

Eiffel et OCaml—deux langages, deux approches
pour obtenir la qualité des logiciels

Ross Gardiner

18 février 2016

Table des matières

Introduction	2
Eiffel	4
La conception par contrat	4
La séparation commande-requête	5
La sécurité contre le dérèférencement de pointeurs nuls	6
Réutilisabilité et extensibilité	7
OCaml	10
La programmation « fonctionnelle » et l’immuabilité par défaut . . .	10
L’inférence de types et la concision	11
L’application partielle et la composition de fonctions	12

Introduction

De nos jours, les logiciels sont partout : portables, avions, lave-vaisselles, lampes. Pour citer Marc Andreessen, « les logiciels mangent le monde ». Malheureusement, ces logiciels sont souvent bogués, même dangereux.

Depuis de nombreuses années, l'industrie du logiciel essaye de trouver des méthodes pour améliorer la qualité de son logiciel. Plusieurs éminents informaticiens français ont développé des nouveaux langages à cette fin. Dans ce projet, je décrirai deux langages de programmation : Eiffel et OCaml. Les deux tentent d'assurer la sécurité contre les bogues et les comportements inattendus, mais ils ont des approches assez différentes. Je vais examiner et comparer leurs fonctionnements et leurs caractéristiques.

Bien qu'OCaml—datant de 1996—soit plus récent qu'Eiffel—de 1986—ses origines sont plutôt plus anciennes. OCaml dérive de ML, développé à Édimbourg en 1973. OCaml lui-même a débuté à l'INRIA, un institut de recherche français. Eiffel a été créé par Bertrand Meyer, un informaticien français qui est actuellement professeur de génie logiciel à l'ETH Zurich.

Eiffel est un langage « orienté objet », ce qui signifie qu'il modèle le monde comme une collection d'objets. Chaque objet peut effectuer des actions (« commandes »), et chaque objet a des propriétés (« requêtes »).

Par contre, OCaml est un langage multi-paradigme—mais, premièrement, c’est un langage fonctionnel. Dans un tel langage, les informations et les actions sont strictement séparées. Nous examinerons les conséquences de ces conceptions dans le reste du projet.

Eiffel

Eiffel a été conçu avec un seul objectif : améliorer la qualité de logiciel. La plupart de sa fonctionnalité est destinée à servir cet objectif.

La conception par contrat

La caractéristique la plus inhabituelle d'Eiffel est « la conception par contrat ». Très souvent, lorsque les informaticiens veulent prouver qu'un algorithme est correct, ils utilisent les « préconditions », les « postconditions » et les « invariants ». Si on examine un fragment de code, une précondition est un fait qui doit être vraie juste avant l'exécution du code. De même, une postcondition doit être vraie juste après l'exécution. Un invariant doit être vrai en tout temps.

La plupart des langages n'encodent pas ces concepts mathématiques, mais Eiffel l'incite. La Figure 1 montre un exemple. Nous définissons une classe qui représente un compte bancaire. C'est très simple. Premièrement, il y a un attribut `solde` qui stocke la somme d'argent qui est actuellement dans le compte. En outre, il y a une routine `dépose` qui prend en argument un montant d'argent pour ajouter au compte. Le bloc `require` contient une précondition : nous ne pouvons pas ajouter un montant négatif au compte. Le bloc `ensure` contient

une postcondition : après avoir déposé un montant, le solde du compte doit être le solde initial plus la somme ajoutée.

```
1  class COMPTE_BANCAIRE
2
3  feature
4    solde: INTEGER
5
6    dépose (somme: INTEGER)
7      require
8        somme >= 0
9      do
10        solde := solde + somme
11      ensure
12        solde = old solde + somme
13    end
14  end
```

FIGURE 1 – Un exemple d'utiliser une précondition et une postcondition ¹

La séparation commande-requête

Cet exemple montre aussi le principe de séparation commande-requête. `solde` est une requête—elle retourne le solde, mais elle ne peut pas changer l'état du compte. En revanche, `dépose` est une commande. Elle ne donne un résultat, mais elle changera l'état du compte. Ce principe rassure le programmeur que l'extraction du solde ne causera pas des effets secondaires inattendus.

1. Adapté de <https://archive.eiffel.com/doc/online/eiffel50/intro/language/invitation-07.html>

```

1  class APPLICATION
2
3  feature
4    principal
5      local
6        compte: COMPTE_BANCAIRE
7      do
8        print (compte.somme)
9      end
10 end

```

FIGURE 2 – Un exemple d'un déréférencement d'un pointeur nul

```

1  class APPLICATION
2
3  feature
4    principal
5      local
6        compte: COMPTE_BANCAIRE
7      do
8        create compte
9        print (compte.somme)
10      end
11 end

```

FIGURE 3 – Un exemple d'une initialisation correcte d'un pointeur

La sécurité contre le déréférencement de pointeurs nuls

Eiffel empêche aussi un problème rencontré souvent dans d'autres langages—le déréférencement de pointeurs nuls. Ceci se produit lorsqu'on crée une référence mais ne l'attache jamais à un objet. Comparez les figures 2 et 3. Figure 2 montre une situation courante dans les autres langages. Une référence `compte` est déclarée par la ligne 6, mais aucun objet réel est créé. Quand l'instruction `print (compte.somme)` (ligne 8) est exécutée, le logiciel plantera. Figure 3 est une ver-

sion correcte : compte est initialisé par l'instruction `create compte` (ligne 8). Eiffel détecte automatiquement la version incorrecte, qui empêche beaucoup de plantages.

Réutilisabilité et extensibilité

Deux des grands principes d'Eiffel sont réutilisabilité et extensibilité. Ces mots résume l'idée que les logiciels doivent—autant que possible—être réutilisés ou modifiés plutôt que développés à partir de zéro. Le but est—dans tout nouveau système—de maximiser la quantité du code qui a déjà fait ses preuves.

Eiffel offre toutes les méthodes normales pour permettre la réutilisabilité et l'extensibilité—par exemple l'héritage, les bibliothèques communes et les classes génériques. Cependant, Eiffel a une approche intéressant pour faire l'héritage multiple.

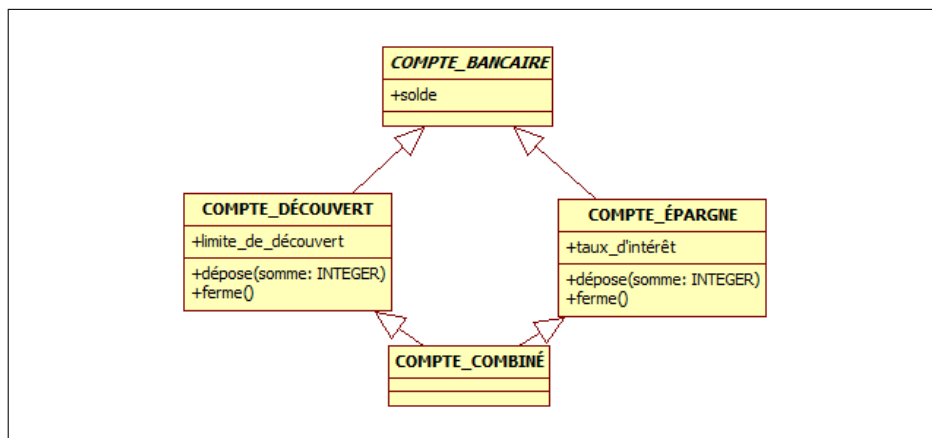


FIGURE 4 – Un exemple du héritage multiple

Dans la figure 4 il y a un exemple du héritage multiple. Nous définissons un simple compte bancaire avec juste un solde. Les types plus précis de compte sont un compte avec un solde à découvert, un compte d'épargne et un compte

qui combine les deux. Les deux classes COMPTE_DÉCOUVERT et COMPTE_ÉPARGNE définissent les routines supplémentaires `dépose(somme: INTEGER)` et `ferme`. Que devrait faire le COMPTE_COMBINÉ quand ces routines s'appellent ? Eiffel propose quelques solutions.

```
1  class COMPTE_COMBINÉ
2    inherit COMPTE_DÉCOUVERT
3    redefine
4      dépose
5    rename
6      ferme as ferme_découvert
7  end
8
9  inherit COMPTE_ÉPARGNE
10  redefine
11    dépose
12  rename
13    ferme as ferme_épargne
14  end
15
16  feature
17    dépose (somme: INTEGER)
18    do
19      ...
20    end
21  end
```

FIGURE 5 – Un exemple d'héritage multiple avec Eiffel

La figure 5 montre deux des outils de gestion d'héritage multiple. `redefine dépose` (ligne 3) signifie que les mises en oeuvre de `dépose` (par `COMPTE_DÉCOUVERT` et `COMPTE_ÉPARGNE`) sont à être ignorées et remplacées par une nouvelle définition dans `COMPTE_COMBINÉ`.

Par ailleurs, nous pouvons aussi renommer les fonctionnalités : ici, le `ferme` de `COMPTE_DÉCOUVERT` devient `ferme_découvert` (ligne 5). Quoiqu'on fasse, Eiffel garantit que les préconditions, les postconditions et les invariants sont observés.

Ce système permet la réutilisation de plusieurs classes dans une seule classe dérivée.

OCaml

Comparé à Eiffel, OCaml n'est pas aussi concentré sur la prévention des erreurs. Malgré cela, une partie significative de sa fonctionnalité est utile pour écrire du code stable.

La programmation « fonctionnelle » et l'immutabilité par défaut

OCaml incitera fortement un style de la programmation « fonctionnelle ». Le style traditionnel de la programmation est « impératif ». Dans celle-là, on évite de causer des effets secondaires—alors que dans celle-ci les effets secondaires sont la norme. Un effet secondaire clé est la modification des données (la « mutation »).

La figure 6 montre un organigramme qui est mis à jour dans un langage impératif. Après l'opération, la référence originale (organigramme) pointe à la même structure d'arbre, mais la structure a été modifiée. N'importe quel code qui utilise cet organigramme doit savoir que d'autres parties de code peuvent le modifier.

La figure 7 montre un organigramme dans un langage fonctionnel. L'opération

ne modifie pas l'arbre original, mais elle plutôt crée une nouvelle référence et elle fabrique une nouvelle copie seulement de la partie d'arbre qui change. Les deux versions du organigramme peuvent bien coexister. Le code qui utilise la référence organigramme peut supposer que les données ne changent jamais.

Nous reconnaissons depuis peu que les structures des données immuables sont souvent préférable. Elles facilitent de raisonner sur le comportement d'un programme parce qu'on n'a pas à se soucier de la possibilité que les données peuvent être changées de manière imprévue.

L'inférence de types et la concision

OCaml et Eiffel sont langages à typage statique. Cela signifie qu'on sait le type de chaque valeur dans un programme—par exemple : un nombre entier, un nombre réel, une chaîne de caractères, un organigramme. Cependant, il y a une différence clé. En Eiffel, on doit indiquer le type d'une valeur très explicitement.

La figure 8 montre une fonction en OCaml qui construit un message. La fonction prend deux arguments : `verbeux`, qui indique si le résultat devrait être verbeux, et `message`, qui est le message verbeux. Le message non-verbeux est une chaîne vide. On ne précise jamais les types exacts de `verbeux` ou `message` : OCaml les calcule automatiquement. `verbeux` doit être booléen, également `message` doit être une chaîne de caractères.

Comparez avec Eiffel (la Figure 9) : on doit écrire `: BOOLEAN` et `: STRING` pour indiquer les types. Le code est beaucoup plus long. La force d'OCaml est qu'il permet aux programmeurs d'écrire des programmes concis et compréhensible. Ces qualités rendent la maintenance plus facile.

2. Adapté de <https://www.cis.upenn.edu/~sweirich/icfp-plmw15/slides/pottier.pdf>

L'application partielle et la composition de fonctions

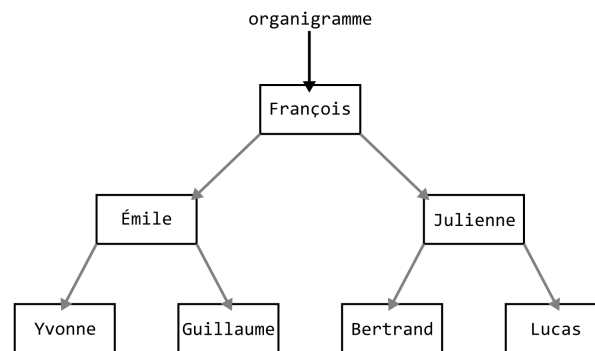
Un autre aspect intéressant d'OCaml est l'application partielle de fonctions. Ceci utilise la curryfication, qui transforme une fonction à plusieurs arguments à une série de fonctions à un argument. Par exemple, considérez la fonction `add x y = x + y`, qui prend deux nombres entiers et renvoie la somme. Évidemment, `add 2 3` renvoie 5.

On peut considérer cette fonction comme `(entier, entier) -> entier` : elle prend deux entiers et renvoie un entier. En fait, OCaml considère la fonction comme `entier -> (entier -> entier)` : une fonction qui prend un entier et renvoie une deuxième fonction, qui prend un entier et renvoie un entier.

On peut appliquer la fonction partiellement ainsi : `add6 = (add 6)`. Ceci rend `add2` comme une fonction `entier -> entier`. Elle prend un entier et le renvoie plus six. Par exemple, `add6 3` renvoie 9.

L'application partielle, entre autres outils, est un mécanisme puissant de combinaison de fonctions. Cela permet la réutilisation simple et fiable du code, un facteur important pour la qualité.

Avant



Après

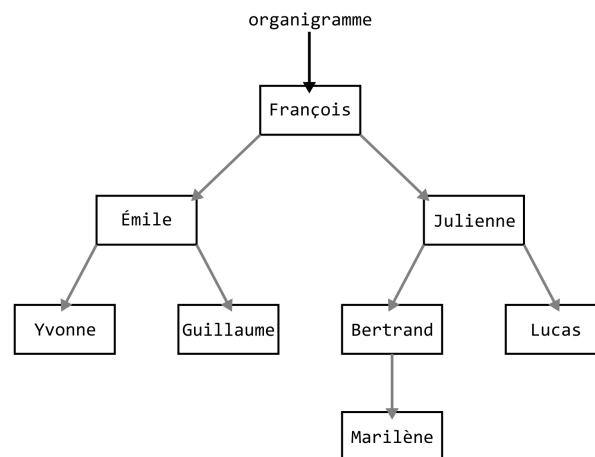


FIGURE 6 – Un exemple d'une structure des données mutable

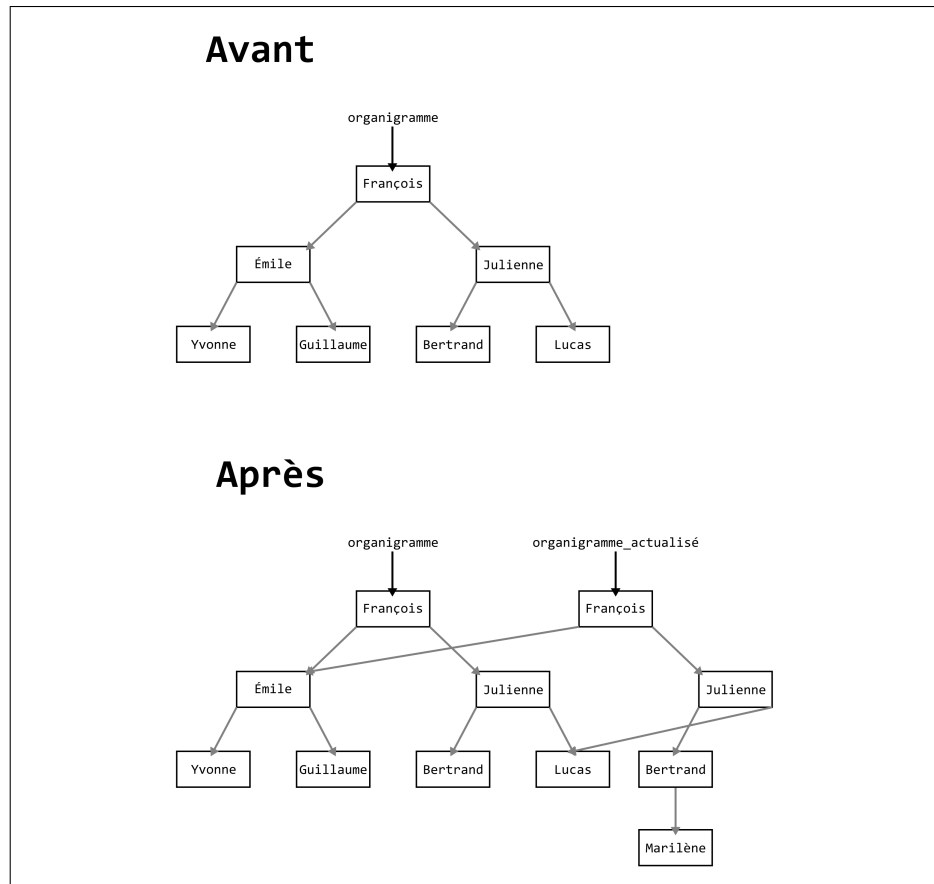


FIGURE 7 – Un exemple d’une structure des données immuable

```

1 let construis_message verbeux message =
2   if verbeux then message else ""

```

FIGURE 8 – Un exemple de l’inférence de types en OCaml²

```
1  construis_message (verbeux: BOOLEAN;  
2                      message: STRING): STRING  
3  do  
4      if verbeux then  
5          Result := message  
6      else  
7          Result := ""  
8      end  
9  end
```

FIGURE 9 – Une requête qui équivaut à la Figure 8