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 à la 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)
- Documents
  - https://foad.ensicaen.fr/course/view.php?id=62
- Discipline
  - Pas de contrôle de présence en CM et TD
    - En contre-partie respect de l'enseignant !
  - Contrôle en TP avec 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**

Résumé du cours de première année

supposé acquis !

- Génie logiciel
  - Un paradigme de conception : **Conception Orientée Objet**
  - Un formalisme de modélisation : **UML**
    (diagrammes cas d'utilisation, classe, paquet, séquence, état-transition)
  - 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, algorithme)
      - 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

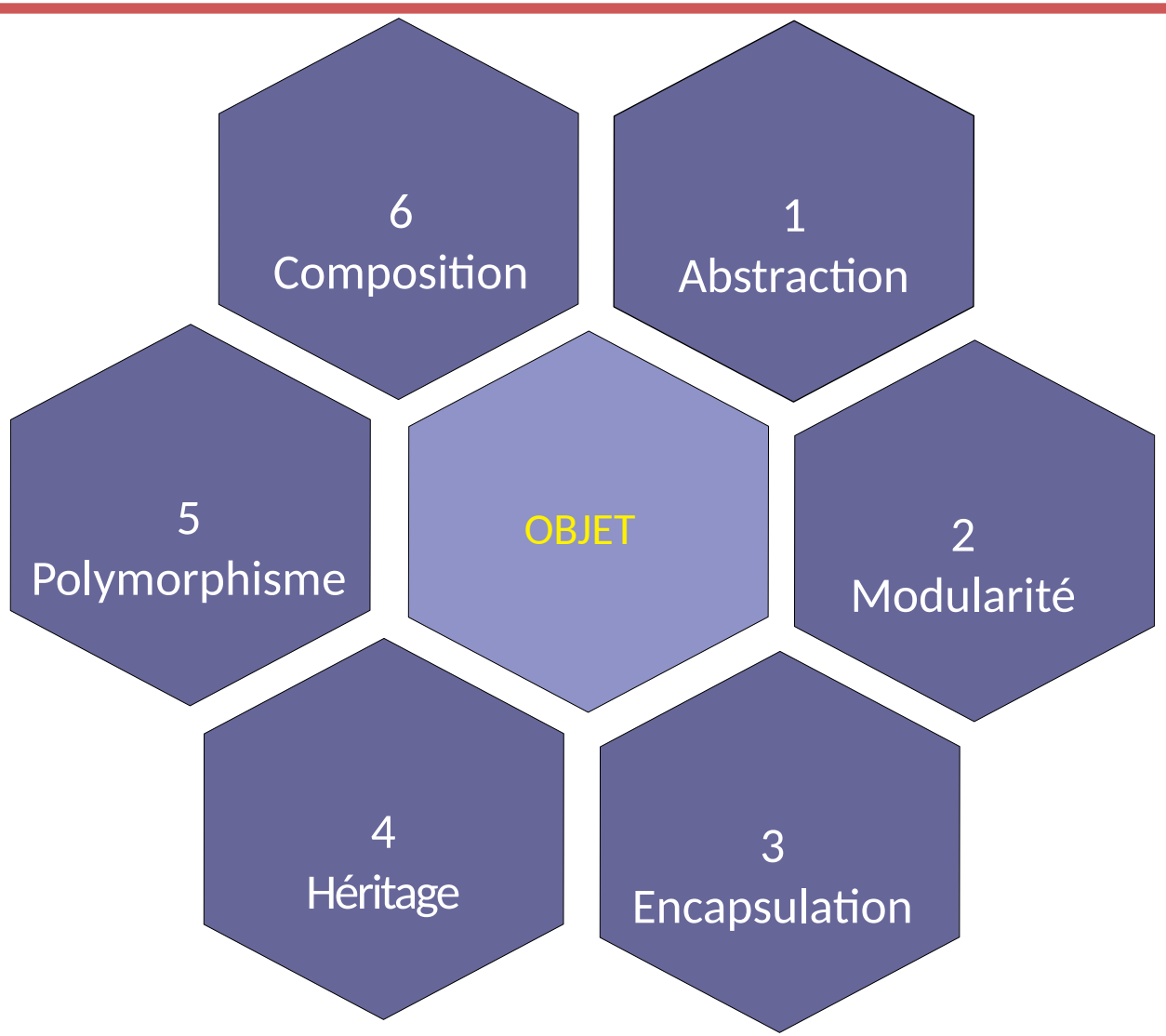
Limites du paradigme objet pour la conception

2I1AC3 : Génie logiciel et Patrons de conception

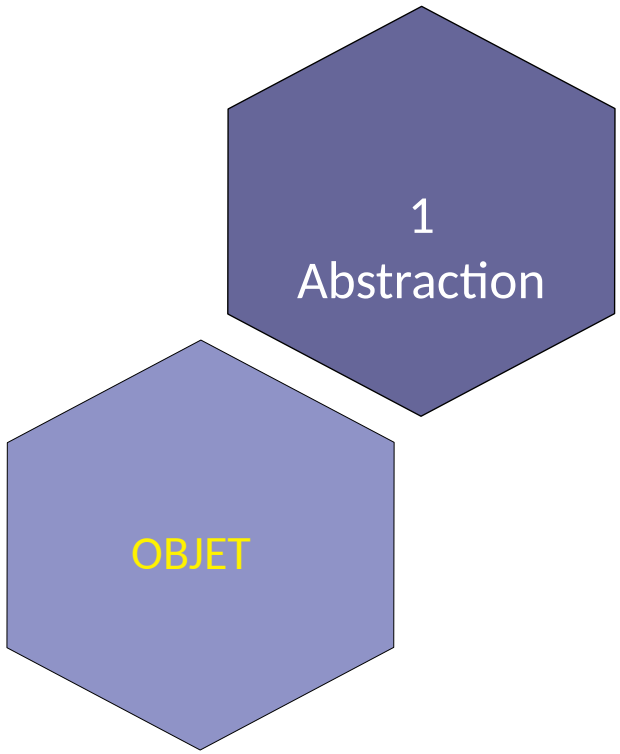
Régis Clouard, ENSICAEN - GREYC

« 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

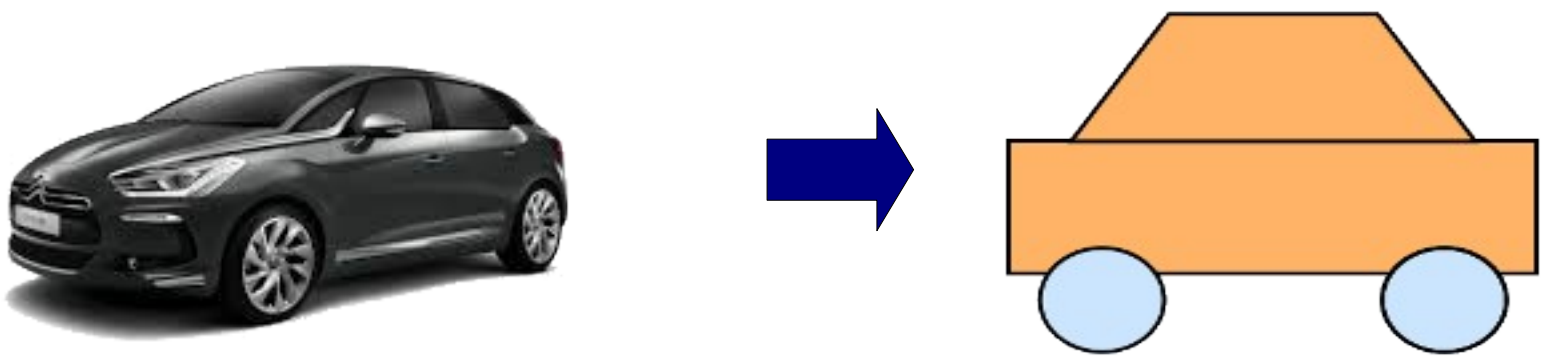


Les six piliers de la conception objet

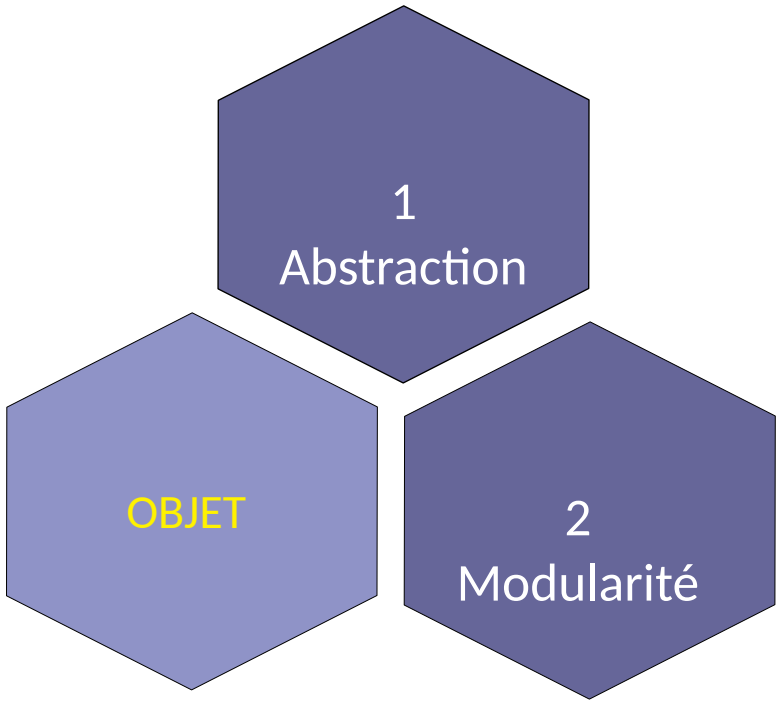


Abstraction

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



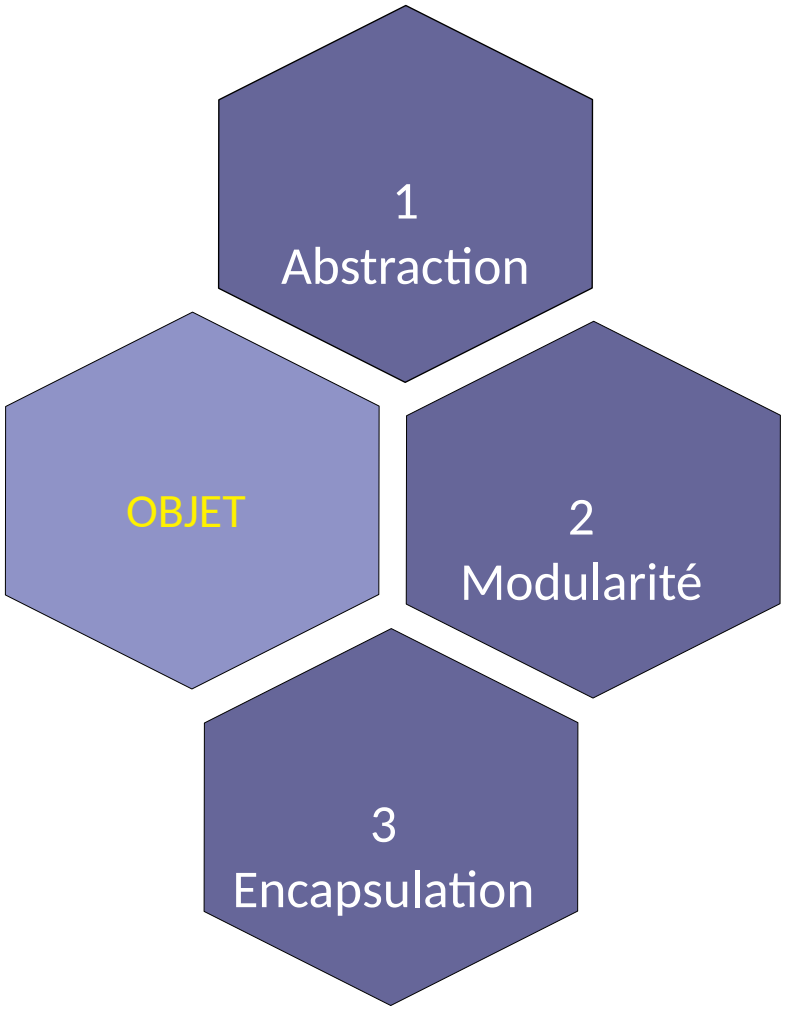
Les six piliers du paradigme objet



Modularité

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

Les six piliers du paradigme objet



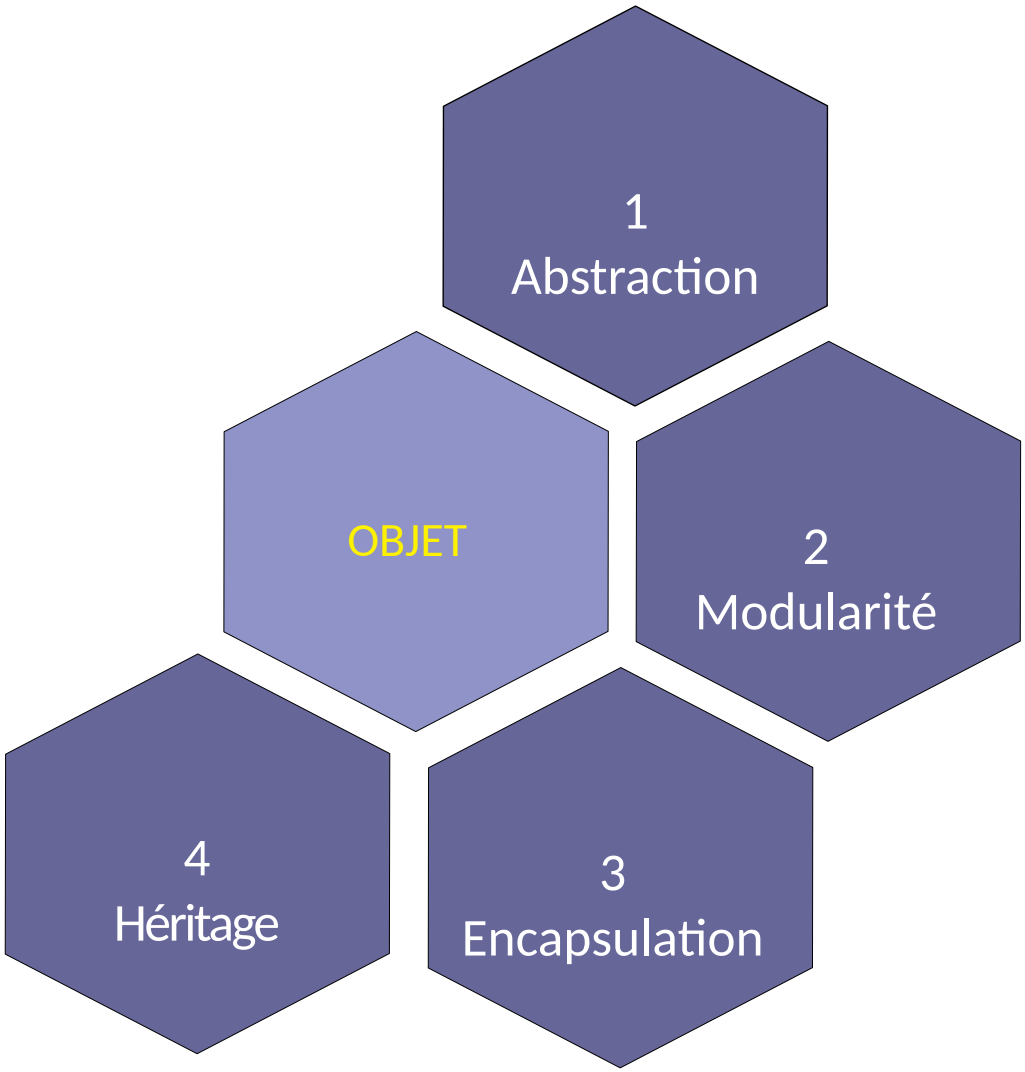
Encapsulation

- Principe de séparation de l'interface contractuelle d'une abstraction avec son implémentation

  → Masquer les détails de l’implémentation

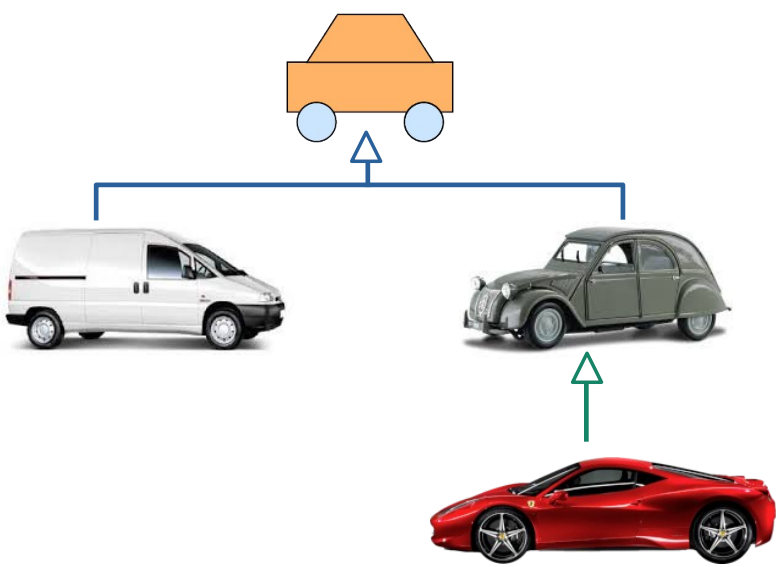
  → Permet de repousser l’implémentation le plus tard possible et d’en changer

Les six piliers du paradigme objet

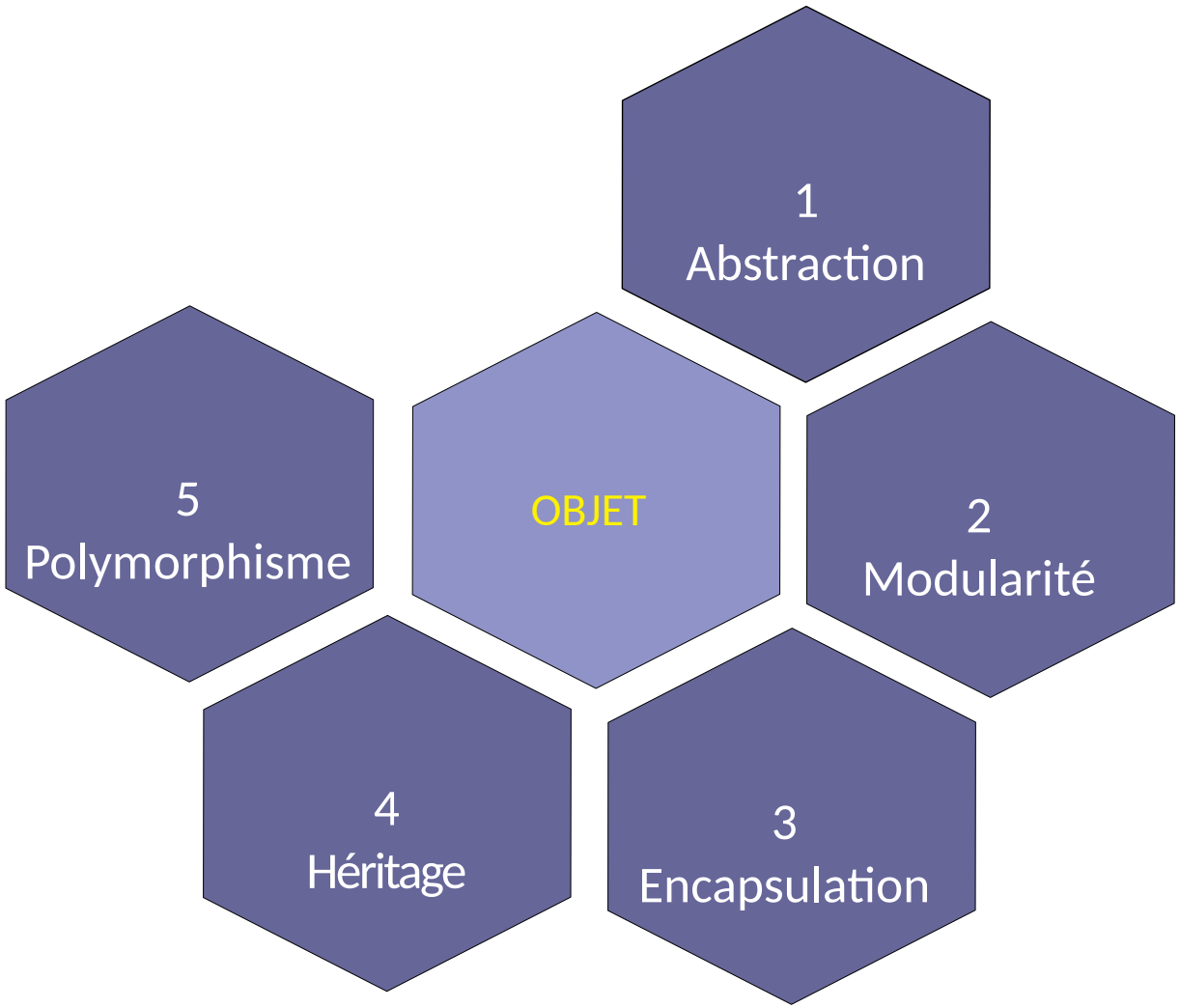


Héritage

- 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

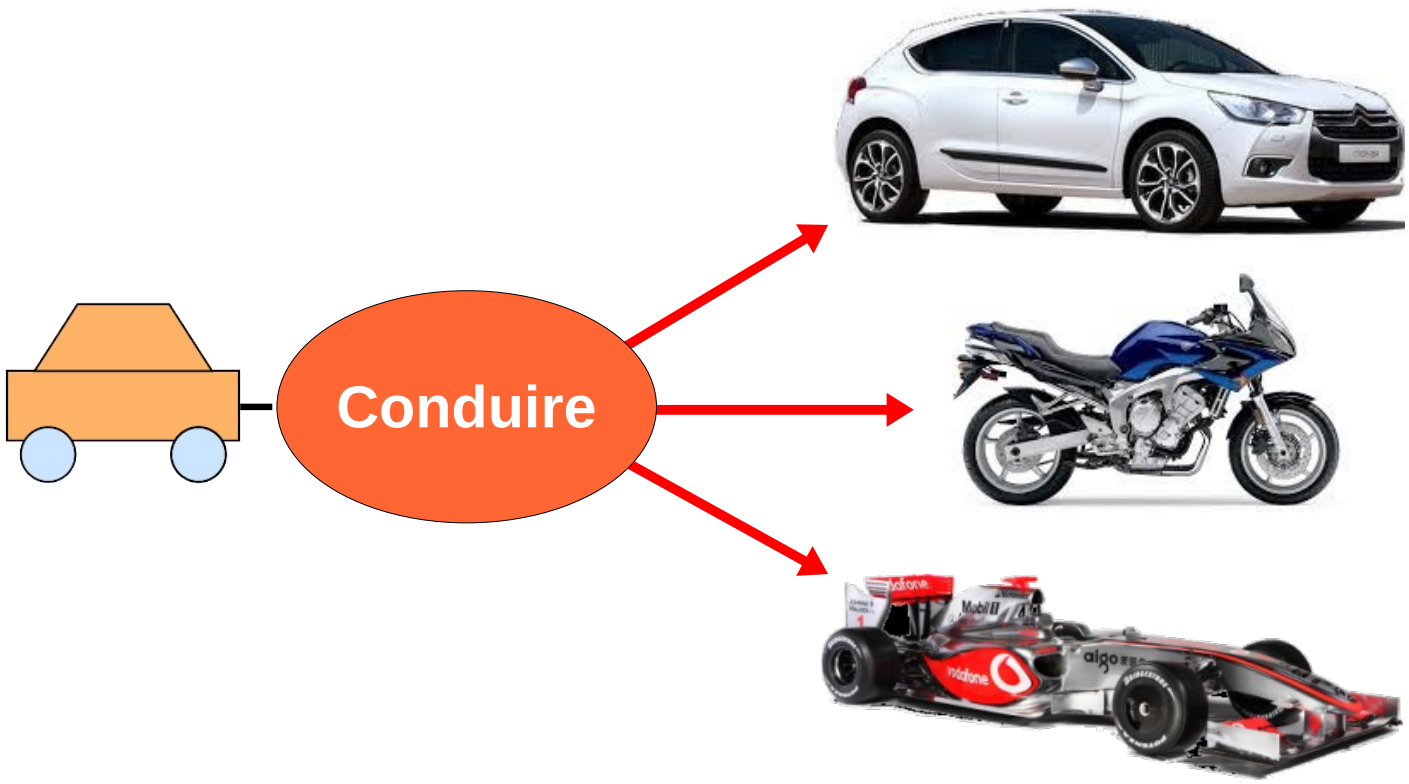


Les six piliers du paradigme objet

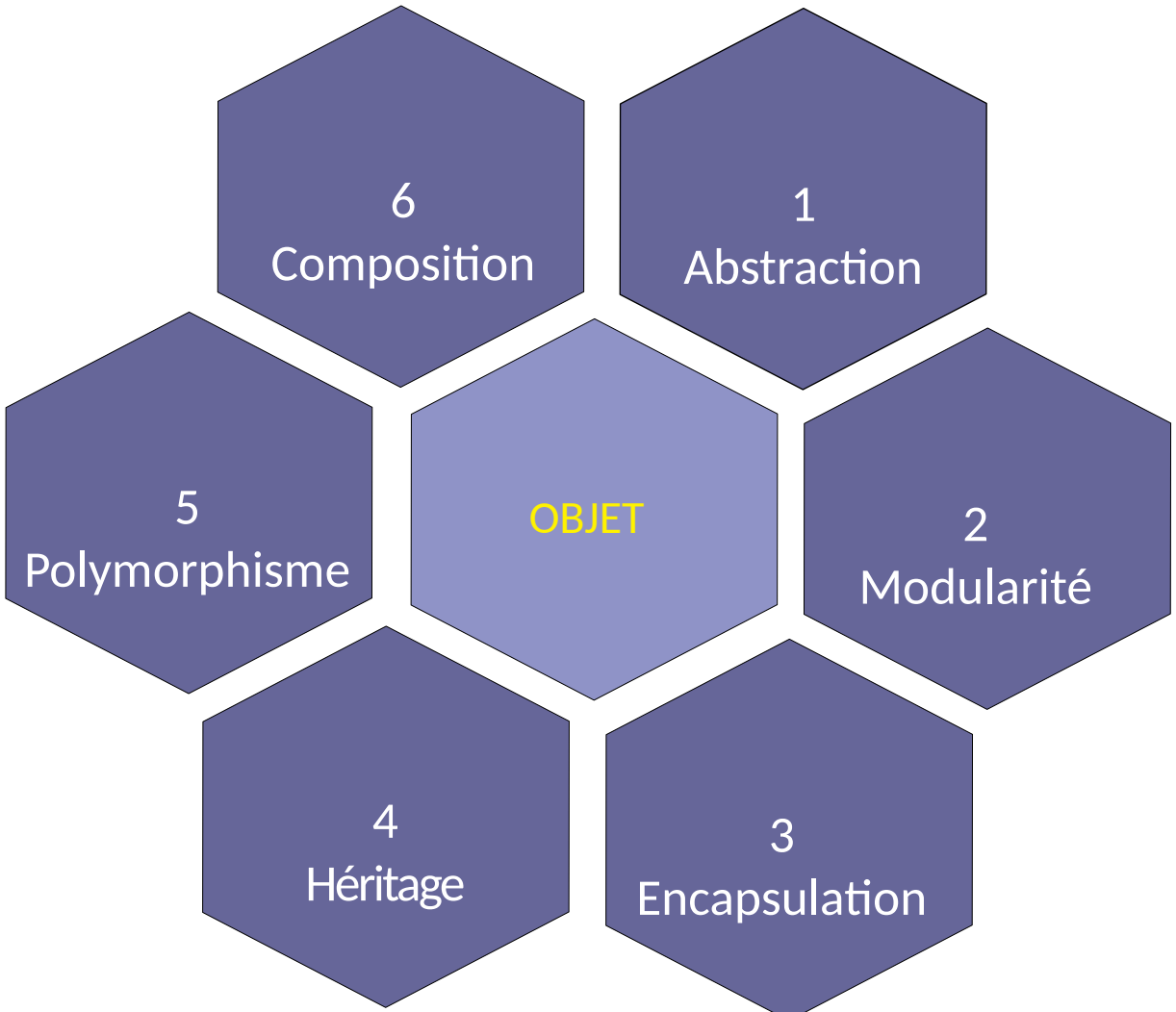


Polymorphisme

- Capacité d'objets appartenant à des classes différentes à répondre à l'appel d'une **méthode** de même nom, chacune avec le comportement adapté à 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 d'énormes progrès en termes de taille et de réussite de projet
- Apports :
  - **Développabilité**
    - confort avec lequel le logiciel peut être développé
  - **Extensibilité**
    - faculté d'étendre les fonctionnalités d'un logiciel sans compromettre son intégrité et sa fiabilité
  - **Maintenabilité**
    - facilité avec laquelle on peut corriger des erreurs ou des manques
  - **Réutilisabilité**
    - aptitude d'un logiciel à être réutilisé, en tout ou en partie, pour de nouvelles applications

Périls de l'approche objet

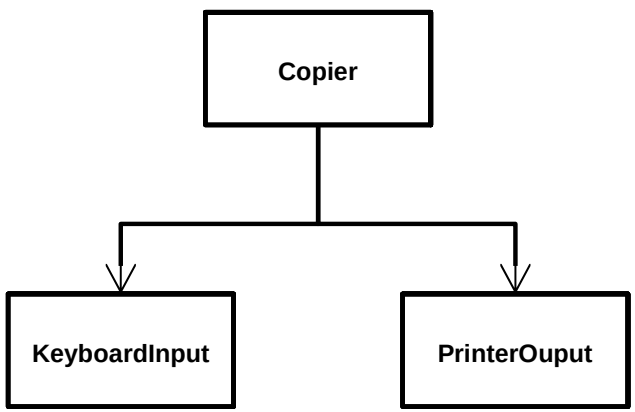
- Le paradigme et le langage seuls ne suffisent pas à assurer ces promesses
  - Par exemple : l'encapsulation est très souvent mise à mal par l'utilisation d'attributs non privés ou d'accesseurs/mutateurs (*getter/setter*) que permet le langage
- Périls d'une mauvaise conception objet
  - **Rigidité**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

Exemple du pourrissement d'une conception (1)

- Besoin client : une application qui copie les caractères lus au clavier vers l'imprimante

```
class Copier {
    private KeyboardInput _reader = new KeyboardInput();
    private PrinterOutput _writer = new PrinterOutput();
    void copy() {
        while ( (int ch = _reader.getChar()) != EOF) {
            _writer.print(ch);
        }
    }
}
```

```
Copier copier = new Copier();
copier.copy();
```



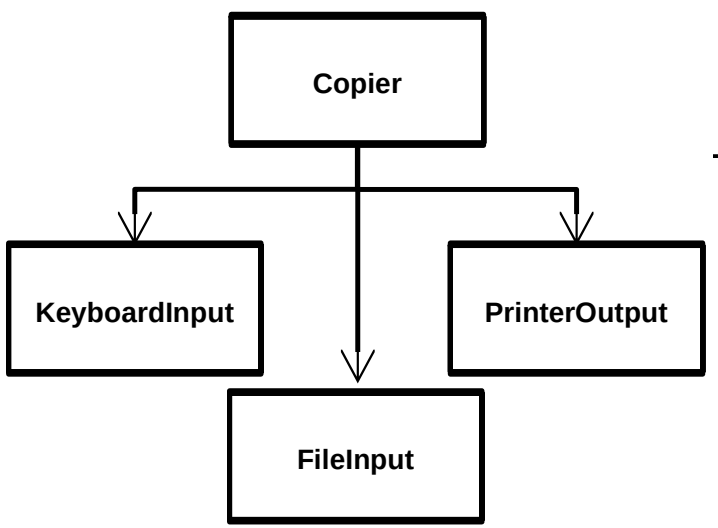
- Une vrai succès !

Exemple du pourrissement d'une conception (2)

- Nouveau besoin client : lecture à partir d'un fichier (code rétrocompatible)

```
class Copier {
    static bool readInFile = false;              // remember to clear
    private KeyboardInput _keyboardReader = new KeyboardInput();
    private FileInput _fileReader = new FileInput();
    private PrinterOutput _writer = new PrinterOutput();
    void copy() {
        while ((int ch = readInFile ? _fileReader.getChar()
                                    : _keyboardReader.getChar()) != EOF) {
            _writer.print(ch);
        }
    }
}
```

```
Copier copier = new Copier();
Copier.readInFile = true;
copier.copy();
Copier.readInFile = false; // Ne pas oublier
```

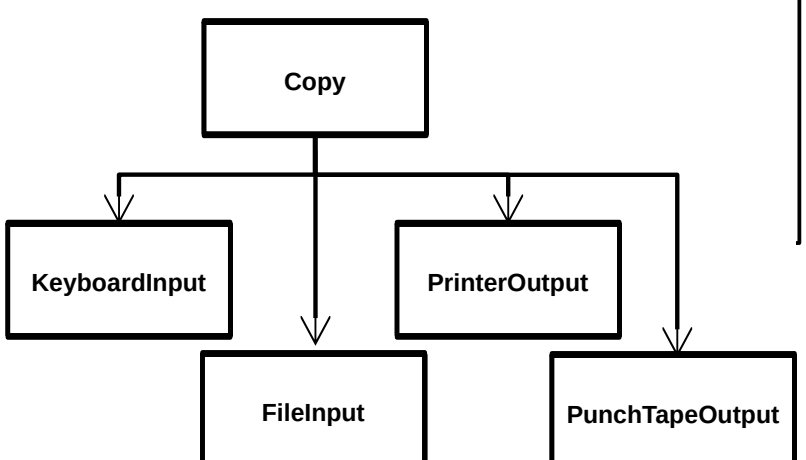


Exemple du pourrissement d'une conception (3)

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

```
class Copier {
    static bool readInFile = false;              // remember to clear
    private KeyboardInput _keyboardReader = new KeyboardInput();
    private FileInput _fileReader = new FileInput();
    static bool writeOnTape = false;             // remember to clear
    private PrinterOutput _printerWriter = new PrinterOutput();
    private PunchTapeOutput _punchTapeWriter = new PunchTapeOutput();
    void copy() {
        while((int ch = fileInReader ? _fileReader.getChar()
                                     : _keyboardReader.getChar())) != EOF) {
            writeOnTape? _punchWriter.print(ch)
                       : _printerWriter.print(ch)
        }
    }
}
```

```
Copier copier = new Copier();
Copier.writeOnTape = true;
copier.copy();
Copier.writeOnTape = false; // Ne pas oublier
```

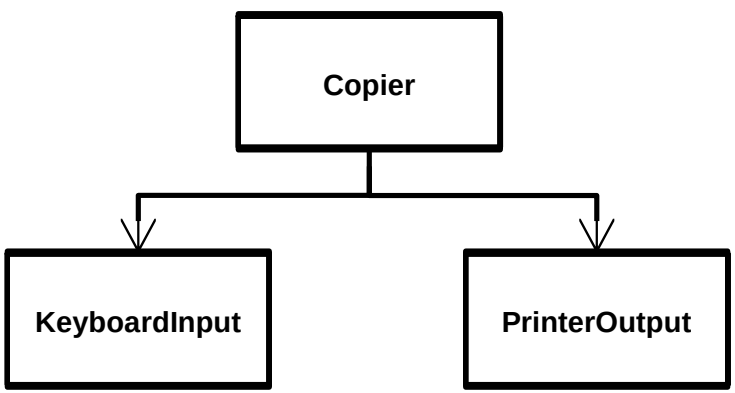


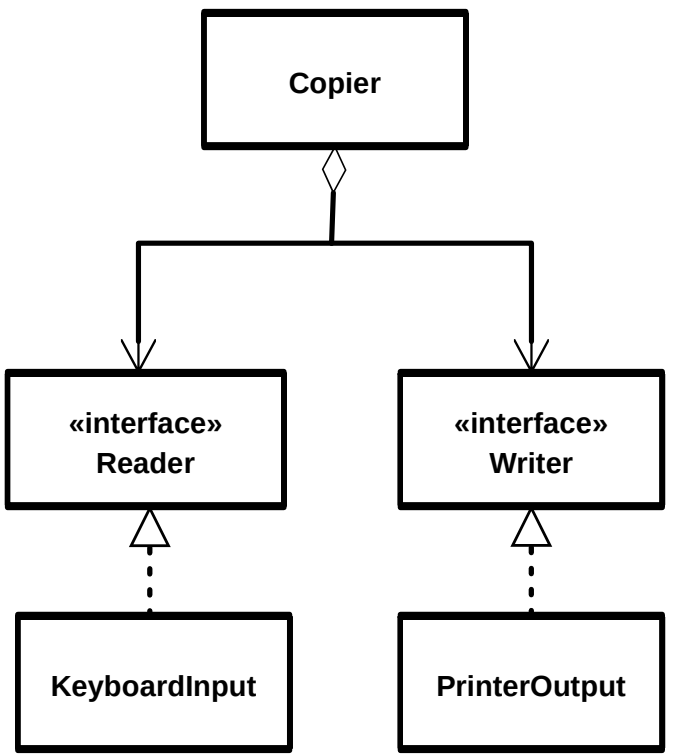
Sentez vous le relent de pourrissement (*rotten code*) ?

- **Rigidité** (logiciel difficile à faire évoluer)
  - Pour utiliser le service, il faut positionner une variable qui n'est pas explicitement liée au service (uniquement par un commentaire)
    - Le prototype des services n'est d'aucune aide pour positionner ces variables
  - Si on veut ajouter une nouvelle entrée : catastrophe
  - La parallélisation est impossible à cause des variables globales
- **Fragilité** (logiciel difficile à maîtriser)
  - Si on oublie de repositionner la valeur de la variable, le programme peut ne plus fonctionner correctement à d'autres endroits
    - Le compilateur n'est d'aucune aide pour signaler l'oubli
- **Immobilité** (logiciel difficile à réutiliser)
  - Impossible de réutiliser Copy sans embarquer les 4 autres classes (`KeyboardInput`, `PrinterOutput`, `FileInput`, `PunchTapeOutput`) mêmes si vous voulez n'en utiliser que 2

Refonte de la conception (*refactoring*)

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





Code résultant


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

Utilisation

- Création d'une copie

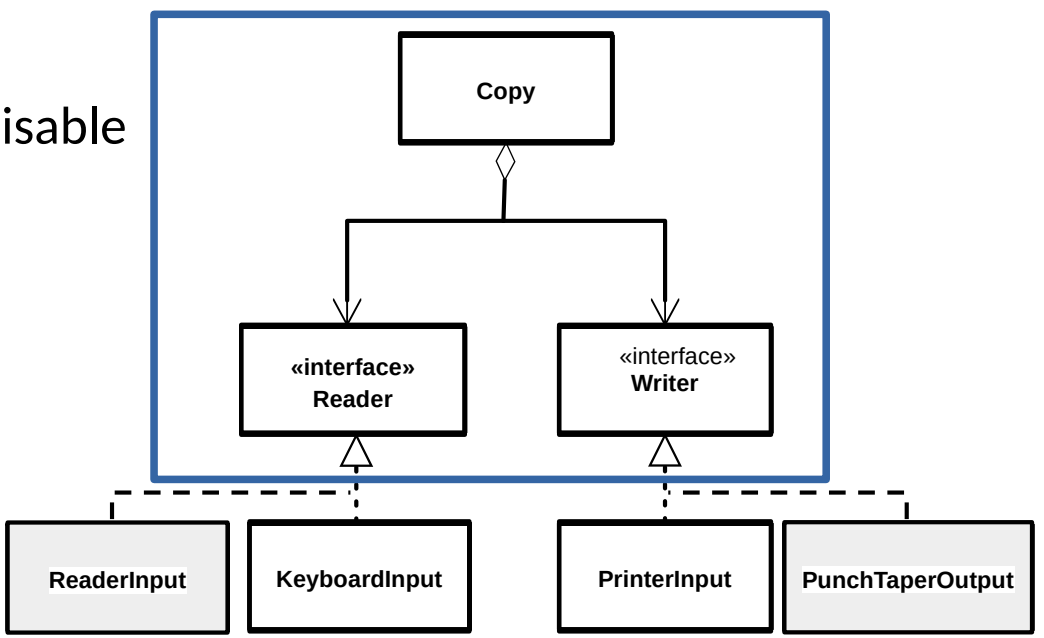
```
Copier copier1 = new Copier(); // KeyBoardInput(), PrinterOutput
Copier copier2 = new Copier(new KeyBoardInput(), new PrinterOutput());
Copier copier3 = new Copier(new FileInput(), new PrinterOutput());
Copier copier4 = new Copier(new FileInput(), new PunchTapeOutput());
```

- Utilisation du service grâce au polymorphisme

```
copier1.copy()
copier2.copy()
copier3.copy()
copier4.copy()
```

Bilan de la nouvelle conception

- Qu'avons nous gagner ?
  - ~~Rigidité~~ : Facilité d'ajout d'une autre entrée ou d'une autre sortie.
  - ~~Fragilité~~ :
    - Plus de variable locale à positionner sans information pour cela.
    - Fonctionnement vérifiable par le compilateur (plus de condition d'utilisation en commentaire).
    - Fonctionnalité parallélisable.
  - ~~Immobilité~~ : 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 de 1re année : extensions IntelliJ pour aider à améliorer la qualité d'une conception
  - **SonarQube** : une référence dans le domaine de l'analyse de code
  - **PMD** : ensemble de métriques pour la cohésion et le couplage

Critère 1. Cohésion (*Cohesion*)

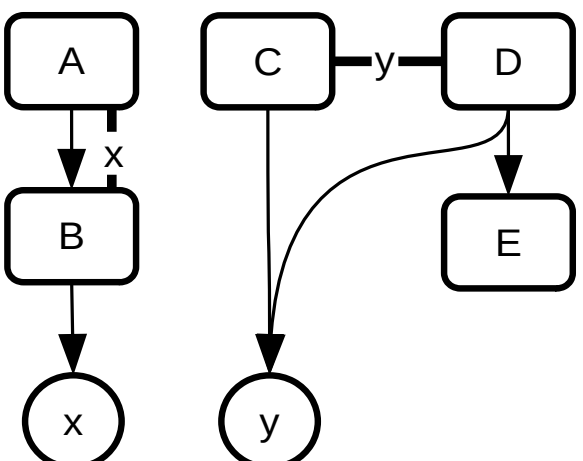
- Caractérise l'étendue de la responsabilité d'un module

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

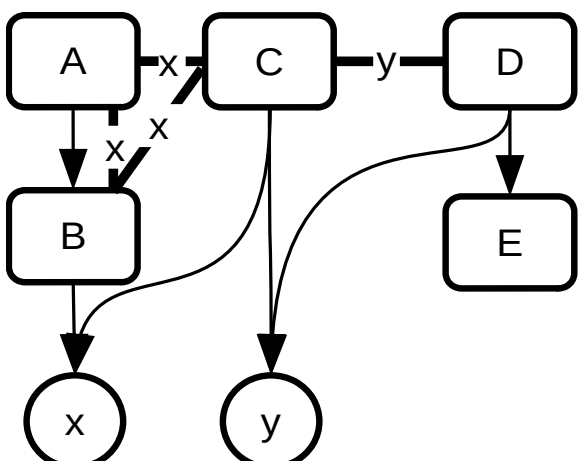
- Indice
  - La liste des attributs est un bon indicateur du degré de cohésion

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

Objectif du cours

- Rappel : la valeur d'un logiciel ne vient pas forcément de la fonctionnalité en elle-même, mais de la capacité à pouvoir changer le comportement de son application facilement afin de pouvoir s'adapter au mieux aux exigences du client

> "If you give me a program that works
> but is (practically) impossible to change,
> it won't work after the requirements change,
> and I won't be able to make it work.
> If you give me a program that doesn't work
> but is easy to change,
> then I can make it work."
> R. Martin, 2012