Usages et perspectives de la programmation fonctionnelle

Ph. Narbel

Université de Bordeaux, LaBRI

JDev 2015

version 1.2

Situation

La programmation fonctionnelle est à la mode...

Pourquoi donc ce changement de point de vue des programmeurs et des créateurs de langages, comment le justifier objectivement ?

Programmation fonctionnelle à la mode ?

- Langages fonctionnels... :
  - Langages "*mainstream*" : C#2.0-5.0, C++11/14, Java 8.
  - Langages outre Lisp : Javascript, Python, Smalltalk, Ruby, OCaml/F#, Scala, Groovy, D, Erlang, Clojure, Go, Swift, etc.
- Quelques livres très récents sur la programmation fonctionnelle :

Être fonctionnel...

I. Comment définir
ce qu'est un langage fonctionnel,
et ce qu'est la programmation fonctionnelle ?

Définir la programmation fonctionnelle

- La **programmation fonctionnelle** : une programmation **basée sur les fonctions et leurs compositions**, et donc aussi **basée sur la décomposition en fonctions**.

- Cet effet de composition/décomposition est la marque d'un "**paradigme de programmation**" (un moyen de modéliser, de construire des solutions).

- Ceci dit, tous les langages possèdent la notion de fonction...

  $\Rightarrow$ **la programmation fonctionnelle consiste à exploiter des fonctions avec des propriétés particulières**, propriétés qui leur donnent des moyens augmentés de bonne composition/décomposition.

Propriétés particulières des fonctions

- **Deux propriétés possibles des fonctions** :
  1. La "**pureté**" : les fonctions ont des résultats qui ne dépendent strictement que de leurs arguments, sans autre effet externe.
     $\Rightarrow$ **cloisonnement, localisation, stabilité, déterminisme, compositionnalité**.
  2. La "**(citoyenneté) de première classe**" : les fonctions ont un statut de valeur.
     $\Rightarrow$ **flexibilité d'utilisation, compositionnalité**.

- La **programmation fonctionnelle** consiste à exploiter l'une et/ou l'autre de ces deux propriétés.
- Un **langage fonctionnel** : un langage qui permet et favorise la programmation fonctionnelle.

Pureté

- Être une **fonction pure**, c'est donc **engendrer des résultats qui ne dépendent que des arguments, sans effet externe**.
- Par ex. voici une fonction pure et une fonction impure :

```
f(i) {
  return i + 4;
}

f(1); --> 5
f(1); --> 5
```

```
j = 4;

g(i) {
  j = i + j;
  return j;
}

g(1); --> 5
g(1); --> 6
```

- Par ex. *log* est pure ; *random* et *print* sont "impures" ;

Première classe

Être de **première classe**, c'est donc **avoir le même statut qu'une valeur telle qu'un entier ou un caractère**, c'est-à-dire :

1. **Pouvoir être nommé, affecté (et typé)** :

```
x := sin;
```

2. **Pouvoir être défini et créé à la demande** :

```
x := (function x --> x + 1);
```

3. **Pouvoir être passé en argument à une fonction** :

```
f(sin);
```

4. **Pouvoir être le résultat d'une fonction** :

```
(f(3))(5);
```

5. **Pouvoir être stocké dans une structure de données qlcq.** :

```
array := {log, exp, tan};
```

La notion de fermeture

- **La pureté et la première classe sont des propriétés indépendantes l'une de l'autre**.
- Mais, pour être pure et/ou de première classe, une fonction doit parfois être transformée en **fermeture** (*closure*), i.e. l'association du code de la fonction avec un environment de définitions.
- Par ex., la fonction suivante est transformée en fonction pure et transportable dans un autre contexte, lorsqu'on lui associe l'environnement `{ i=0, j=1, +='code de +' }` :

```
i = 0;  j = 1;
h(k) {
  return i + j + k;
}
```

- $\Rightarrow$ **La marque d'un langage fonctionnel** : transformer automatiquement les définitions de fonctions en fermetures.

Être fonctionnel...

- **Les définitions précédentes indiquent que** :
  - La pureté est surtout une discipline d'écriture des fonctions.
  - Les cinq caractéristiques de la première classe n'ont pas besoin d'être toutes satisfaites pour améliorer le statut des fonctions.
  - Un langage fonctionnel n'a pas besoin de l'être exclusivement.
  - La pureté et la première classe peuvent être favorisés dans des langages pour lesquels ils ne l'étaient pas initialement.
- $\Rightarrow$ **La programmation fonctionnelle est soluble dans l'eau des autres paradigmes**.

De l'utilité d'être fonctionnel

II. Quelles sont les conséquences pratiques de la pureté et de la première classe des fonctions (en termes de développement de programmes) ?

Conséquences de la pureté

La pureté induit la caractéristique principale suivante :

- **Indépendance au contexte de l'application d'une fonction** : cette application ne nécessite pas la prise en compte d'un environnement, d'un état particulier.

Avec une conséquence importante :

- **Indépendance à l'ordre des applications dans les expressions constituées de fonctions pures** : chaque sous-expression est évaluable n'importe quand, et remplaçable par son résultat n'importe quand (cf. **transparence référentielle**).

```
f(g(3, h("ici", 9.1), log(1), r('a', 'c')), 10)
f(g(3, h("ici", 9.1), 0, r('a', 'c')), 10)
```

$\Rightarrow$ Stabilité lors de la composition des fonctions
(cf. définition du paradigme fonctionnel).

Conséquences de la pureté

L'indépendance au contexte et à l'ordre d'évaluation :

a. **Formalisations facilitées de la notion de fonction** (pas d'états)
Une application de fonction peut devenir une règle de réécriture.
$\Rightarrow$ λ-calculs, preuves de cohérence, confluence, terminaison, etc.

b. **Typages plus complets et plus représentatifs du comportement des fonctions** : toute sous-expression renvoie un résultat (typable).
Ce typage peut s'ajouter aux formalisations ci-dessus.

c. **Parallélisation naturelle** : indépendance directe des calculs.

Conséquences de la pureté

d. **Lisibilité, maintenabilité améliorées** : cloisonnement, localisation, stabilité, déterminisme des calculs. Par exemple (rappel) :

```
f(i) {
  return i + 4;
}
```

```
j = 4;

g(i) {
  j = i + j;
  return j;
}
```

Et donc certes, `f(1)-f(1)==0`, mais quoi de `g(1)-g(1)` ?...
Et donc certes, `f(1)+f(1) = 2*f(1)`, mais quoi de `g(1)+g(1)` ?...

e. **Tests facilités** : tests en boîte noire immédiats, constructions simplifiées d'environnements spécifiques de tests (i.e. *fixturing, stubbing, mocking, dummying* simplifiés).

Conséquences de la pureté

f. **Optimisations possibles** :

i. **Mémorisation de valeurs** : les résultats des fonctions étant déterministes, ils peuvent être stockés, mis en cache (*caching*).

ii. **Contrôle de l'évaluation (cf. nécessité, paresse)**, par ex. :

```
f(x, y, z) {
    if (x > 0) y else z;
}
```

```
f(1, 1, exp(1000000000));
--> if (1 > 0) 1 else exp(1000000000) --> 1;

f(1, 1, 1/0);
--> if (1 > 0) 1 else 1/0              --> 1;
```

Conséquences de la pureté

- **Néanmoins, la pureté rend certaines choses compliquées** (mais non pas impossibles), par exemple :
  - La gestion des structures de données (cf. la **persistance**).
  - La gestion de la mémoire explicite (cf. les **ramasses-miettes**).
  - La définition des entrées/sorties, du traitement d'erreurs, (cf. les **monades**).
- **Il est possible de ne programmer qu'avec des fonctions pures** (par ex. Haskell est même un langage qui l'impose).

Indiquer explicitement la pureté

- **Certains langages permettent d'annoter les fonctions pour indiquer explicitement qu'elles sont pures.**
- Par exemple, en C# :

```
[Pure]
bool F(int i) {
    return (i + 4) > 0;
}

Contract.Requires (F(0));
```

- Et par exemple, dans la documentation officielle de C# :
"*Toutes les méthodes appelées dans un contrat doivent être pures ; autrement dit, elles ne doivent pas mettre à jour un état préexistant.*"

Conséquences de la première classe

La première classe des fonctions est déterminée par (rappel) :

0. **Pouvoir être nommé, affecté (et typé)**.
1. **Pouvoir être défini et créé à la demande**.
2. **Pouvoir être passé en argument à une fonction**.
3. **Pouvoir être le résultat d'une fonction**.
4. **Pouvoir être stocké dans une structure de données qlcq.**.

$\Rightarrow$ Nouveaux moyens de composition des fonctions
(cf. définition du paradigme fonctionnel).

Conséquences de la première classe

Pouvoir être passée en argument à une fonction induit la possibilité de **généralisation/abstraction fonctionnelle** (cf. aussi "**programmation d'ordre supérieur**") :

```
two_times_exp(x) {
    return exp(exp(x));
};
```

↓

```
two_times_f(x, f) {
    return f(f(x));
};
```

⇒ Tout "morceau de code" dans une fonction est remplaçable par une abstraction, i.e. un paramètre fonctionnel.
(c'est l'effet principal de la première classe pour les fonctions).

Conséquences de la première classe

Autre exemple de généralisation fonctionnelle (en Javascript) (implémentation d'un processus itératif particulier) :

```
function iterateUntil(init) {
  var ret = [];
  var result = init * 1000;
  while (result > 2000) {
    ret.push(result);
    result = result * 1000);
  }
  return ret;
};
```

```
function iterateUntil(fun, check, init) {
  var ret = [];
  var result = fun(init);
  while (check(result)) {
    ret.push(result);
    result = fun(result);
  }
  return ret;
};
```

(dû à M. Fogus)

Conséquences de la première classe

La généralisation fonctionnelle dans un cadre statiquement typé (e.g. Java, C++, C#) nécessite des types de fonctions
(en fait, les autres propriétés de la première classe aussi) :

```
float two_times_exp(float x) {
    return exp(exp(x));
}
```

↓

```
float two_times_f(float x, fun_t_float_float f) {
    return f(f(x));
}
```

Première classe et généricité

La généralisation fonctionnelle dans un cadre statiquement typé nécessite souvent des types génériques de fonctions
(car elle induit des fonctions avec des codes moins spécifiques) :

```
float two_times_f(float x, fun_t_float_float f) {
    return f(f(x));
}
```

```
T1 two_times_f(T1 x, fun_t(T1, T1) f) {
    return f(f(x));
}
```

Première classe et généricité

- $\Rightarrow$ **Définitions de types génériques de fonctions en C#** sous forme de types de **délégués** (cf. .NET System) :

```
Func(TResult) Delegate
Func(T1,TResult) Delegate
Func(T1,T2,TResult) Delegate
...
Func(T1,T2,T3,T4,T5,T6,T7,T8,T9,....,
                                T15,T16,TResult) Delegate
Action Delegate
Action(T1) Delegate
Action(T1,T2) Delegate
...
Action(T1,T2,T3,T4,T5,T6,T7,T8,T9,....,T15,T16) Delegate
```

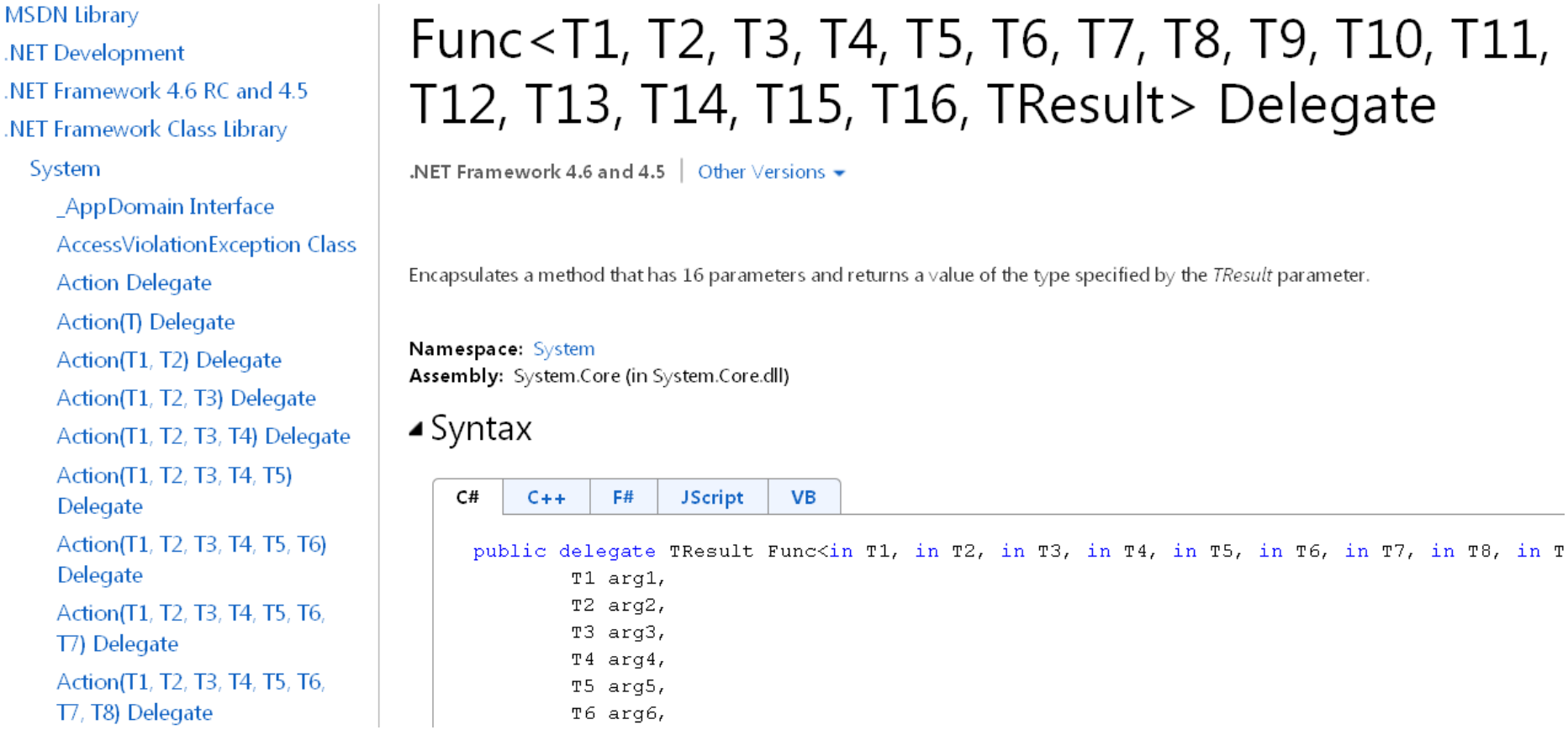
- **L'exemple précédent en C#** :

```
T1 TwoTimesFun<T1>(T1 x, Func<T1, T1> f) {
    return f(f(x));
}

TwoTimesFun(3.0, Math.Exp));   //call
```

Première classe et généricité

Extrait de la documentation de `.NET System...` :



MSDN Library
.NET Development
.NET Framework 4.6 RC and 4.5
.NET Framework Class Library
System
_AppDomain Interface
AccessViolationException Class
Action Delegate
Action(T) Delegate
Action(T1, T2) Delegate
Action(T1, T2, T3) Delegate
Action(T1, T2, T3, T4) Delegate
Action(T1, T2, T3, T4, T5) Delegate
Action(T1, T2, T3, T4, T5, T6) Delegate
Action(T1, T2, T3, T4, T5, T6, T7) Delegate
Action(T1, T2, T3, T4, T5, T6, T7, T8) Delegate

Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15, T16, TResult> Delegate

.NET Framework 4.6 and 4.5 | Other Versions

Encapsulates a method that has 16 parameters and returns a value of the type specified by the *TResult* parameter.

Namespace: System
Assembly: System.Core (in System.Core.dll)

Syntax

C# | C++ | F# | JScript | VB

```
public delegate TResult Func<in T1, in T2, in T3, in T4, in T5, in T6, in T7, in T8, in T
        T1 arg1,
        T2 arg2,
        T3 arg3,
        T4 arg4,
        T5 arg5,
        T6 arg6,
```

Première classe et généricité

- ⇒ **Définitions des types génériques de fonctions en Java 8** sous forme d'**interfaces fonctionnelles**, i.e. interfaces avec une seule méthode abstraite (cf. `java.util.function`) (mais donc sans surcharge de nommage comme en C#...) :

```
BiConsumer<T,U>, BiFunction<T,U,R>, BinaryOperator<T>, BiPredicate<T,U>,
BooleanSupplier, Consumer<T>, DoubleBinaryOperator, DoubleConsumer,
DoubleFunction<R>, DoublePredicate, DoubleSupplier, DoubleToIntFunction,
DoubleToLongFunction, DoubleUnaryOperator, Function<T,R>, IntBinaryOperator,
IntConsumer, IntFunction<R>, IntPredicate, IntSupplier, IntToDoubleFunction,
IntToLongFunction, IntUnaryOperator, LongBinaryOperator, LongConsumer,
LongFunction<R>, LongPredicate, LongSupplier, LongToDoubleFunction,
LongToIntFunction, LongUnaryOperator, ObjDoubleConsumer<T>, ObjIntConsumer<T>,
ObjLongConsumer<T>, Predicate<T>, Supplier<T>, ToDoubleBiFunction<T,U>,
ToDoubleFunction<T>, ToIntBiFunction<T,U>, ToIntFunction<T>, ToLongBiFunction<T,U>,
ToLongFunction<T>, UnaryOperator<T>
```

- **L'exemple précédent en Java** :

```
<T1> T1 twoTimesFun(T1 x, Function<T1, T1> f) {
    return f.apply(f.apply(x));
}

twoTimesFun(3.0, x -> Math.exp(x));   //call
```

Première classe et généricité

Extrait de la documentation de l'API Java SE8...

(`java.util.function`) :

Interface Summary

Interface	Description
BiConsumer<T,U>	Represents an operation that accepts two input arguments and returns no result.
BiFunction<T,U,R>	Represents a function that accepts two arguments and produces a result.
BinaryOperator<T>	Represents an operation upon two operands of the same type, producing a result of the same type as the operands.
BiPredicate<T,U>	Represents a predicate (boolean-valued function) of two arguments.
BooleanSupplier	Represents a supplier of `boolean`-valued results.
Consumer<T>	Represents an operation that accepts a single input argument and returns no result.
DoubleBinaryOperator	Represents an operation upon two `double`-valued operands and producing a `double`-valued result.
DoubleConsumer	Represents an operation that accepts a single `double`-valued argument and returns no result.
DoubleFunction<R>	Represents a function that accepts a double-valued argument and produces a result.
DoublePredicate	Represents a predicate (boolean-valued function) of one `double`-valued argument.

Première classe et généricité

Remarque : **l'exploitation des propriétés de la première classe peut être allégée par des mécanismes d'inférence de type** :

- Langages avec inférence complète des types génériques fonctionnels :

```
two_times_f (x, f) = f (f (x));;                    // Haskell
    --> two_times_f : (t, (t -> t)) -> t

let two_times_f (x, f) = f (f (x));;                // OCaml
    --> val two_times_f : 'a * ('a -> 'a) -> 'a
```

- Langages avec plus ou moins d'inférence de type :

```
auto f = [](double x) {return exp(exp(x));};// C++
def f[T] = ((x:T, f:T=>T) => f(f(x)))               // Scala
```

- C# offre un "var", mais trop faible pour les types de fonctions ; Java n'a encore que quelques zestes d'inférence de type.

Conséquences de la première classe

Pouvoir être défini et créé à la demande est réalisé par les "**fonctions anonymes**" (par défaut, les valeurs sont sans nom), aussi appelées **lambdas** ou **fermetures** :

```
x -> x + 1                                  // Java
x => x + 1                                  // C#
[](int x) -> { return x + 1; }              // C++
function(x) { return x + 1 }                // Javascript
lambda x: x + 1                             // Python
\x -> x + 1                                 // Haskell
[ :x | x + 1 ]                              // Smalltalk
x => x + 1                                  // Scala
fun x -> x + 1                              // OCaml
```

Les fonctions anonymes servent à "nourrir" à la volée les variables, les données, les paramètres fonctionnels, par ex. :

```
two_times_f(1, (x -> x + 1))  -->  3
```

Conséquences de la première classe

- **La généralisation fonctionnelle + fonctions anonymes** :

  $\Rightarrow$ **intégration facile de nouveau code dans code existant.**
  $\Rightarrow$ **flexibilité du code, maintenabilité, extensibilité.**

  C'est une des principales utilisations des fonctions de première classe dans la programmation de tous les jours.

- Exemple principal d'application : la généralisation des traitements itératifs sur les structures de données par trois types de fonctions appelées **maps**, **reduces/folds**, et **filtres** :

Conséquences de la première classe

Généralisation fonctionnelle des traitements itératifs :

a. Les "**maps**" : transformations uniformes de structures (cf. "**itérateurs internes**").

```
list_map (list, f) {
    if (isEmpty list)
        return emptyList;
    else
        return addFirst(f(getFirst(list)),
                        list_map(getTail(list), f));
}
```

```
list_map([1,2,3,4], (x -> x + 1))  --> [2,3,4,5]
list_map([1,2,3,4], (x -> x * 2))  --> [2,4,6,8]
```

Conséquences de la première classe

Généralisation fonctionnelle des traitements itératifs :

b. Les "**reduces**" (ou "**folds**") : calculs de valeurs à partir d'un parcours sur tous les éléments d'une structure.

```
list_reduce (list, f, startvalue) { ... }
```

```
list_reduce ([1,2,3,4], ((x,y) -> x + y), 0)  --> 10
list_reduce ([1,2,3,4], ((x,y) -> x + 1), 0)  --> 4
```

Conséquences de la première classe

Généralisation fonctionnelle des traitements itératifs :

c. Les **filtres** : extractions des éléments d'une structure qui vérifient un prédicat.

```
list_filter (list, pred) { ... }


list_filter([1,2,3,4], (x -> x > 2))  --> [3,4]
list_filter([1,2,3,4], (x -> x < 2))  --> [1]
```

Conséquences de la première classe

- $\Rightarrow$ **La plupart des langages fonctionnels proposent des maps/reduces/filtres dans leur bibliothèques**
  (en une sorte de "couteau suisse" des structures de données) :

```
map, reduce, filter              // Java
Select, Aggregate, Where         // C#
map, reduce, filter              // Javascript
map, fold, filter                // OCaml
collect:, reduce:, select:       // Smalltalk
```

- Et souvent avec beaucoup de dérivés,
  par exemple dans `Underscore.js` :

```
each, reduceRight, find, where, findWhere, reject, every,
some, contains, invoke, pluck, max, min, sortBy, groupBy,
indexBy, countBy, shuffle, sample, toArray, size, partition.
```

Conséquences de la première classe

Les ensembles de fonctions maps/reduces/filters sont aussi utilisés pour énoncer des compositions de traitements à la manière de requêtes bases de données :

- **En C#** (cf. LINQ, Language Integrated Query) :

```
IEnumerable<int> q = giraffes
  .Select(g => g.Age * 3);
  .Where(g => g.Age > 2)
  .OrderByDescending(g => g.Age)
```

- **En Java 8** (java.util.stream.ReferencePipeline) :

```
Stream<Integer> q = giraffes.stream()
  .map(g -> g.age * 3);
  .filter(g -> g.age > 2)
  .sorted((g1,g2) -> Integer.compare(g1.age, g2.age))
```

Conséquences de la première classe

Pouvoir être le résultat d'une fonction permet :

- Adaptations de fonctions, définitions d'appels particuliers (cf. **fonctions "around"**, et aussi "**décorateurs**" en Python) :

```
list_map_scaling(k) {
  return function(l) {
    return list_map(l, function(n) {
                         return (n * k);
                       });
  };}
```

- **Applications partielles** (cf. **fonctions "curryfiées"**) :

```
curried_add(a) {
  return function(b) {
    return a + b;
  };}

var add5 = curried_add(5);
add5(3) --> 8
```

Conséquences de la première classe

Pouvoir être stocké dans une structure de données permet :

- Gestion d'ensembles de fonctions, structurés en listes, tables, arbres, graphes, etc. (cf. "**modularité dynamique**").

```
graphic_actions := [
      ('f1', (x, y) -> action1(x, y)),
      ('f2', (x, y) -> action2(x, y)),
      ('f3', (x, y) -> action3(x, y)),
      ...
];
```

- Passages d'arguments sous forme d'ensembles de fonctions (au lieu de simples fonctions) :

```
graphic_environment(graphic_actions);
```

- $\Rightarrow$ **Programmation orientée-données** (*data-driven*).

Conséquences de la première classe

Certaines techniques de programmation fonctionnelle utilisent la pureté + la première classe :

- **Les techniques "mapreduce"** : traitements itérés généralisés à base de fonctions pures permettant une parallélisation facile.

```
reduce (map(list, purefun1), purefun2)
```

- **Le contrôle de l'évaluation par émulation fonctionnelle** : expressions pures encapsulées dans des fonctions sans paramètres (évaluation à la demande par appel de ces fonctions).

```
expression  ⇝  function() { expression }
```

Par exemple :

```
benchMarkAndTest(function() { exp(1000000000); })
```

III. Pourquoi le paradigme fonctionnel est-il désormais souvent intégré au paradigme objet ?

Programmation fonctionnelle et objet

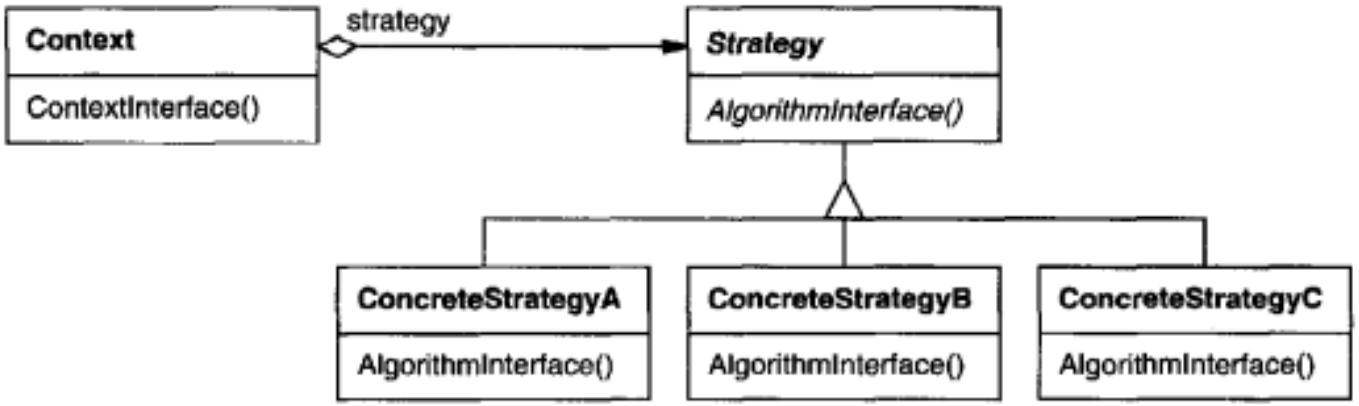
- **Les langages objets principaux Java, C++ et C# sont basés sur les "classes-modules"** (*"classes as modules"*).
- Une des idées fortes du développement en programmation objet : **les actions de maintenance, d'extension, d'adaptation peuvent passer par l'héritage et la composition de classes** (on évite ainsi toute modification du code existant).
- $\Rightarrow$ Parfois trop lourd... ! Prolifération de classes... !
- Par ex. en Java < 8, pour contrer cet effet : utilisation de **classes internes anonymes**.
- **La programmation fonctionnelle est une solution à ce problème.**

Un design pattern classique : la Stratégie

- Par ex. le design pattern "**Stratégie**" consiste à "*laisser un algorithme varier indépendamment des clients qui l'utilisent*".
- **Tactique objet d'implémentation** : composition d'objets.

```
class Context {
    Strategy strategy;
    Context(Strategy s) {strategy = s;}
    float context(float x) {
        return strategy.algorithm(x);
    }
}

Composition c = new Composition(instanciatedStrategy);
```



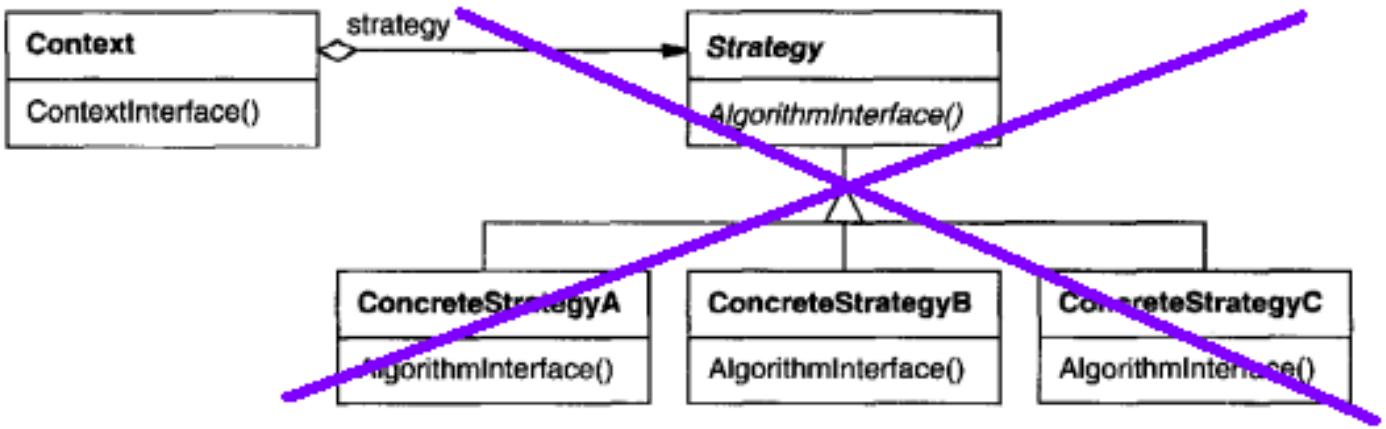
(issu du GoF)

Un design pattern classique : la Stratégie

- **Tactique fonctionnelle d'implémentation** : passage de fonctions en paramètre (cf. première classe) (en Java 8) :

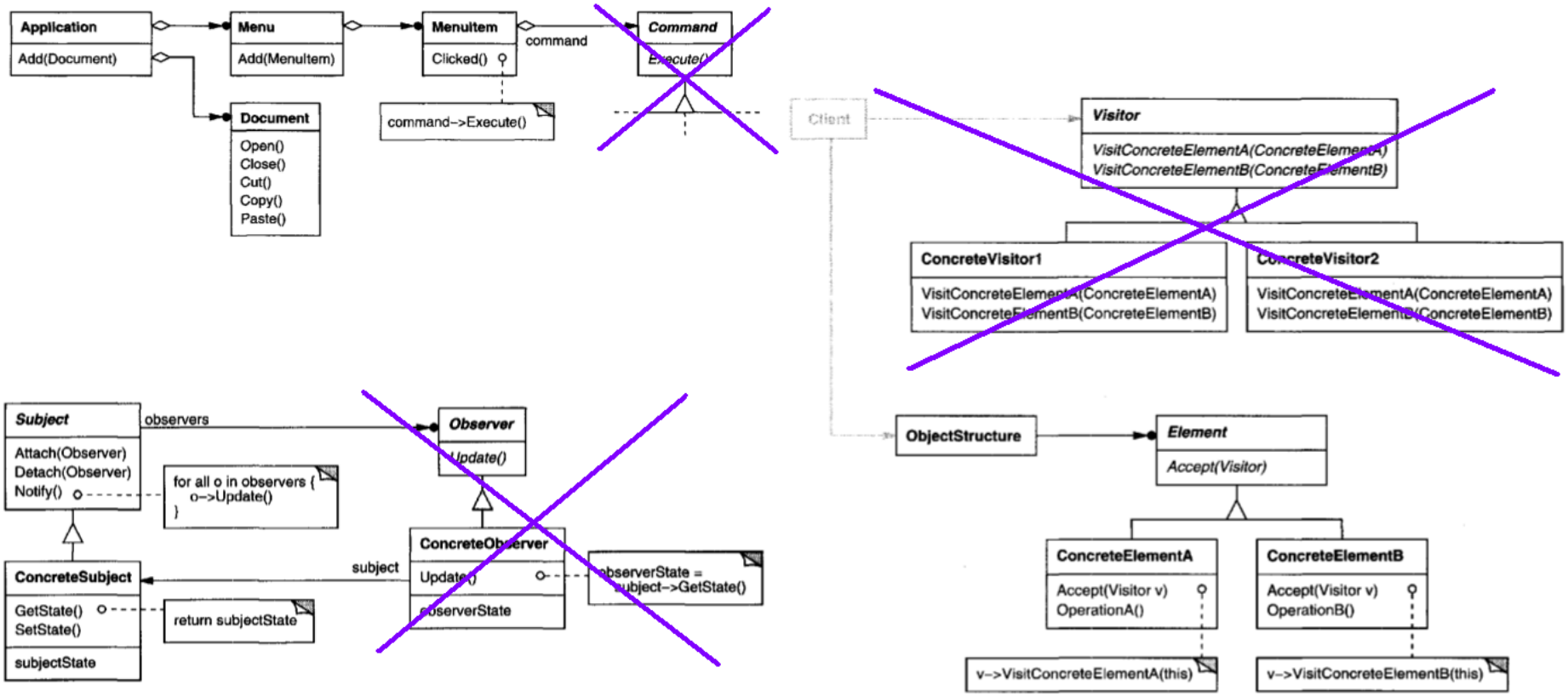
```
class Context  {
   Function<Float, Float> algorithm;
                  // predefined functional interface
   Context(Function<Float, Float> a) {algorithm = a;}
   float context(float x) {
      return algorithm.apply(x);
   }
}

Context c = new Context(x -> x + 1);
```



Autres design patterns...

D'autres design patterns, e.g. Commande, Observateur, Visiteur peuvent bénéficier de la première classe des fonctions :



⇒ La granularité fine de la programmation fonctionnelle peut agir efficacement sur la granularité générale modulaire.

Intégrer la programmation fonctionnelle

- Règle : **un langage qui intègre la première classe des fonctions doit faciliter la programmation fonctionnelle et la promouvoir** (cf. déf. de langage fonctionnel).
- Par exemple en Java 8 , l'intégration de la programmation fonctionnelle a induit des modifications de la notion d'interface :
  a. Les types de fonctions sont représentés par des **interfaces fonctionnelles**, instanciables par des lambdas.
  b. Les interfaces avec implémentations de **méthodes par défaut** :
     - $\Rightarrow$ Interfaces fonctionnelles plus complexes.
     - $\Rightarrow$ L'API Java 8 compatible ascendante avec les API Java < 8.
- Par ex., le "couteau suisse" sur les structures de données :

  $\Rightarrow$ Dans `Collection<E>`, ajoûts suivants (par rapport à Java 7) :

```
default boolean removeIf(Predicate<? super E> filter)
```

  $\Rightarrow$ Dans `Iterable<E>`, ajoûts suivants :

```
default void forEach(Consumer<? super T> action)
```

Ce qu'on peut retenir ?

- **La programmation fonctionnelle est bonne pour le développement de programmes** : pureté et première classe induisent une part de stabilité, déterminisme, testabilité, cloisonnement, fluidité d'utilisation, compositionalité, généralisation, extensibilité, etc.
- **La programmation fonctionnelle est bonne pour l'objet-modulaire** : elle met un peu d'huile dans les rouages, et amoindrit parfois la complexité des architectures.
- **La programmation fonctionnelle est soluble** : pureté et première classe des fonctions peuvent être considérées, incluses et facilitées dans n'importe quel langage.