Ecrire des classes fiables
Assertions et débogage

11

Il existe deux grands types d'erreurs lorsqu'on écrit des classes: (i) les erreurs apparaissant lors de la compilation et (ii) les erreurs lors de l'exécution du programme.

1. Erreurs de compilation

L'erreur la plus courante et la plus simple à corriger est celle qui apparaît lors de la compilation. Le compilateur indique alors le type d'erreur et la ligne où celui-ci a trouvé un problème. Dans la majorité des cas, il s'agit

- ✔ d'erreurs typographiques.
- ✔ d'oublis de déclaration de variables.
- ✔ d'appels de méthodes non existantes.
- ✔ de non respect des types et/ou du nombre d'arguments lors de l'utilisation de méthodes.

Dans la majorité des cas, le compilateur est très dithyrambique dans ses commentaires et est d'une aide précieuse.

2. Erreurs lors de l'exécution du programme

2.1. Arrêt du programme à l'exécution

Cette fois, la compilation se déroule sans erreur mais à l'exécution, le programme "plante". L'erreur la plus courante est l'oubli de la création d'un objet qui se traduit par le message suivant:

```
*** Error at Run Time ***: Call with a Void target.
```

Il est assez facile de corriger le bogue en consultant l'état de l'objet au moment du "plantage".

```
2 frames in current stack.
=====  Bottom of run-time stack  =====
<system root>
Current = TST#0x8775038
        [ str = Void
        ]
line 7 column 2 file /home/jct/bioinfo/work/tst.e
=====================================
make TST
Current = TST#0x8775038
        [ str = Void
        ]
line 9 column 8 file /home/jct/bioinfo/work/tst.e
=====   Top of run-time stack    =====
Line : 9 column 4 in /home/jct/bioinfo/work/tst.e.
*** Error at Run Time ***: Call with a Void target.
```

Dans le message précédent, on peut lire qu'il y a eu une utilisation d'un objet Void dans la ligne 9. L'objet en question str est à l'état Void; donc il est l'auteur du "plantage".

```
Current = TST#0x8775038
        [ str = Void
        ]
```

3. Comportement instable du programme

Une fois passé ce stade, il reste les erreurs qui ne font pas "planter" le programme mais qui ne donne pas le résultat escompté. Il faut alors s'assurer que les objets ont un "comportement" normal correspondant aux attentes du créateur de classes.

Les erreurs peuvent survenir à deux niveaux:

- ✔ au niveau du programmeur « client » qui utilise incorrectement les classes.
- ✔ au niveau de l'utilisateur qui n'utilise pas le programme de façon correcte.

3.1. Assertions: Garde-fous pour le programmeur client

Une assertion est un test qui s'il est incorrect déclenche l'arrêt de l'exécution du programme. Il joue le rôle d'une sorte de filtre – placé dans une routine ou une classe – qui va bloquer et arrêter l'exécution du programme s'il y a eu une utilisation anormale.

ATTENTION!!! L'assertion ne sert que pendant la **phase de test** de classes ou du programme et sera "enlevée" une fois que le programme est en **phase de production**. Il est donc impératif que l'assertion n'ait aucun rôle dans le comportement d'une classe ou d'un programme.

La syntaxe est la suivante: Il est d'usage d'associer au test de l'assertion, un message qui sera affiché par le programme au moment de l'arrêt du programme. La syntaxe d'une assertion en Eiffel est la suivante:

my_tag_explaining_the_assertion : condition

L'étiquette *my_tag_explaining_the_assertion* est un seul mot (on utilise donc souvent le caractère souligné "_" pour faire une mini-phrase résumant la condition de l'assertion. L'assertion est équivalente au pseudo-code suivant:

```
if condition = False then
    io.put_string("my_tag_explaining_the_assertion%N")
    die_with(1)  -- End of program
end
```

Une assertion classique est de vérifier que l'argument de votre routine est non vide. Par exemple, pour une méthode foo(arg : MY_CLASS) l'assertion peut alors ressembler à:

```
    arg_not_void : arg /= Void
```

3.1.1. Assertions au niveau des routines

Les routines en Eiffel peuvent être réglementées par des pré- et des post-conditions qui sont des assertions placées respectivement dans les blocs **require** et **ensure**. Une routine classique peut donc être organisée de la manière suivante:

```
foo(arg1 : INTEGER; arg2 : MY_CLASS) is
    require
          -- Here are the pre-conditions (first assertion block)
    local
          -- Declaration of local variables
    do
          -- code
    ensure
          -- Here are the post-conditions (last assertion block)
    end
```

Le programmeur "client" doit utiliser la routine foo en respectant les clauses d'un « contrat » entre lui et le créateur de classes via la routine foo. Si les clauses du contrat sont respectées (les pre-conditions et les post-conditions sont correctes), alors la routine est exécutée. Dans le cas contraire, le contrat est rompu, l'exécution s'arrête. Ce schéma de fonctionnement est nommé par le fondateur du langage Eiffel Bertrand Meyer, le « design by contract ».

L'exemple typique est l'utilisation de la méthode sqrt qui calcule la racine carrée d'un nombre. La ligne suivante est incorrecte:

(-1).sqrt -- *Nombre négatif interdit*

```
class TST

creation
    make

feature
    make is
          do
                io.put_double( (-1).sqrt); io.put_new_line
          end
end -- end of class TST
```

A l'exécution du programme, le message suivant est affiché:

```
*** Error at Run Time ***: Require Assertion Violated.
3 frames in current stack.
=====  Bottom of run-time stack  =====
<system root>
Current = TST#0x9241038
line 7 column 2 file /home/jct/bioinfo/work/tst.e
=====================================
make TST
Current = TST#0x9241038
line 9 column 7 file /home/jct/bioinfo/work/tst.e
=====================================
sqrt INTEGER_8
Current = -1
Result = 0.000000
```

```
line 238 column 9 file
/usr/local/SmartEiffel/lib/numeric/integer_general.e
=====   Top of run-time stack    =====
*** Error at Run Time ***: Require Assertion Violated.
```

A la première ligne du message, le commentaire indique qu'une assertion de type **require** a été violée à la ligne 9 dans la méthode *make*. Les trois dernières lignes montrent que l'assertion provient de la classe INTEGER_GENERAL. Si on cherche dans EiffelDoc, la méthode *sqrt* de INTEGER_GENERAL, on trouve les lignes suivantes:

```
  sqrt: REAL
      -- Square root of Current.
      require
         Current >= 0
```

La clause require indique que le nombre réel en cours (Current) doit être positif ou nul.

Il existe aussi le mot-clé **check** pour insérer une assertion en milieu de routines.

```
add_chain(a_child : EBO_NODE) is
    local
          a_chain : EBO_CHAIN
    do
          a_chain ?= a_child
          check
                a_chain /= Void
          end
          ........................
    end
```

Dans ce cas, la tentative d'affectation *a_chain ?= a_child* peut poser problème si *a_child* n'est pas de type EBO_CHAIN. Le créateur de classes a prévu que dans cette routine, la tentative d'affectation doit toujours réussir. Pendant la phase de test, il ajoute une asssertion pour vérifier si ses prévisions sont bien respectées.

Remarque: Il existe d'autres types d'assertions, comme par exemple, au niveau des boucles (**from** ...**until** ...**loop**...) mais celles-ci ne seront pas détaillées car assez peu utilisées dans les projets libres Eiffel.

3.1.2. Assertions au niveau des classes

La clause **invariant** se situe à la fin d'une classe. A l'intérieur de cette clause, on trouve les propriétés de la classe qui restent vraies à tout moment de la vie de l'objet.

Par exemple, l'interface publique de STRING – obtenue avec **short** – est la suivante:

```
class interface STRING
   --
   -- Resizable character STRINGs indexed from 1 to count.
   --

.................................

invariant

    0 <= count;

    count <= capacity;

    capacity > 0 implies storage.is_not_null;
```

```
end of STRING
```

Dans une STRING, il existe trois propriétés invariantes dont la première par exemple, vérifie si le nombre de caractères d'une STRING est **toujours** supérieur ou égal à zéro.

3.2. Exceptions: Garde-fous pour l'utilisateur du programme

Il arrive des cas où le créateur de classes souhaite fiabiliser son programme face à l'utilisateur. Dans ce cas, il utilise les exceptions. Le fonctionnement est le suivant; au cas où une erreur fatale apparaît dans un programme ("l'exception"), le programme doit pouvoir intercepter ce problème pour permettre de continuer de façon transparente.

Remarque: Contrairement aux assertions, les exceptions restent dans le programme même en phase de production.

En Eiffel, les mécanismes d'exception fonctionnent au niveau de la routine et sont définies dans la clause **rescue**. L'interception de l'exception est réalisée dans la clause **rescue**. De plus, le mot clé **retry** permet de relancer l'exécution de la routine tant que le problème n'est pas résolu.

```
foo(arg1 : INTEGER; arg2 : MY_CLASS) is
    require
          -- Here are the pre-conditions (first assertion block)
    local
          -- Declaration of local variables
    do
          -- code
    ensure
          -- Here are the post-conditions (last assertion block)
    rescue
          -- Here are the exception handling
    end
```

L'exemple suivant présente une utilisation des exceptions. La routine *ask_filename* attend que l'utilisateur saisisse un nom de fichier. Si le nom est correct – c'est à dire qu'il existe réellement un fichier – la routine continue son exécution sinon le mot clé **retry** fait relancer l'exécution de *ask_filename*.

```
ask_filename is
    local
          name : STRING
    do
          io.read_string(name);
    rescue
          if not file.connect(name) then
                retry
          end
    end
```

4. Débogage, déverminage

Le débogage consiste à suivre pas à pas l'exécution du programme pour trouver et corriger le problème.

4.1. Simplissime

4.1.1. Débogage artisanal

La méthode la plus naturelle est d'afficher le contenu d'un objet, ou d'une variable à un instant donné au cours de l'exécution du programme. Les fichiers sources fleurissent ainsi de:

```
io.put_string(once "DEBUG " + my_object.out + "%N")
```

qui permettent dans la majorité des cas de corriger les bogues disséminés dans les classes. L'inconvénient est qu'il faut ensuite commenter ou détruire toutes les lignes comportant ces commandes d'affichage lorsque la phase de débogage est terminée.

4.1.2. Débogage artisanal et ... simple

Dans Eiffel, il existe une facilité pour inclure des lignes de commande qui ne seront exécutées que si les classes sont compilées en mode débogage. Il suffit pour cela d'encadrer le bloc de code utilisé au moment du débogage par **debug** et **end** à l'intérieur d'une méthode . Le bloc de code ainsi délimité ne sera exécuté que si l'option debug sera active dans le fichier ACE correspondant.

```
gl_begin(i : INTEGER) is
    do
          debug
                io.put_string(once "gl_begin(" + i.out + ")" + "%N")
          end
          do_the_job
    end
```

Il faut ensuite indiquer au compilateur de prendre en compte ces blocs de débogage en utilisant l'option debug_check.

```
compile my_class.e -o my_class -debug_check
```

Dans un fichier ACE, il faut ajouter dans la section **default**, l'option **debug(yes).**

Dans certains cas où vous voulez segmenter vos zones de débogage. Par exemple, vous souhaitez tester séparément le « moteur », de la partie graphique de votre programme et ne lancer que les blocs de débogage correspondant au graphique. Vous pouvez associer au mot **debug**, une clé sous forme d'un mot et indiquer au compilateur quels sont les blocs à prendre en compte. Par exemple, la méthode graphique précédente pourrait avoir un bloc GRAPHICS qui est définie de la manière suivante:

```
gl_begin(i : INTEGER) is
    do
          debug("GRAPHICS")
                io.put_string(once "gl_begin(" + i.out + ")" + "%N")
          end
          do_the_job
    end
```

Dans le fichier ACE, il faut ajouter dans la section **default**, l'option **debug("GRAPHICS").**

4.2. Sophistiqué

Sous SmartEiffel, vous pouvez utiliser un vrai débogueur qui malheureusement fonctionne uniquement en ligne de commande et ne dispose d'aucune interface graphique. A mon humble avis, il s'agit ici du dernier recours car il est d'un maniement compliqué et est réservé uniquement aux experts en informatique.

La première étape consiste à compiler le programme en mode "SmartEiffel debugger" par la commande en mode ligne:

```
-sedb                Enable sedb, the SmartEiffel debugger
```

Par exemple,

```
compile tst.e -o tst -sedb
```

A l'exécution du programme:

```
./tst
```

Le message de bienvenue du "SmartEiffel debugger" s'affiche:

```
```