Chapitre I

Introduction à la programmation orientée objets

Crise du logiciel

Constat

Les entreprises sont inondées de données.

La complexité des problèmes à traiter a augmenté considérablement.

⇒ Équipes de développement importantes.

Un seul individu ne peut comprendre toutes les subtilités d'un modèle de conception.

Conséquences

Le logiciel est généralement livré en retard.

Ses coûts excèdent souvent les budgets prévus.

Le logiciel est souvent défectueux et difficile à modifier.

Un programme utilisé sera modifié ⇒ On augmente sa complexité, à moins de lutter contre ce phénomène.

Crise du logiciel

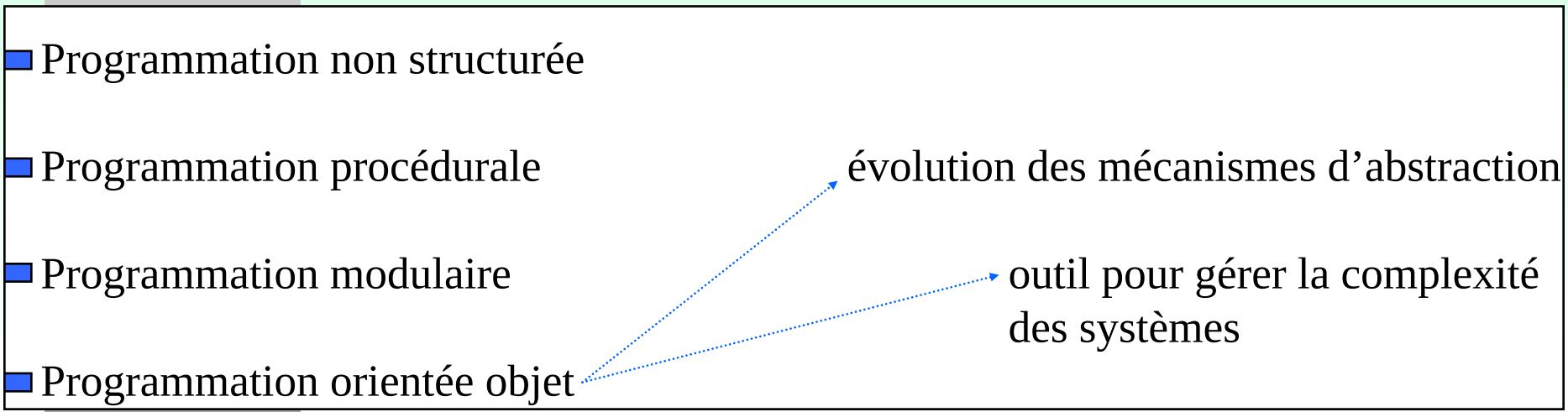
Défi majeur

Construire rapidement du logiciel de qualité.

- L'analyse, la conception et la programmation orientées objet visent à relever ce défi.
- Cela ne résoudra pas la crise du logiciel mais cela diminue la complexité des programmes.

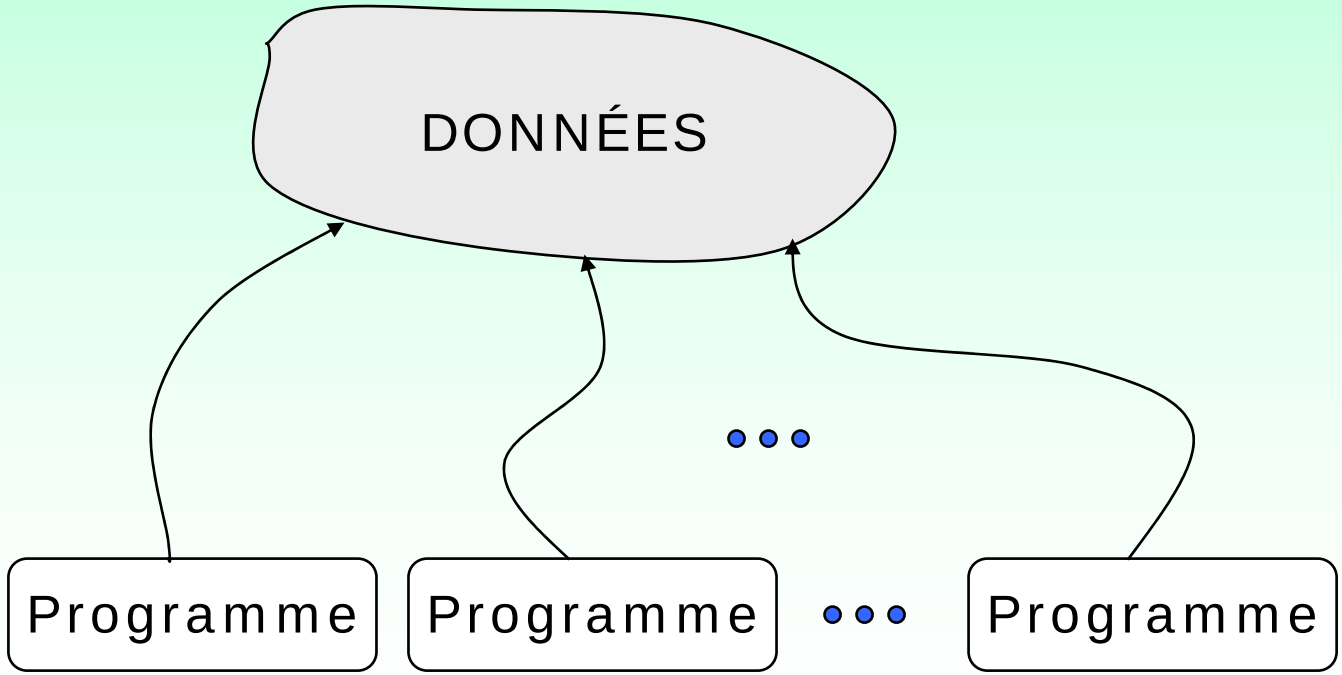
Évolution des langages de programmation

- On retrouve 2 aspects dans les programmes : les algorithmes et les données.
- Ceci n'a pas changé au cours de l'histoire de l'informatique.
- Ce qui a évolué, c'est la relation existant entre eux, qui s'appelle « paradigme de programmation ».



Programmation non structurée

- Les premiers langages de programmation étaient généralement constitués d'une suite d'instructions s'exécutant de façon linéaire: programmation non structurée.



Programmation non structurée

Caractéristiques (Yves Roy):

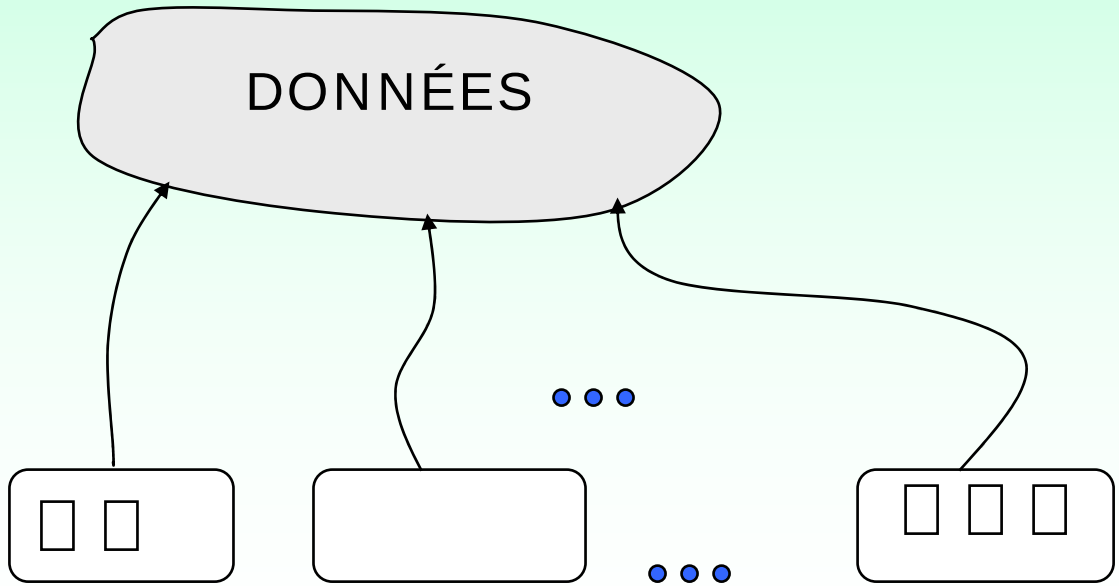
- Difficile d'écrire de grosses applications
- Difficile à maintenir la cohérence des données dans le temps
- Risque associé au partage de données à grande échelle
- Duplication des traitements communs. Cela ne favorise pas la réutilisation du code.
- Données globales
- Couplage important des données

etc.

Programmation procédurale

- D'autres langages plus évolués sont apparus (PASCAL, C, FORTRAN, etc.) préconisant une approche procédurale.

Il s'agit de découper un programme en une série de fonctions lesquelles ont pour but de réaliser un traitement particulier : programmation procédurale.



Programmation procédurale

Avantages:

- facilite la maintenance d'un programme
- favorise la modularité des programmes, un problème complexe pouvant être découpé en plusieurs sous-problèmes
- ces fonctions peuvent constituer les éléments d'une boîte à outils qui pourront être réutilisés pour les nouveaux développements.
- introduction de mécanismes de passage de paramètres.

Désavantages:

- dissocie les données des fonctions qui les manipulent.
- données globales
- couplage élevé des données
- aucune technique permettant de spécialiser des fonctions existantes.

Définition d'une pile

Programmation procédurale

```
#include <iostream.h>
```

Données

```
const int taille_maximale = 100;
float vecteur[taille_maximale ];
int haut = 0;
```

Fonctions

```
bool Pile_vide()                    { return (haut == 0); }
bool Pile_pleine()                  { return (haut == taille_maximale ); }
void Initialiser_pile()             { haut = 0; }
void Inserer_pile(float valeur)     { vecteur[haut] = valeur; haut += 1; }
float Acces_pile()                  { return vecteur[haut - 1];}
float Enlever_pile()                { haut = haut – 1; return vecteur[haut];}
```

Utilisation

```
void main ();
{       Initialiser_pile();  Inserer_pile(2.3f); Inserer_pile(3.4f); Inserer_pile(6.3f);
        while (!Pile_vide()) { cout << Enlever_pile() << endl; }
}
```

Programmation modulaire

Pour pallier à ces inconvénients, on propose une décomposition en modules.

- Les modules communiquent entre eux par des appels de fonctions seulement et non en partageant des données.
- Des données peuvent être échangées en passant par les paramètres des fonctions.

Définition d'une pile

Interface

```
bool Pile_vide();
bool Pile_pleine();
void Initialiser_pile();
void Inserer_pile(float valeur);
float Acces_pile();
float Enlever_pile();
```

Programmation modulaire

Implantation

```
const int taille_maximale = 100;
float vecteur[taille_maximale];
int haut = 0;

bool Pile_vide()                    { return (haut == 0); }
bool Pile_pleine()                  { return (haut == taille_maximale ); }
void Initialiser_pile()             { haut = 0; }
void Inserer_pile(float valeur)     { vecteur[haut] = valeur; haut += 1; }
float Acces_pile()                  { return vecteur[haut - 1];}
float Enlever_pile()                { haut = haut – 1; return vecteur[haut];}
```

Programmation modulaire

Utilisation

```
#include "Pile.h"
#include <iostream.h>

void main ();
{       Initialiser_pile();  Inserer_pile(2.3f); Inserer_pile(3.4f); Inserer_pile(6.3f);
        while (!Pile_vide()) { cout << Enlever_pile() << endl; }
}
```

Programmation modulaire

Avantages:

- Les données locales au module ne peuvent être endommagées accidentellement de l'extérieur.
- L'utilisation d'un module ne nécessite pas que l'on sache comment le module est implanté.
- Compilation séparée.
- Compréhension plus facile d'un système.
- Facilite le travail des équipes de développement.
- Un début d'encapsulation d'information à l'intérieur d'un module.
- Il existe une portion publique accessible à l'extérieur du module.
- Il existe une portion privée accessible à l'intérieur du module.

Désavantages:

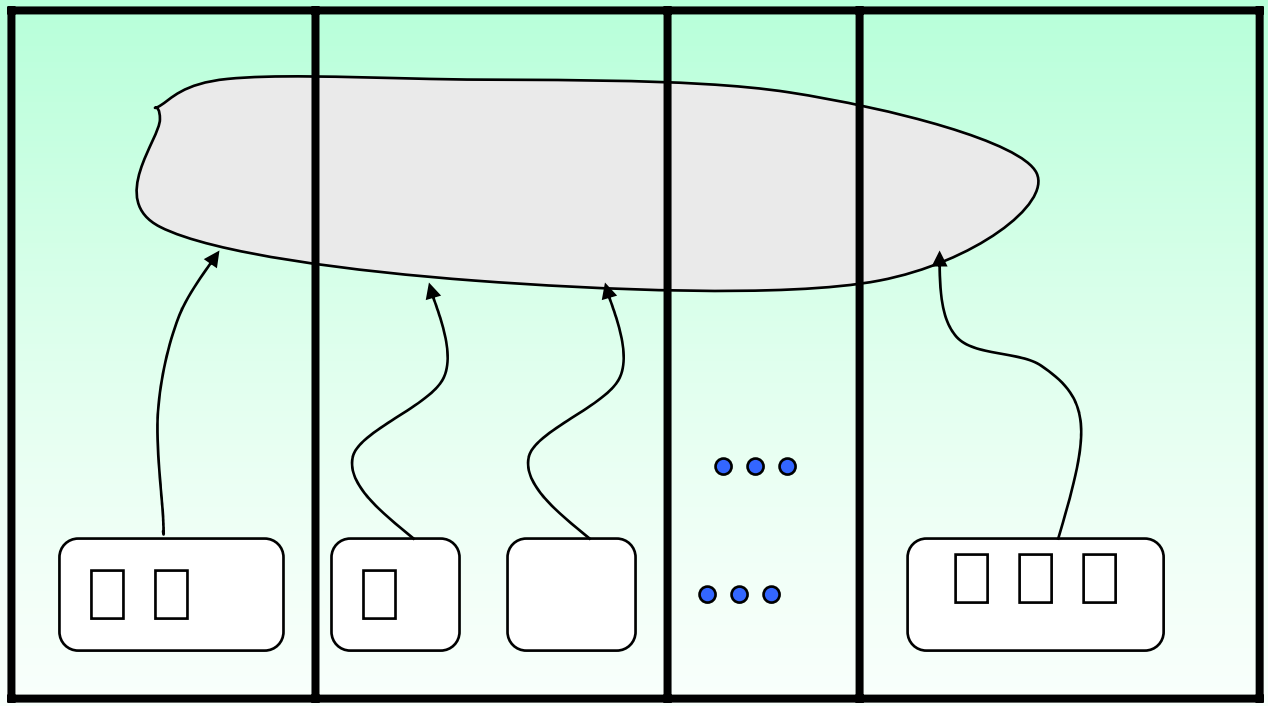
- Une seule pile à la fois.
- Besoin de 2 piles ⇨ duplication du code

Programmation modulaire

MODULES

DONNÉES

Sous-programmes
et
programmes



Types de données abstraits

- Type de donnée défini par un programmeur et manipulé comme un type de donnée de base du langage.
- Implantation : utilisation du concept de module tout en permettant de créer plusieurs instances de ce type.
- Les opérations qui peuvent être utilisées pour manipuler une variable de ce type doivent être disponibles.
- Les données faisant partie de ce type doivent être protégées :
  elles sont traitées uniquement par les opérations définies plus haut.
- On doit pouvoir créer plusieurs variables (instances) de ce type.
- Le type de donnée abstrait peut être implanté à l'aide de la notion de classe.

Définition d'une pile

Types de données abstraits

Interface

```
const int taille_maximale = 100;
struct Pile
{
        float vecteur[taille_maximale];
        int haut = 0;
};

bool Pile_vide(Pile * P);
bool Pile_pleine(Pile * P);
void Creer_pile(Pile * P);
void Inserer_pile(Pile * P, float valeur);
float Acces_pile(Pile * P);
float Enlever_pile(Pile * P);
```

Définition d'une pile

Types de données abstraits

Implantation

```
#include "Pile.h"

bool Pile_vide(Pile * P)                    { return ((*P).haut == 0); }
bool Pile_pleine(Pile * P)                  { return ((*P).haut == taille_maximale ); }
void Creer_pile(Pile * P)                   { (*P).haut = 0; }
void Inserer_pile(Pile * P, float valeur)       { (*P).vecteur[(*P).haut] = valeur;
                                                  (*P).haut += 1; }
float Acces_pile(Pile * P)                  { return (*P).vecteur[(*P).haut - 1];}
float Enlever_pile(Pile * P)                { (*P).haut -= 1;
                                              return (*P).vecteur[(*P).haut];}
```

Définition d'une pile

Types de données abstraits

Utilisation

```
#include "Pile.h"
#include <iostream.h>

void main ()
{
        Pile Q;
        Initialiser_pile(&Q);
        Inserer_pile(&Q, 2.3f); Inserer_pile(&Q, 3.4f); Inserer_pile(&Q, 6.3f);
        while (!Pile_vide(&Q)) { cout << Enlever_pile(&Q) << endl; }
}
```

Programmation orientée objets

- Programmation orientée objets

Buts:
- améliorer la réutilisation des outils existants
- favoriser la spécialisation de ces mêmes outils.
- s'éloigner de la machine et se rapprocher du problème à résoudre.
- développer une approche mieux adaptée à la résolution de problèmes.
- Nouvelle façon de penser la décomposition de problèmes et l'élaboration de solutions.

⟹ Une nouvelle entité est créée, appelée **objet**, afin de regrouper les fonctions et les données.

⟹ Peu ou pas de variables globales

⟹ Hiérarchie de données ➡ Hiérarchie de fonctions ➡ Hiérarchie de classes

Concepts de la programmation orientée objets

L’approche objet s’appuie sur 3 techniques fondamentales:

- **l’encapsulation**
- **l’héritage**
- **le polymorphisme**

L'ENCAPSULATION

EXEMPLE :

- Mon chat miaule, marche et mange :

  Nous connaissons les manifestations extérieures du miaulement, mais la façon dont le son est produit nous est masqué.

- Le chat devient alors une **abstraction** dont on ne montre seulement que quelques comportements.

- Une interface représente cette abstraction.

  L'interface énonce les services offerts et les détails de fonctionnement sont cachés.

- L'emphase est mise sur la compréhension des éléments importants, sans se soucier de l'implantation.

L ’ENCAPSULATION

- Permet de réunir au sein d’une même entité appelée classe, des données membres (variables ou attributs) et des traitements (fonctions membres ou comportements).
- Les objets de ces classes ont la propriété de masquer l’information.
- Un système de protection permet de contrôler l’accès aux données et traitements de la classe.
  - On peut cacher le fonctionnement interne d’une classe et éviter que les objets de cette classe soient utilisés de façon non conforme.
- Vu de l’extérieur, l’interface d’un objet (aussi appelé protocole) constitue la liste de tous les services (i.e. les méthodes) auxquels les autres objets ont accès.
  - On ne peut modifier l’état d’un objet directement, on doit passer par l’interface.

L ’ENCAPSULATION

Catégories d’opérations dans l’interface d’un objet

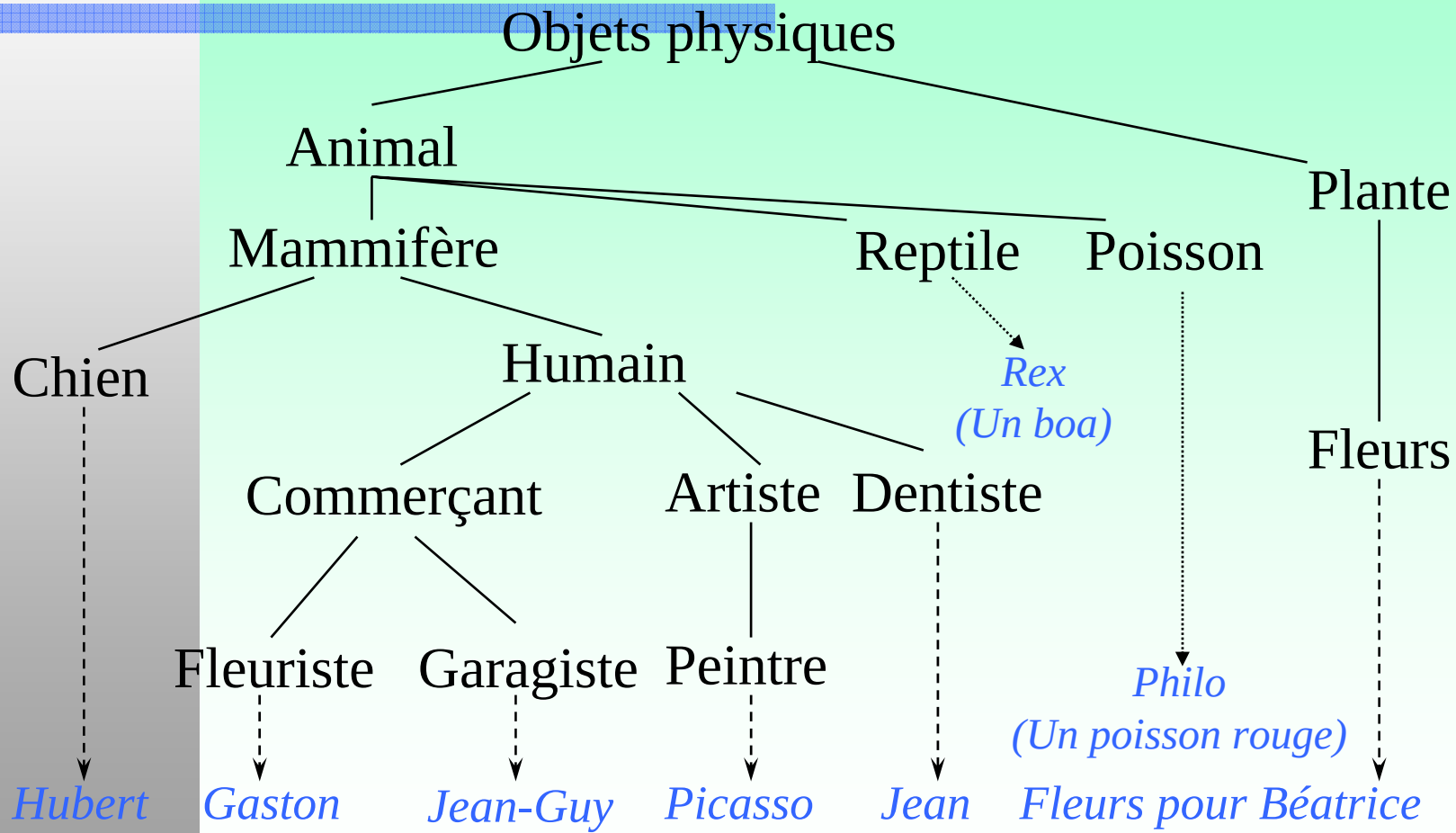
- **Constructeurs** : créer et initialiser un nouvel objet.
- **Destructeurs** : libérer les ressources et détruire l’objet.
- **Accès** : accède à de l’information mais n’altère pas l’état de l’objet.
- **Assignation** : altère l’état d’un objet.
- **Opérateur de comparaison** : comparer des objets entre eux.
- **Itérateur** : permet de parcourir une collection d’objets dans un ordre donné.
- **Copie et clonage** : copie à partir d’un autre objet ou clonage de l’objet lui-même.
- **Entrée / sortie**.

L'HÉRITAGE

EXEMPLE :

- Plusieurs **comportements** d'un fournisseur de bois de chauffage sont connus, non pas parce que je sais qu'il est fournisseur de bois de chauffage, mais aussi parce qu'il est un **commerçant**.
- Je m'attend à ce qu'il me demande de l'argent pour ses services et qu'il me donne un reçu en échange.
- Ces comportements sont aussi vrais pour l'épicier, le coiffeur ou le préposé au club vidéo.
- Les comportements d'un commerçant sont aussi ceux de l'épicier, du coiffeur, …
- En organisant nos connaissances ainsi, cela permet une compréhension rapide du problème.

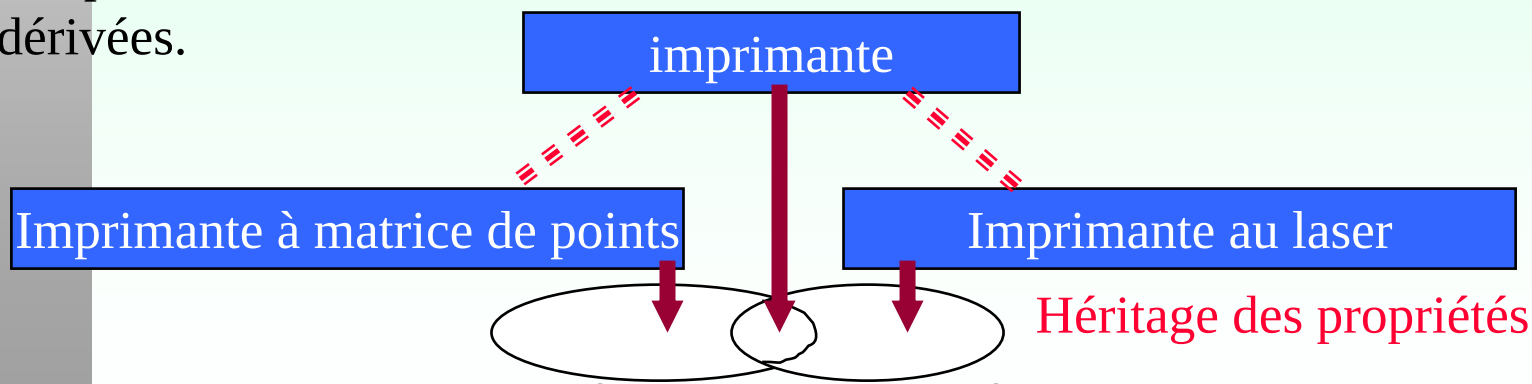
L'HÉRITAGE


Objets physiques
Animal
Plante
Mammifère
Reptile
Poisson
Chien
Humain
Rex
(Un boa)
Fleurs
Commerçant
Artiste
Dentiste
Fleuriste
Garagiste
Peintre
Philo
(Un poisson rouge)
Hubert
Gaston
Jean-Guy
Picasso
Jean
Fleurs pour Béatrice

L'HÉRITAGE

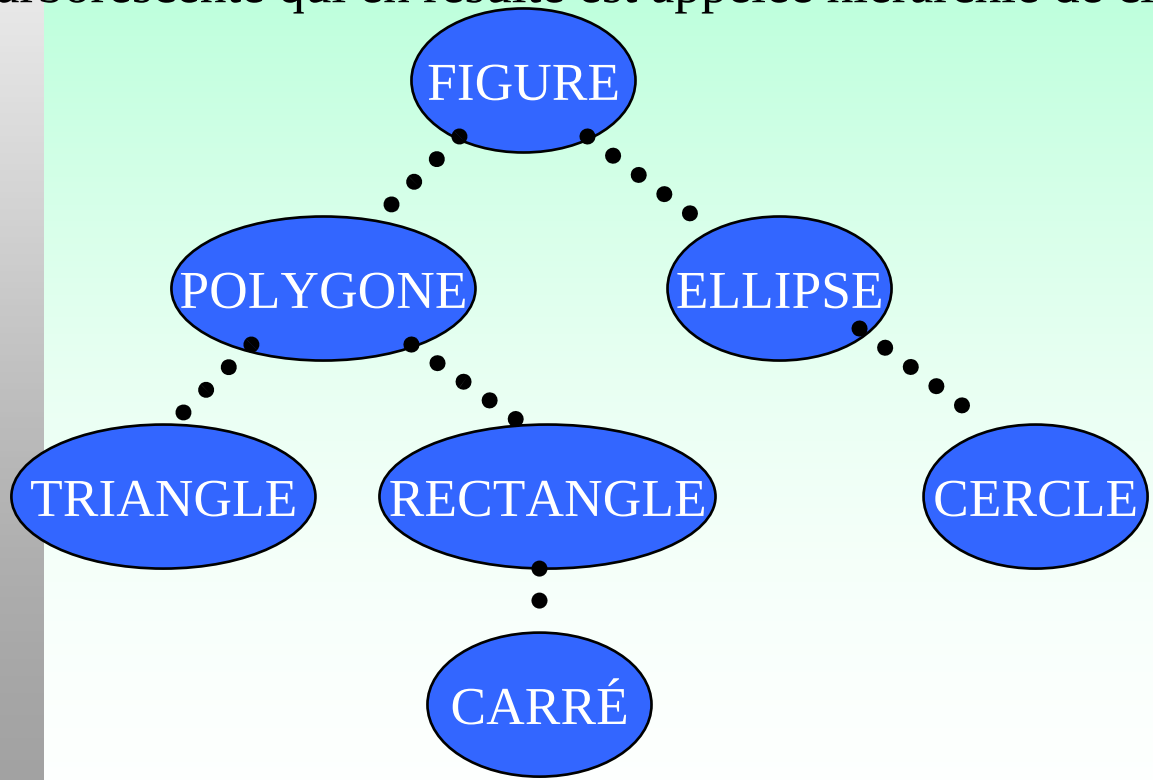
- Permet de réutiliser les classes existantes
  - On peut définir une nouvelle classe à partir d'une autre.
    - La classe dérivée pourra bénéficier
      - des attributs (variables)
      - des comportements (fonctions)

      de la classe de base dont elle hérite.
- Permet de spécialiser une classe de base en renfermant d'autres éléments dans les classes dérivées.



Hiérarchie de classes

- Une classe peut hériter d'une classe qui est elle-même une sous-classe et ainsi de suite.
- La structure arborescente qui en résulte est appelée hiérarchie de classes.



LE POLYMORPHISME

- EXEMPLE :

- Les chiens, les serpents et les poissons partagent tous un comportement commun hérité de la classe des animaux : ils ont la capacité de **se mouvoir**.

- Tous le font, mais chacun à sa manière :
  - le chien marche
  - le serpent rampe
  - le poisson nage.

- Si on donne l'ordre de se mouvoir, chaque animal va pouvoir répondre à cet ordre, **à sa manière propre**.

- S'applique aux fonctions membres des classes.
- Permet à 2 objets de classes différentes de réagir différemment au même appel de méthode.

LES CRITÈRES DE QUALITÉ D'UN LOGICIEL

Les problèmes que les scientifiques sont amenés à résoudre sont souvent extrêmement complexes et généralement très évolutifs.

On doit prendre en compte plusieurs critères de qualité d'un logiciel.

Étude plus exhaustive dans [Bouché Max, La démarche objet Concepts et outils. Chap. 5, section 4, 1994].

Fiabilité

Aptitude d'un logiciel à se comporter de manière valide et robuste.

Réalise exactement les tâches définies par sa spécification

Fonctionne même dans des conditions anormales.

LES CRITÈRES DE QUALITÉ D'UN LOGICIEL

Étude plus exhaustive dans [Meyer B., Object-Oriented Software Construction. Chap. 1, 1997].

Compatibilité	Aptitude des logiciels à pouvoir être combinés les uns avec les autres.
Extensibilité	Facilité d'adaptation d'un logiciel aux changements de spécification.
Réutilisabilité	Aptitude d'un logiciel à pouvoir être réutilisé en tout ou en partie pour de nouvelles applications. Soit **chercher**(Element x, Table T),

- variations de type
- variations dans le choix des structures de données et algorithmes
- regroupement de procédures
- indépendance p/r à la représentation des données

- Utilisez une approche « jeu de construction » quand vous concevez des programmes.
- Servez-vous des pièces existantes chaque fois que c'est possible.
- La réutilisation de logiciels est fondamentale à la POO.

LES CRITÈRES DE QUALITÉ D'UN LOGICIEL

Efficacité	Utilisation optimale des ressources matérielles.
Portabilité	Facilité avec laquelle le système peut être transféré dans divers environnements logiciels et matériels.

Utiliser les fonctions et les classes de la bibliothèque standard au lieu d'en écrire des versions comparables assurera efficacité et portabilité.

Cohérence	Une classe décrit une seule abstraction. Toutes les opérations doivent remplir un seul objectif.

Exemple : Une classe Vecteur qui renfermerait une méthode « Affichage d'un point à l'écran. »

Intégrité	Aptitude du système à se prémunir contre des accès ou modifications non autorisés.
Logiciel complet	Support de toutes les opérations qui ont du sens.

LES CRITÈRES DE QUALITÉ D'UN LOGICIEL

Opérations primitives

Les opérations ne sont pas décomposables en plus petites opérations.

Exemple : Une classe Vecteur qui renfermerait une méthode « Normaliser » laquelle possède localement une fonction qui calcule la norme d'un vecteur.

Interface consistante

Les méthodes doivent être consistantes entre elles sur la base de leur nom, arguments, valeur de retour, etc.

Exemple : La présence des méthodes suivantes :
« Acces_Matricule » et « Set_Nom ».

Facilité d'apprentissage et d'utilisation

Programmes simples Évitez les usages bizarres qui « étirent » le langage inutilement

USAGE DE C++

- C++ est un langage supportant plusieurs paradigmes.

- C++ permet d'utiliser l'approche procédurale de C et possède toutes les caractéristiques d'un langage orienté objets.

Quelques langages orientés objet (Yves Roy)

ABCL/1	ABE	Acore	Act	Actor
Actors	Actra	Ada	Argus	ART
Berkeley Smalltalk	Beta	Blaze	Brouhaha	C with Classes
C++	C_talk	Cantor	Clascal	Classic Ada
CLOS	Cluster 86	Common Loops	Common Objects	Common ORBIT
Concurrent Prolog	Concurrent Smalltalk	CSSA	CST	Director

Quelques langages orientés objet (Yves Roy)

Distributed Smalltalk	Eiffel	Emerald	Exper Common Lisp	Extended Smalltalk
Felix Pascal	Flavors	FOOPlog	FOOPS	FRL
Galileo	Garp	GLISP	Gypsy	Hybrid
Inheritance	InnovAda	Intermission	Java	KL-One
KRL	KRS	Little Smalltalk	LOOPS	Lore
Mace	MELD	Mjolner	ModPascal	Neon

Quelques langages orientés objet (Yves Roy)

New Flavors	NIL	O-CPU	OakLisp	Oberon
Object Assembler	Object Cobol	Object Lisp	Object Logo	Object Oberon
Object Pascal	Objective-C	ObjVLisp	OOPC	OOPS+
OPAL	Orbit	Orient84/K	OTM	PCOL
PIE	PL/LL	Plasma II	POOL-T	PROCOL
Quick Pascal	QuiclTalk	ROSS	SAST	SCOOP

Quelques langages orientés objet (Yves Roy)

SCOOPS	Self	Simula	SINA	Smalltalk
Smalltalk AT	Smalltalk V	Smallworld	SPOOL	SR
SRL	STROBE	T	Tellis/Owl	Turbo Pascal 5.x
Uniform	UNITS	Vulcan	SLISP	Zoom/VM