



01

Chapitre

Limites du paradigme objet pour la conception

2I1AC3 : Génie logiciel et Patrons de conception

Régis Clouard, ENSICAEN - GREYC

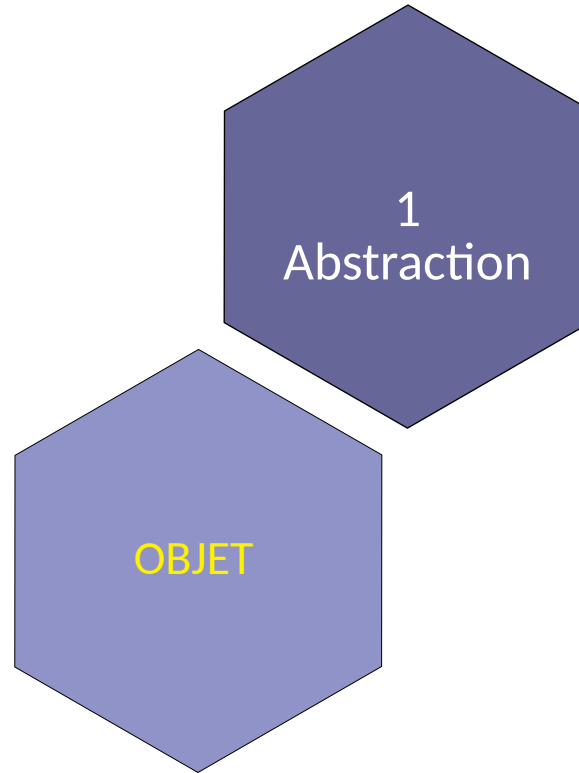
« La perfection n'est atteinte, non pas lorsqu'il n'y a plus rien à ajouter, mais lorsqu'il n'y a plus rien à enlever. »

Antoine de Saint-Exupéry

Rappel : les six piliers du paradigme objet

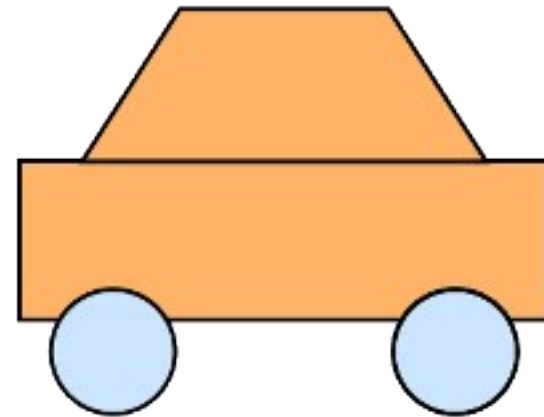
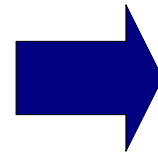


Les six piliers de la conception objet

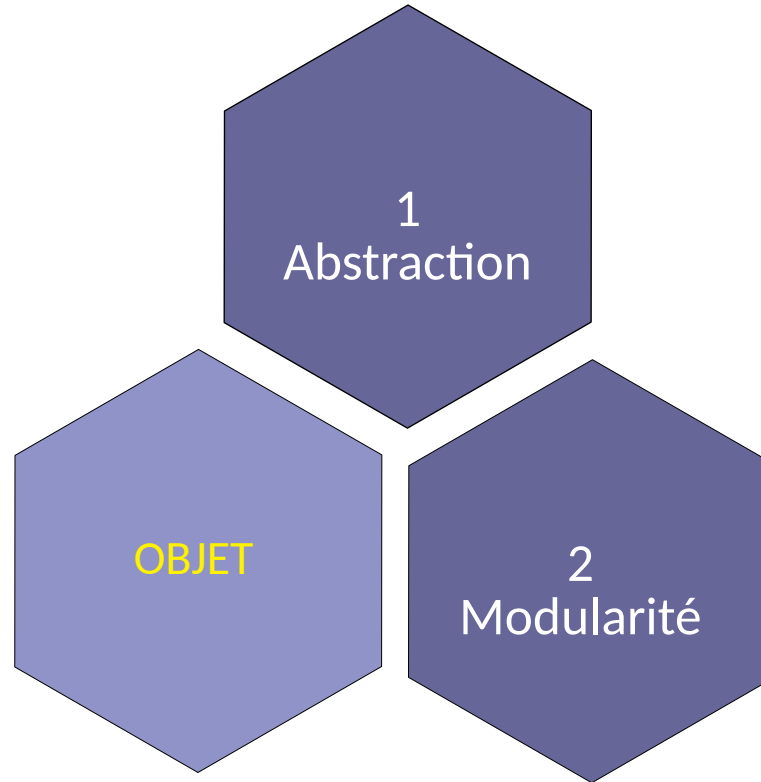


Abstraction

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

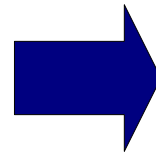


Les six piliers du paradigme objet

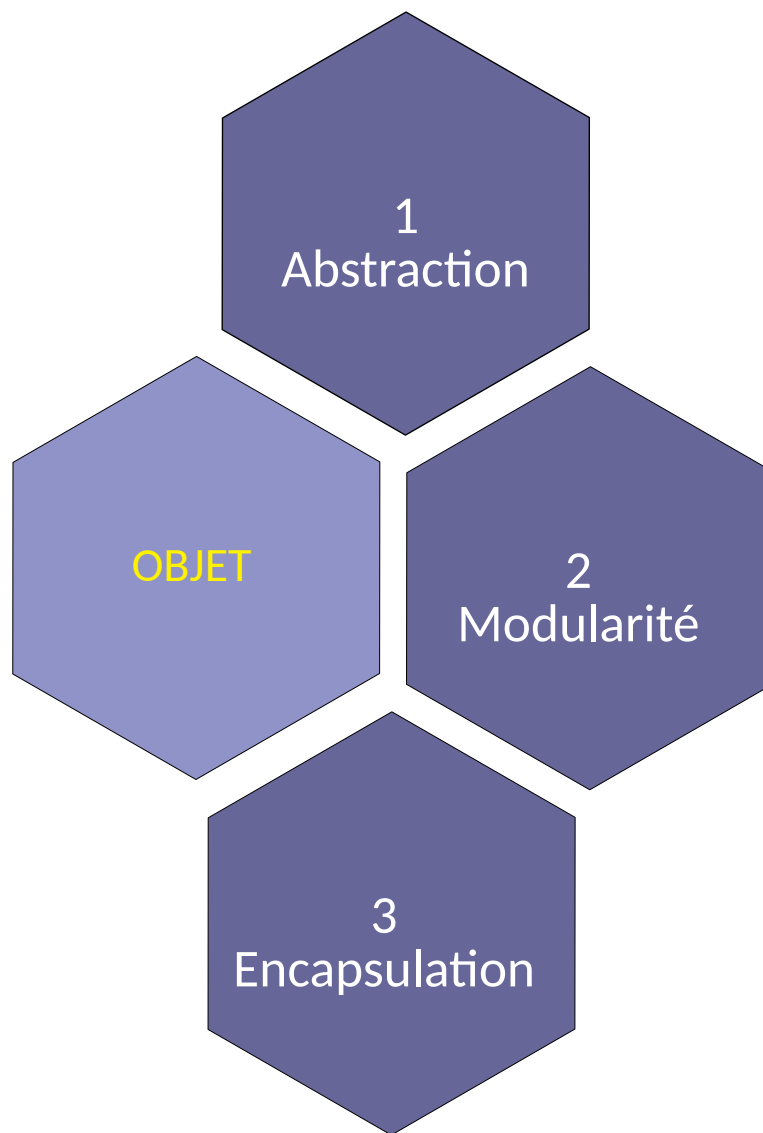


Modularité

- Procédé de construction d'un système par assemblage de modules compacts
 - Module = Fonction, Classe, Paquet, Composant ou Nœud



Les six piliers du paradigme objet

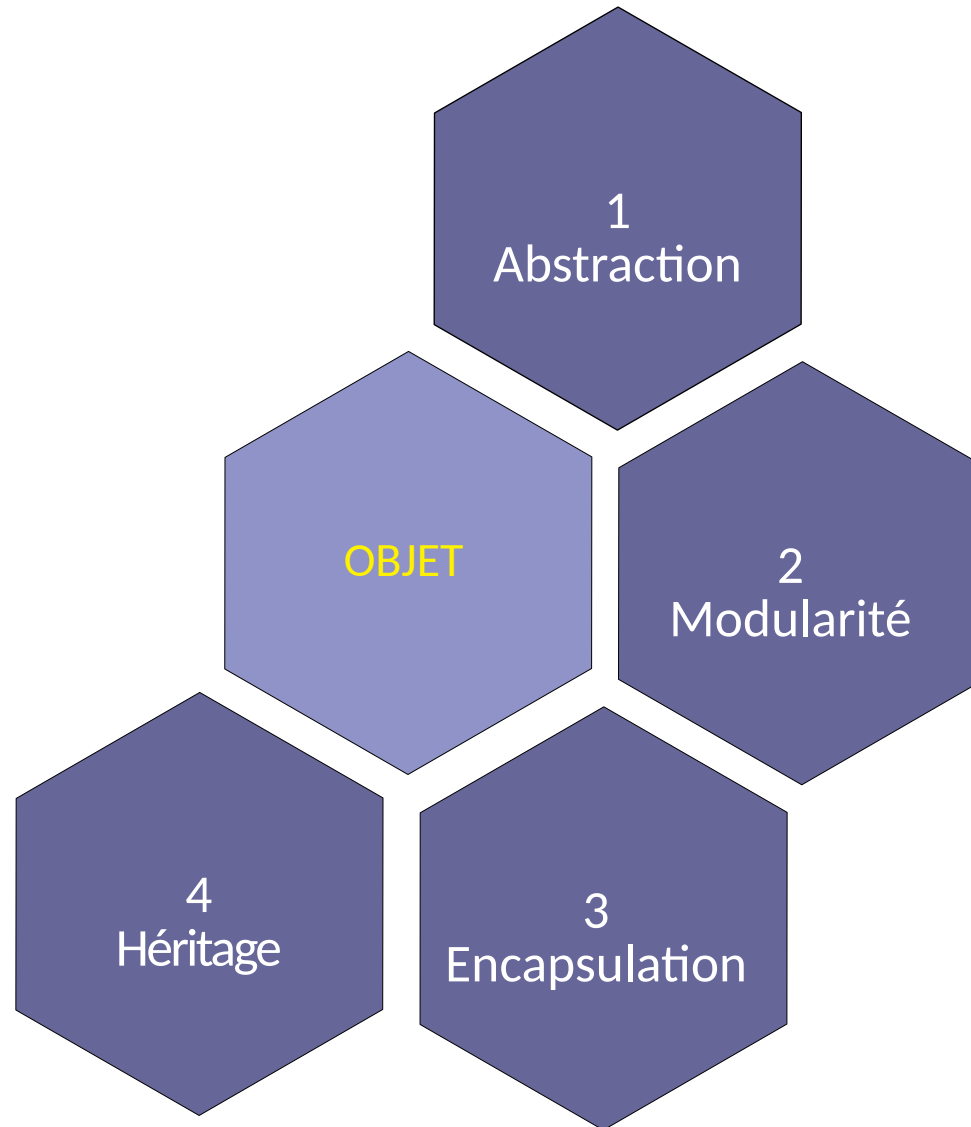


Encapsulation

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

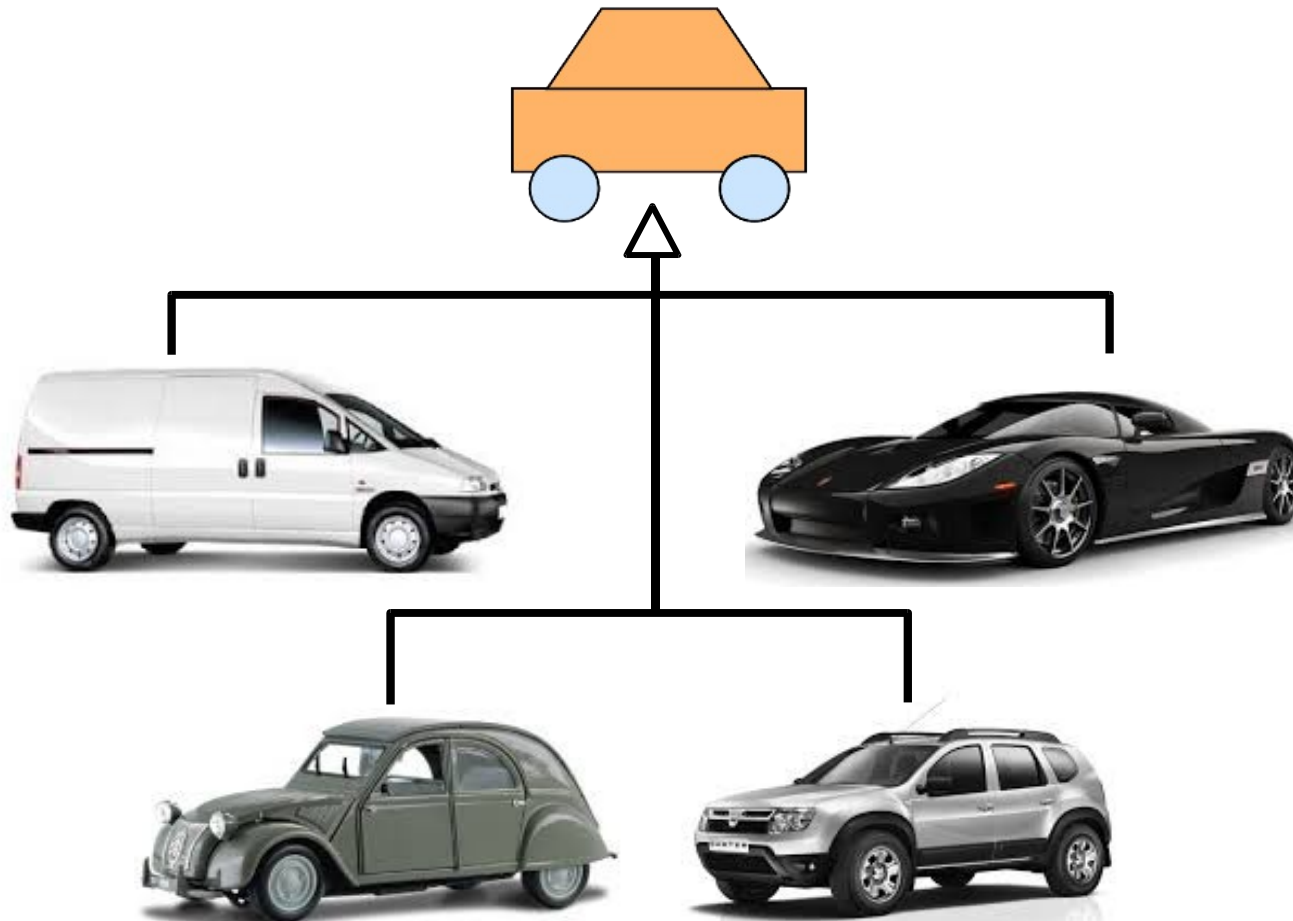


Les six piliers du paradigme objet

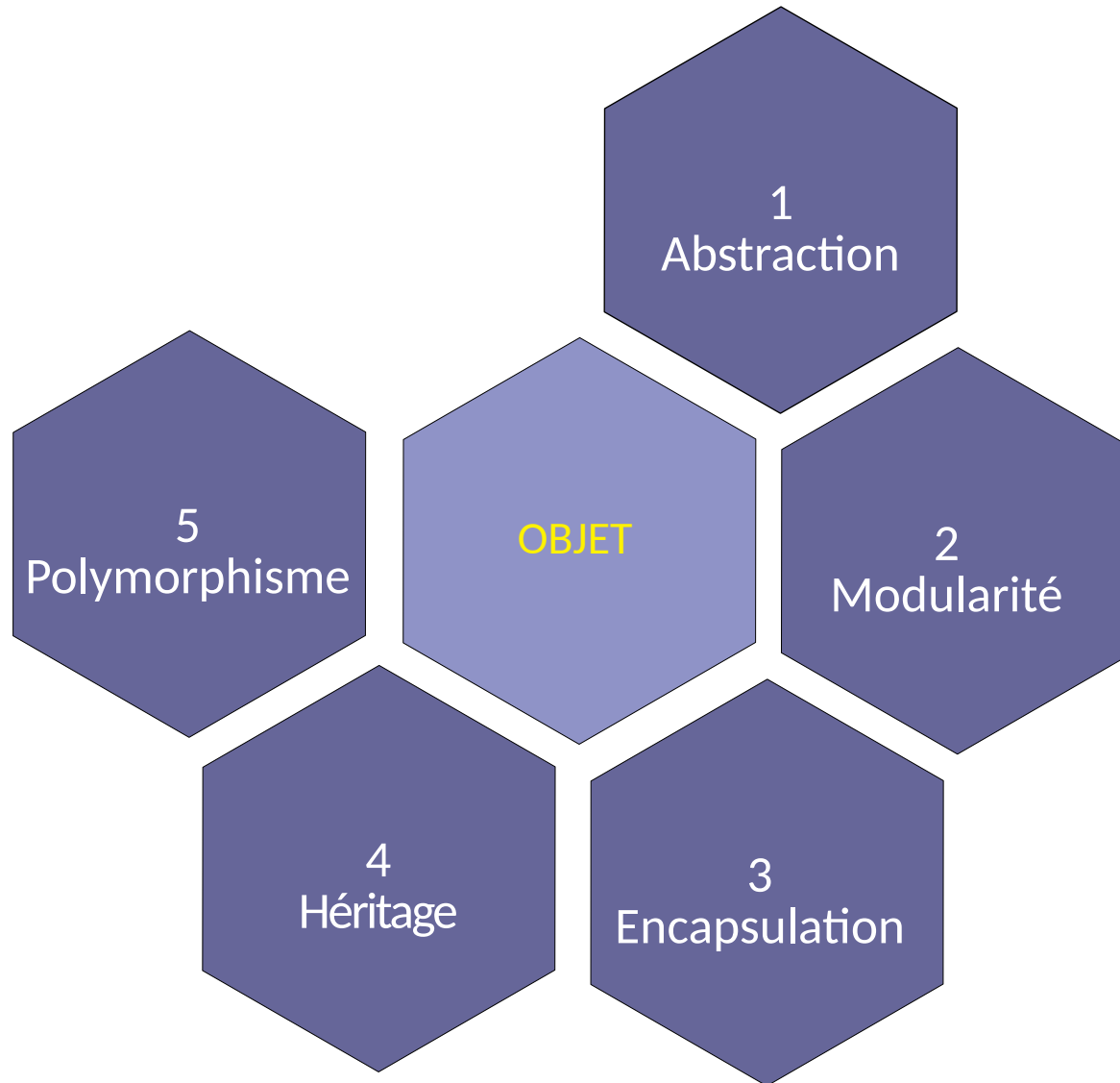


Héritage

- Procédé de réutilisation par lequel une classe est obtenue par extension de l'implémentation d'une classe existante

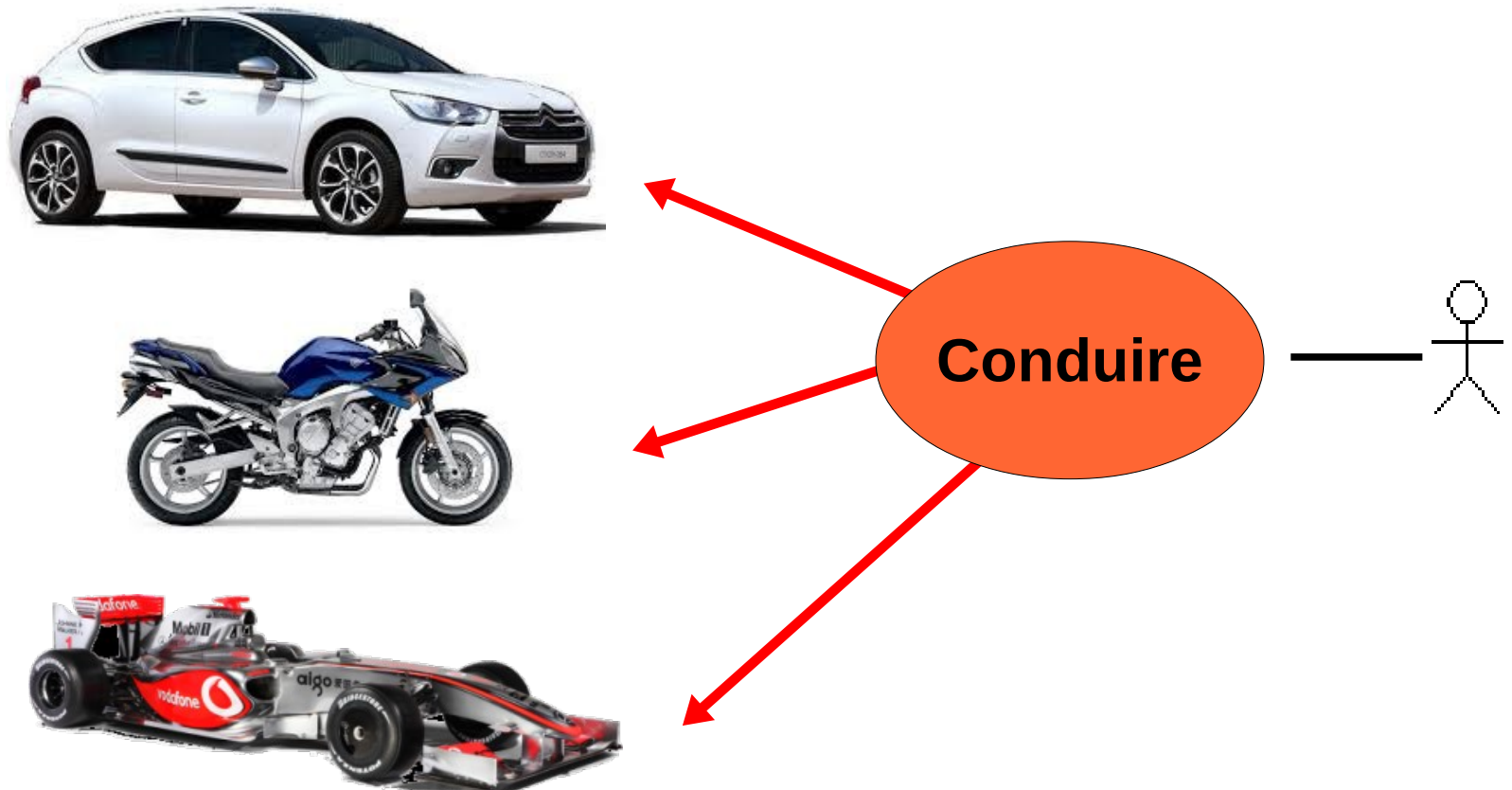


Les six piliers du paradigme objet

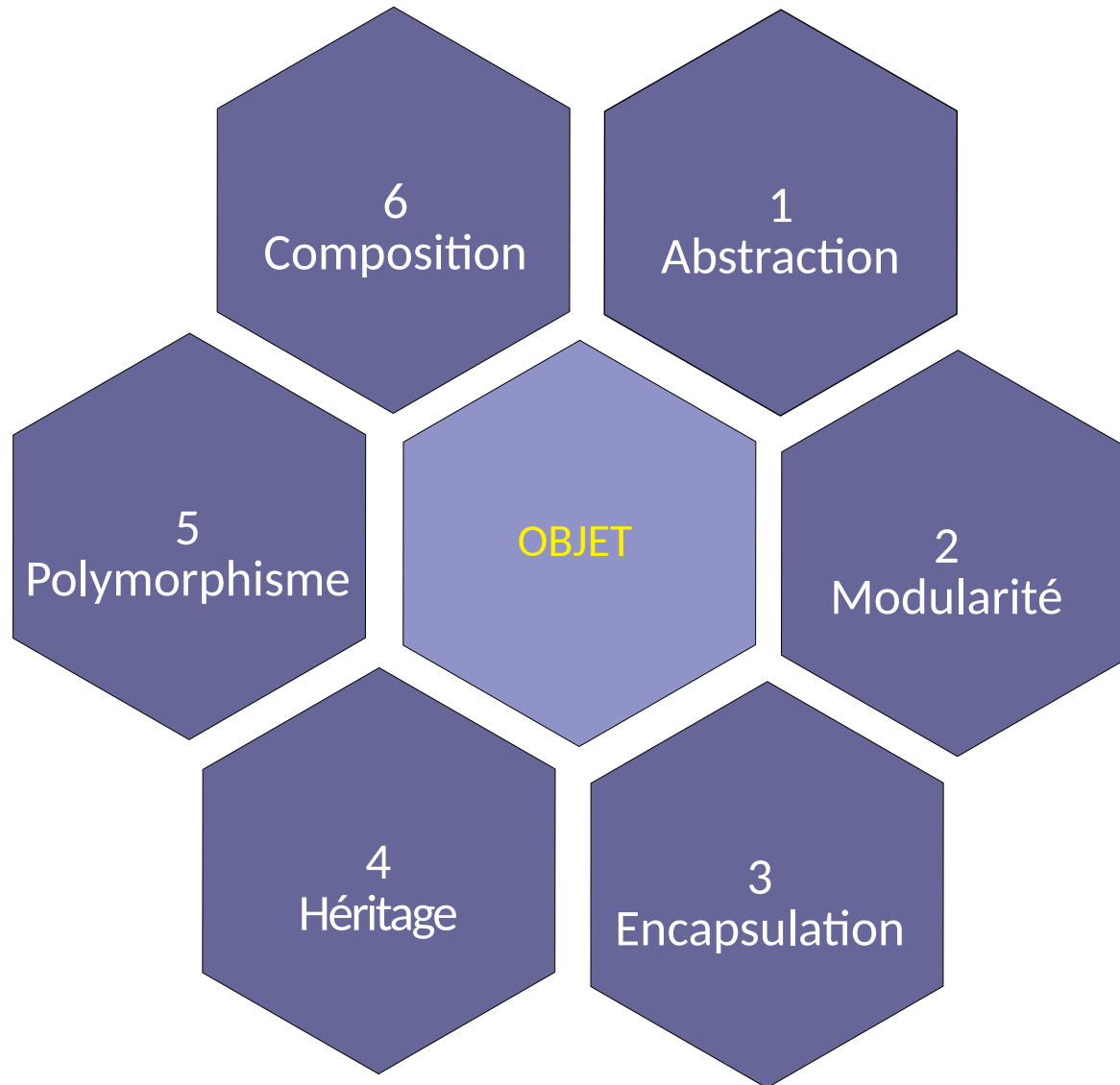


Polymorphisme

- 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.

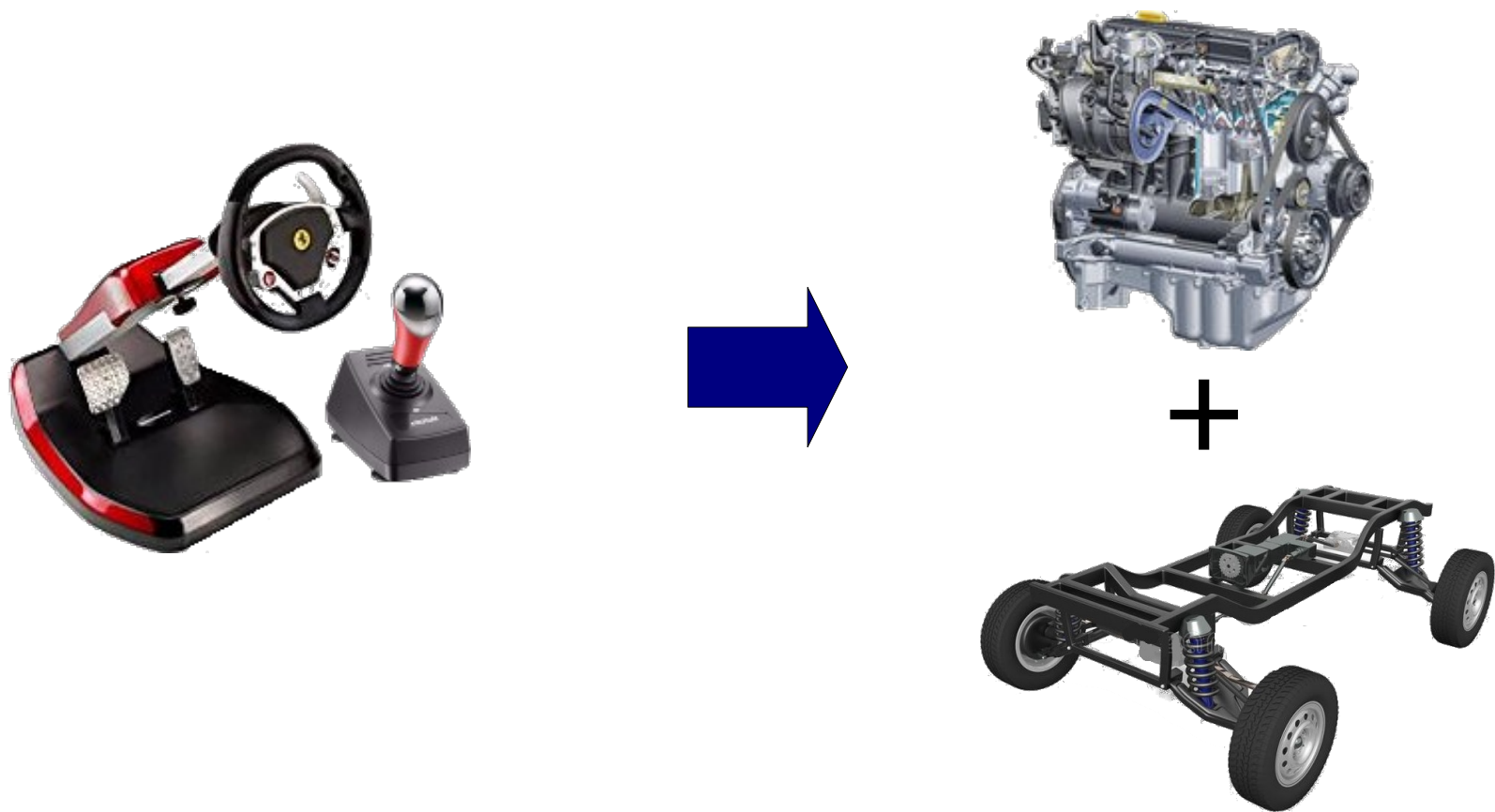


Les six piliers du paradigme objet



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 pour ambition :
 - **Développabilité** : confort avec lequel le logiciel peut être développé
 - ▶ Grâce à l'abstraction et la modularité
 - **Extensibilité** : faculté d'étendre simplement 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 paquet a la qualité de la réutilisabilité si et seulement si le réutilisateur n'a pas besoin de regarder le code pour le réutiliser (autre que l'interface publique)
- Le paquet doit être pensé puis réutilisé comme s'il s'agissait d'une archive compilée (eg. .jar, .dll)

Périls de l'approche objet

- Mais, 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'accessseurs / mutateurs

Périls de l'approche objet

- Le paradigme objet n'empêche pas de faire une conception présentant les défauts suivants :
 - **Rigidité** : logiciel difficile à faire évoluer
 - **Fragilité** : logiciel difficile à maîtriser
 - **Immobilité** : logiciel difficile à réutiliser

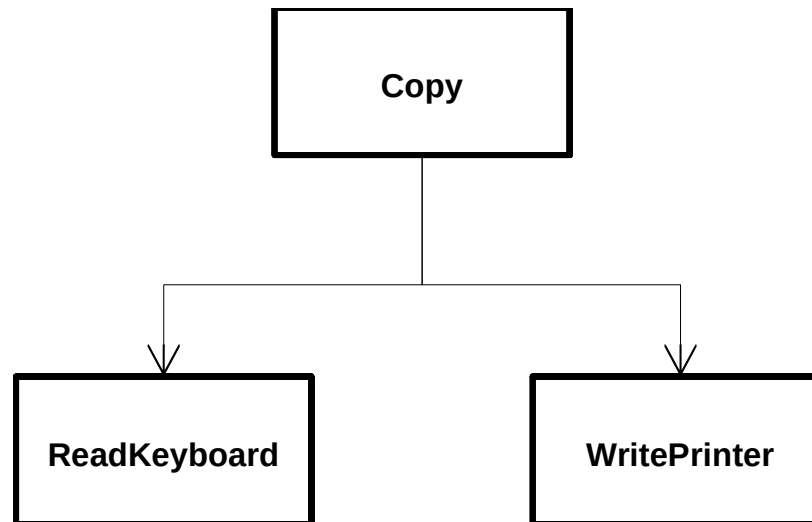
Exemple du pourrissement d'une conception

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

Première version

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

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



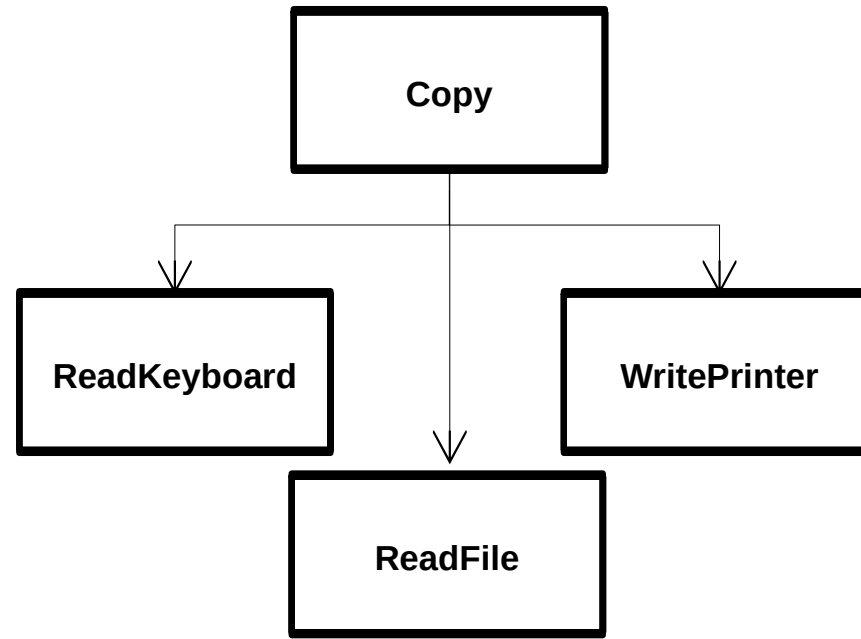
- Une vrai succès !

Seconde version

- Lecture à partir d'un fichier

```
bool GtapeReader = false; // remember to clear

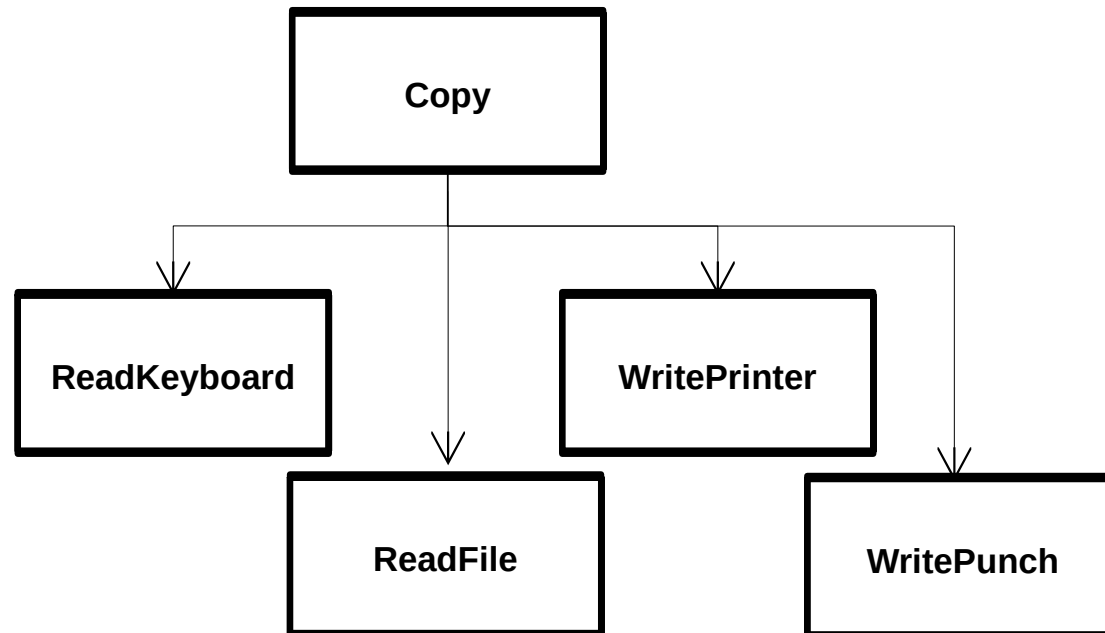
void copy(void) {
    int ch;
    while ((ch = GtapeReader ? ReadFile.getChar()
                             : ReadKeyboard.getChar()) != EOF) {
        WritePrinter.print(ch);
    }
}
```



Troisième version

- Imprimer sur un ruban perforé pour aveugle

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



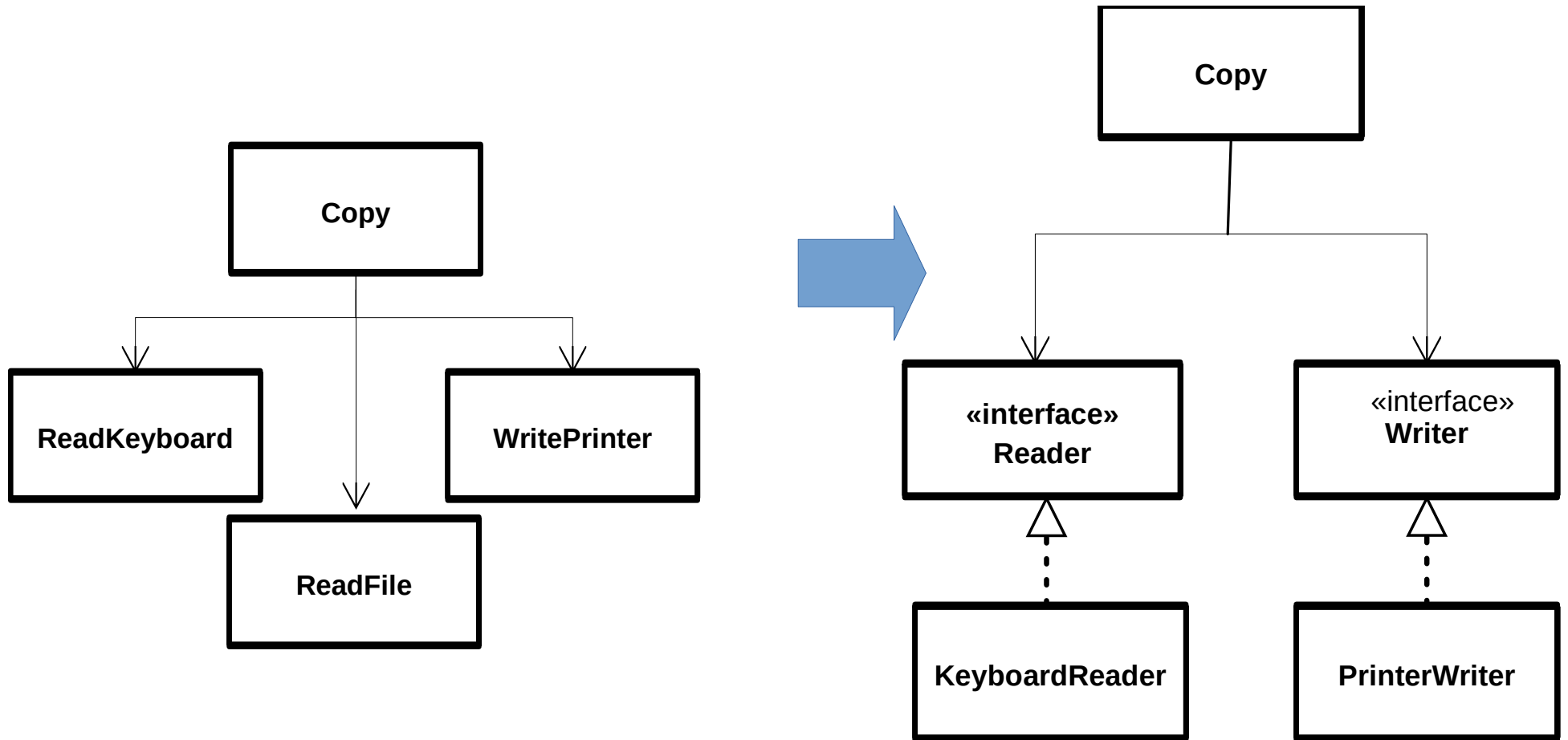
Sentez vous le relent de pourrissement ?

23

- Rigidité : tout est intrinsèquement lié
- Fragilité : variables globales
- Immobilité : un bloc non sécable

Refonte (refactoring)

- La première version est la même (principe YAGNI)
- Mais le 2^e conduit à une refonte tout en étant rétrocompatible




```
interface Reader {
    char read();
}
interface Writer {
    void write(char c);
}
class Copy {
    Reader itsReader;
    Writer itsWriter;
    Copy() { // Par défaut
        itsReader = new KeyboardReader();
        itsWriter = new PrinterWriter();
    }
    Copy(Reader r, Writer w) {
        itsReader = r;
        itsWriter = w;
    }
    void copy(void) {
        while( (int c = itsReader.read()) != EOF ) {
            itsWriter.write(c);
        }
    }
}
```

Critères de qualité d'une conception

- Deux critères absolus de qualité :
 - 1) Cohésion
 - 2) Couplage
- Ils portent sur les modules
- Ces deux critères sont difficiles à mesurer automatiquement
 - Il existe des métriques qui permettent d'apprécier ces critères, mais ces métriques ne sont que des indicateurs mais pas des mesures de qualité absolues

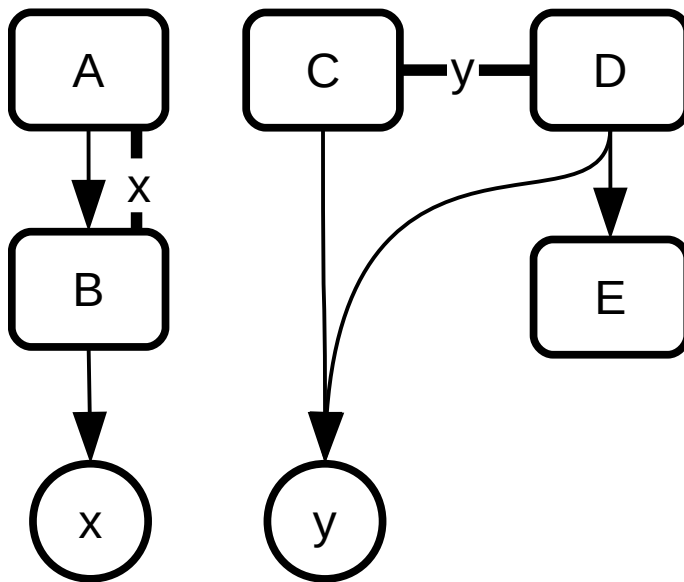
Critère 1. Cohésion

- 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

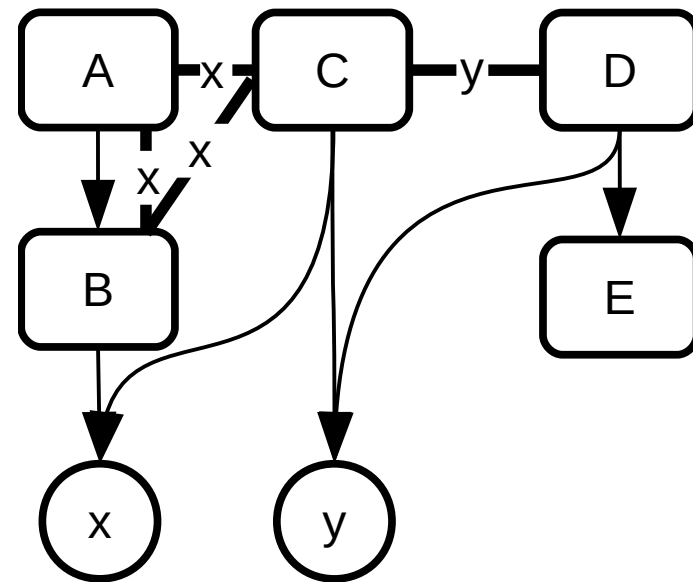
Exemple d'une métrique

■ 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

- Caractérise la force d'interaction entre les modules
- Questions associées :
 - Comment les modules collaborent 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)

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* : $ATFD > 5\%$

- 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*) - ie. Virer à l'usine à gaz
 - 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