

- Simple interprétation du code-octet au fur et à mesure de son exécution.
- JIT, « Just In Time Compilation » : pendant l'exécution du programme, la VM traduit (une bonne fois pour toutes) en code natif optimisé les morceaux de code qui s'exécutent souvent
- AOT, « Ahead Of Time Compilation » : "on" compile tout ou partie du code-octet vers des instructions natives avant son exécution

La JVM HotSpot (JVM par défaut depuis Java 3), fait du JIT. Depuis $\sim$ Java 9, Oracle expérimente AOT via GraalVM.

Le « programme » Java

Architecture du code source

Le code source : fichier `.java` ∈ paquetage ∈ module ∈ projet

- la base : fichiers « source » `.java` et éventuellement ressources diverses (images, sons, polices de caractères, etc.) ;
- fichiers regroupés en **paquetages** (matérialisés par des sous-répertoires) ;
- si on utilise JPMS (Java ≥ 9), paquetages regroupés en **modules** (décrits dans les fichiers `module-info.java`) [1] ;
- paquetages (et modules) souvent regroupés en « **projets** » [2].
  Un tel projet est typiquement un répertoire muni d'un ou plusieurs fichiers de configuration (propriétaires à l'outil utilisé).

1. Dans IntellIJ IDEA, ces modules correspondent désormais aux modules du projet, subdivision déjà proposée par cet IDE, avant Java 9.
2. C.-à-d. un ensemble de packages ou modules partageant une configuration commune dans un IDE (comme Eclipse, NetBeans, IntellIJ IDEA, ...) ou dans un moteur de production (make, ant, maven, gradle, ...). Dans Eclipse, en plus, un « espace de travail » regroupe les projets apparaissant dans une même fenêtre.

Le « programme » Java

Architecture d'un programme compilé/distribuable

Le code compilé :

- organisé de façon similaire au code source.
- Mais, à chaque un fichier `.java` correspond (au moins) un fichier `.class`.
- Un programme compilé est distribuable via une ou des archives `.jar`[1].
- Si on utilise JPMS, il y a exactement un fichier `.jar` par module.

1. C'est en réalité un fichier `.zip` avec quelques méta-données supplémentaires.

Question(s) de style

- Aucun programme n'est écrit directement dans sa version définitive.
- Il doit donc pouvoir être facilement modifié par la suite.
- Pour cela, ce qui est déjà écrit doit être **lisible et compréhensible**.
  - lisible par le programmeur d'origine
  - lisible par l'équipe qui travaille sur le projet
  - lisible par toute personne susceptible de travailler sur le code source (pour le logiciel libre : la Terre entière !)

Les commentaires [1] et la javadoc peuvent aider, mais rien ne remplace un code source bien écrit.

1. Si un code source contient plus de commentaires que de code, c'est en réalité assez "louche".

Question de goût ?

- “être lisible” $\rightarrow$ évidemment très subjectif
- un programme est lisible s’il est écrit tel qu’“on” a l’habitude de les lire
- $\rightarrow$ habitudes communes prises par la plupart des programmeurs Java (d’autres prises par seulement par telle ou telle organisation ou communauté)

Langage de programmation $\rightarrow$ comme une langue vivante !

Il ne suffit pas de connaître par cœur le livre de grammaire pour être compris des locuteurs natifs (il faut aussi prendre l’accent et utiliser les tournures idiomatiques).

Une hiérarchie de normes

Habitudes dictées par :

1. le compilateur (la syntaxe de Java [1])
2. le guide [2] de style qui a été publié par Sun en même temps que le langage Java ($\rightarrow$ conventions à vocation universelle pour tout programmeur Java)
3. les directives de son entreprise/organisation
4. les directives propres au projet

… et ainsi de suite (il peut y avoir des conventions internes à un package, à une classe, etc.)

» et enfin... le bon sens ! [3]

Nous parlerons principalement du 2ème point et des conventions les plus communes.

1. L'équivalent du livre de grammaire dans l'analogie avec la langue vivante.
2. À rapprocher des avis émis par l'Académie Française ?
3. Mais le bon sens ne peut être acquis que par l'expérience.

Nommer les entités
(classes, méthodes, variables, ...)

Règles de capitalisation pour les noms (auxquelles on ne déroge pratiquement jamais) :

- ... de classes, interfaces, énumérations et annotations [1] ⟶ `UpperCamelCase`
- ... de variables (locales et attributs), méthodes ⟶ `lowerCamelCase`
- ... de constantes (**`static final`** ou valeur d'**`enum`**) ⟶ `SCREAMING_SNAKE_CASE`
- ... de packages ⟶ tout en minuscules sans séparateur de mots [2]. Exemple : `com.masociete.biblitothequetruc` [3].

⟶ rend possible de reconnaître à la première lecture quel genre d'entité un nom désigne.

1. c.-à-d. tous les types référence
2. "_" autorisé si on traduit des caractères invalides, mais pas spécialement encouragé
3. pour une bibliothèque éditée par une société dont le nom de domaine internet serait `masociete.com`

Nommer les entités

Codage, et langue

- **Se restreindre aux caractères suivants :**
  - `a-z`, `A-Z` : les lettres minuscules et capitales (non accentuées),
  - `0-9` : les chiffres,
  - `_` : le caractère soulignement (seulement pour `snake_case`).

  Explication :
  - `$` (dollar) est autorisé mais réservé au code automatiquement généré ;
  - les autres caractères ASCII sont réservés (pour la syntaxe du langage) ;
  - la plupart des caractères unicode non-ASCII sont autorisés (p. ex. caractères accentués), mais aucun standard de codage imposé pour les fichiers `.java`.[1]
- **Interdits :** commencer par `0-9` ; prendre un nom identique à un mot-clé réservé.
- **Recommandé :** Utiliser l'Anglais américain (pour les noms utilisés dans le programme **et** les commentaires **et** la javadoc).

1. Or il en existe plusieurs. En ce qui vous concerne : il est possible que votre PC personnel et celle de la salle de TP n'aient pas le même réglage par défaut → incompatibilité du code source.

Nommer les entités

Nature grammaticale (1)

Nature grammaticale des identifiants :

- types (→ notamment noms des classes et interfaces) : nom au singulier
  ex : `String`, `Number`, `List`, ...
- classes-outil (non instanciables, contenu statique seulement) : nom au pluriel
  ex : `Arrays`, `Objects`, `Collections`, ... [1]
- variables : nom, singulier sauf pour collections (souvent nom pluriel) ; et booléens (souvent adjectif ou verbe au participe présent ou passé). ex :

```
int count = 0; // noun (singular)
boolean finished = false; // past participle
while (!finished) {
    finished = ...;
    ...
    count++;
    ...
}
```

1. attention, il y a des contre-exemples au sein même du JDK : `System`, `Math`... oh !

Nommer les entités

Nature grammaticale (2)

Les noms de méthodes contiennent généralement **un verbe**, qui est :

- `get` si c'est un accesseur en lecture ("getteur") ; ex : `String getName();`
- `is` si c'est un accesseur en lecture d'une propriété booléenne ;
  ex : `boolean isInitialized();`
- `set` si c'est un accesseur en écriture ("setteur") ;
  ex : `void getName(String name);`
- tout autre verbe, à l'indicatif, si la méthode retourne un booléen (méthode prédicat) ;
- à l'impératif [1], si la méthode sert à effectuer une action avec **effet de bord** [2]
  `Arrays.sort(myArray);`
- au participe passé si la méthode retourne une version transformée de l'objet, sans modifier l'objet (ex : `list.sorted()`).

1. ou infinitif sans le "*to*", ce qui revient au même en Anglais
2. c.-à-d. mutation de l'état ou effet physique tel qu'un affichage ; cela s'oppose à fonction pure qui effectue juste un calcul et en retourne le résultat

Nommer les entités

Concision versus information

- Pour tout idenficateur, il faut trouver le bon compromis entre information (plus long) et facilité à l'écrire (plus court).
- Typiquement, plus l'usage est fréquent et local, plus le nom est court :
  ex. : variables de boucle
  ```
  for (int idx = 0; idx < anArray.length; idx++){ ... }
  ```
- plus l'usage est lointain de la déclaration, plus le nom doit être informatif
  (sont particulièrement concernés : classes, membres publics... mais aussi les paramètres des méthodes !)
  ex. : paramètres de constructeur
  ```
  Rectangle(double centerX, double centerY, double width, double length){ ... }
  ```

Toute personne lisant le programme s'attend à une telle stratégie → ne pas l'appliquer peut l'induire en erreur.

Nombre de caractères par ligne

- On limite le nombre de caractères par ligne de code. Raisons :
  - certains programmeurs préfèrent désactiver le retour à la ligne automatique [1] ;
  - même la coupure automatique ne se fait pas forcément au meilleur endroit ;
  - longues lignes illisibles pour le cerveau humain (même si entièrement affichées) ;
  - certains programmeurs aiment pouvoir afficher 2 fenêtres côte à côte.
- Limite traditionnelle : 70 caractères/ligne (les vieux terminaux ont 80 colonnes [2]).
  De nos jours (écrans larges, haute résolution), 100-120 est plus raisonnable [3].
- Arguments contre des lignes trop petites :
  - découpage trop élémentaire rendant illisible l'intention globale du programme ;
  - incitation à utiliser des identifiants plus courts pour pouvoir écrire ce qu'on veut en une ligne ($\rightarrow$ identifiants peu informatifs, mauvaise pratique).

1. De plus, historiquement, les éditeurs de texte n'avaient pas le retour à la ligne automatique.
2. Et d'où vient ce nombre 80 ? C'est le nombre du de colonnes dans le standard de cartes perforées d'IBM inventé en... 1928 ! Et pourquoi ce choix en 1928 ? Parce que les machines à écrire avaient souvent 80 colonnes... bref c'est de l'histoire très ancienne !
3. Selon moi, mais attention, c'est un sujet de débat houleux !

Indentation

- **Indenter** = mettre du blanc en tête de ligne pour souligner la structure du programme. Ce blanc est constitué d'un certain nombre d'**indentations**.
- En Java, typiquement, 1 indentation = 4 espaces (ou 1 tabulation).
- Le nombre d'indentations est égal à la profondeur syntaxique du début de la ligne $\simeq$ nombre de paires de symboles [1] ouvertes mais pas encore fermées. [2]
- Tout éditeur raisonnablement évolué sait indenter automatiquement (règles paramétrables dans l'éditeur). Pensez à demander régulièrement l'indentation automatique, afin de vérifier qu'il n'y a pas d'erreur de structure !

Exemple :

```
voici un exemple (
    qui n'est pas du Java;
    mais suit ses "conventions
        d'indentation"
)
```

1. Parenthèses, crochets, accolades, guillemets, chevrons, ...
2. Pas seulement : les règles de priorité des opérations créent aussi de la profondeur syntaxique.

Où couper les lignes

- On essaye de privilégier les retours à la ligne en des points du programme “hauts” dans l’arbre syntaxique ($\rightarrow$ minimise la taille de l’indentation).
  P. ex., dans “$(x + 2) * (3 - 9/2)$”, on préfèrera couper à côté de “$*$” $\rightarrow$

```
(x + 2)
* (3 - 9 / 2)
```

- Parfois difficile à concilier avec la limite de caractères par ligne $\rightarrow$ compromis nécessaires.
- $\rightarrow$ pour le lieu de coupure et le style d’indentation, essayez juste d’être raisonnable et consistent. Dans le cadre d’un projet en équipe, se référer aux directives du projet.

Taille des classes

Quelle est la bonne taille pour une classe ?

- Déjà, plusieurs critères de taille : nombre de lignes, nombre de méthodes, ...
- Le découpage en classes est avant tout guidé par l'abstraction objet retenue pour modéliser le problème qu'on veut résoudre.
- En pratique, une classe trop longue est désagréable à utiliser. Ce désagrément traduit souvent une décomposition insuffisante de l'abstraction. [1]
- Conseil : se fixer une limite de taille et décider, au cas par cas, si et comment il faut "réparer" les classes qui dépassent la limite (cela incite à améliorer l'aspect objet du programme).
- En général, pour un projet en équipe, suivre les directives du projet.

1. Le « S » de « SOLID » : *single reponsibility principle*/principe de responsabilité unique.

Taille des méthodes

- Pour une méthode, la taille est le nombre de lignes.
- Principe de responsabilité unique [1] : une méthode est censée effectuer une tâche précise et compréhensible.
  $\rightarrow$ Un excès de lignes
  - nuit à la compréhension ;
  - peut traduire le fait que la méthode effectue en réalité plusieurs tâches probablement séparables.
- Quelle est la bonne longueur ?
  - Mon critère [2] : on ne peut pas bien comprendre une méthode si on ne peut pas la parcourir en un simple coup d'œil
    $\rightarrow$ faire en sorte qu'elle tienne en un écran ($\sim$ 30-40 lignes max.)
  - En général, suivre les directives du projet.

1. Oui, là aussi !
2. qui n'engage que moi !

Nombre de paramètres des méthodes

Autre critère : le nombre de paramètres.

Trop de paramètres (>4) implique :

- Une signature longue et illisible.
- Une utilisation difficile (“ah mais ce paramètre là, il était en 5e ou en 6e position, déjà ?”)

Il est souvent possible de réduire le nombre de paramètres

- en utilisant la surcharge,
- ou bien en séparant la méthode en plusieurs méthodes plus petites (en décomposant la tâche effectuée),
- ou bien en passant des objets composites en paramètre

  ex : un `Point p` au lieu de `int x, int y`.

  Voir aussi : patron “monteur” (le constructeur prend pour seul paramètre une instance du `Builder`).

Et ainsi de suite

- Pour chaque composant contenant des sous-composants, la question “combien de sous-composants ?” se pose.
- “Combien de packages dans un projet (ou module) ?”
  “Combien de classes dans un package ?”
- Dans tous les cas essayez d’être raisonnable et homogène/consistent (avec vous-même... et avec l’organisation dans laquelle vous travaillez).

Commentaires

Plusieurs sortes de commentaires (1)

- En ligne :

```
int length; // length of this or that
```

Pratique pour un commentaire très court tenant sur une seule ligne (ou ce qu'il en reste...)

- en bloc :

```
/*
 * Un commentaire un peu plus long.
 * Les "*" intermédiaires ne sont pas obligatoires, mais Eclipse
 * les ajoute automatiquement pour le style. Laissez-les !
 */
```

À utiliser quand vous avez besoin d'écrire des explications un peu longues, mais que vous ne souhaitez pas voir apparaître dans la documentation à proprement parler (la JavaDoc).

Commentaires

Plusieurs sortes de commentaires (2)

- en bloc JavaDoc :

```
/**
 * Returns an expression equivalent to current expression, in which
 * every occurrence of unknown var was substituted by the expression
 * specified by parameter by.
 *
 * @param var   variable that should be substituted in this expression
 * @param by    expression by which the variable should be substituted
 * @return      the transformed expression
 */
Expression subst(UnknownExpr var, Expression by);
```

Commentaires

La JavaDoc

À propos de la JavaDoc :

- Les commentaires au format JavaDoc sont compilables en documentation au format HTML (dans Eclipse : menu “Project”, “Generate JavaDoc...”).
- Pour toute déclaration de type (classe, interface, enum...) ou de membre (attribut, constructeur, méthode), un squelette de documentation au bon format (avec les bonnes balises) peut être généré avec la combinaison `Alt`+`Shift`+`J` (toujours dans Eclipse).
- Il est **indispensable** de documenter tout ce qui est public.
- Il est **fortement recommandé** de documenter tout ce qui n’est pas privé (car utilisable par d’autres programmeurs, qui n’ont pas accès au code source).
- Il est **utile** de documenter ce qui est privé, pour soi-même et les autres membres de l’équipe.

Patrons de conception (1)

ou *design patterns*

- Analogie langage naturel : patron de conception = figure de style
- Ce sont des stratégies standardisées et éprouvées pour arriver à une fin.
  ex : créer des objets, décrire un comportement ou structurer un programme
- Les utiliser permet d'éviter les erreurs les plus courantes (pour peu qu'on utilise le bon patron !) et de rendre ses intentions plus claires pour les autres programmeurs qui connaissent les patrons employés.
- Connaître les noms des patrons permet d'en discuter avec d'autres programmeurs. [1]

1. De la même façon qu'apprendre les figures de style en cours de Français, permet de discuter avec d'autres personnes de la structure d'un texte...

Patrons de conception (2)

ou *design patterns*

- Quelques exemples dans le cours : décorateur, délégation, observateur/observable, monteur.
- Patrons les plus connus décrits dans le livre du “Gang of Four” (GoF) [1]
- Les patrons ne sont pas les mêmes d’un langage de programmation à l’autre :
  - les patrons implémentables dépendent de ce que la syntaxe permet
  - les patrons utiles dépendent aussi de ce que la syntaxe permet :
    quand un nouveau langage est créé, sa syntaxe permet de traiter simplement des situations qui autrefois nécessitaient l’usage d’un patron (moins simple).
    Plusieurs concepts aujourd’hui fondamentaux (comme les « classes », comme les énumérations, ...) ont pu apparaître comme cela.

1. E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns : Elements of Reusable Object-oriented Software*, 1995, Addison-Wesley Longman Publishing Co., Inc.