ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Chapitre 1
Limites du paradigme objet pour la conception

2I1AC3 : Génie logiciel et Patrons de conception

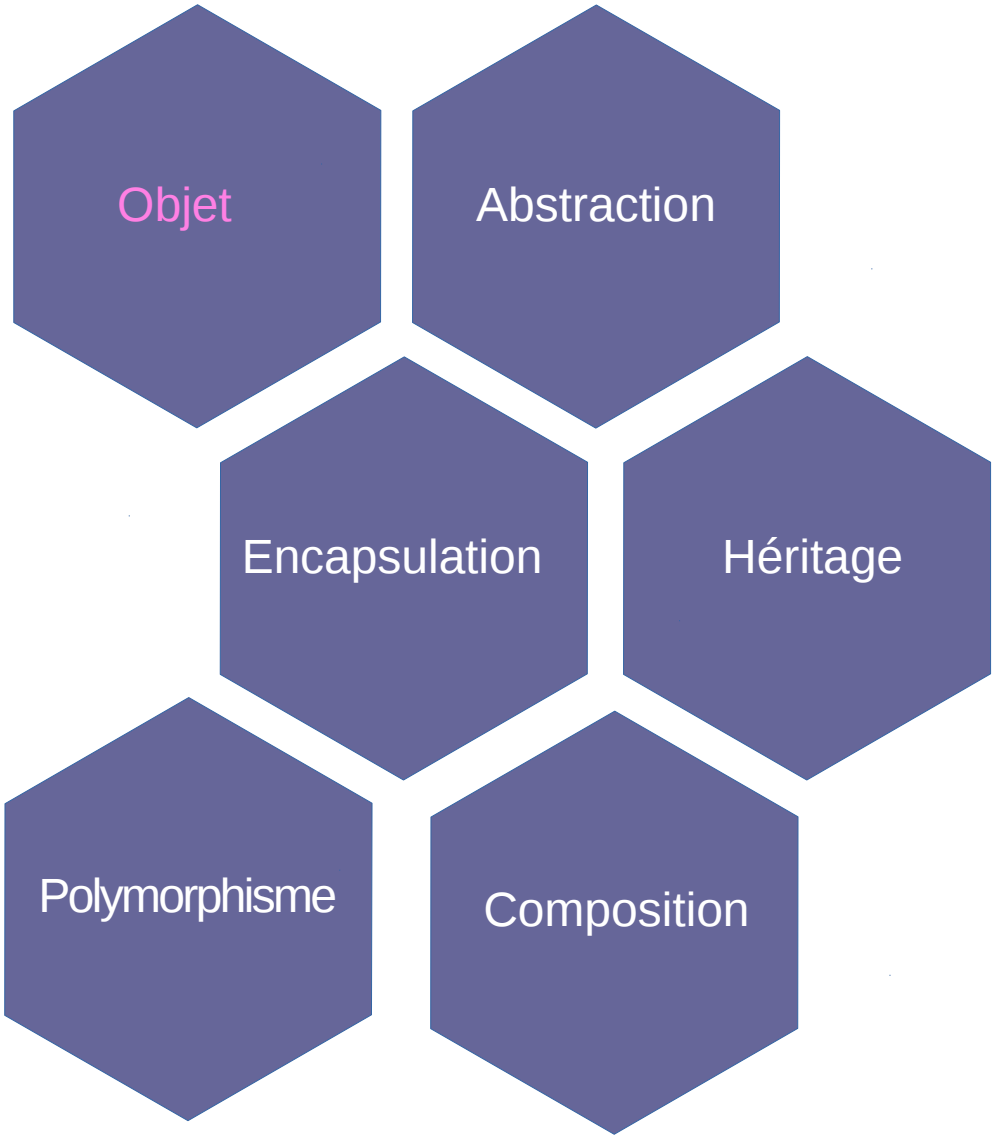
Régis Clouard, ENSICAEN - GREYC

« 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

Organisation

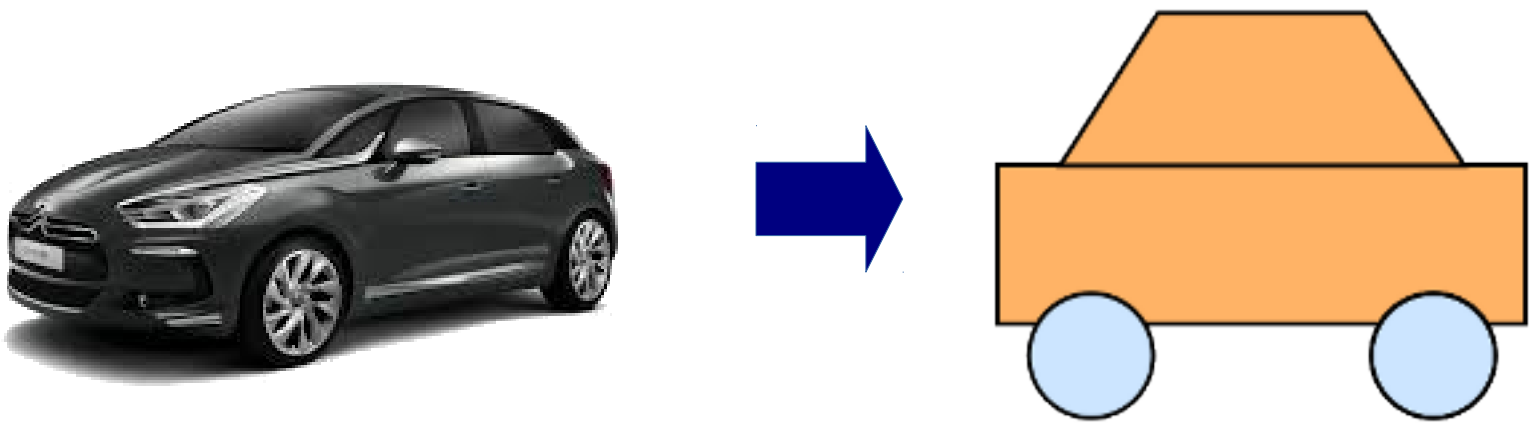
- Objectif du cours
  - Vers une conception des logiciels de qualité professionnelle.
  - Valoriser la « beauté » d'une conception : **artisanat du logiciel** (*software craftsmanship*).
- Prérequis
  - Conception orientée objet.
  - Langage de modélisation : UML.
  - Langage de programmation orientée objet : Java.
- Charge de travail
  - 9 CM (+ 2h de git)
  - 8 TD
  - 16 TP (projet) (+ 2h de git)

Les cinq piliers de la conception objet



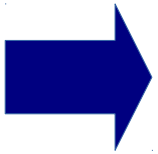
Abstraction

- Procédé de construction d'un modèle simplifié d'un système complexe tout en conservant ses fonctions essentielles.
  - Identifier le **comportement attendu** des objets d'intérêt du système.
  - Focaliser sur la vue externe des objets.



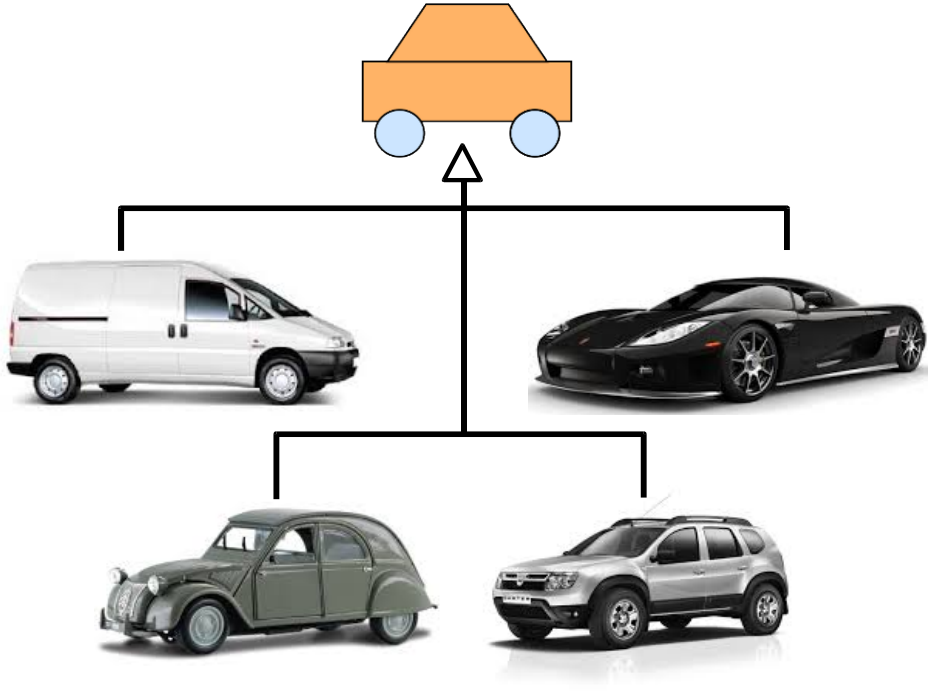
Encapsulation

- Principe de séparation de l'interface contractuelle d'une abstraction avec son implémentation.
  - Emballer l'information de manière à masquer la représentation interne.
  - Toutes les interactions avec l'objet sont faites par l'intermédiaire d'une **interface publique** des fonctions (les services).



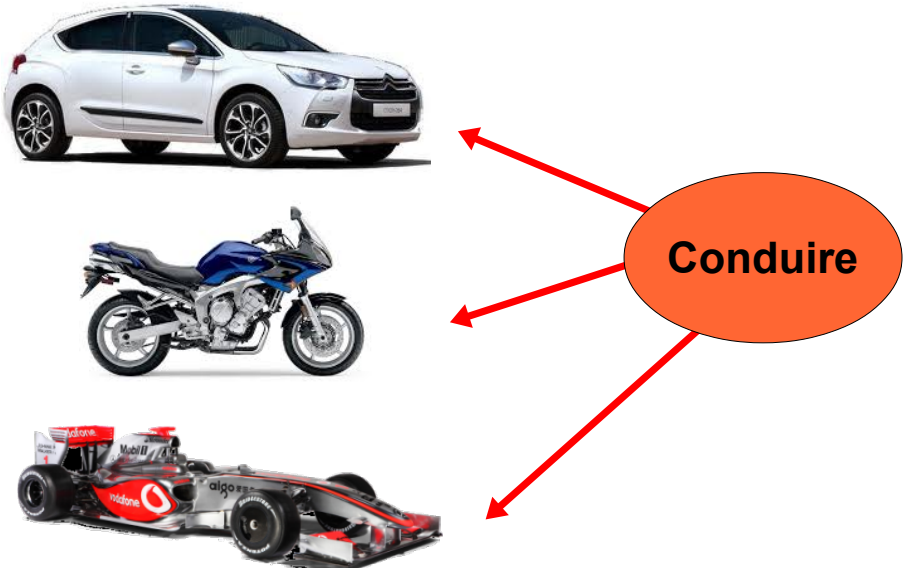
Héritage

- Procédé de réutilisation par lequel une nouvelle fonction est obtenue par extension de l'implémentation d'une classe existante.
  - La classe de généralisation capture explicitement les attributs et les méthodes communes.
  - La classe de spécialisation étend l'implémentation avec des attributs et des méthodes supplémentaires.



Polymorphisme

- Capacité des objets appartenant à des classes différentes à répondre aux appels de méthodes de même nom, chacun selon le comportement spécifique de sa classe.
  - Les objets doivent présenter une interface compatible avec la même signature de méthode.
  - Le programme n'a pas à connaître la classe exacte de l'objet à l'avance, de sorte que le comportement est mis en œuvre lors de l'exécution.



Composition

- Procédé de réutilisation, par lequel une nouvelle fonction est obtenue en créant un objet composé d'autres objets.
  - La composition encapsule plusieurs objets à l'intérieur d'un autre.
  - La nouvelle fonction est obtenue en déléguant sa réalisation à différents objets de la composition.

Promesses du paradigme objet

- Le paradigme de conception orientée objet (COO) a pour ambition :
  - **Extensibilité** : faculté d'étendre simplement les fonctionnalités d'un logiciel sans compromettre son intégrité et sa fiabilité.
    - La modularité, l'abstraction et le polymorphisme permettent de décomposer une grosse application en les sous-parties plus autonomes et plus faciles à étendre.
  - **Maintenabilité** : simplicité de correction et de modification du logiciel.
    - Grâce à l'encapsulation, le code n'a besoin d'être changé qu'à un seul endroit.
  - **Réutilisabilité** : aptitude d'un logiciel à être réutilisé en tout ou en partie pour de nouvelles applications.
    - Le développement d'un nouveau système tire avantage de l'assemblage, par héritage ou composition, de modules intègres.

Périls de l'approche objet

- Mais, le paradigme et les langages de conception orientée objet seuls ne suffisent pas à assurer ces promesses.
  - Par exemple : l'encapsulation est très souvent mises à mal par l'utilisation d'attributs non privés ou d'accesseurs / mutateurs.
- Sans règles et principes supplémentaires, une conception finit inévitablement en « plat de spaghettis » qui induit :
  - **Rigidité**
    - Difficile à faire évoluer. L'extension impacte de nombreuses parties de l'application. Coût d'extension élevé.
  - **Fragilité**
    - Difficile à maîtriser. Le changement provoque des erreurs éparses non prévisibles. Modification de plus en plus risquée.
  - **Immobilité**
    - Difficile à réutiliser. Coût de développement élevé puisqu'il faut repartir de zéro à chaque fois.

Critères de qualité d'une conception

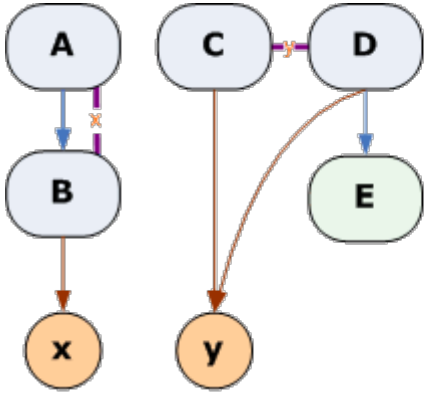
- Deux critères :
  1) Cohésion
  2) Couplage
- Portent sur la notion de module :
  - Classe, Méthode, Paquet, Composant ou Nœud.

Critère 1. Cohésion

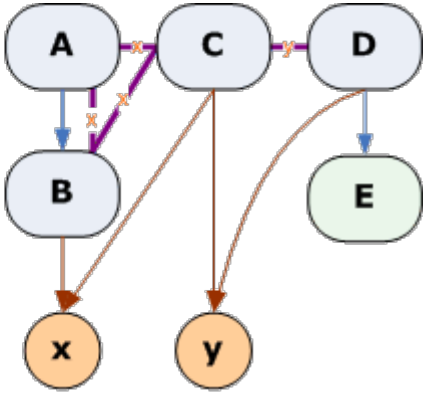
- Mesure l'étendue de la responsabilité d'un module.
  - Le degré d'indissociabilité des éléments d'un module.
- Questions associées
  - Quel est le but du module ?
  - Fait-il une ou plusieurs choses ?
- Indice
  - La liste des attributs est un bon indicateur du degré de cohésion.
- Il faut maximiser le degré de cohésion dans une conception.

Exemple d'une métrique

- Tight Class Cohesion (TCC)
  - Compter le nombre relatif de paires de méthodes qui accèdent directement aux même attributs de la classe.
    - Permet d'identifier les groupes de méthodes indépendantes.
  - Mesure : TCC = NDC / NP (*God class* : 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

- Mesure la force d'interaction entre les modules.
  - Le degré de dépendance de chaque module aux autres modules.
- Questions associées :
  - Comment les modules collaborent ensemble ?
  - Qu'ont-ils besoin de connaître les uns des autres?
- 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).
- Il faut minimiser le couplage dans une conception.

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.
  - *God class* : 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 – tourner en plat de spaghetti (*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 :
    - le développement,
    - la maintenance,
    - la réutilisation.
- Intégrer le fait que le code est malléable et doit sans cesse être remodelé pour garantir la meilleure qualité.