

Notes de cours

Ecole Nationale d'Ingénieurs de Brest

Programmation par objets

— Les concepts —

J. Tisseau

– 1994 –

Tables des matières

■ La qualité du logiciel	3
■ Une société de modules	11
■ Les types abstraits de données	21
■ Les objets	31
■ Analyse et conception orientées objets	43
■ Les langages orientés objets	51
■ Bibliographie	65

La qualité du logiciel

- **La qualité du logiciel**
 1. Définition
 2. Validité et robustesse
 3. Extensibilité et réutilisabilité
 4. Compatibilité et intégrité
 5. Efficacité et portabilité
 6. Ouvert ou fermé ?
- Une société de modules
- Les types abstraits de données
- Les objets
- Analyse et conception orientées objets
- Les langages orientés objets
- Bibliographie

Définition

La qualité d'un produit est son aptitude à satisfaire aux besoins de l'utilisateur au moindre coût et dans les meilleurs délais.

La qualité d'un logiciel concerne aussi bien les phases d'analyse, de conception, de développement que de maintenance du logiciel.

On estime souvent que 70% du coût d'un logiciel est consacré à sa maintenance.

Validité et robustesse

Validité : aptitude du logiciel à réaliser exactement les tâches définies par sa spécification.

Problème : les spécifications sont difficiles à formaliser rigoureusement.

Robustesse : aptitude d'un logiciel à fonctionner même dans des conditions anormales.

La robustesse, c'est l'assurance que dans des situations anormales le logiciel ne va pas déclencher une "catastrophe".

Extensibilité et réutilisabilité

Extensibilité : faculté d'adaptation d'un logiciel aux changements de spécification.

- **simplicité de la conception** : une architecture simple sera toujours plus facile à modifier qu'une architecture complexe.
- **décentralisation** : plus les modules d'une architecture logicielle sont autonomes, plus il est probable qu'une modification simple n'affectera qu'un seul module, ou un nombre restreint de modules, plutôt que de déclencher une réaction en chaîne sur tout le système.

Réutilisabilité : aptitude d'un logiciel à être réutilisé en tout ou en partie pour de nouvelles applications.

Ne pas réinventer la roue !

Programmer *moins* pour programmer *mieux* !

Compatibilité et intégrité

Compatibilité : aptitude des logiciels à pouvoir être combinés les uns avec les autres.

La compatibilité ne peut être obtenue que par une conception homogène et l'utilisation de conventions normalisées pour la communication entre programmes.

- formats de fichiers normalisés (comme les fichiers UNIX)
- structures de données normalisées (comme les listes Lisp)
- interfaces normalisées (comme sur le Mac)
- ...

Intégrité : aptitude du logiciel à protéger ses différents composants contre des accès ou des modifications non autorisés.

Efficacité et portabilité

Efficacité : aptitude du logiciel à utiliser de manière optimale les ressources du matériel (processeurs, mémoires internes et externes, voies de communication, ...).

Portabilité : facilité avec laquelle un logiciel peut être adapté à différents environnements matériels et logiciels.

Ouvert ou fermé ?

Fermeture	Ouverture
Validité Réutilisabilité Intégrité Efficacité	Robustesse Extensibilité Compatibilité Portabilité

Le compromis par les modules

- Il faut produire du logiciel plus modulaire.
- Un module doit être ouvert *et* fermé.

Programme \equiv Société de modules

Qualité du logiciel

Une société de modules

- La qualité du logiciel
- **Une société de modules**
 1. Communications inter-modules
 2. Décomposabilité modulaire
 3. Composabilité modulaire
 4. Compréhensibilité modulaire
 5. Continuité modulaire
 6. Protection modulaire
 7. Vers les objets ...
 8. Conception par objets
- Les types abstraits de données
- Les objets
- Analyse et conception orientées objets
- Les langages orientés objets
- Bibliographie

Communications inter-modules

1. Les modules doivent correspondre à des unités syntaxiques du langage.
2. Tout module doit communiquer avec aussi peu d'autres que possible.
3. Si deux modules communiquent, ils doivent échanger aussi peu d'information que possible.
4. Chaque fois que deux modules communiquent, cela doit ressortir clairement de la définition de ces modules.
5. Toute information concernant un module doit être privée, sauf si elle est explicitement déclarée publique (*masquage de l'information*).

Dans cette société de modules, les modules se parlent peu, leur conversation est limitée à peu de mots et leurs dialogues sont publics et se font à haute voix !

Décomposabilité modulaire

Décomposer un nouveau problème en plusieurs sous-problèmes dont les solutions peuvent être recherchées séparément.

Exemple : la conception descendante

Contre-exemple : un module d'initialisation

Composabilité modulaire

Production d'éléments de logiciel qui peuvent être combinés librement les uns avec les autres pour produire de nouveaux systèmes, éventuellement dans un environnement très différent de celui pour lequel ils ont été initialement conçus.

Exemple : bibliothèques de sous-programmes

Contre-exemple : préprocesseurs

Compréhensibilité modulaire

Produire des modules dont chacun peut être compris isolément par un lecteur humain.

Exemple : commandes UNIX

Contre-exemple : dépendances séquentielles

Continuité modulaire

Une petite modification de la spécification du problème n'amène à modifier qu'un seul module du système.

De telles modifications ne doivent pas avoir d'impact sur l'architecture du système, c'est-à-dire sur les relations entre les modules.

Exemple : constantes symboliques

Contre-exemple : tableaux statiques

Protection modulaire

Une condition anormale se produisant à l'exécution d'un module doit rester localisée à ce module.

Exemple : validation des entrées à partir de la source

Contre-exemple : exceptions indisciplinées

Vers les objets ...

1. Les systèmes sont découpés en modules sur la base de leurs *structures de données*.
2. Les modules doivent être décrits comme des implémentations de *types abstraits de données*.
3. Tout type est un module et tout module est un type.
4. Un module peut être défini comme une extension ou une restriction d'un autre module (*héritage*).
Un module peut hériter de plus d'un autre module (*héritage multiple*).
5. Une même entité de programme peut faire référence, au cours de son exécution, à différents modules (*polymorphisme*).

Lors de l'exécution, une opération doit pouvoir avoir des comportements différents dans des modules différents (*liaison dynamique*).

Conception par objets

La conception par objets est une méthode de construction de logiciel qui fonde l'architecture des systèmes sur les objets qu'ils manipulent plutôt que sur la fonction qu'ils assurent.

Abstraire par les données

Module \equiv Type

La programmation par objets est la construction de logiciels sous forme de collections structurées d'implémentations de types abstraits de données.

Société de modules

Types abstraits de données

- La qualité du logiciel
- Une société de modules
- **Les types abstraits de données**
 1. **T.A.D.**
 2. **Exemple de TAD : les piles**
 3. **Services d'un TAD**
 4. **Sémantique d'un TAD**
 5. **Contrôle/Commande**
 6. **Types concrets de données**
- Les objets
- Analyse et conception orientées objets
- Les langages orientés objets
- Bibliographie

T.A.D.

Question : Comment définir une classe d'objets ?

- complètement
- précisément
- indépendamment de toutes représentations physiques (de toutes implémentations informatiques)

Réponse : Par les types abstraits de données.

Un TAD est une classe de structures de données décrite par un ensemble de services disponibles et par les propriétés formelles de ces services.

TAD :

...

Services :

...

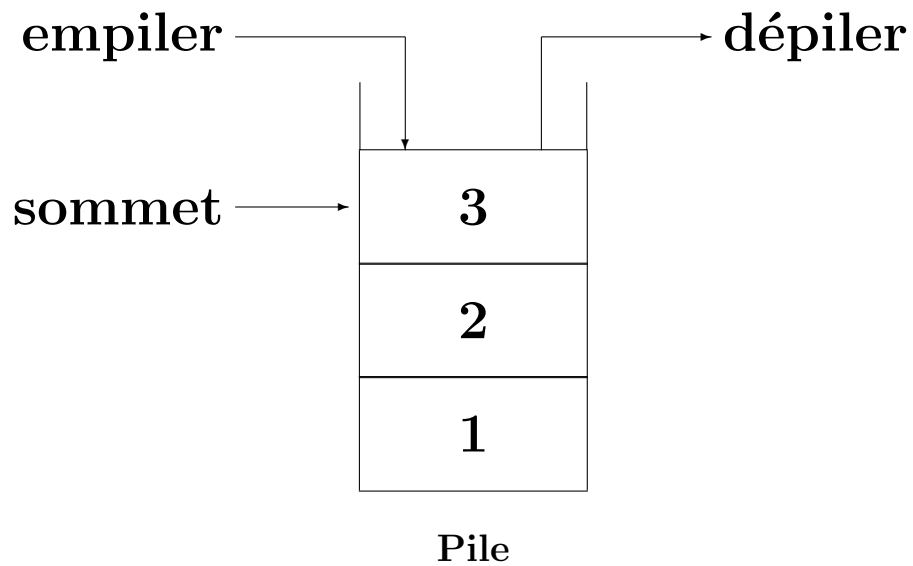
Préconditions :

...

Axiomes :

...

Exemple de TAD : les piles



Pile : structure LIFO (*Last In, First Out*)

TAD Pile

TAD : $Pile[X]$

Services :

$empty$: $Pile[X] \longrightarrow Booleen$

top : $Pile[X] \longrightarrow X$

$push$: $X \times Pile[X] \longrightarrow Pile[X]$

pop : $Pile[X] \longrightarrow Pile[X]$

Préconditions :

$\forall p \in Pile[X],$

top : $\neg empty(p)$

pop : $\neg empty(p)$

Axiomes :

$\forall x \in X, \forall p \in Pile[X],$

$\neg empty(push(x, p))$

$top(push(x, p)) = x$

$pop(push(x, p)) = p$

Services d'un TAD

Les services d'un TAD T sont spécifiés comme des fonctions mathématiques

$$f : A_1 \times A_2 \times \cdots \times A_m \longrightarrow B_1 \times B_2 \times \cdots \times B_n$$

Observateur : une fonction f pour laquelle T n'apparaît qu'à gauche de la flèche accède aux attributs des éléments de ce type.

$$\cdots \times T \longrightarrow \cdots$$

$$\begin{aligned} \text{empty} & : \text{Pile}[X] \longrightarrow \text{Booleen} \\ \text{top} & : \text{Pile}[X] \longrightarrow X \end{aligned}$$

Transformateur : une fonction f où T apparaît des deux côtés de la flèche donne de nouveaux éléments du même type T à partir des éléments précédents.

$$\cdots \times T \longrightarrow T \times \cdots$$

$$\begin{aligned} \text{push} & : X \times \text{Pile}[X] \longrightarrow \text{Pile}[X] \\ \text{pop} & : \text{Pile}[X] \longrightarrow \text{Pile}[X] \end{aligned}$$

Sémantique d'un TAD

Donner une signification (une sémantique) aux différents services du TAD.

Préconditions : Les fonctions ne sont pas nécessairement définies pour tous les objets du TAD considéré (fonctions partielles).

Les préconditions indiquent les conditions d'applicabilité de chaque fonction partielle.

$$\begin{aligned} &\forall p \in \text{Pile}[X], \\ \text{top} & : \neg \text{empty}(p) \\ \text{pop} & : \neg \text{empty}(p) \end{aligned}$$

Sémantique d'un TAD

Axiomes : Le rôle des axiomes est d'adjoindre des propriétés sémantiques aux TAD.

$$\begin{aligned} \forall x \in X, \forall p \in \text{Pile}[X], \\ \neg \text{empty}(\text{push}(x, p)) \\ \text{top}(\text{push}(x, p)) = x \\ \text{pop}(\text{push}(x, p)) = p \end{aligned}$$

TAD = Signature + Sémantique

Signature : nom et services

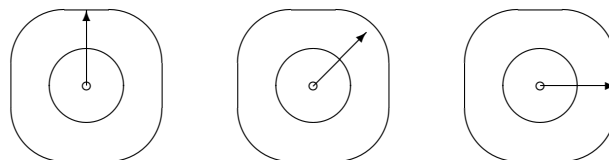
Sémantique : axiomes et préconditions

Contrôle/Commande

Observateurs



Transformateurs



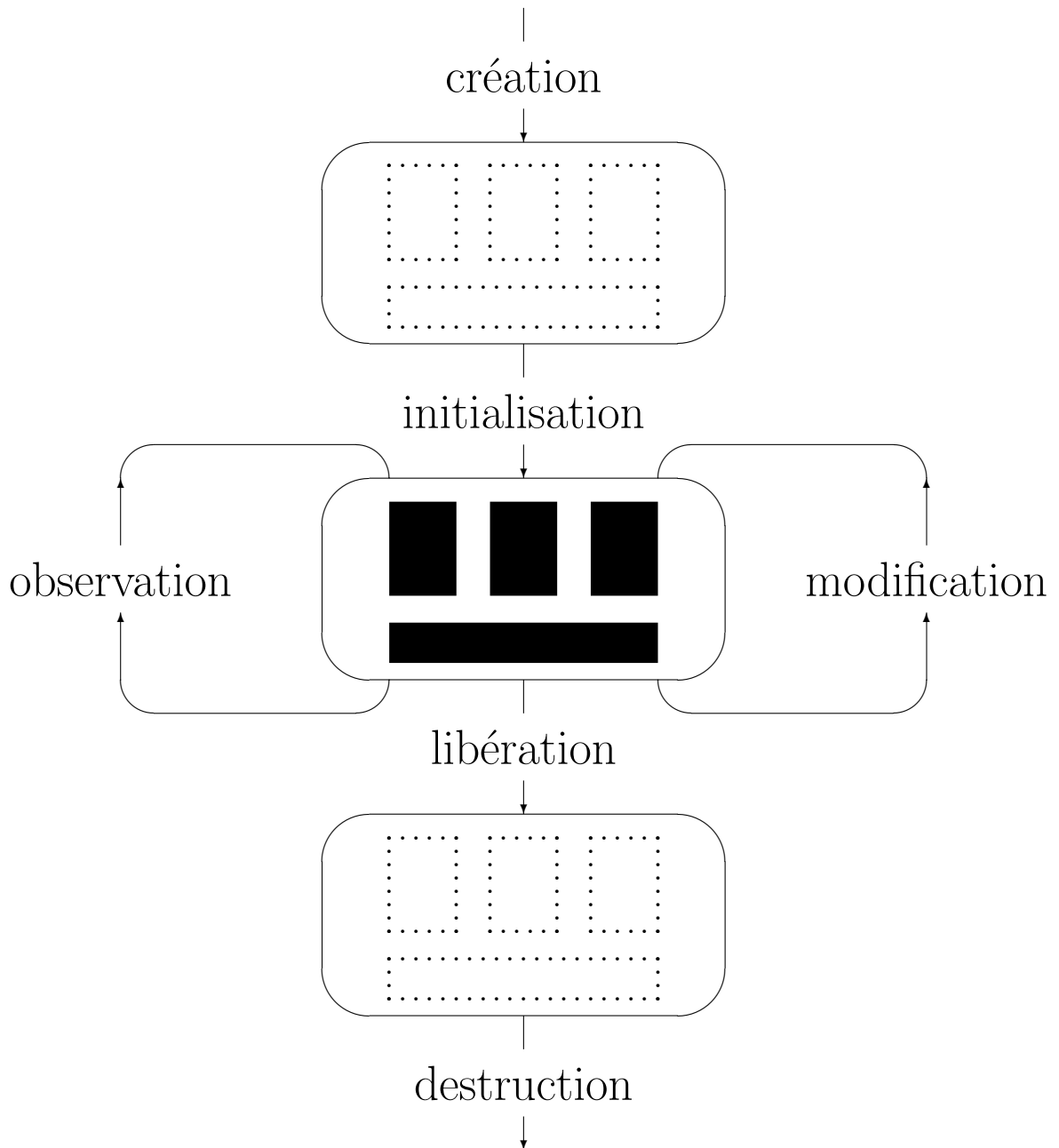
- Un observateur (*contrôle*) donne certaines indications sur l'état interne du TAD.

$$top : Pile[X] \longrightarrow X$$

- Un transformateur (*commande*) modifie l'état interne du TAD.

$$push : X \times Pile[X] \longrightarrow Pile[X]$$

Types concrets de données



Les objets

- La qualité du logiciel
- Une société de modules
- Les types abstraits de données
- **Les objets**
 1. Etat et comportement
 2. Les classes
 3. L'instanciation
 4. L'envoi de messages
 5. L'héritage
- Analyse et conception orientées objets
- Les langages orientés objets
- Bibliographie

Etat et comportement

Un objet comprend :

1. une partie structurelle qui décrit son *état* et ses *liens* avec les autres objets ;
2. une partie opérationnelle qui décrit son *comportement*.

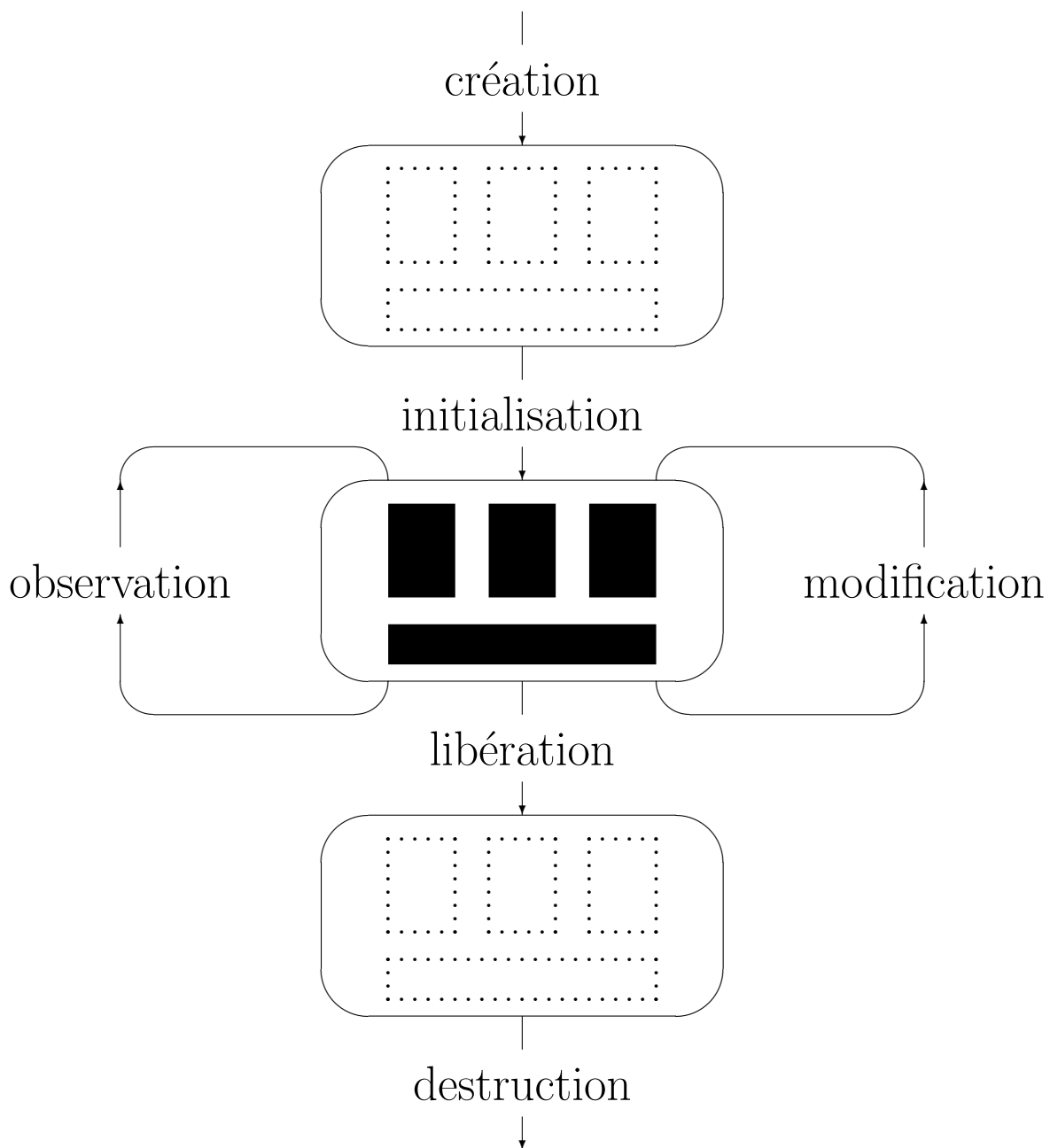
Les classes

- Une classe est la description d'une famille d'objets qui ont la même structure et les mêmes comportements.
- Une classe est une sorte de moule à partir duquel sont générés les objets (ou *instances*).
- Une classe est constituée :
 1. d'un ensemble de champs (ou *attributs*) qui décrivent la structure de ses instances ;
 2. d'un ensemble d'opérations (ou *méthodes*) qui sont applicables à ses instances, et qui décrivent leur comportement.

L'instanciation

- La classe est l'entité conceptuelle qui décrit l'objet. Elle sert de modèle pour construire ses représentants physiques (*instances*).
- Le dictionnaire des méthodes est détenu par la classe.
- Le dictionnaire des variables est divisé en deux:
 1. la moitié contenant les noms des variables est détenue par la classe ;
 2. la moitié répertoriant les valeurs appartient en propre à chaque instance.

La vie des objets



L'envoi de messages

- Pour agir sur un autre objet, un objet doit utiliser une des méthodes appartenant à l'interface de cet autre objet, et lui *envoyer un message* qui demande l'exécution de la méthode en question.
- Le message est une requête dont la satisfaction est à la charge de l'objet auquel elle est adressée.

`send(objet, message)`

- L'envoi de message doit spécifier :
 1. le *receveur* du message (l'objet que l'on veut activer) ;
 2. le *sélecteur* de la méthode à activer ;
 3. les *arguments* sur lesquels la méthode activée doit s'appliquer.

`send(receveur, selecteur, args)`

- L'envoi de message est un appel de procédure : la méthode du *receveur* associée au *sélecteur* est appliquée aux *arguments*.

`call(methode(selecteur, classe(receveur)), args)`

L'héritage

- La relation d'héritage entre classes permet de partager efficacement les connaissances.
Les connaissances les plus générales sont ainsi mises en commun dans des classes qui sont ensuite *spécialisées* par définitions de sous-classes successives contenant des connaissances de plus en plus spécifiques.
- La relation d'héritage est transitive : les caractéristiques des classes supérieures sont héritées par les classes inférieures.
- Les relations d'héritage entre classes forment un graphe d'héritage.

La relation d'héritage

Point de vue ensembliste : la relation d'héritage décrit une inclusion d'ensembles.

$$\forall x, x \in \text{ArticleDeLuxe} \Rightarrow x \in \text{Article}$$

Point de vue logique : l'appartenance à une classe est exprimée par un prédicat ; la relation d'héritage est un théorème logique.

$$\forall x, \text{ArticleDeLuxe}(x) \Rightarrow \text{Article}(x)$$

Point de vue conceptuel : la relation d'héritage indique une spécialisation.

ArticleDeLuxe est une sorte d'Article

Spécialisation par héritage

La spécialisation d'une classe peut être réalisée selon deux techniques :

1. l'*enrichissement* : la sous-classe est dotée de nouvelles variables et/ou de nouvelles méthodes ;
2. la *surcharge* : une (ou plusieurs) méthode héritée est redéfinie.

L'héritage simple

L'héritage multiple

Analyse et Conception

- La qualité du logiciel
- Une société de modules
- Les types abstraits de données
- Les objets
- Analyse et conception orientées objets
 1. Cycle de vie
 2. Analyse orientée objets
 3. Conception orientée objets
 4. De la méthode
 5. Métriques
 6. Méthodes et outils
- Les langages orientés objets
- Bibliographie

Cycle de vie

La cascade :

La spirale :

Analyse orientée objets

Modélisation du système indépendamment de l'environnement d'implémentation.

- Trouver les objets
 - Terminologie du domaine
 - Types d'objets (temporaire, permanent, persistant, générique, spécifique, ...)
- Organiser les objets
 - Héritage
 - Composition
 - Collaboration
- Interactions entre objets
 - Interfaces publiques
- Opérations sur les objets
 - Observateurs
 - Modificateurs

Conception orientée objets

Modélisation du système dans son environnement d'implémentation.

- Identifier l'environnement d'implémentation
 - Le système cible
 - Le langage de programmation
 - Les composants logiciels réutilisables
 - Les générateurs d'interfaces
 - Les SGBD
- Etablir les diagrammes d'interactions
 - Emetteur/Récepteur
 - Séquentialité/Parallélisme
 - Evénements
- Décrire les comportements des objets
 - Graphe de transitions d'états

De la méthode

1. **Niveau initial** : pas de méthode.
2. **Niveau répétable** : méthode non formalisée.
3. **Niveau défini** : processus de développement formalisé et documenté.
4. **Niveau géré** : mesures des différentes caractéristiques du processus et du produit.
5. **Niveau optimisé** : optimisation du processus et du produit grâce aux mesures du niveau 4.

U.S.A. (1990)

niveau 1	:	85%
niveau 2	:	10%
niveau 3	:	5%
niveau 4	:	0%
niveau 5	:	0%

Métriques

Métriques du processus :

- Temps de développement total
- Nombre de types différents de fautes
- Temps passé à modifier les modèles
- ...

Métriques du produit :

- Nombre de classes
- Nombre de classes réutilisées
- Nombre total de messages envoyés
- Profondeur des hiérarchies d'héritage
- Nombre moyen d'opérations héritées
- Nombre de classes dont dépend une classe particulière
- Nombre de classes qui dépendent d'une classe particulière
- ...

Méthodes et outils

Vous pouvez utiliser une méthode sans outil, mais pas un outil sans méthode.

HOOD (*Hierarchical Object–Oriented Design*) de l'Agence Spatiale Européenne.

Exemple d'outil : **Stood**

OOA/OOD (*Object–Oriented Analysis, Object–Oriented Design*) de Coad et Yourdon.

Exemple d'outil : **GraphTalk/OOA/OOD**

OOD (*Object–Oriented Design*) de Booch.

Exemple d'outil : **ObjectCenter**

OMT (*Object Modeling Technique*) de Rumbaugh et al. .

Exemple d'outil : **ObjectTeam/Paradigme**

Analyse et Conception

Les langages à objets

- La qualité du logiciel
- Une société de modules
- Les types abstraits de données
- Les objets
- Analyse et conception orientées objets
- **Les langages orientés objets**
 1. **Caractéristiques générales**
 2. **Le langage Smalltalk**
 3. **Le langage Eiffel**
 4. **Le langage C++**
- Bibliographie

Caractéristiques générales

Points communs :

1. *Modélisation* : classes/instances
2. *Activation* : messages/méthodes
3. *Construction* : héritage/composition

Différences :

1. *Typage*: fort ou faible ?
2. *Attributs*: simples ou complexes ?
3. *Envoi de messages*: dynamique ou statique ?
 - *liaison dynamique* : le lien message/méthode est traité à l'exécution
 - *liaison statique* : le lien message/méthode est traité à la compilation
4. *Héritage*: simple ou multiple ?
5. *Métaclases*: les classes sont-elles des objets ?
6. *Ramasse-miettes*: automatique ou manuel ?
7. ...

Un premier graphe d'héritage

Le langage Smalltalk

Origines

Concepteur : **A. Kay**

Date de naissance : . **Début des années 1970**

Lieu de naissance : .. **Xerox PARC**

But initial : **Outil personnel de gestion
d'informations**

Les principes

- Tout est objet.
- Tout objet est instance d'une classe.
Une classe est instance d'une autre classe : sa métaclasse.
- Un objet est uniquement accessible par envoi de message.

Le système Smalltalk-80

- A la fois langage, système d'exploitation et environnement de programmation.
- Nombreuses classes prédéfinies.

Une classe en Smalltalk

```
"Definition de la classe Pile"
Object subclass: #Pile
  instanceVariableNames: 'contenu'

"Methodes de classe"
creer
  "retourne une instance de Pile"
  ^ (self new) initialiser

"Methodes d'instance"
initialiser
  contenu <- OrderedCollection new

empiler: unObjet
  contenu addLast: unObjet

depiler
  contenu removeLast

sommet
  ^ contenu last
```

Le langage Eiffel

Origines

Concepteur : **B. Meyer**
Date de naissance : . **Fin des années 1980**
Lieu de naissance : .. **ISE – Santa Barbara**
But initial : **Langage pour le Génie
Logiciel**

Particularités

- Classes et objets
- Héritages simple et multiple
- Polymorphisme et liaison dynamique
- Généricité
- Programmation par contrat
- Traitement des exceptions
- Environnement de programmation

Une classe en Eiffel

```
class Pile[T]
  vide, empiler, depiler, sommet
feature
  representation : TABLEAU[T];
  taille_max : INTEGER;
  nb_elements : INTEGER;
  Create(n : INTEGER) is
    do
      if n > 0 then taille_max := n end;
      representation.Create(1,taille_max)
    end; -- Create
  vide : BOOLEAN is
    do
      Result := (nb_elements = 0)
    end; -- vide
  sommet : T is
    require
      not vide
    do
      Result :=
        representation.entree(nb_elements)
    end; -- sommet
```

Une classe en Eiffel

```
empiler(x : T) is
  do
    nb_elements := nb_elements + 1;
    representation.entre(nb_elements, x)
  ensure
    not vide;
    sommet := x;
    nb_elements = old nb_elements + 1
  end; -- empiler
depiler is
  require
    not vide
  do
    nb_elements := nb_elements - 1
  ensure
    nb_elements := old nb_elements - 1
  end; -- depiler
```

Le langage C++

Origines

Concepteur : **B. Stroustrup**

Date de naissance : . **Début des années 1980**

Lieu de naissance : . **Laboratoires Bell (ATT)**

But initial : **Programmation par objets
en C**

Le premier +

- Compatibilité avec le langage C
- Typage fort
- Références
- Surdéfinition des fonctions
- Généricité

Le deuxième +

- Classes et objets
- Héritages simple et multiple
- Polymorphisme et liaison dynamique

Mon premier programme C++

E.N.I. Brest

premier.C

1/1

```
/* un premier commentaire */

#include <iostream.h>

int main(void)          // un autre commentaire
{
    cout << "Hello world" << '\n' ;
    return 0;
}
```

- **2 types de commentaires**

`/* ... */` (sur plusieurs lignes)

`// ...` (en fin de ligne)

- **directive du préprocesseur**

`#include <iostream.h>`

- **en-tête de la fonction principale**

`int main(void)`

- **corps de la fonction principale**

`{ cout << "Hello world" << '\n' ; return 0 ; }`

Une classe C++ : interface

E.N.I. Brest

pile.h

1/1

```
#ifndef PILE_H
#define PILE_H

const unsigned int MAXELEMS = 20;

class Pile {
public :
    Pile(int = MAXELEMS);           // un constructeur
    ~Pile(void);                   // le destructeur
    int empty(void);
    double top(void);
    void push(double);
    double pop(void);
private :
    double* _elems;
    int _top;
    int _capacity;
} ;

#endif
```

Une classe C++ : implémentation

E.N.I. Brest

pile.C

1/1

```
// version simplifiée
#include "pile.h"

Pile::Pile(int n)
:   _top(-1) , _capacity(n)    // initialisations
{   _elems = new double[n]; }

Pile::~~Pile(void)
{   delete [] _elems; }

int Pile::empty(void)
{   return (_top == -1) ; }

double Pile::top(void)
{   return _elems[_top]; }

void Pile::push(double elem)
{   _elems[++_top] = elem; }

double Pile::pop(void)
{   return _elems[_top--]; }
```

Mon deuxième programme C++

E.N.I. Brest

deuxieme.C

1/1

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

int main(void)
{
    Pile p;
    double x, y;

    cin >> x >> y;
    cout << "x= " << x << " , y= " << y << '\n';

    p.push(5.2); p.push(x); p.push(y);

    while(! p.empty()) {
        cout << "sommet = " << p.pop() << '\n';
    }
    return 0;
}
```

Compilation et exécution

E.N.I. Brest	compilation	1/1
--------------	--------------------	-----

```
$ CC -c pile.C
$ CC -c deuxieme.C
$ CC deuxieme.o pile.o -o deuxieme
$
```

E.N.I. Brest	exécution	1/1
--------------	------------------	-----

```
$ deuxieme
8.2 7.9
x= 8.2 , y= 7.9
sommet = 7.9
sommet = 8.2
sommet = 5.2
$
```

Bibliographie

- La qualité du logiciel
- Une société de modules
- Les types abstraits de données
- Les objets
- Analyse et conception orientées objets
- Les langages orientés objets
- **Bibliographie**

Références

Booch G. (1992) *Conception orientée objets et applications.*

Addison–Wesley — 1992

Coad P., Yourdon E. (1991) *Object–Oriented Analysis.*

Prentice Hall — 1991

Coad P., Yourdon E. (1991) *Object–Oriented Design.*

Prentice Hall — 1991

Ferber J. (1990) *Conception et programmation par objets.*

Hermes — 1990

HOOD (1989) *HOOD Reference Manual.*

European Space Agency — 1989

Jacobson et al. (1993) *Le génie logiciel orienté objet.*

Addison–Wesley — 1993

Références

- Masini G. et al.** (1989) *Les langages à objets. Langages de classes, langages de frames, langages d'acteurs.*
InterEditions — 1989
- Mével A., Gueguen T.** (1987) *Smalltalk-80.*
Eyrolles — 1987
- Meyer B.** (1990) *Conception et programmation par objets. Pour du logiciel de qualité.*
InterEditions — 1990
- Rumbaugh J. et al.** (1991) *Object-Oriented Modeling and Design.*
Prentice Hall — 1991
- Shlaer S., Mellor S.J.** (1988) *Object-Oriented Systems Analysis.*
Prentice Hall — 1988
- Stroustrup B.** (1992) *Le langage C++.*
Addison-Wesley — 1992