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

Ross Gardiner (rg14820)

Table des matières

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 [1] et OCaml [2]. 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[1]—soit plus récent qu'Eiffel—de 1986[3]—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'EPFZ.

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.

1. `https://www.eiffel.com/`
2. `http://ocaml.org/`

Eiffel

Eiffel a été conçu avec un seul objectif : améliorer la qualité logicielle. 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 `déposer` qui prend en argument un montant d'argent à ajouter au compte. Le bloc `require` contient une précondition : nous ne pouvons pas ajouter un montant négatif au compte. Le block `ensure` contient une postcondition : après avoir déposé un montant, le solde du compte doit être le solde initial plus la somme ajoutée.

La séparation commande-requête

Cet exemple montre aussi le principe de « séparation commande-requête ». `solde` est une requête—elle renvoie le solde, mais elle ne peut pas changer l'état du compte. En revanche,

3. Adapté de `https://archive.eiffel.com/doc/online/eiffel50/intro/language/invitation-07.html`

```
1  class COMPTE_BANCAIRE
2
3  feature
4    solde: INTEGER
5
6    déposer (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'utilisation d'une précondition et d'une postcondition [3]

déposer est une commande. Elle ne donne un résultat, mais elle changera l'état du compte. Ce principe garantit aux programmeurs que l'extraction du solde ne causera pas d'effets secondaires inattendus.

```
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

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 lui attache jamais à un objet. Comparez les figures 2 et 3. La figure 2 montre une situation courante dans les autres langages. Une référence `compte` est déclarée par la 6^{e} ligne, mais aucun objet réel n'est créé. Quand l'instruction `print (compte.somme)` (ligne 8) est exécutée, le logiciel plante. La figure 3 est une version correcte : `compte` est initialisé par l'instruction `create compte` (ligne 8). Eiffel détecte automatiquement la version incorrecte. Cette détection empêche beaucoup de plantages.

```
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

Réutilisabilité et extensibilité

Deux des grands principes d'Eiffel sont la réutilisabilité et l'extensibilité[4]. 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 intéressante pour gérer l'héritage multiple.

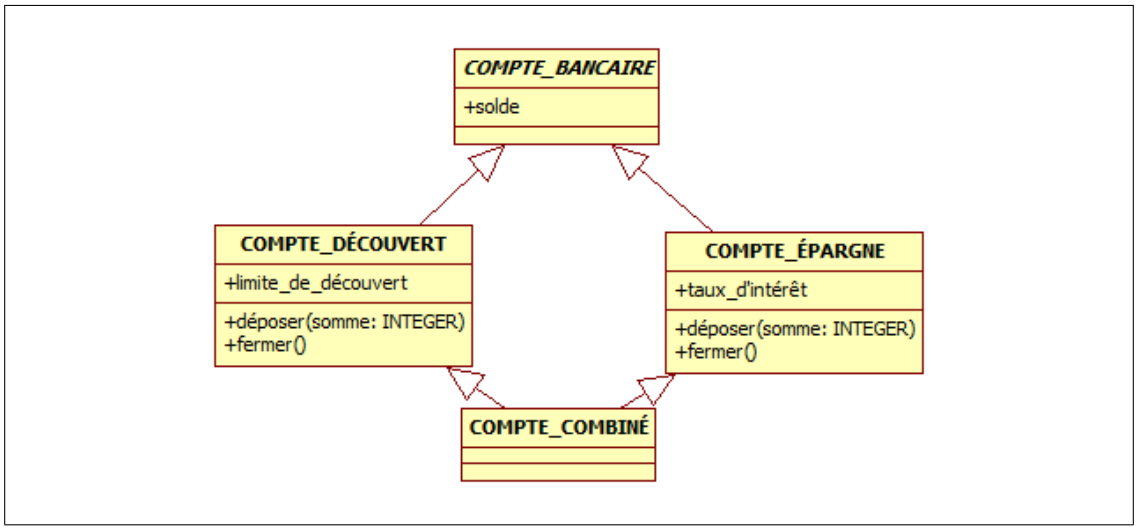

FIGURE 4 – Un exemple du héritage multiple

Dans la figure 4, il y a un exemple d'héritage multiple. Nous définissons un compte bancaire simple avec juste un solde. Les types plus précis de compte sont un compte avec un solde à découvert, un compte épargne et un compte qui combine les deux. Les deux classes `COMPTE_DÉCOUVERT` et `COMPTE_ÉPARGNE` définissent les routines supplémentaires `déposer(somme: INTEGER)` et `fermer`. Que devrait faire le `COMPTE_COMBINÉ` quand ces routines sont appelées ? Eiffel propose quelques

solutions.

```
1  class COMPTE_COMBINÉ
2    inherit COMPTE_DÉCOUVERT
3      redefine
4        déposer
5      rename
6        fermer as fermer_découvert
7    end
8
9    inherit COMPTE_ÉPARGNE
10     redefine
11       déposer
12     rename
13       fermer as fermer_épargne
14   end
15
16   feature
17     déposer (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 déposer` (ligne 3) signifie que les mises en oeuvre de `déposer` (par `COMPTE_DÉCOUVERT` et `COMPTE_ÉPARGNE`) doivent 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 ne porte pas 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'immuabilité par défaut

OCaml incitera fortement un style de programmation « fonctionnelle ». Le style traditionnel de la programmation est « impératif ». Dans la programmation fonctionnelle, on évite de causer des effets secondaires ; dans la programmation impératif, 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 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 crée plutôt une nouvelle référence et elle fabrique seulement une nouvelle copie de la partie d'arbre qui change. Les deux versions d'organigramme peuvent 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 préférables. Elles facilitent à 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.

4. Adapté de `https://www.cis.upenn.edu/~sweirich/icfp-plmw15/slides/pottier.pdf`

L'inférence de types et la concision

OCaml et Eiffel sont des langages à typage statique. Cela signifie qu'on connaît 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. À Eiffel, on doit indiquer le type de chaque valeur très explicitement.

La figure 8 montre une fonction 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, `message` doit également 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 compréhensibles. Ces qualités rendent la maintenance plus facile.

L'application partielle et la composition de fonctions

Un autre aspect intéressant d'OCaml est l'application partielle de fonctions. Elle utilise la curryfication, qui transforme une fonction à plusieurs arguments en 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 construit `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.

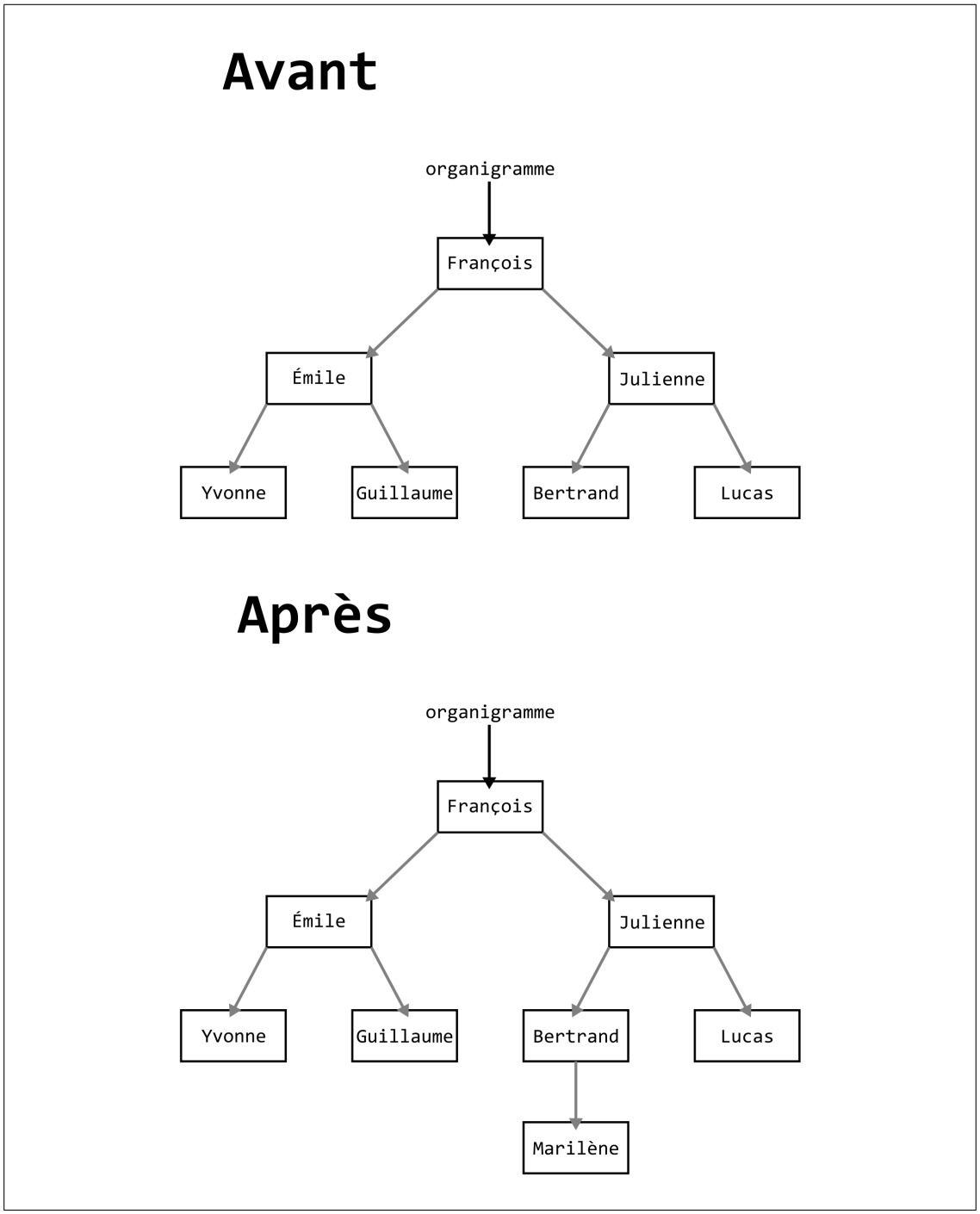

FIGURE 6 – Un exemple d'une structure des données mutables

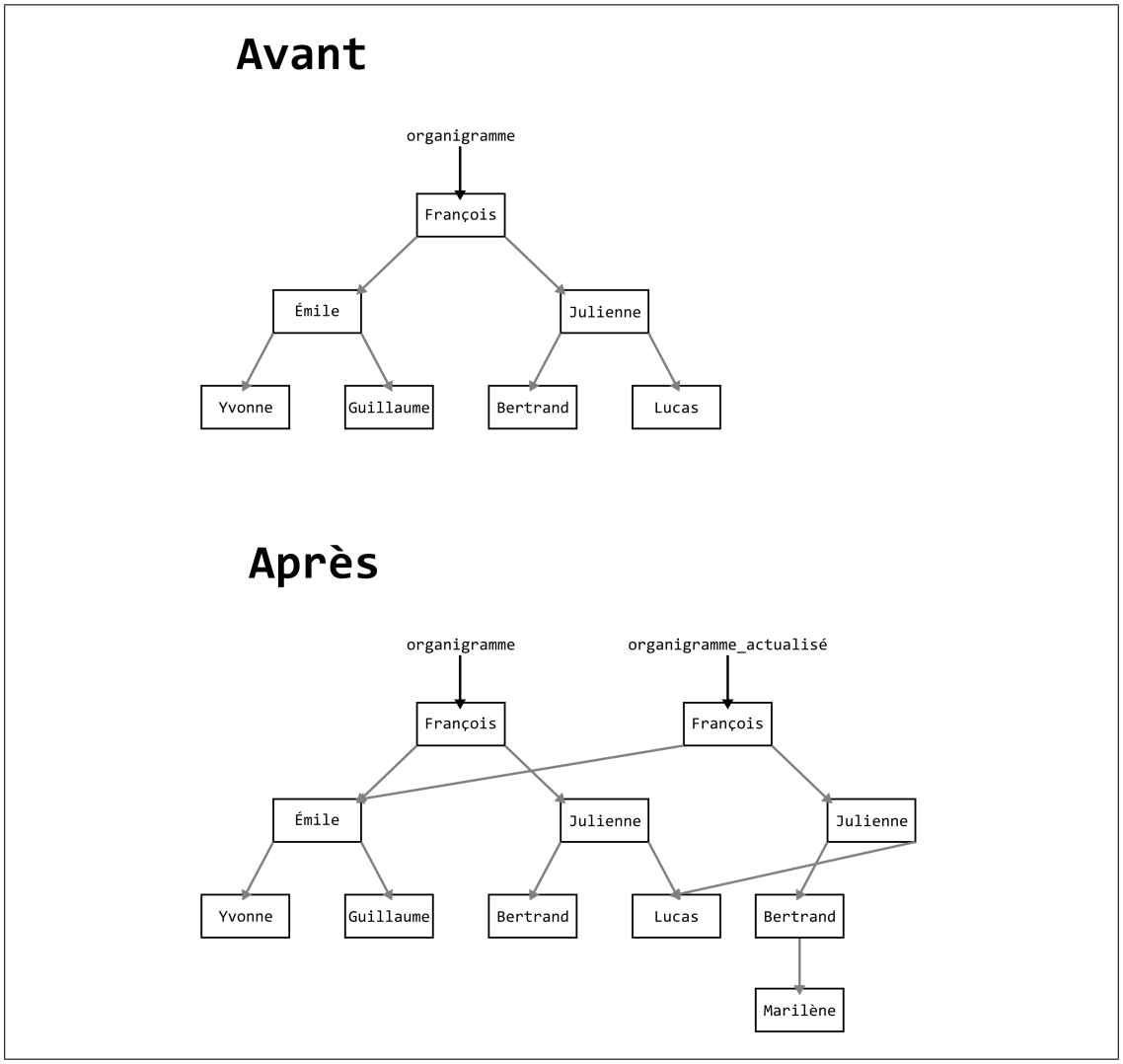

FIGURE 7 – Un exemple d'une structure des données immuables

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

FIGURE 8 – Un exemple de l'inférence de types en OCaml [4]

```
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

Une comparaison

Il est intéressant d'observer que, même si OCaml encourage naturellement un style strict de programmation, il ne l'impose pas formellement. Par contre, les programmes Eiffel ne sont pas écrits dans un style très strict, mais ils respectent rigoureusement les autres règles (les préconditions, les postconditions, les pointeurs nuls etc.). Comparons comment les deux langages proposent des différents types de sécurité :

L'organisation et l'encapsulation

Une bonne organisation est essentielle pour faciliter l'entretien. De plus, il est important d'encapsuler les parties d'un système afin qu'elles n'interfèrent pas entre elles ni exposent leur fonctionnement interne.

Dans OCaml, l'unité organisationnelle est le `module`. Dans Eiffel, c'est la classe (`class`). La différence principale est qu'une classe signifie un objet ; il peut exister de nombreuses instances d'une classe. Les `modules` signifient une collection d'éléments liés—par exemple la définition d'un type `liste` et les opérations relatives comme `filtrer` et `insérer`. On peut instancier la `liste` elle-même, mais pas le module entier.

Il y a des avantages et des inconvénients dans les deux approches. Lorsque l'on utilise les `modules`, il est souvent plus difficile de trouver toutes les opérations disponibles ou de comprendre comment les `modules` interagissent. Le fonctionnement interne est souvent exposé, ce qui peut être confus.

D'autre part, les classes fournissent d'habitude uniquement les opérations pertinentes, ce qui aide à une bonne encapsulation en cachant les détails de mise en œuvre. Cependant les instances d'une classe maintiennent un état interne qui peut être désynchronisé et changé de manière imprévue.

La réutilisation du code

Pouvoir réutiliser facilement du code est une panacée. Le code réutilisé est déjà conçu, déjà écrit et déjà testé. Tous les fonctionnalités qui aident la réutilisabilité sont bénéfiques.

Les deux langages ont des mécanismes très différents de réutilisation. Les unités de réutilisation d'OCaml sont des fonctions. On peut composer des fonctions (peut-être écrites par des personnes différentes) pour créer une chaîne de transformations de données. L'unité d'Eiffel est la classe. Les classes forment un réseau interconnecté, dans lequel chaque classe parle à plusieurs autres. Cette interdépendance rend la réutilisation plus difficile, malgré de nombreux outils pour l'héritage. Parce que les fonctions d'OCaml ont d'habitude moins de dépendances, elles sont plus simples à réutiliser.

Le système de types et l'exactitude du logiciel

Le logiciel est de haute qualité seulement s'il est correct. Un bon système de types est, par exemple, un moyen crucial pour assurer que les données dans un programme sont du type correct. Les valeurs imprévues causent des pannes et des comportements incorrects.

Alors que les deux langages sont fortement typés, il ne fait aucun doute que le typage d'OCaml a plus de rigueur académique. OCaml a les types algébriques et les types fonctions, la curryfication, l'application partielle et l'inférence de types.

Les deux langages empêchent le déréférencement de pointeurs nuls, un avantage énorme par rapport aux autres langages plus populaires. On peut dire que l'approche d'OCaml est supérieure, parce qu'il offre des meilleures abstractions comme le type `option`. Ce type indique explicitement que la valeur peut être vide, obligeant le programmeur à soutenir ce cas. La méthode d'Eiffel—obligeant l'initialisation de chaque référence—est plus rudimentaire.

Bien que ces fonctionnalités supplémentaires d'OCaml n'améliorent pas nécessairement sa sécurité, elles permettent l'écriture de code plus concis et plus réutilisable.

Cependant Eiffel offre néanmoins une proposition intéressante : des préconditions, des postconditions, des invariants. Ces fonctionnalités agissent un peu comme un système de types étendu, un peu comme des tests unitaires. L'inconvénient est que les erreurs sont trouvées à l'exécution, pas à la compilation.

Conclusion

Il y a donc un compromis : il faut choisir l'ensemble des caractéristiques qui conviennent au problème à résoudre. Dans l'ensemble, OCaml semble être un choix meilleur pour les projets académiques, mathématiques ou conceptuellement complexes. Eiffel pourrait être une meilleure solution pour des projets de grandes entreprises qui sont assez simple mais très grands. Dans le monde réel, aucun des deux n'est pas très populaire. Cependant il semble qu'OCaml et d'autres langages ML sont plus utilisés[5].

Il y a aussi d'autres langages qui essayent de résoudre les mêmes problèmes. Scala [5], inventé par Martin Odersky (professeur à l'EPFL), combine l'orientation objet (comme Eiffel) et la programmation fonctionnelle (comme OCaml). Une critique fréquente est que cette combinaison rend Scala trop complexe. L'Inria a plusieurs projets marquants à côté d'OCaml, y compris Pharo [6] (qui porte sur le débogage interactif) et Coq [7] (un assistant de preuve utilisant une théorie des types d'ordre supérieur). Sur un plan global Haskell [8] (profondément influencé par OCaml) gagne en popularité grâce à son système de types incroyablement puissant. Depuis la publication d'Eiffel, l'automatisation de test—qui est similaire aux postconditions et les invariants—est devenu universelle.

En conclusion, donc, on peut dire que le vrai pouvoir d'Eiffel et d'OCaml sont les idées qu'ils ont établi. Une nouvelle génération de langages—Haskell, Rust [9], Elm [10], F# [11], Scala—ont pris ces idées. Espérons qu'ensemble ils amélioreront le logiciel.

5. http://www.scala-lang.org/
6. http://pharo.org/
7. https://coq.inria.fr/
8. https://www.haskell.org/
9. https://www.rust-lang.org/
10. http://elm-lang.org/
11. http://fsharp.org/

Glossaire

algorithme une série d'actions pour accomplir une tâche. 3

bibliothèque une collection de routines qui peuvent être partagées entre les programmes. 5

bogue un défaut d'un logiciel. 2

déréférencement la conversion d'une référence à l'information stockée à cet endroit. 1, 4

encapsulation l'encapsulation est la protection des données d'un objet contre l'accès direct. Au lieu de cela, l'objet fournit des façons contrôlées pour l'accès.. 1, 11

EPFL l'École polytechnique fédérale de Lausanne. 13

EPFZ l'École polytechnique fédérale de Zurich (en allemand *Eidgenössische Technische Hochschule Zürich*). 2

fonction une fonction est un objet mathématique—une relation qui transforme les éléments d'un ensemble en un autre ensemble. Dans certains langages, une fonction peut avoir d'autres effets.. 8, 12

héritage multiple le mécanisme par lequel des classes héritent de fonctionnalités de multiples autres classes. 5

immuabilité un objet immuable ne peut pas être modifié. 1, 7

Inria l'Institut national de recherche en informatique et en automatique. 2, 13

Bibliographie

[1] Xavier Leroy OCaml.org. *Une histoire d'OCaml*. URL : https://ocaml.org/learn/history.fr.html (visité le 15/03/2016).

[2] Eiffel Software. *Eiffel Documentation*. Anglais. URL : http://www.eiffel.org/documentation (visité le 15/03/2016).

[3] Eiffel Software. *Frequently Asked Questions on the Eiffel Language*. Anglais. URL : https://www.eiffel.com/resources/faqs/eiffel-language/#classic-how (visité le 15/03/2016).

[4] Eiffel Software. *Invitation à Eiffel - 2 Principes de conception*. Trad. de l'anglais par Cédric Babault. 2006. URL : http://cedric.babault.free.fr/Eiffel/Introduction/invitation-03.html (visité le 15/03/2016).

[5] TIOBE. *TIOBE Index*. Anglais. Mar. 2016. URL : http://www.tiobe.com/tiobe_index (visité le 15/03/2016).