Génie Logiciel
et
Patrons de conception

« La perfection n'est atteinte,
non pas lorsqu'il n'y a plus rien à ajouter,
mais lorsqu'il n'y a plus rien à enlever. »
Antoine de Saint-Exupéry

Objectif du cours

- Vers une conception des logiciels de qualité professionnelle
- Sensibilisation à « l'excellence technique »
  - Cf. Artisanat du logiciel (*software craftsmanship*)

- **Remarque** : les outils de génération de code donnent de moins en moins d'importance au langage de programmation mais de plus en plus d'importance à la conception

Organisation du cours

- Charge de travail
  - 9 h CM → une grande partie du cours est faite au tableau (modélisation UML)
  - 8 h TD
  - 16 h TP (projet)
- Discipline
  - Pas de contrôle de présence en CM et TD
    - En contre-partie respect de l'enseignant !
  - Par contre, sanction des absences et des retards en TP
- Examen
  - Le cahier de TD et les Coding Dojo tiennent lieu d'annales d'examen
  - Document autorisé : une feuille **A4,** recto/verso, **manuscrite**, 2D, libre

Résumé du cours de première année

- Génie logiciel
  - Un paradigme de conception : **Conception Orientée Objet**
  - Un formalisme de modélisation : **UML**
    (diagrammes cas d'utilisation, classe, paquet, séquence, état-transition)
  - Langage de programmation orientée objet : **Java**
    (et donc C++, C#, Python, Dart…)
  - Une méthode de gestion de projet : **Agilité**
- Agilité : un ensemble de pratiques régulées
  - Gestion de projet basée sur un cycle de développement itératif et incrémental
  - Code propre (*clean code*)
  - Auto-documentation du code (architecture, algorithmique)
  - Développement dirigé par les tests (*TDD*)
  - Programmation par pair (*pair programming*)
  - DevOps (*git*, *gitlab CI/CD*)

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

01
Chapitre

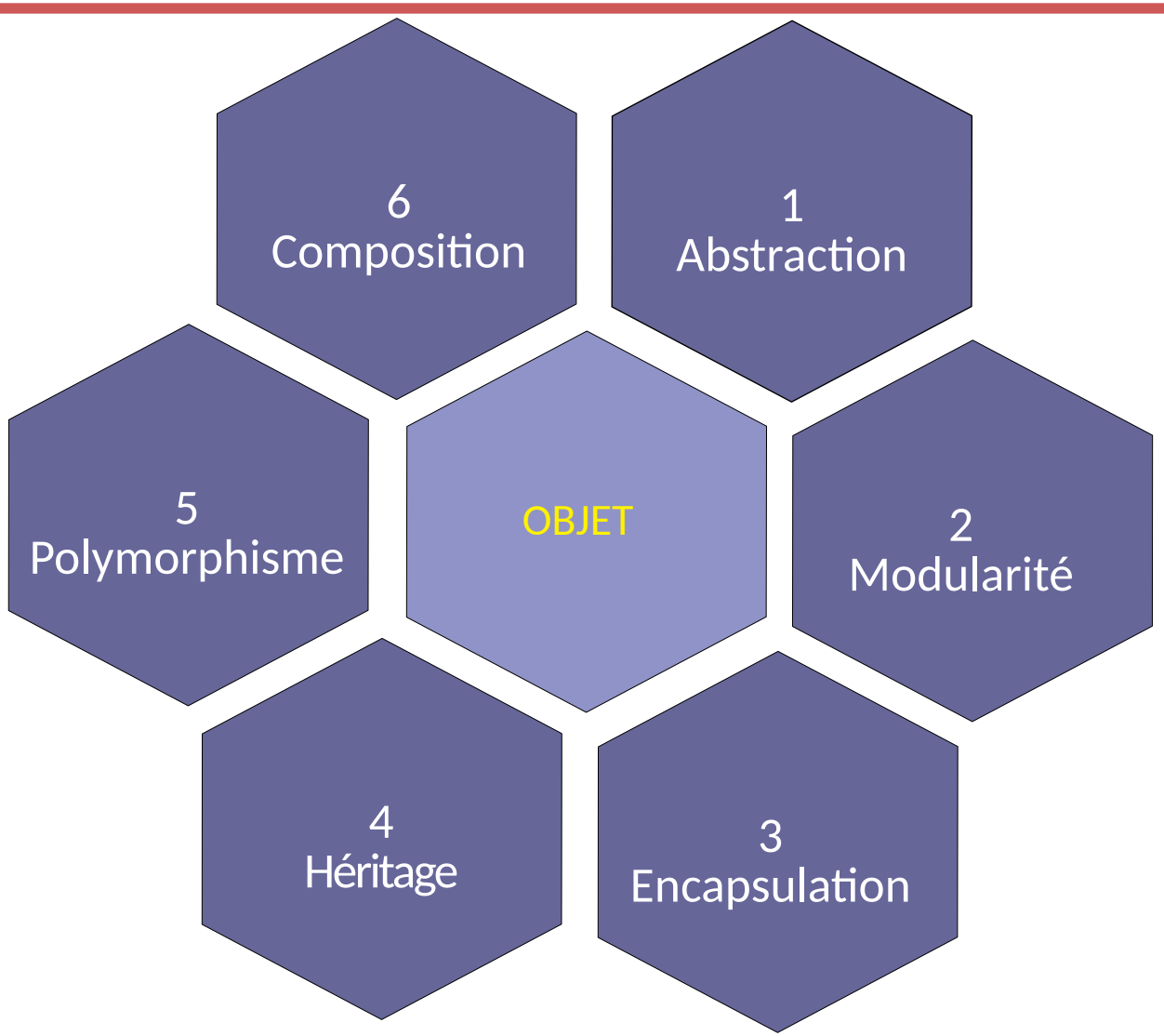
Limites du paradigme objet pour la conception

2I1AC3 : Génie logiciel et Patrons de conception

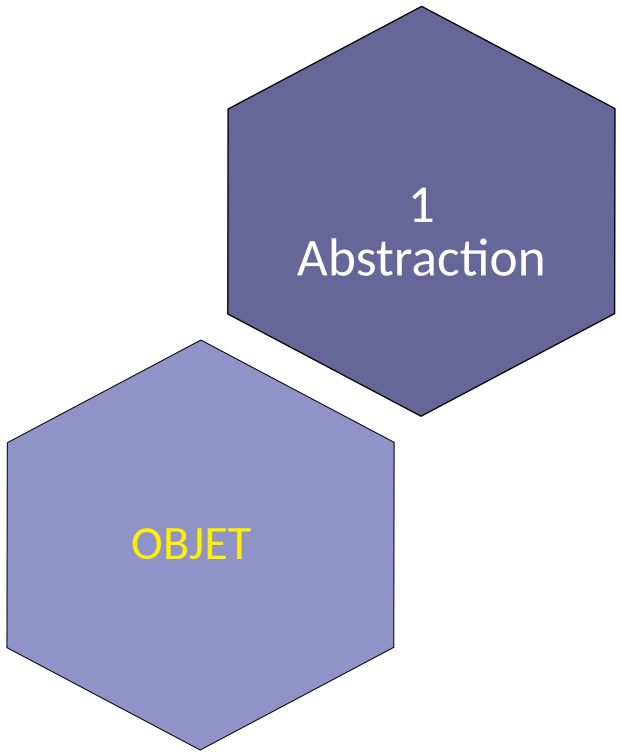
Régis Clouard, ENSICAEN - GREYC

« La perfection n'est atteinte,
non pas lorsqu'il n'y a plus rien à ajouter,
mais lorsqu'il n'y a plus rien à enlever. »
Antoine de Saint-Exupéry

Rappel : les six piliers du paradigme objet

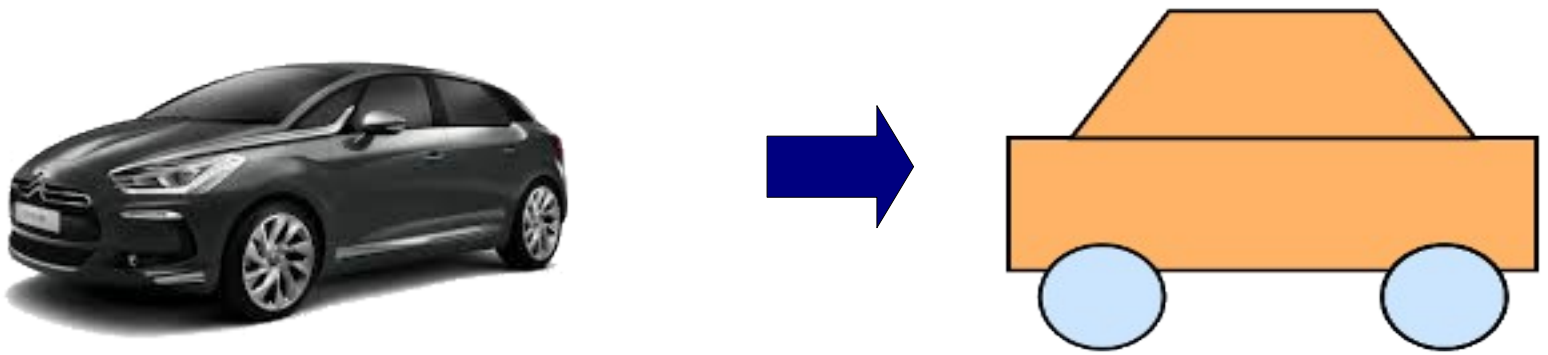


Les six piliers de la conception objet

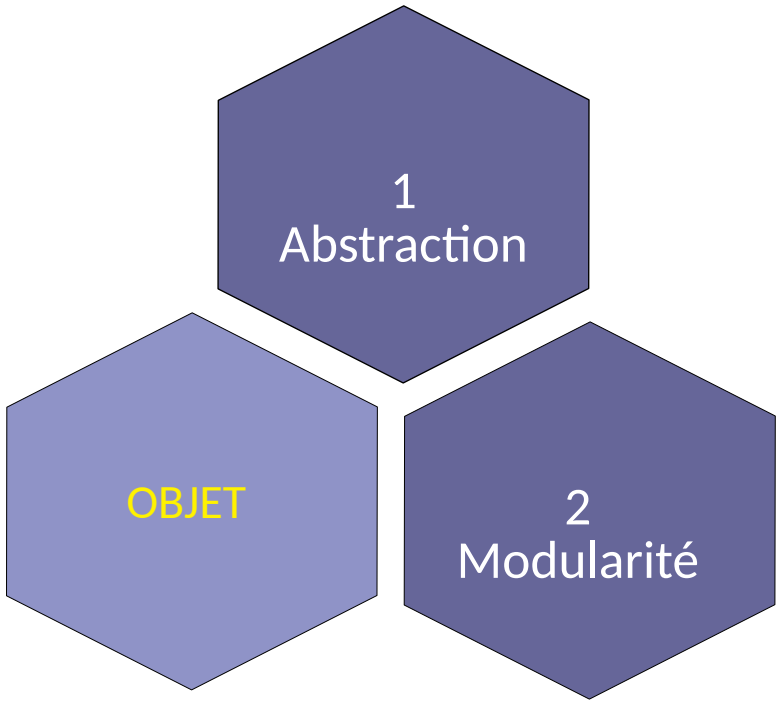


Abstraction

- Procédé de construction d'un modèle simplifié d'un système réel complexe tout en conservant ses fonctions essentielles



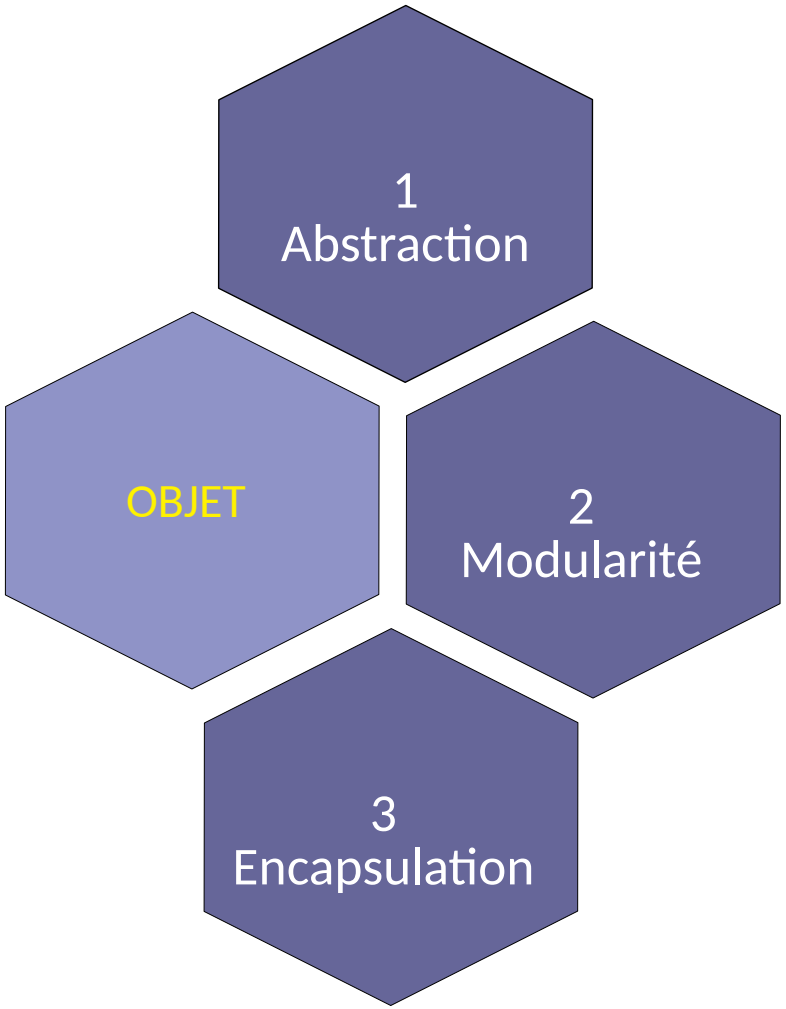
Les six piliers du paradigme objet



Modularité

- Procédé de construction d'un système complexe par assemblage de modules compacts plus simples
  - Module = Fonction, Classe, Paquet, Composant, Nœud
    → Propose une approche cartésienne de la résolution de problème

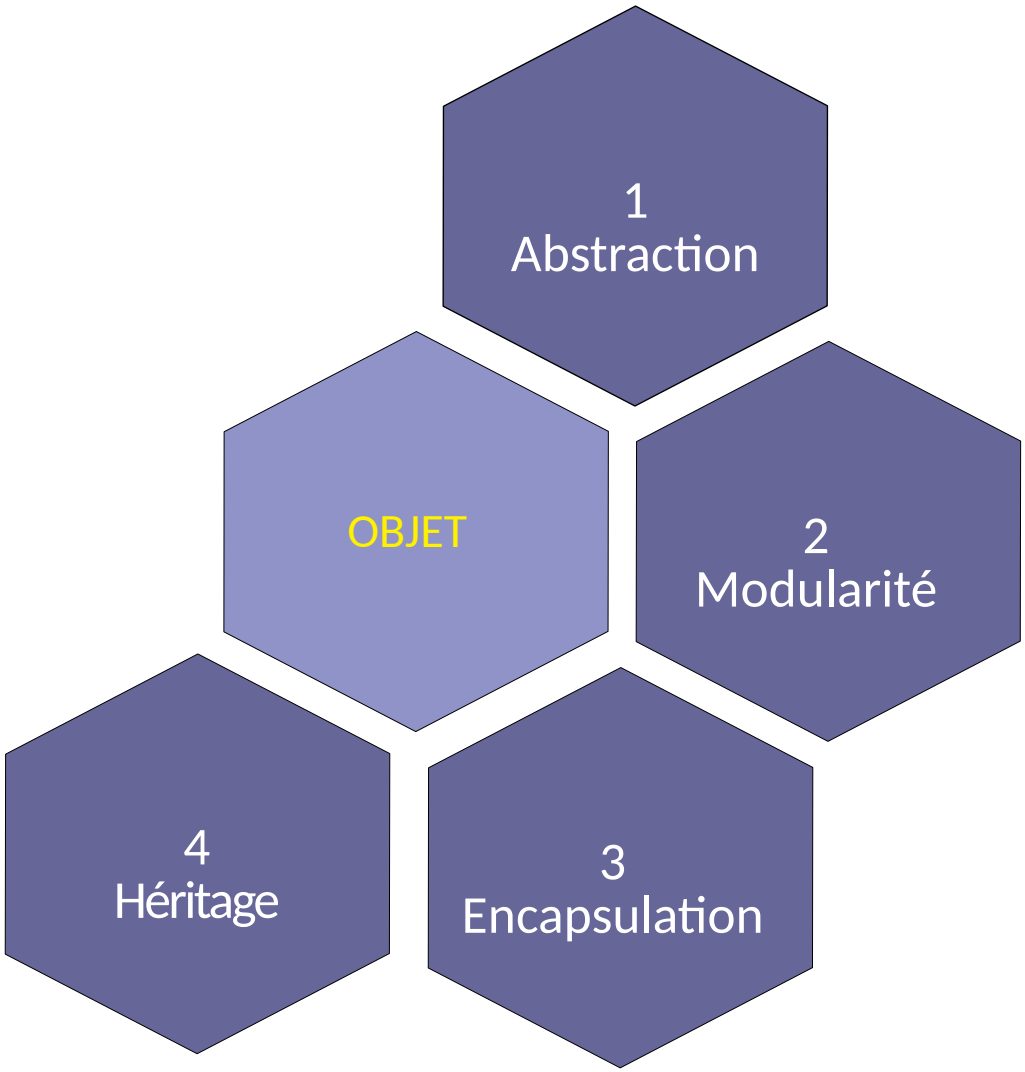
Les six piliers du paradigme objet



Encapsulation

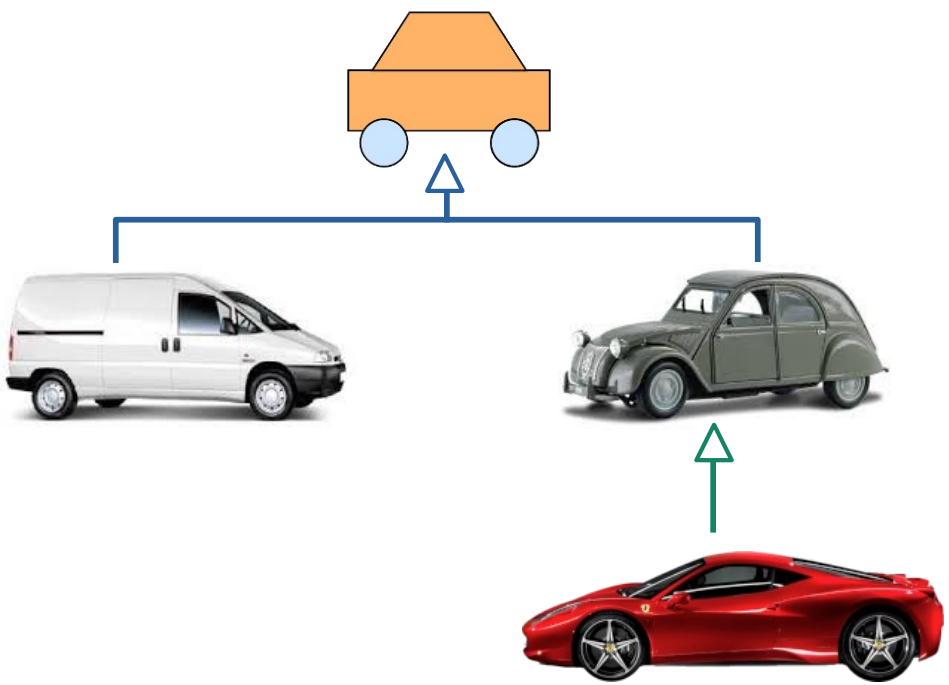
- Principe de séparation de l'interface contractuelle d'une abstraction avec son implémentation
  - → Masquer les détail de l'implémentation
  - → Permet de repousser l'implémentation le plus tard possible

Les six piliers du paradigme objet

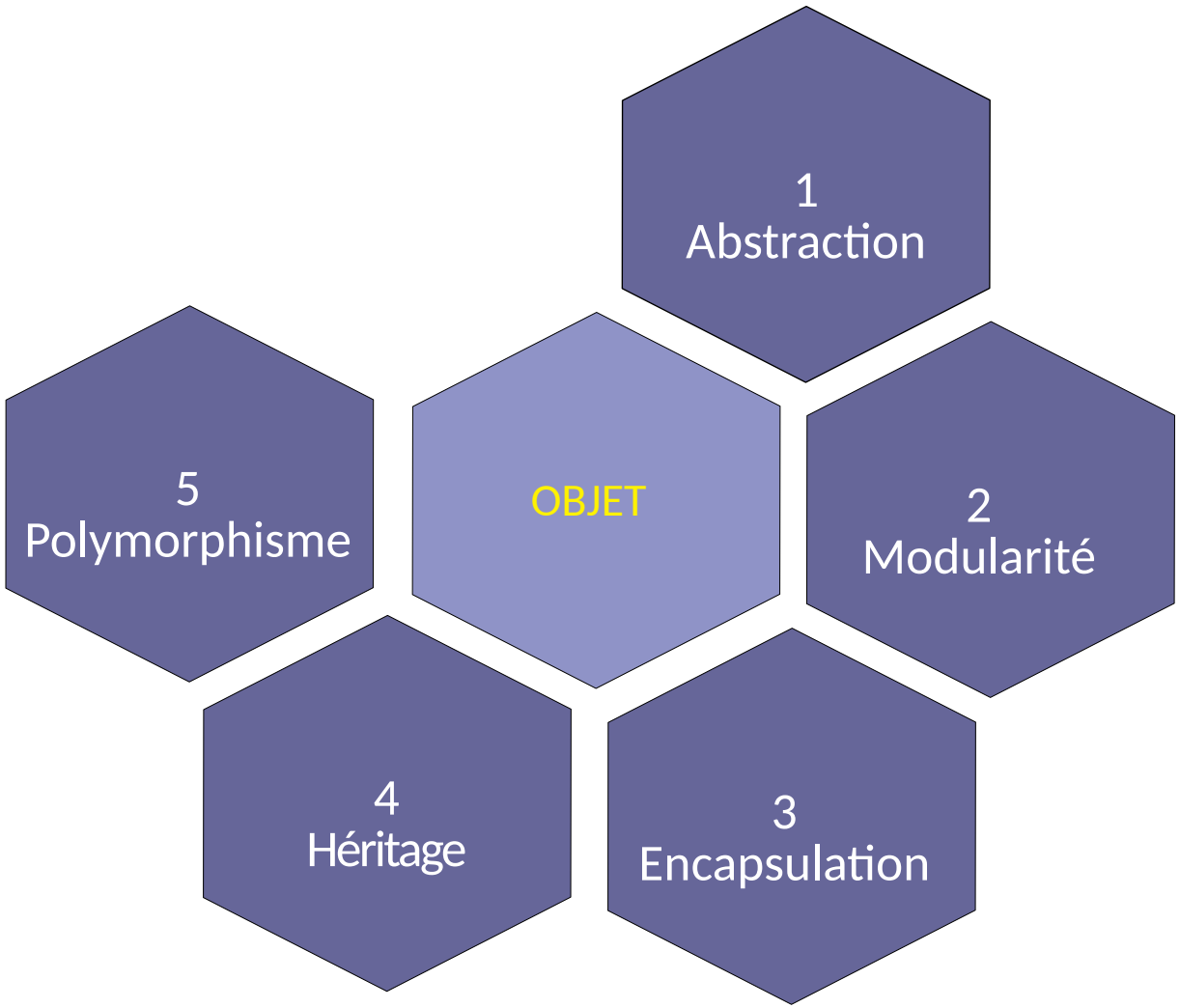


Héritage

- Procédé de **généralisation/spécialisation** par lequel une classe est obtenue à partir d'autres classes
  - La **généralisation** est le processus de factorisation des éléments communs à plusieurs classes dans une nouvelle classe
  - La **spécialisation** est le processus de création d'une nouvelle classe par extension de l'implémentation d'une classe existante

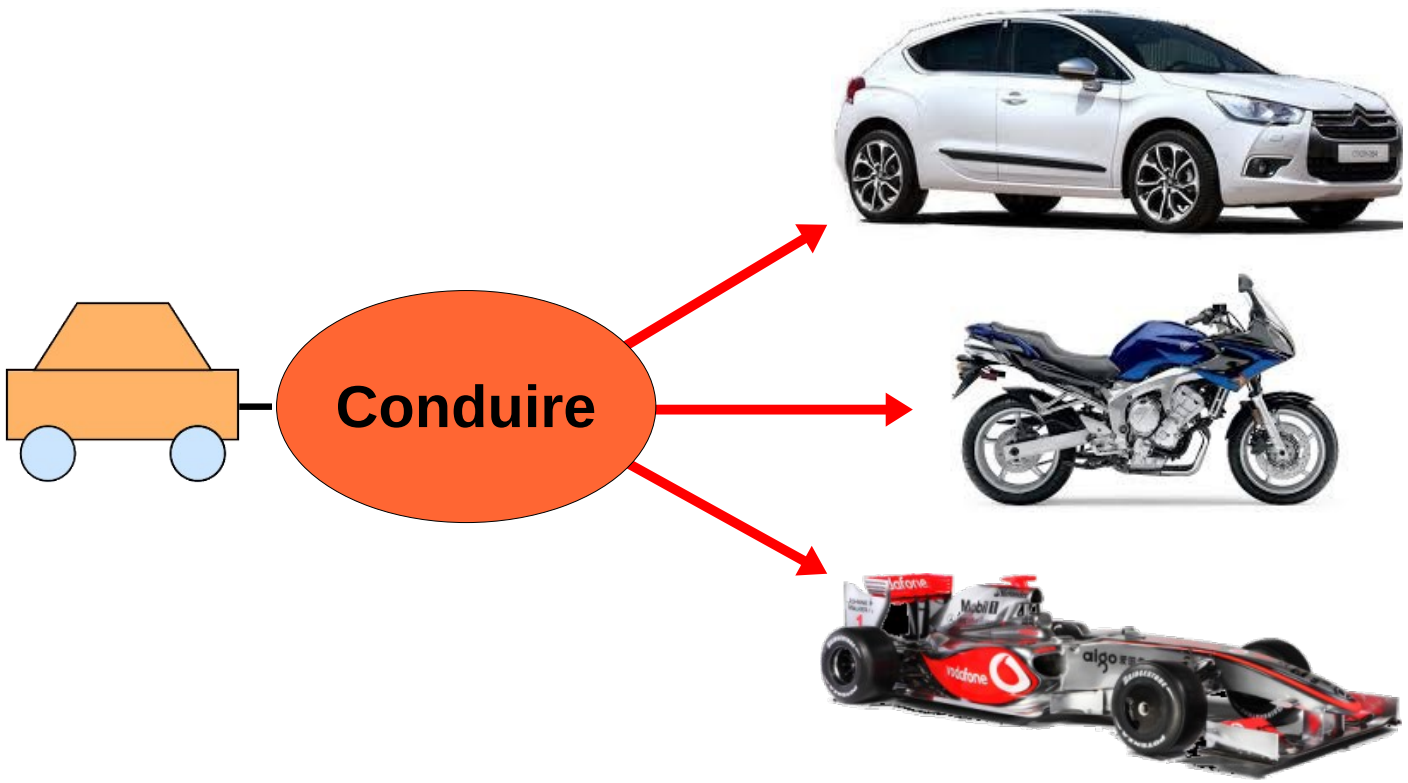


Les six piliers du paradigme objet

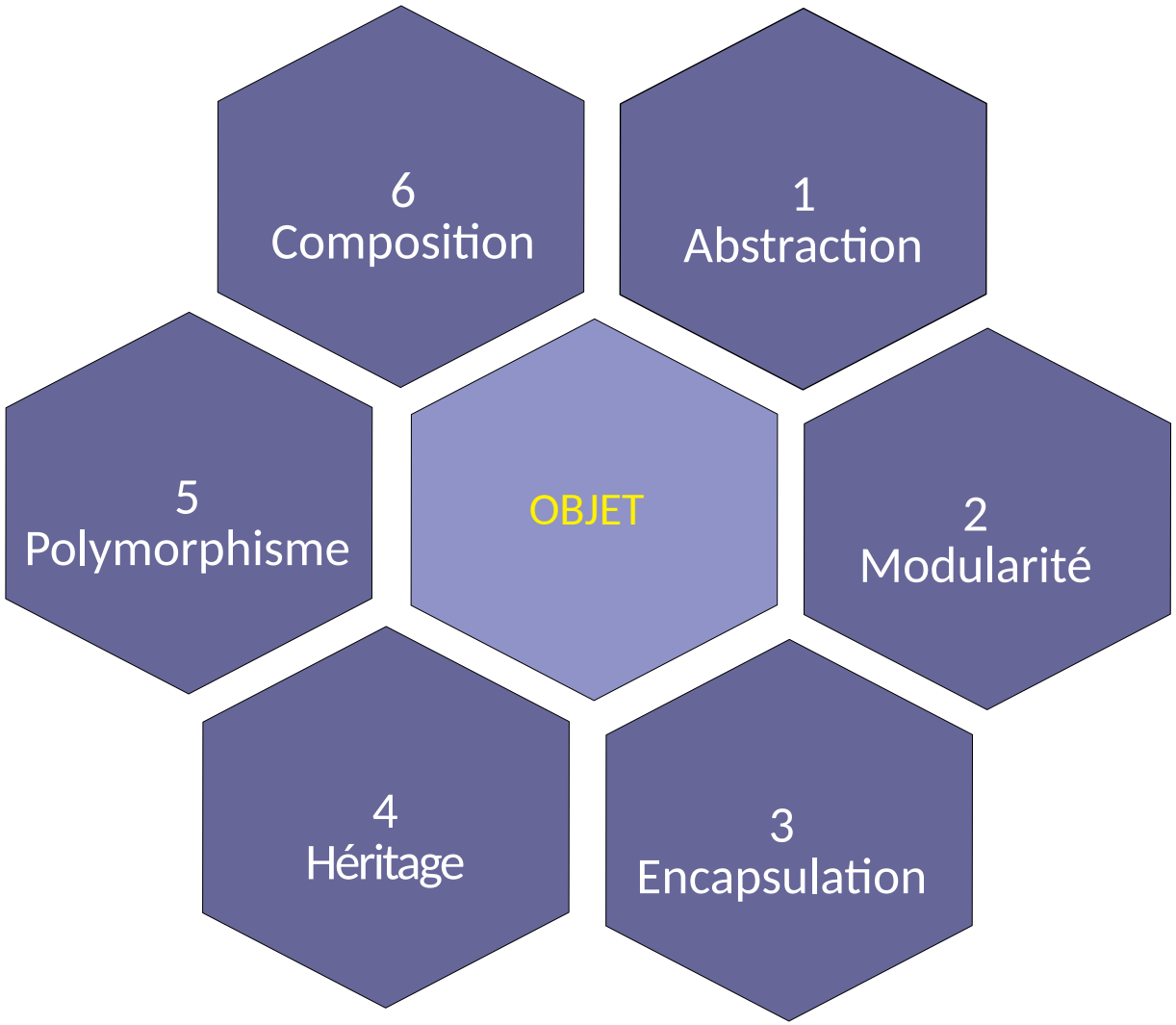


Polymorphisme

- Capacité des objets appartenant à des classes différentes à répondre aux appels de **méthodes** de même nom, chacune selon le comportement spécifique de sa classe

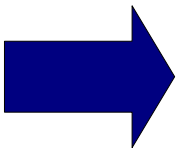


Les six piliers du paradigme objet



Composition

- Procédé de **réutilisation** par lequel une nouvelle fonctionnalité est obtenue en combinant les services de plusieurs objets



+

Promesses du paradigme objet

- Le paradigme de conception orientée objet (COO) a permis de faire des progrès énormes en termes de taille et de réussite de projet.
- Apports :
  - **Développabilité** : confort avec lequel le logiciel peut être développé
    - Grâce à l'abstraction et la modularité
  - **Extensibilité** : faculté d'étendre les fonctionnalités d'un logiciel sans compromettre son intégrité et sa fiabilité
    - Grâce à la modularité, l'héritage et le polymorphisme
  - **Maintenabilité** : facilité avec laquelle on peut corriger des erreurs ou des manques
    - Grâce à l'encapsulation et la composition
  - **Réutilisabilité** : aptitude d'un logiciel à être réutilisé en tout ou partie pour de nouvelles applications
    - Grâce à l'héritage et la composition

Définition de la réutilisabilité

- Réutilisabilité
  - La recopie de code n’est pas de la réutilisation. Le code copié devient du code normal
  - Un code réutilisable s’entend comme une archive compilée (eg, .jar en Java, module en Python, package en Dart, bundle en Php…)
  - Un code a la qualité de la réutilisabilité si et seulement si le réutilisateur n'a pas besoin de regarder le code pour le réutiliser (l’interface publique doit suffire)

Périls de l'approche objet

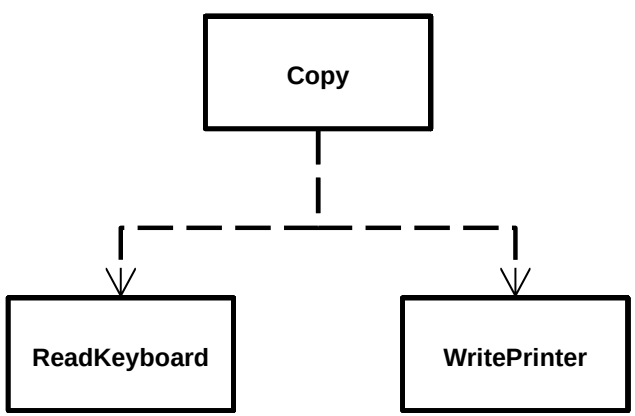
- Le paradigme et le langage seuls ne suffisent pas à assurer ces promesses.
  - Par exemple : l'encapsulation est très souvent mise à mal par l'utilisation d'attributs non privés ou d'accesseurs/mutateurs (*getter/setter*)
- Périls d'une mauvaise conception objet
  - **Rigidité** : logiciel difficile à faire évoluer
  - **Fragilité** : logiciel difficile à maîtriser
  - **Immobilité** : logiciel difficile à réutiliser
- Un logiciel mal conçu tend inévitablement vers le pourrissement (*software rot*)
  - Le pourrissement fait que les développeurs craignent de plus en plus de modifier le logiciel. Mais un logiciel qui n'évolue pas meurt.

Exemple du pourrissement d'une conception

- Besoin client : Une API qui copie les caractères lus au clavier vers l'imprimante

```
class Copy {
    void copy() {
        while ( (int ch = ReadKeyboard.getChar()) != EOF) {
            WritePrinter.print(ch);
        }
    }
}
```

```
Copy().copy();
```



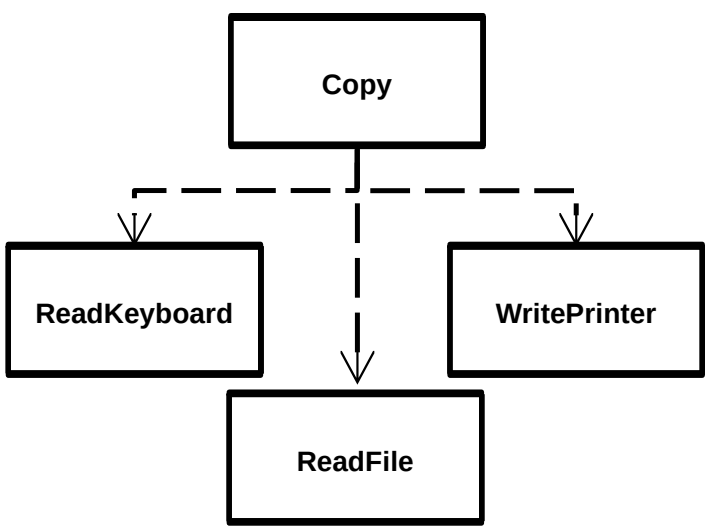
- Une vrai succès !
  - Cela veut dire que l'on ne peut plus changer la signature de la méthode

Exemple du pourrissement d'une conception

- Nouveau besoin client : lecture à partir d'un fichier

```
class Copy {
    static bool tapeReader = false; // remember to clear
    void copy() {
        while ((int ch = tapeReader ? ReadFile.getChar()
                                    : ReadKeyboard.getChar()) != EOF) {
            WritePrinter.print(ch);
        }
    }
}
```

```
Copy.tapeReader = true;
Copy().copy();
Copy.tapeReader = false; // Ne pas oublier
```

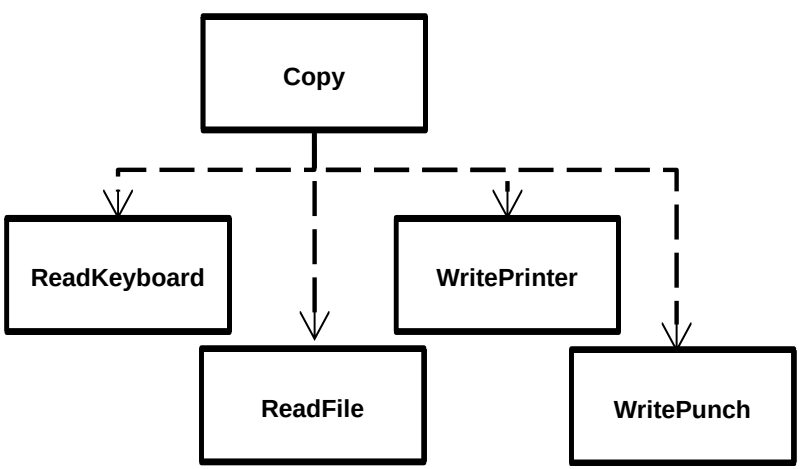


Exemple du pourrissement d'une conception

- Troisième demande client : Imprimer sur un ruban perforé pour aveugle

```
class Copy {
    static bool tapeReader = false;     // TODO remember to clear
    static bool tapePunch = false;      // TODO remember to clear
    void copy() {
        while((int ch = tapeReader ? ReadFile.getChar()
                                   : ReadKeyboard()) != EOF) {
            tapePunch ? WritePunch.print(ch) : WritePrinter.print(ch);
        }
    }
}
```

```
Copy.tapePunch = true;
Copy().copy();
Copy.tapePunch = false; // Ne pas oublier
```

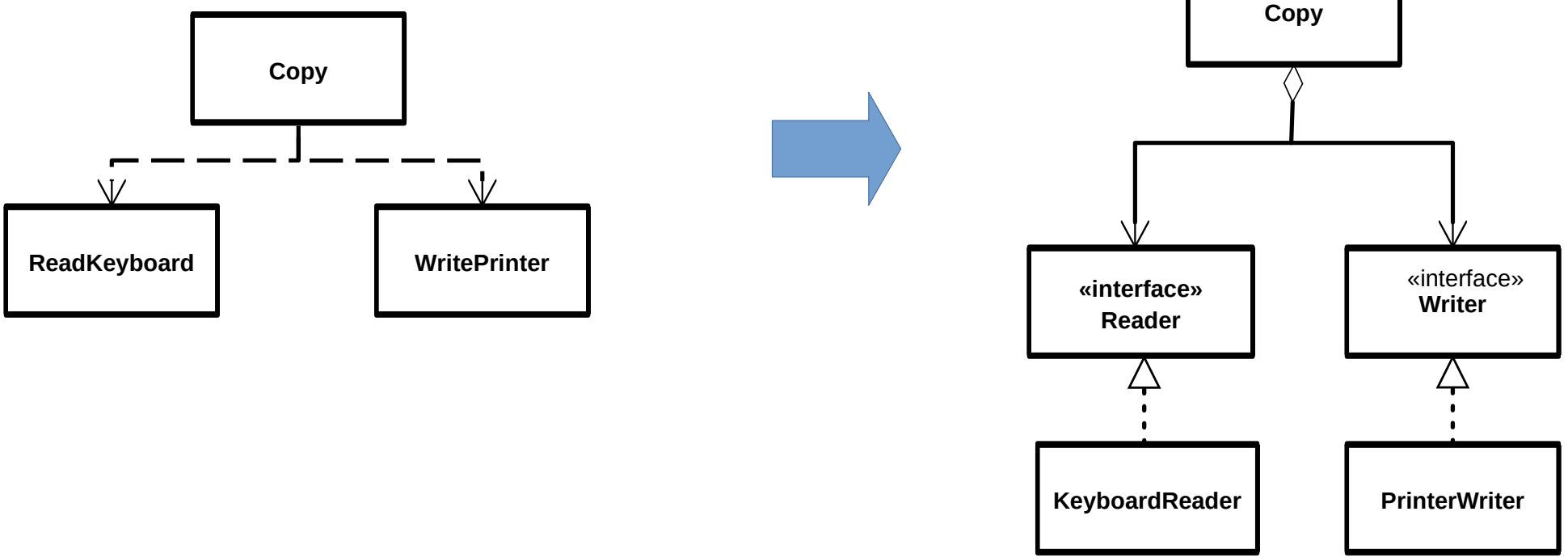


Sentez vous le relent de pourrissement (*rotten code*) ?

- Rigidité (logiciel difficile à faire évoluer)
  - Changements à plusieurs endroits sans liens explicites (plusieurs lignes `Copy.tapePunch=true;` à plusieurs endroits)
  - Parallélisations impossibles à cause des variables globales
- Fragilité (logiciel difficile à maîtriser)
  - Si on oublie de remettre le flag, le programme ne fonctionne correctement plus à d'autres endroits et rien n'indique de positionner ce flag (uniquement un commentaire)
  - Les tests sont rendus plus difficiles parce que les variables globales agissent par effets de bords (pas dans la signature de méthode à tester)
- Immobilité (logiciel difficile à faire évoluer)
  - Impossible de réutiliser Copy sans embarquer les autres classes mêmes si vous ne voulez pas les utiliser (`ReadKeyboard`, `WritePrinter`, `WritePunch`)

Refonte de la conception (*refactoring*)

- La 1re version est la même que précédemment (principe YAGNI)
- Mais la 2e conduit à une refonte, tout en étant rétrocompatible

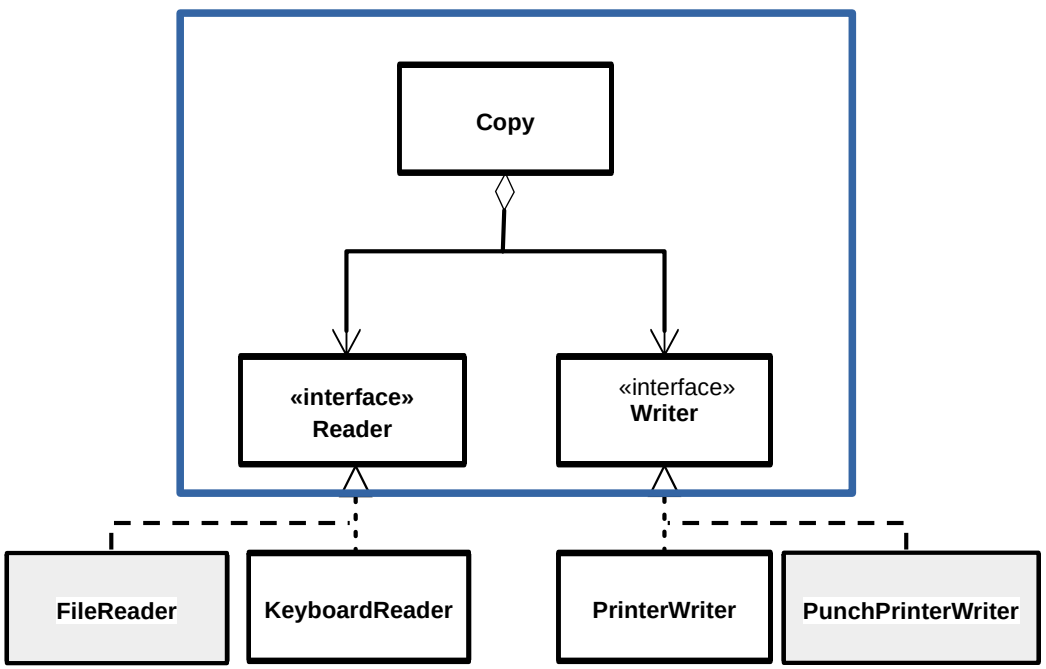


Code résultant


```
public interface Reader {
    char read();
}
public interface Writer {
    void write(char c);
}
public class Copy {
    private Reader _reader;
    private Writer _writer;
    Copy() { // rétrocompatible
        _reader = new KeyboardReader();
        _writer = new PrinterWriter();
    }
    Copy(Reader r, Writer w) { // Nouveau
        _reader = r;
        _writer = w;
    }
    void copy() { // rétrocompatible
        while( (int c = _reader.read()) != EOF ) {
            _writer.write(c);
        }
    }
}
```

Bilan de nouvelle conception

- Qu'avons nous gagner ?
  - ~~Rigidité~~ : Plus de variable locale (parallélisable, testable)
  - ~~Fragilité~~ : Vérifiable par le compilateur (plus de condition d'utilisation en commentaire)
  - ~~Immobilité~~ : ajout d'une autre entrée ou d'une autre sortie
  - Réutilisabilité : La classe Copy est réutilisable seule (avec ses 2 interfaces)
  - Rétrocompatibilité



Critères de qualité d'une conception

- Deux critères absolus de qualité :
  1) Cohésion (*cohesion*)
  2) Couplage (*coupling*)
- Ils portent sur les modules
- Malheureusement, ces deux critères sont difficiles à mesurer automatiquement
  - Il existe toutefois des métriques qui permettent d'apprécier ces critères, mais elles ne sont que des indicateurs pas des mesures absolues pour ces critères
- Rappel : extensions IntelliJ pour mesurer la qualité d'une conception
  - **SonarQube** : une référence dans le domaine de l'analyse de code
  - **PMD** : métriques pour la cohésion et le couplage

Critère 1. Cohésion (*Cohesion*)

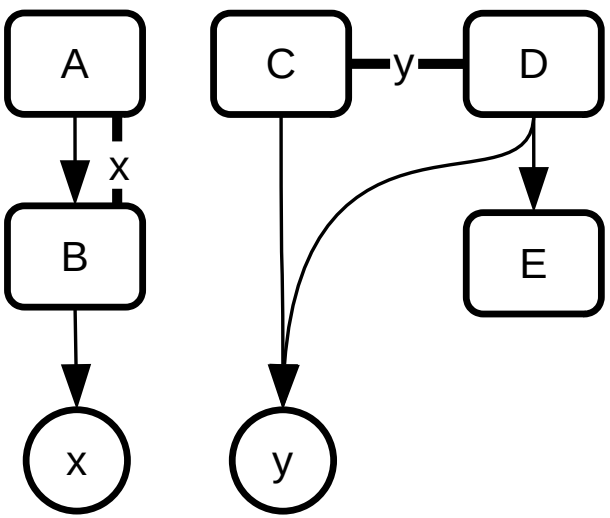
- Caractérise l'étendue de la responsabilité d'un module

- Questions associées
  - Quel est le but du module ?
  - Fait-il une ou plusieurs choses ?
- Il faut maximiser le degré de cohésion dans une conception

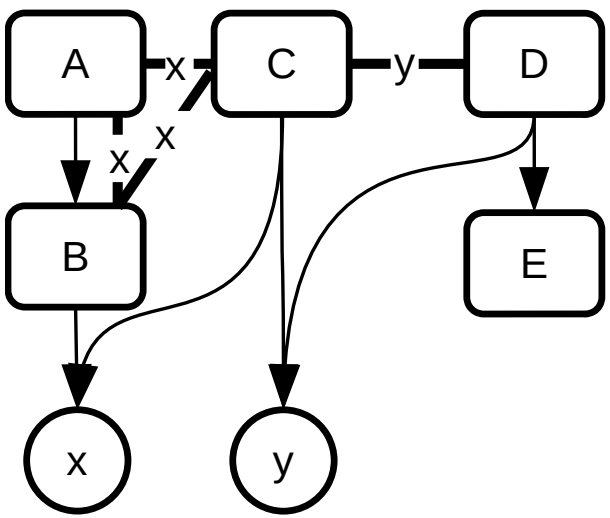
- Indice
  - La liste des attributs est un bon indicateur du degré de cohésion

Exemple d'une métrique

- Tight Class Cohesion (TCC)
  - Compter le nombre relatif de paires de méthodes qui accèdent directement aux mêmes attributs de la classe
    - Permet de repérer les groupes de méthodes indépendantes
  - Mesure : TCC = NDC / NP (*erreur de cohésion* : TCC < 1/3)
    - NDC : le nombre de paires de méthodes directement liées
    - NP : le nombre total de paires de méthodes



Faible cohésion. TCC = 2/10



Forte cohésion. TCC = 4/10

Critère 2. Couplage (*coupling*)

- Caractérise la force d'interaction entre les modules

- Questions associées :
  - Comment les modules collaborent-ils ensemble ?
  - Qu'ont-ils besoin de connaître les uns des autres ?
- Il faut minimiser le couplage dans une conception

- Indice
  - La liste des importations est un bon indicateur de la force de couplage
    - e.g, nombre d'inclusions (#include en C++, C#), import (Java, Python, Dart)

Exemple d'une métrique

- Access To Foreign Data (ATFD)
  - Compter la proportion de méthodes d’une classe qui accèdent directement ou via des accesseurs/mutateurs à des attributs d'autres classes
  - *Erreur de couplage* : ATFD > 5 %

Objectif du cours

- Développer un esprit critique sur la conception logicielle
  - Savoir évaluer la qualité d'une conception et se rendre compte qu'elle a tendance à pourrir (*software rot*)
  - Savoir produire une conception qui soit :
    - extensible
    - maintenable
    - réutilisable
  - Savoir réutiliser une conception :
    - Connaître et savoir apprécier l'existant
    - Adapter une solution existante
  - Savoir organiser son code en paquets pour favoriser :
    - la développabilité
    - l'extensibilité
    - la maintenance
    - la réutilisation
- Intégrer le fait que le code est malléable et doit sans cesse être remodelé pour garantir la meilleure qualité : cohésion - couplage