1) Introduction

Dans les années 80, la programmation structurée répond de moins en moins aux besoins des développeurs car les applications deviennent de plus en plus complexes :

- temps de programmation élevé,
- difficulté de maintenir les programmes,
- coût élevé de développement et de maintenance,
- peu de réutilisation du code déjà écrit,
- des problèmes de sécurité liés aux réseaux…

Des chercheurs (informaticiens, mathématiciens, linguistes, logiciens,…) ont tenté de savoir en quoi la programmation structurée avait atteint ses limites, et sont arrivés à la conclusion qu'il faudrait pouvoir mieux représenter les objets du monde réel. C'est ainsi que sont apparus les premiers langages Orientés Objets : Eiffel, C++, Java, C#…

La réutilisation du code est l'objectif principal, la technique consistant à construire des applications en assemblant des briques logicielles. Ces briques sont écrites directement par le programmeur ou alors issues de bibliothèques de classes d'objets, dans ce cas le programmeur utilise des boites noires dont il connaît les spécifications (l'interface) mais pas le code.

Tout ceci a comme conséquences la réduction de la taille du codage, une facilité de maintenance, mais demande par contre une adaptation des développeurs à la Programmation Orientée Objets (POO), non seulement dans les outils de conception (UML), les techniques de programmation mais également dans les méthodes de travail (plus collaboratif).

La programmation orientée objets repose sur trois concepts fondamentaux qu'on appelle "**le paradigme objet**" :

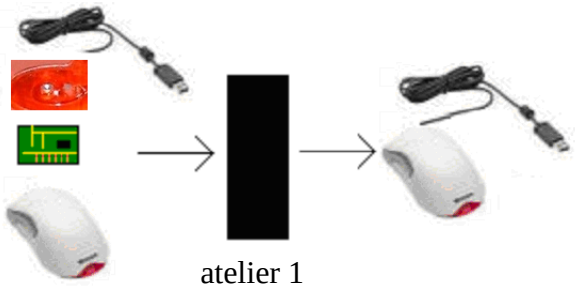
- L'encapsulation,
- l'héritage,
- le polymorphisme (**poly**=plusieurs, **morphe**=forme).

Paradigme : *Un **paradigme** est une représentation du monde, une manière de voir les choses, un modèle cohérent de vision du monde qui repose sur une base définie.*

2) Le concept de l'encapsulation

Exemple d'un objet du monde réel : une souris d'ordinateur

☞ Imaginons une usine qui assemble des souris :



pièces détachées assemblage souris

Une fois terminée, l'exemplaire de la souris possède des caractéristiques et des fonctionnalités :

• modèle : EasyMouse+ • n° de série : 98050509 • couleur : blanc	Comment elle est (propriétés)
• bouger • cliquer à droite • cliquer à gauche	Son savoir-faire (méthodes)
• détecter mouvement • envoyer impulsions électriques	

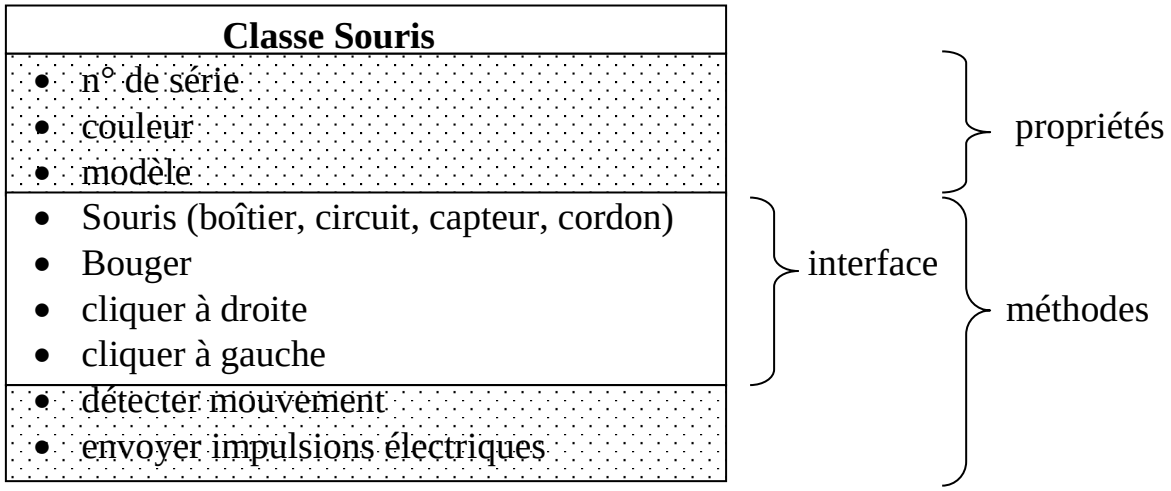
accessible (public)	non accessible de l'extérieur (privé)

Chaque souris est unique, c'est un "**objet**" !

Les caractéristiques et le savoir-faire sont regroupés (embarqués) dans la souris elle-même. Ce concept est appelé "**encapsulation**". Lorsque l'on clique à gauche, l'effet s'exécute sur la souris qu'on utilise (et pas sur celle du voisin pourtant du même modèle) !

Un utilisateur manipule la souris grâce à la partie publique de son savoir-faire, c'est "**l'interface**". les propriétés et les méthodes privées quant à elles, ne sont pas accessibles par l'utilisateur mais uniquement utilisées par le mécanisme interne de la souris.

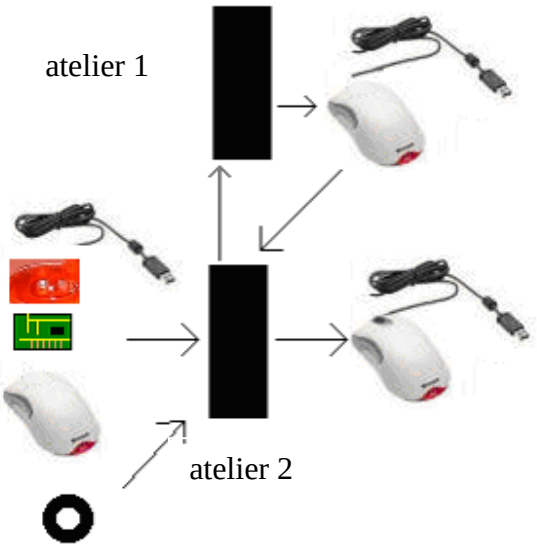
L'atelier 1 sert à construire une sorte de souris, il va utiliser pour cela la "**classe**" Souris qui est une description plus générale de ce que doit comporter tout objet souris (les propriétés et les méthodes).



Il faut noter qu'une méthode "Souris" a été rajoutée, c'est elle qui permet de créer un objet souris il s'agit du "**constructeur**" :

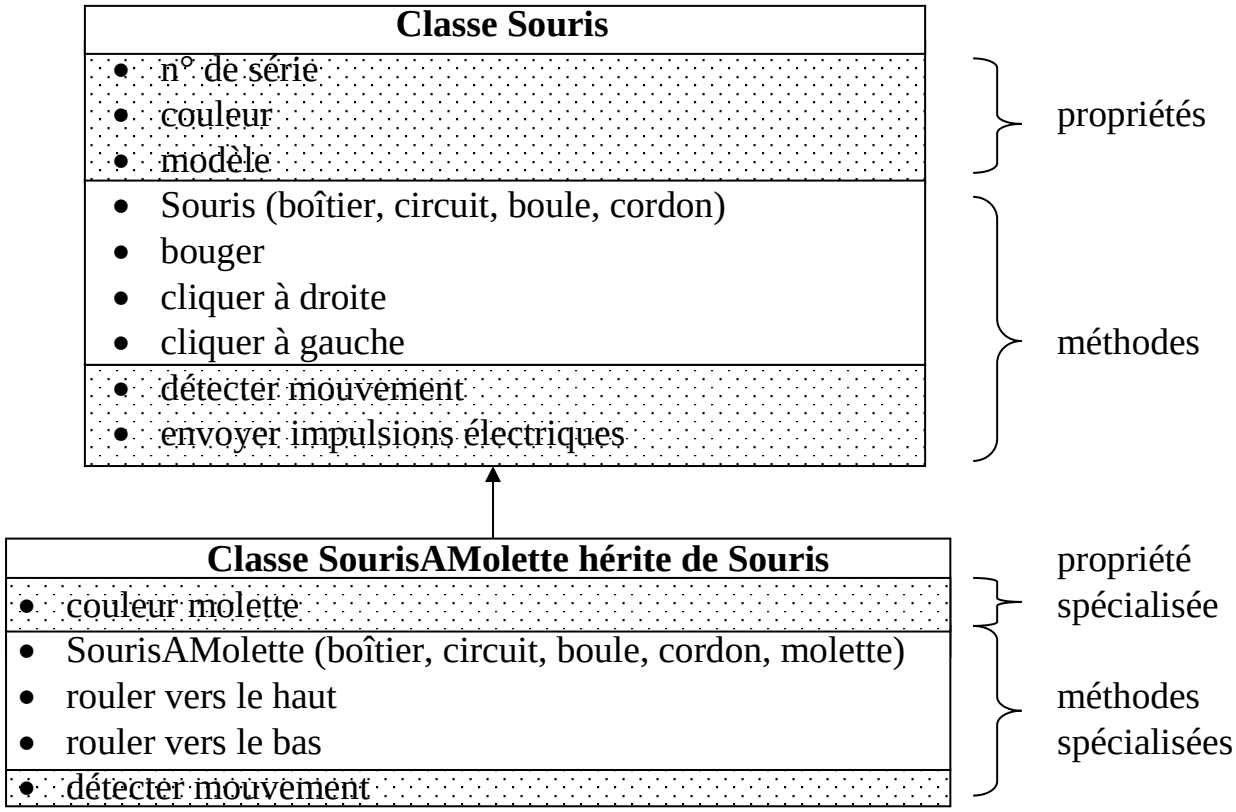
3) Les concepts de l'héritage et du polymorphisme

Imaginons maintenant le fonctionnement d'une nouvelle activité "souris à molette" dans un atelier 2. L'usine sait fabriquer des souris sans molette dans l'atelier 1, ce serait dommage de se passer de cette compétence !



Le principe est de sous-traiter l'assemblage de la souris dans l'atelier 1 et de rajouter simplement la molette et les méthodes qui permettent de la gérer dans l'atelier 2:

Nouvelle représentation des classes :



Remarque : La méthode 'détecter mouvement' de 'SourisAMolette' est capable de détecter en plus les mouvements de la molette.

Une souris à molette possède des propriétés et des méthodes spécialisées mais se comporte également comme une souris (puisque c'en est une). Si l'on clique à gauche avec une souris à molette, la méthode 'cliquer à gauche' de 'Souris' va s'appliquer automatiquement car 'SourisAMolette' a hérité des propriétés et des méthodes de 'Souris'. Ce concept est appelé "**héritage**".

La méthode 'détecter mouvement' de la classe 'sourisAMolette' a été redéfinie (enrichie) afin de pouvoir détecter les mouvements de la molette. Cette redéfinition est rendu possible par le concept du "**polymorphisme**". Lors de l'appel de cette méthode, c'est telle ou telle version qui va s'exécuter en fonction de la nature de l'objet qui va l'appeler (souris ou sourisAMolette).