Une panoramique de concepts et langages orientés-objet

Roberto Di Cosmo

3 juin 1996

Table des matières

Chapitre 1

Panoramique et comparaison de langages et notions de l'Orienté Objet

On souhaite proposer dans ce chapitre une petite panoramique de notions, problèmes et solutions dans le monde des objets à travers la comparaison des approches retenues dans des langages largement utilisés dans le monde de l'industrie (ce qui ne signifie pas forcément que ces langages soient de qualité, mais cela est un autre discours). On va s'intéresser plus précisément à `C++`, `Objective-C` et `Java`, dont on donnera une introduction sommaire. On verra aussi que ces langages sont largement différents dans plusieurs de leurs aspects fondamentaux, et que cela vient de choix précises de conception.

On ne souhaite pas restreindre notre attention aux seuls problèmes posés pour le typage statique des programmes, (et cela d'autant plus que `Objective-C` admet aussi du typage dynamique), mais le typage reste un atout majeur, et dans `C++`, `Objective-C` et `Java` on fait des efforts pour que cela au moins en partie faisable, donc on donnera quand même du poids a ces aspects.

1.1 Rappel sur l'origine de l'orienté objet

La programmation à grande échelle nécessite une méthodologie et un support qui permettent au moins

- la compilation séparée
- la protection des détails d'implémentation
- la réutilisation du code

Pour obtenir cela, on retrouve dans différents langages des mécanismes tels que

- compilation séparée: un programme peut se découper en morceaux que l'on peut compiler séparément (ex: Modula-2, TurboPascal, etc.), cela est le minimum.
- types de données abstraits: on peut cacher une partie de l'implémentation en ne fournissant à l'extérieur qu'une interface abstraite (ex: packages dans ADA).
- héritage: on peut réutiliser du code en étendant et composant des définitions préexistantes (cela concerne plutôt la réutilisation du code au niveau des sources, comme par exemple avec les `generic packages` de ADA).
- polymorphisme: on peut réutiliser du code en écrivant des fonctions génériques capables d'opérer sur des données de types différents (cela concerne plutôt la réutilisation du code à l'exécution, par exemple ML). On distingue normalement entre

  polymorphisme vertical c'est celui de ML: on utilise du code ayant un type générique avec des instances de ce type, obtenues normalement par substitution de variables de type.

  polymorphisme horizontal c'est celui qui est répandu le plus dans le monde à objets: on utilise du code avec des objets contenant plus d'informations que celles requises par le code (ex: enregistrements avec plus de champs), mais c'est aussi ce qui se passe par exemple dans le mécanisme de `signature matching` pour les modules de `SML` ou `Objective Caml`. Cela correspond souvent à ce qu'on appelle *sous-typage* en théorie des systèmes de types.

Les difficultés et la grande complexité de la plupart des langages dits à objets viennent de l'ambition de ces langages de capturer *tous* ces aspects à la fois, et cela en utilisant une seule notion, celle d'objet.

On va voir comment nos trois langages traitent ces problèmes dans ce qui suit, en les comparant élément par élément, problème par problème.

1.2 Héritage, sous-typage et conformance

Il est nécessaire, pour pouvoir se démêler dans la bué vaseuse des concepts qui polluent le monde à objets, de distinguer clairement au moins entre la relation d'héritage, la relation de sous-typage et une forme plus faible de celle-ci qu'on appelle des fois la relation de "conformance".

héritage la classe B est dérivée de la classe A (on réutilise le code, et on établi une relation `is-a`)

polymorphisme contextuel les objets de la classe B peuvent être utilisés dans un certain ensemble de contextes pas spécialement prévus pour eux.

sous-typage les objets de la classe B peuvent être utilisés partout où on peut utiliser des objets de la classe A

conformance les objets de la classe B savent répondre à un certain nombre de messages (où l'on prend en compte le type du résultat et ceux des paramètres) et peuvent être utilisés partout où il suffit de savoir répondre à ces messages.

On a des exemples désormais classiques qui montrent comment ces notions sont différentes.

l'héritage n'est pas du sous-typage l'exemple classique de ce phénomène est un programme qui contient une classe point et une classe point-coloré, les deux ayant une méthode pour tester l'égalité de deux points.
Voici comment on écrirait un petit programme dans ce genre en `C++`:

```
#include <fstream.h>
class Point {
protected:
  int x;
public:
  void setx(int y) {x=y;};
  int getx() {return x;};
  int eq(Point q) {return (q.getx() == x);}
};

class ColorPoint: public Point {// n'est pas un sous-type de Point!
  char *c;
public:
  void setcol(char *d) {free(c); c=(char *)malloc(strlen(d)); strcpy(c,d);};
  char *getcol() {return c;};
  int eq(ColorPoint q) {if (strcmp(q.getcol(),c)==0)
                           {return Point::eq(q);} else {return 0;};}
};

main()
{
  Point      p;
  ColorPoint q;


  p.setx(3);
  q.setx(4);
  q.setcol("blue");
  cout
    <<p.eq(p)<<"\n"
```

```
          <<p.eq(q)<<"\n"
//          <<q.eq(p)        ERREUR: p n'a pas de composante couleur!
              <<q.eq(q)<<"\n"
                <<"Point at "<<p.getx()<<" "<<q.getcol()
                  <<" Colorpoint at "<<q.getx()<<" "<<"\n";;
}
```

l'héritage n'est pas de la conformance le même exemple que ci-dessus montre cela aussi par rapport au protocole `eq`.

la conformance n'est pas du sosu-typage la conformance est rélative à à un protocole donné qui n'est pas forcement né d'une classe, donc cd n'est pas exactement la même chose que le sous-typage, mais cela revient quand même au sous-typage par rapport à une classe hypothétique qui a exactement les méthodes du protocole.

1.3 Terminologie

Tous les langages n'adoptent pas la même terminologie. Voici un court rappel.

Notion	Java	Objective-C	C++
objet	object	object	object
classe	class	class	class
methode	method	method	member function
variable d'instance	instance variable	instance variable	data member
self	this	self	this
super	super	super	—

Ensuite, voyons les différents traits et problèmes typiques de la programmantion avec objets et comment ils sont résolus (ou pas) par ces langages.

1.4 Un exemple dans chaque langage

Voyons comment on peut déclarer une classe *Point* et une classe *PointColoré* en relation d'héritage dans chacun des trois langages.

1.4.1 Point et ColorPoint in C++

```
#include <stdio.h>
#include <fstream.h>
class Point {// interface
int x;
      public:
void setx(int y);
int getx();
};

//implementation
int Point::getx() {return x;};
void Point::setx(int y) {x=y;};

class ColorPoint: public Point {//interface
char *c;
      public:
void setcol(char *d);
char *getcol();
};

//implementation
void ColorPoint::setcol(char *d) {free(c);
                                  c=(char *)malloc(strlen(d));
                                  strcpy(c,d);};
char *ColorPoint::getcol() {return c;};

main()
{
  Point      p;
  ColorPoint q;

  p.setx(3);
  q.setx(4);
  q.setcol("blue");
  cout<<"Point at "
      <<p.getx()
      <<" "<<q.getcol()
      <<" Colorpoint at "
      <<q.getx()
      <<"\n";;
}
```

1.4.2 Point et ColorPoint in `Objective-C`

```
#import <appkit/appkit.h>

@interface Point : Object
{// instance variables
 int x;
}
- (int) getx;
- setx:(int) y;
@end

@implementation Point
- (int) getx
        { return x;}
- setx:(int) y
        { x=y; return self;}
@end

@interface ColorPoint : Point
{// instance variables
char * col;
}
- (char *) getcol;
- setcol:(char *) c;
@end

@implementation ColorPoint : Point
- (char *) getcol
        { return col;}
- setcol:(char *) c
        { free(col);
          col=(char *)malloc(strlen(c));strcpy(col,c);
          return self;}
@end

main() {
id p, cp;

p=[[Point new] setx: 3];
cp=[[[ColorPoint new] setx: 4] setcol: "blue"];

printf("Point at %d, %s ColorPoint at %d\n",
       [p getx], [cp getcol], [cp getx]);
exit(0);
}
```

1.4.3 Point et ColorPoint in Java

```
class Point extends Object {
        protected int x;
        void setx(int x) { this.x = x;}
        int  getx() { return x;}
}

class ColorPoint extends Point {
        protected String c;
        void setx(String c) { this.c = c.clone();}
        String getcol() { return c;}
}

  Point p;
  ColorPoint q;

  p.setx(3);
  q.setx(4);
  q.setcol("blue");
  System.out.println("Point at "
                     +p.getx().toString()
                     +" "+q.getcol()
                     +" Colorpoint at "
                     +q.getx().toString()
                     +"\n");;
```

1.5 Bref rappel des opérations de base dans les trois langages

Dans le tableau qui suit, `A` et `B` sont des classes, `a` et `b` des identificateurs d'objets de ces classes, et `m, m1, m2` des noms de messages.

Notion	**Java**	Commentaire
décl. de classe	class A {def. }	
décl. de méthode	class A {m1(..){}; ..};	seulement dans la classe
heritage	A extends B	
instanciacion	A a;	
envoi de messages	a.m(parametres)	
initialisation	A a(initparametres);	appelle le constructeur
constructeurs	A() {};	
et aussi	A(params) {};	constructeurs surchargés
destructeurs	finalize {};	

Notion	**Objective-C**	Commentaire
décl. de classe	@interface A {decl.}	seulement la déclaration
décl. de classe	@implementation A {def.}	implémentation
décl. de méthode	- m1: {};	méthode de l'objet
décl. de méthode	+ m1: {};	méthode de la classe
heritage	A : B	
instanciacion	a = [A new];	message à l'objet classe
envoi de messages	[a m: par1 n: par2 etc.];	
initialisation	a = [[A alloc] init: pars];	
constructeurs	- init	
destructeurs	- free	

Notion	**C++**	Commentaire
décl. de classe	class A {def. }	
décl. de méthode	class A {m1(..){};	dans la classe
ou aussi	A:: m1 () {};	séparément
heritage	A : public B	
instanciacion	A a;	
initialisation	A a(initparametres);	appelle le constructeur
envoi de messages	a.m(params)	
constructeurs	A() {};	
et aussi	A(params) {};	constructeurs surchargés
destructeurs	A() {};	

La syntaxe de `Objective-C` est clairement derivée de `Smalltalk`, et celle de `Java` a été astucieusement modelée sur celle de `C++`.

1.6 Modularité

Dans les trois langages, on peut déclarer séparément une classe et son implémentation, mais avec les distinctions suivantes:

- `Java` va plus loin, et distingue nettement la notion d'*interface* de celle de définition de classe, en gardant la première et en supprimant la deuxième.

- en `C++`, en raison de l'approche très statique de ce langage, et son soucis d'efficacité, la compilation séparée est souvent une chimère: si on modifie une super-classe, on peut être obligé de recompiler tout le code qui lui fait référence (sous-classes, friends etc.), cela est connu comme le problème de la *superclasse fragile*

De plus, `Java` propose une notion de *package* pour grouper des définitions de classes reliées, qui ressemble aux modules de Modula-2 (aussi connu comme le système de modules du pauvre, "poor man's module system" en anglais).

1.7 Cacher l'information

Dans les trois langages, on peut limiter l'accès aux variables d'instance et aux méthodes en utilisant les mots clefs suivants:

private accessible seulement dans cette classe,

protected accessible seulement dans cette classe et ses sous-classes,

public accessible partout.

Cela donne un contrôle assez fin sur l'encapsulation.
Cependant, `C++` et `Java` disposent aussi d'un quatrième niveau, qu'on appelle le niveau des `friends`. Cela revient à permettre l'accès aux données à toute classe, fonction ou méthode qui est un *friend* (une amie), mais qui n'a aucune relation avec l'héritage. En `C++` cela se fait en définissant explicitement à l'intérieur d'une classe les noms de ces privilégiés, comme dans

```
class ColorPoint: public Point {
private int color;
...
friend void antialias (ColorPoint *c,ColorPoint *d);
...
}

void antialias (ColorPoint *c,ColorPoint *d)
{
c->color=(c->color+d->color)/2;
}
```

En `C++`, toute donnée est par défaut `private`, tandis qu'en `Objective-C` ell'est par défaut `public` et en `Java` accessible aux amis, mais à différence de `C++`, en `Java` on considère `friend` seulement les méthodes qui sont dans le même package.

Le choix du niveau d'accessibilité n'est pas facile: toute donnée déclarée `private` est effectivement invisible aux sous-classes, ce qui protège les sous-classes de tout changement apporté à la classe de base, mais cela signifie aussi que pour accéder à ces données les sous-classes doivent passer par une interface composée de méthodes, et cela nuit à l'efficacité. C'est là qu'on doit déjà commencer à choisir comment mélanger ADT et OO.
Un bon exemple de ce dilemme se retrouve dans un fragment de programme comme celui-ci:

```
class IntList {
??? int data[];
...
public hd { return data[0];}
...
}
class OrdIntList extends IntList {
...
public sortAscending { ??? }
}
```

Si on veut protéger l'implémentation de `IntList`, on ne pourra pas savoir que les données sont rangées dans un vecteur, ce qui nous empêchera d'utiliser un algorithme de tri efficace dans `OrdIntList`. Si par contre on ne la protège pas, on pourra être efficace dans `OrdIntList`, mais tout changement de représentation dans `IntList` se propagera aussi dans `OrdIntList`.
Dans ce cas spécifique, la meilleure solution semble être de garder l'implémentation protégée, et de fournir dans `IntList` des méthodes

```
public asVector: { // renvoie une repr\'esentation de la liste comme vecteur }
public fromVector: { // initialise la liste a partir d'un vecteur }
```

Ces méthodes donneront un accès efficace pour des opérations de tri, sans casser la barrière de modularité.

1.8 Héritage et polymorphisme vertical

Dans le cadre de l'héritage, on essaye de fournir des mécanismes qui permettent de réutiliser le code au maximum, et pour cela on est souvent amenés à compliquer énormement la notion d'héritage, vu que c'est là qu'on fourre tout mécanisme de réutilisation de source. Dans différents langages on trouve différents mécanismes pour obtenir des buts similaires, que nous résumons brièvement ici.

1.8.1 Classes dont on aurait envie de designer plus qu'un parent

On veut souvent réutiliser du code de *plusieurs* autres classes en principe non reliés: par exemple, la classe `pointcoloré` pourrait être vue comme une sous-classe à la fois de `point` et de `couleur`, pour reutilisrer du code. Certains langages permettent de faire cela, à travers l'*héritage multiple*

Cependant, il y a souvent des cas où on veut bien qu'un objet donné puisse répondre à des messages qui ne sont pas déclarés dans sa classe, mais sans besoin de définir une sous-classe à chaque fois: c'est le cas par exemple d'un `ministre` qui dans certains conditions doit accepter un message `négocier`, même s'il n'est pas capable de négocier. On pourrait alors définir une sous-classe `ministrenégociateur` qui hérite à la fois de `ministre` et de `négociateur`, mais on risque de voir vite proliférer la hiérarchie des classe avec la nécessité de `ministre-expert`, `ministre-anglophone`, et plein d'autres mélanges possibles. Dans ce type de situation, il est souvent plus naturel de "passer" le message (anglais: forward) à d'autres objets d'autres classes qui savent le gérer: on fait la du *message forwarding*.

héritage multiple : il est supporté dans la version 3.0 et suivantes de `C++`, il est absent dans `Java` et `Objective-C`.

message forwarding il est supporté explicitement dans `Objective-C`, où chaque objet qui ne reconnaît pas un message qu'on lui adresse se voit offrir la possibilité de passer ce message à quelq'un d'autre à travers l'envoi par le système d'un message `forward`. On peut bidouiller pour le simuler dans `Java` et `C++` à travers l'*object containement*, mais l'élégance du mécanisme est perdue. Notez que pour que cela puisse fonctionner on a besoin de pouvoir passer des messages comme paramètres à run-time.Voici un exemple en `Objective-C`:

```
- forward:(SEL)aSelector :(marg_list)argFrame
{if ( [monDiplomat respondsTo:aSelector] )
        return [monDiplomat performv:aSelector :argFrame];
    else if ( [monInterprete respondsTo:aSelector] )
             return [monInterprete performv:aSelector :argFrame];
         else  return self; }
```

Le mécanisme du `message-forwarding` permet de simuler l'héritage et quelques aspects de l'héritage multiple, tout en gardant un nombre réduit de classes.

1.8.2 Comment traiter de façon uniforme objets provenant de classes non reliés

Il arrive souvent d'avoir besoin de traiter de façon uniforme des objets de classes en principe non reliées, mais qui savent tous répondre aux messages qui leur seront envoyés dans un contexte précis. C'est par exemple le cas d'une liste d'objets graphiques que l'on veut imprimer: on peut avoir des `rectangles` et des `cercles-colorés`, tous capables de s'afficher en répondant au message `affiche`, mais quel sera bien le type d'une liste de ces objets? Une approche typique est de factoriser les messages communs dans une classe artificielle (par exemple, `graphicobject`) que l'on rend ensuite super-classe de toutes les classes dont les instances peuvent se retrouver dans la même liste. Normalement, on ne pourra pas instancier `graphicobject`, parce qu'il ne corresponde pas à des objets, mais à une *abstraction* conceptuelle, et il faut donc du support pour vérifier que l'on ne prend pas d'instances des classes abstraites.

En `C++`, cela se fait en déclarant la classe abstraite ou partie de ses composantes comme `virtual` (cela étant quand même un gros hack, vu que c'est aussi utilisé pour le contrôle de la liaison tardive), et en utilisant cette classe comme super-type dans la liste.

au fait, une classe abstraite est une classe qui contient au moins une méthode `m` qui est *pure virtual*, c'est à dire déclarée comme:

```
virtual m =0;
```

Toutefois, il y a des situations dans lesquelles on n'a aucun intérêt à utiliser ce type de bricolage, en raison de l'absence de sens de la super-classe abstraite. C'est par exemple le cas des objets pouvant répondre aux messages `copy` et `paste`, qui copient dans ou depuis le tampon la sélection courante, cela pouvant être du texte, un objet graphique, un font, ou , en général, n'importe quel objet sur lequel ces opérations ont un sens, et il risque d'y en avoir beaucoup. Utiliser l'approche de la classe abstraite oblige souvent, en plus, à implémenter l'héritage multiple.

C'est pour cela qu'on retrouve dans les langages plus récents des mécanismes plus propres pour capturer la notion de "conformance" à un protocole de communication, c'est le cas des `protocols` dans `Objective-C`, repris pratiquement inchangés dans `Java`.

classes abstraites/virtuelles sont fournies explicitement en `C++` et `Java`, et admises en `Objective-C`, mais sans support syntactique spécifique

Voici un exemple en `Java`:

```
abstract class GraphicObject extend Object {
abstract void affiche();
}

class Rectangle extends GraphicObject {
. . . // instance variables etc.
void affiche() {
```

```
// routine d'affichage
};
}
```

Voici un exemple en `C++`:

```
virtual class GraphicObject {
public:
virtual void affiche();
}

class Rectangle : public GraphicObject {
. . . // instance variables etc.
void affiche() {
// routine d'affichage
}
}
```

protocoles sont fournis en `Objective-C`, et repris tel quels dans `Java`, où ils sont appelés *interfaces*.
Voici un exemple en `Objective-C`:

```
@protocol Affichable
- (void) affiche;
@end

@interface AffichableList : List
- addObject:(id<Affichable>)anObject;
- affiche;
@end

@implementation AffichableList
- addObject:(id<Affichable>)anObject
{ [super addObject: anObject];};
- affiche
{ [self makeObjectsPerform: @selector(affiche)];};
@end
```

Il faut noter que, comme le typage de `Objective-C` n'est pas sûr, on pourrait tout simplément utiliser une liste sans se soucier de la "conformance" de ses éléments, et c'est cela la pratique en `Objective-C`, mais les protocoles permettent d'introduire un peu de sûreté dans le typage.

Les protocoles deviennent cependant nécessaires lorsque on travaille avec des objets distribués.

En `Java`, un exemple typique d'interface est l'interface `Runnable`[1], qui déclare les méthodes nécessaires pour qu'un objet puissse être exécuté dans un thread.

Leur présence dans `Java` est nécessaire, en raison de la limitation à l'héritage simple qui empêche d'utiliser le mécanisme des classes abstraites trop à la légère (notamment, un heritage artificiel utilisé pour faire des classes abstraites peut gêner un vrai heritage).

Ce problème est en général assez complexe, parce-que il a souvent à voir avec la nécessité de distinguer clairement entre la relation d'héritage, la relation de sous-typage et une forme plus faible de celle-ci que c'est la relation de "conformance" vue en 1.2.

1.8.3 La nécessité du polymorphisme vertical

Finalement, il y a plein de cas où on a besoin de recuperer le pouvoir expressif du polymorphisme vertical pour éviter une prolifération massive du nombre de classes dans le programme. Un exemple est encore bien sûr la liste, mais on peut choisir autre chose, comme la classe `Matrix` de `Objective-C` qui fournit une matrice de cellules affichables. Si on veut vraiment que le typage soit vérifié statiquement, on est obligé de demander quelque chose du type des cellules, et si on n'a rien d'autre que l'héritage sous la main, on se retrouve vite avec une sous-classe de `Matrix` pour tout type de cellules dans le programme.

Il y a plusieurs façons de répondre à ce problème: on examinera ici les classes paramétriques et le vrai polymorphisme à la ML.

classes paramétriques dans `C++`, on fournit à l'utilisateur le mécanisme des *classes paramétriques* (aussi appelé *genericity*) pour simuler le polymorphisme vertical, mais il s'agit essentiellement d'un gros hack qui se résout en une macro-expansion. Il n'y a pas là le partage de code possible en ML, et on se retrouve à peu près avec quelque chose de moins que les *generic packages* de ADA.
Voici un exemple très simple en `C++` pour des points à coordonnées entières ou flottantes.

```
#include <fstream.h>
//------------------------------
template <class T>
class Point
{private:
   T x;
  public:
```

[1] http: //java.sun.com/java.sun.com/JDK-1.0/api/java.lang.Runnable.html

```
  T getx();
  void setx(T x);
};

template<class T>
T Point<T>::getx() { return x; }

template<class T>
void Point<T>::setx(T v) { x=v; }

//------------------------------

main()
{
Point<int> p;
Point<float> q;
p.setx(3);
q.setx(4.33);
cout<<p.getx()<<" "<<q.getx()<<"\n";
exit(0);
}
```

vrai polymorphisme à la ML Comparez le code `C++` que l'on a été obligé d'écrire ci-dessus avec le fragment `Objective Caml` suivant, qui exploite le vrai polymorphisme vertical:

```
#class 'a point x =
#val mutable x = x
#method setx (y:'a) = x <-y
#end;;
class 'a point ('a) = val x : 'a method setx : 'a -> unit end

#new point 3;;
- : int point = <obj>

#new point 4.33;;
- : float point = <obj>
```

1.9 Délégation

C'est un paradigme de la programmation avec objets dans lequel on crée les objets et on les modifies sans utiliser les classes comme générateurs, mais seulement comme types. Cela n'est pas le paradigme choisi par la plupart des langages industriels, donc on ne détaillera pas.

Cependant, certains systèmes, comme OpenStep écrit en `Objective-C`, utilisent le mot délégation pour indiquer le fait que certains objets ont aussi une liste de délégués, auxquels on passe les messages qu'on n'a pas envie de traiter, ou des messages de synchronisation. Cela est différent du "forwarding". Voir la documentation de `NEXTSTEP` pour plus d'information.

1.10 Liaison tardive et dynamique

Dans les langages orientés objets, le polymorphisme et l'héritage ne peuvent marcher correctement que si la méthode exécutée par un objet qui reçoit un message dépend du type de l'objet à run-time et non pas du type statique connu à la compilation.

Cela n'est pas la conduite par défaut de `C++`: là, une méthode (disons *affiche*) qui n'est pas déclarée `virtual` est liée statiquement, et un *colorpoint* qui reçoit le message *affiche* ne va pas afficher la couleur.

Si par contre on prend en compte le type de l'objet à runtime, on voit que ce type est souvent, en réalité, plus petit (un sous-type) du type statiquement connu à la compilation. Si c'est ce type plus petit que l'on prend en compte pour sélectionner le message, on parle alors de *liaison tardive*, ce qui s'explique bien par la liaison tardive de self dans le modèle à enregistrements. Cela se fait en `C++` pour les méthodes déclarés `virtual`, pas un joli nom, mais cela vient du fait que ils sont souvent utilisés pour définir des super-classes abstraites, où la liaison tardive est nécessaire.
Cependant, la solution `C++` reste aussi un hack: cela se voit avec le problème de la super-classe fragile 1.6.

Toutefois, dans les deux cas qu'on vient de voir, on peut tout compiler de façon à pouvoir faire marcher le programme sans besoin de garder l'information de type à l'exécution: il suffit d'utiliser une table de fonctions construite en parallèle avec la hiérarchie d'héritage.
Cependant, il y a des situations où on doit travailler avec l'assumption de *monde ouvert* (on ne connaît pas tout à la compilation) et non pas celle trop restrictive de *monde fermé* (on connaît pas tout à la compilation). Ce genre de situation se présente souvent dans le dessin de GUI (interface graphique utilisateur), où la classe d'un objet qui reçoit un message `cut` ne peux normalement pas être connue à la compilation.

Il y a des langages, comme `Objective-C`, où l'on permet, pour gérer ces situations, la *liaison dynamique*, c'est à dire, il est possible que à runtime l'objet qui se trouve à la place d'une variable dans le programme source aie un type qui n'est même pas un sous-type de celui connu à la compilation. Cela exige une étape supplémentaire ou l'on cherche le message dans une liste chainée de tableaux, sans figer l'adresse physique du code comme en `C++`.

La documentation de `Java` semble indiquer que cela est possible dans `Java` aussi.

1.11 Caractéristiques des classes

Ces langages se distinguent aussi pour la gestion des classes; voici quelques points de repère.

variables de classe il est des fois pratique de pouvoir partager au niveau de la classe des informations qui seraient identiques dans toutes les instances. Cela est fait de façon transparente, par le compilateur, pour les méthodes, qui ne sont pas répliqués à chaque instance, mais cela peut servir aussi si l'on veut garder la version de la classe.
`C++` et `Java` fournissent cela explicitement, à l'aide du qualificateur `static` que l'on peut préfixer aux déclarations de variables d'instance.
`Objective-C` n'a pas de support explicite pour cela, mais toute variable `static` déclarée dans le même fichier où on a declaré une classe se conduit comme variable de classe.

méthodes de classe il s'agit de méthodes qui peuvent travailler seulement sur les variables de classe (dans le cas de `C++` et `Java`, declarés avec `static`), ou de methodes de l'objet classe dans le cas de `Objective-C`, déclarés avec un `+` plutôt qu'un `-`.

objets classes/containement Certains langages prévoient que toute classe soit représentée à run-time par un *objet classe*, et en général on peut ensuite demander à un objet quelle est sa classe. `Objective-C` et `Java` disposent de la notion d'objet classe, mais non `C++`, qui élimine pendant la compilation presque toute information de type.
Cela permet un certain niveau d'*object introspection*.

1.12 Extension "verticale" du code

Il est souvent nécessaire de rajouter de la fonctionnalité à une classe déjà définie, c'est-à-dire, rajouter des méthodes, ce qui correspond à l'extension verticale du code. Si on n'a à disposition que la notion de classe et l'héritage, cela nous oblige à chaque coup à redéfinir une nouvelle sous-classe, avec le désavantage évident que les objets obtenus en instanciant l'ancienne classe n'ont pas accès aux nouvelle méthodes.

Or, certains langages, dont `Objective-C`, mais pas `C++` ni `Java`, fournissent un mécanisme simple pour faire cela, avec la notion d'extension verticale à l'aide de ce qu'on appelle en `Objective-C` des *catégories*.

En `Objective-C`, une catégorie se déclare comme une classe, mais avec le nom de la catégorie entre parenthèses après le nom de la classe que l'on étend, et on peut déclarer *seulement* des nouvelles méthodes, et non pas des nouvelles variables d'instance (cela permet de faire travailler les nouvelles méthodes sur les anciens objets): ces méthodes ont accès à toutes les variables d'instance de la classe d'origine.

```
@interface ClassName ( CategoryName )
 declarations des methodes de la categorie
@end

@implementation ClassName ( CategoryName )
 definitions des methodes de la categorie
@end
```

Comme on n'a pas besoin du code de la classe que l'on étend, ce mécanisme permet d'étendre les fonctionnalités d'un kit préexistant, et comme les nouvelles méthodes sont héritées par les sous-classes, cela est un mécanisme strictement plus puissant que l'héritage, si on n'a pas accès au code, vu que cela permet de faire travailler des vieux objets (préexistants) avec les nouvelles méthodes.

Par exemple, voici le fichier des déclarations des extensions de la classe *List* qui vient avec NEXTSTEP (sans code), extensions qui font partie du `MiscKit`:

```
//      MiscListExtensions.h -- various extensions to List
//              Originally written by Drew Davidson and Don Yacktman
//              Copyright (c) 1994 by Drew Davidson.
//              Modified and extended by Don Yacktman for inclusion into the MiscKit.
//                              Version 1.0.  All rights reserved.
//              This notice may not be removed from this source code.
//
//      This object is included in the MiscKit by permission from the author
//      and its use is governed by the MiscKit license, found in the file
//      "LICENSE.rtf" in the MiscKit distribution.  Please refer to that file
//      for a list of all applicable permissions and restrictions.
@interface List(Extensions)
- deepCopy;
- deepCopyFromZone:(NXZone *)zone;
- makeObjectsPerform:(SEL)aSelector with:object1 with:object2;
- makeObjectsPerformInOrder:(SEL)aSelector;
- makeObjectsPerformInOrder:(SEL)aSelector with:anObject;
- makeObjectsPerformInOrder:(SEL)aSelector with:object1 with:object2;
- addLists:firstList, ...;
- mergeLists:firstList, ...;
@end

@interface Object(MiscListForeach)
// Make self perform aSelector once for each object in a List.  The
// object from the List is sent as the first argument to aSelector,
// so aSelector should take a single argument of type id.
- foreach:aList perform:(SEL)aSelector;
// As above, but aSelector should take two arguments of type id.
- foreach:aList perform:(SEL)aSelector with:anObject;
@end
```

```
@interface List (MiscStringSort)
/* Assumes everything in the list is a MiscString -- no check is made.
 * This category is written by Bill Bumgarner and (C) Bill Bumgarner.
 */
- miscStringSort;
- miscStringSortCaseSensitive:(BOOL)aFlag; // added by DAY
@end

// This next category is from R. Todd Thomas and was impl\'emented to
// help support the MiscUser* classes.

@interface List (StringFinding)
- (BOOL)matchString: (const char *)str usingSel: (SEL)comparator;
@end
```

Cela sert aussi pour modulariser le développement d'une classe de grande dimension, en découpant en différents fichiers indépendants les détails des différents aspects de l'implémentation.

- on groupe ensemble les méthodes qui sont reliés, et on sépare celles qui ne le sont pas.
- on permet le développement en parallèle, et l'extension de code déjà écrit: voici un extrait de la documentation de `MiscSearchText` du *MiscKit* pour NEXTSTEP

  > To incorporate this category into your project, instantly adding searching functionality to your existing Text objects, all you need to do is add the files MiscSearchText.m, MiscTBMK.c, and regexpr.c to "Other Sources" in your project file with ProjectBuilder, then add the files MiscSearchText.h, MiscTBMK.h, and regexpr.h to "Headers".

- on a une forme de compilation incrémentale pour la classe.
- on peut configurer différemment une classe pour différentes applications.

1.13 Le paradigme Target/Action

Un des "slogans" de l'orienté-objet est celui de la construction d'applications à l'aide de composantes réutilisables dont on n'a pas forcément le code. Cela dégénère des fois aussi en une sorte de promesse que tout ira bien même si vous codez avant et réfléchissez après, et non le contraire, mais bon.

Un des meilleurs exemples de réalisation de ce slogan est l'*Inteface Builder* qui fait partie de l'environnement de développement de *NEXTSTEP/OPENSTEP*, du à Jean-Marie Hullot. On peut voir à quoi cela ressemble concrétement sur la figure 1.1.

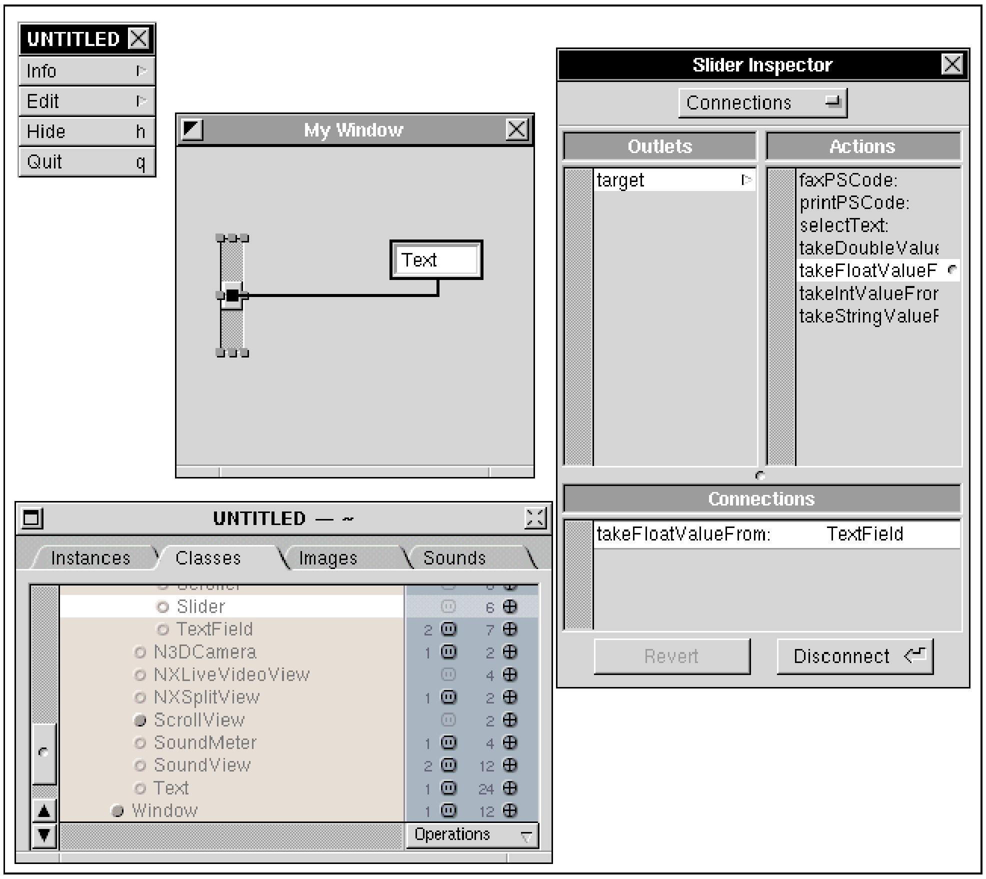

Figure 1.1: Interface Builder et le Target/Action paradigm.

Cet outil exploite au maximum un paradigme bien précis, celui de target/action, qui prévoit que tout objet ayant une interaction avec l'utilisateur dispose de deux variables d'instance, *target* et *action*, configurables à travers des méthodes de l'objet, qui définissent à quelle objet (le target) envoyer quel message (l'action) en réponse à une intervention de l'utilisateur.

Si on vit dans le monde C-like, ou les fonctions sans nom et les variables locales n'existent qu'à un seul niveau, une telle mécanique oblige à demander que le langage fournisse:

- le typage dynamique: on ne sait pas a priori qui sera le target, donc on ne peut assumer rien de mieux que le fait qu'il est un objet, dont le type moins spécifique est *id*, le type de défaut de tout objet non autrement qualifié

```
- target;
- setTarget:anObject;
```

- les méthodes comme valeurs de première classe (passable en paramètre et calculables), dont le type est *SEL*

```
- (SEL)action;
- setAction:(SEL)aSelector;
```

On verra tout de suite qu'on peut conturner cela dans un langage fonctionnel typé, tant qu'on n'a pas bésoin de passer en paramétre à l'objet target l'objet qui declenche l'action. Mais si on veut que le target puisse intéragir avec ceux qui lui envoient l'action, on doit considérer une "action" comme une méthode avec en paramètre le `id` de l'objet qui déclenche l'action: cela est nécessaire pour que les mécanismes de Interface-Builder restent simples. Or, on ne peut pas savoir à l'avance de qui ce message viendra, et alors la déclaration de l'action ne pourra être que

```
- faitqquechose: sender;
```

dans tout objet pouvant recevoir un message qui est une action, où le type de *sender*, l'objet envoyant l'action, est bien, par défaut, `id`.
Cela se fait aussi en Delphi de Borland, où on retrouve des tests dynamiques de type tel `isa`.

Un Interface-Builder typé?

On peut imaginer de pouvoir construire des outils similaires sans ces contraintes si on renonce à la flexibilité que l'on obtient en exposant séparément le target et l'action. On aurait alors, on `Objective Caml`,

```
#class slider p =
#val mutable pos = p
#val mutable action = fun () -> ()
#method pos = pos
#method setAction f = action <- f
#method sliderMoved n = pos <- pos+n; action()
#method stringValue = string_of_int pos
#method print () = print_int pos
#end;;
class slider (int) =
  val mutable action : unit -> unit val mutable pos : int
  method pos : int
  method setAction : (unit -> unit) -> unit
  method sliderMoved : int -> unit
```

```
  method stringValue : string
  method print : unit -> unit
end

#class textfield (s:string) =
#val mutable stringValue = s
#method stringValue = stringValue
#method  setstringValue sv = stringValue <- sv
#end;;
class textfield (string) =
  val mutable stringValue : string
  method stringValue : string method setstringValue : string -> unit
end

#let s = new slider 0;;
val s : slider = <obj>

#let t = new textfield "";;
val t : textfield = <obj>

#s#sliderMoved 3;;
- : unit = ()

#t#stringValue;;
- : string = ""

#s#setAction (fun () -> t#setstringValue s#stringValue);;
- : unit = ()

#s#sliderMoved 3;;
- : unit = ()

#t#stringValue;;
- : string = "6"
```

Cependant, on ne peut pas passer à l'objet recevant l'action l'objet qui la déclenche, et cela réduit la généralité du mécanisme.

1.14 Table comparative sommaire

Notion	Java	Objective-C	C++
classe abstraite	OUI	Partiellement	OUI
classe paramétrique	NO	NO	OUI
objets classe	OUI	OUI	NO
variables de classe	OUI	Partiellement	OUI
méthodes de classe	OUI	OUI	OUI
héritage multiple	NO	NO	OUI
extension verticale	NO	OUI	NO
message forwarding	NO	OUI	NO
liaison tardive	OUI	OUI	OUI
liaison dynamique	il semble	OUI	Partiellement
typage dynamique	il semble	OUI	Partiellement
typage sûr	il semble	Partiellement	il semble
protocoles	OUI	OUI	NO
interface $\neq$ implémentation	Partiellement	OUI	NO
inheritance $\neq$ subtyping	Partiellement	Partiellement	NO
garbage collection	OUI	Partiellement	NO, extension

Index