

PROGRAMMATION OBJET : CONCEPTS AVANCÉS

Cours 8 : Contrat, Test et Preuve

Yann Régis-Gianas
yrg@pps.jussieu.fr

PPS - Université Denis Diderot – Paris 7

12 décembre 2008

MCours.com

Programmation par « contrat »



Le mal de l'ingénierie du logiciel

*Deux « hypocrisies » de l'industrie du logiciel
(E.W. Dijkstra)*

- ▶ La **maintenance** informatique : un logiciel peut-il s'user ?
- ▶ Les fameux **bugs** : pourquoi ne pas les appeler « erreurs » ?

La part perdue du développement logiciel

- ▶ Un programme informatique est la réalisation exécutable d'une **spécification**.
 - ▶ On s'assure qu'un programme n'a pas d'erreur en vérifiant qu'il est **conforme** à sa spécification.
 - ▶ Le **développement de la spécification** devrait avoir autant d'importance que le développement du programme.
- ⇒ L'absence de spécification précise donne l'illusion qu'un dysfonctionnement du programme n'est pas une erreur de la part du programmeur mais, peut-être, un « léger défaut de conception » . . .

La programmation par « contrat »

- ▶ Expliciter la spécification d'un programme permet d'en améliorer la qualité et d'atteindre *in fine* l'objectif d'un logiciel **correct par construction**.
- ▶ Si on voit un programme comme une fonction mathématique f définie sur un ensemble d'entrées I et produisant des résultats dans un ensemble de sorties S , alors une spécification est un **couple de prédicats** (P, Q) tels que :

$$\forall i \in I, P(i) \Rightarrow Q(i, f(i))$$

- ▶ P est la **précondition**, spécifiant le domaine correct d'utilisation.
- ▶ Q est la **postcondition**, spécifiant la relation logique liant l'entrée et la sortie.

Le contrat du pauvre en C++

```
#include <cassert>
#define check_precondition (P) assert (P)
#define check_postcondition (Q) assert (Q)

// input : t
// precondition : t.size () > 0
// output : x
// postcondition :  $x \in t \wedge \forall a, a \in t \Rightarrow x \leq a$ 
int min (std::vector<int> t) {
    check_precondition (t.size () > 0);
    int x = t[0];
    for (int i = 1; i < t.size (); ++i)
        x = t[i] < x ? t[i] : x;
    check_postcondition (
        exists (t.begin (), t.end (), bind2nd (equal<int>(), x))
        && forall (t.begin (), t.end (), bind2nd (greater_equal<int>(), x)));
}
```

Explication

- ▶ La fonction `assert` permet de tester qu'une condition est vérifiée en un point donné du programme.
- ▶ Si elle est invalide, le programme est stoppé en précisant le numéro de ligne de la commande de vérification pour aider au diagnostic.
- ▶ On peut désactiver la vérification en passant l'option `-DNDEBUG` au compilateur.

Déterminer le composant responsable



Responsibility

Première critique :

- ▶ Si une erreur de violation de la postcondition est signalée à l'endroit pertinent, ce n'est pas le cas pour une erreur sur la précondition.
 - ▶ On aimerait qu'une erreur d'utilisation de la fonction soit signalée **au niveau de l'appel** de cette fonction.
- ⇒ Une intégration de ces notions dans le langage permettrait cela.

Une boîte noire, c'est utile . . . mais l'avion a déjà explosé



Seconde critique :

- ▶ Vérifier le respect des préconditions et des postconditions durant l'exécution est tardif et **ne prouve rien** sur les exécutions futures du programme !
 - ▶ De plus, il n'est pas certain que toute spécification soit exécutable !
 - ▶ Enfin, un **argument logique ou mathématique** est parfois nécessaire pour montrer qu'un programme vérifie sa spécification.
- ⇒ L'utilisation d'un **analyseur statique** (similaire à un vérificateur de type) serait plus efficace !

MCours.com

Le cours d'aujourd'hui

1. Le langage `EIFFEL` qui est centré sur la programmation par contrat.
2. Les outils de vérification statique pour les langages à objet.

EIFFEL

- ▶ EIFFEL est un langage de programmation objet basé sur des classes.
- ▶ Il est développé par Bertrand Meyer depuis 1986.
- ▶ Le compilateur SMART EIFFEL est développé par l'INRIA.
- ▶ EIFFEL a les propriétés suivantes :
 - ▶ Il est typé statiquement.
 - ▶ Son système de type traite les polymorphismes d'inclusion et paramétrique borné. Il a aussi quelques traits « exotiques » (covariances des arguments des méthodes et des paramètres de type ...).
 - ▶ Il intègre de l'héritage multiple, fournit des primitives de renommage.
 - ▶ Il fournit un équivalent des fonctions de première classe, appelé des agents.
 - ▶ Son environnement d'exécution inclut un ramasse-miette et est concurrent.
 - ▶ Les méthodes peuvent être des requêtes (qui ne modifient pas l'objet) ou bien des commandes (qui le modifient).
 - ▶ L'accès à un attribut d'une instance suit la même syntaxe que l'appel d'une fonction sans argument (similaires aux propriétés de C#).

⇒ Aujourd'hui : la programmation par contrat en EIFFEL.

Une classe, une méthode et son contrat en EIFFEL

```
class MIN
creation {ANY}
  default_create
feature {ANY}
  is_min (t : ARRAY[INTEGER]; m : INTEGER) : BOOLEAN is
    - ... Définition effacée ...

  min (t : ARRAY[INTEGER]) : INTEGER is
    require - Précondition
      t.count > 0
    local
      x : INTEGER
      i : INTEGER
    do
      x := t.item(0)
      from
        i := t.lower
      until
        i > t.upper
      loop
        if x > t.item(i) then
          x := t.item(i)
        end
        i := i + 1
      end
      Result := x
    ensure - Postcondition
      is_min (t, Result)
    end
end
```

Contrat pour des objets



- ▶ L'environnement d'exécution de `EIFFEL` est équipé pour fournir des rapports d'erreur permettant de déterminer si une erreur provient de la façon dont on appelle une fonction ou bien de sa définition.
- ▶ `EIFFEL` étend ce mécanisme au cas des objets.
- ▶ Plusieurs difficultés apparaissent :
 - ▶ Comment maintenir la cohérence globale d'un objet à travers les appels de différentes méthodes ?
 - ▶ Comment sont véhiculées les spécifications à travers l'héritage ?

Une numérotation configurable de chaîne de caractères

- ▶ On se propose de définir un objet fournissant les services suivants :
 - ▶ `add (s)` : associe un entier k à la chaîne s si ce n'est pas déjà fait. Dans le cas contraire, renvoie l'entier déjà associé à s .
 - ▶ `get (x)` : renvoie la chaîne associée à l'entier x .
 - ▶ `upd (s, i)` : associe l'entier i à la chaîne s si i n'est pas déjà utilisé.
- ▶ On suppose pour le moment que les chaînes de caractères sont initialisées.

Implantation en EIFFEL

```
class NUM
creation {ANY}
  make

feature {}
  s_to_i : HASHED_DICTIONARY[INTEGER, STRING]
  i_to_s : HASHED_DICTIONARY[STRING, INTEGER]
  last_i : INTEGER
  – Les propriétés à vérifier sur ces attributs :
  valid_s_to_i_assoc (v : INTEGER; k : STRING) : BOOLEAN is
    do
      Result := i_to_s.has (v) and then i_to_s.at (v) = k
    end
  valid_i_to_s_assoc (v : STRING; k : INTEGER) : BOOLEAN is
    do
      Result := s_to_i.has (v) and then s_to_i.at (v) = k
    end
  dictionary_are_synchronized : BOOLEAN is
    do
      Result :=
        s_to_i.for_all (agent valid_s_to_i_assoc)
        and i_to_s.for_all (agent valid_i_to_s_assoc)
    end
  last_i_is_sup : BOOLEAN is
    do
      Result := s_to_i.for_all (agent last_i ≥ ?)
    end
end
```

Implantation en EIFFEL (suite)

```
feature {ANY}
  make is
  do
    create s_to_i.make
    create i_to_s.make
    last_i := 0
  ensure
    dictionary_are_synchronized
    last_i_is_sup
  end

  unused (k : INTEGER) : BOOLEAN is
  do
    Result := not (i_to_s.has (k))
  end
```

Implantation en Eiffel (suite)

```
add (s : STRING) : INTEGER is
  require
    s /= Void
    dictionary_are_synchronized
    last_i_is_sup
  do
    if s_to_i.has (s) then
      Result := s_to_i.at (s)
    else
      last_i := last_i + 1
      s_to_i.add (last_i, s)
      i_to_s.add (s, last_i)
      Result := last_i
    end
  ensure
    dictionary_are_synchronized
    last_i_is_sup
    old Current.unused (Result)
    get (x) = s
  end
```

Implantation en Eiffel (suite)

```
get (x : INTEGER) : STRING is
  require
    i_to_s.has (x)
    dictionary_are_synchronized
    last_i_is_sup
  do
    Result := i_to_s.at (x)
  ensure
    dictionary_are_synchronized
    last_i_is_sup
    add (s) = x
  end

upd (x : INTEGER; s : STRING) is
  require
    s /= Void
    dictionary_are_synchronized
    last_i_is_sup
    unused (x)
  do
    s_to_i.put (x, s)
    i_to_s.put (s, x)
  ensure
    dictionary_are_synchronized
    last_i_is_sup
    get (x) = s
    add (s) = x
  end
```

Invariants des objets

- ▶ On remarque que les propriétés `dictionary_are_synchronized` et `last_i_is_sup` doivent être valides **avant** et **après** chaque appel de méthode.
- ▶ Cette propriété est un **invariant** de notre objet.
- ▶ Elle peut être **invalidée au cours de l'exécution** d'une méthode.
- ▶ **EIFFEL** fournit un mécanisme pour déclarer les invariants d'un objet.
- ▶ Dans notre exemple, on rajoute :

invariant

```
dictionary_are_synchronized  
last_i_is_sup
```

à la fin de la définition de la classe.

Un bug, pardon, « une erreur » dans notre exemple !



Une erreur s'est glissée dans notre exemple,
saurez-vous la trouver ?

(en fait, il y en a même deux)

MCours.com

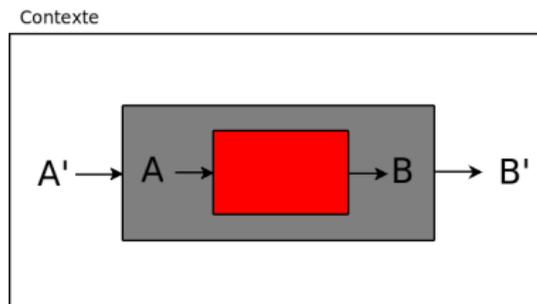
Héritage des spécifications

- ▶ On déclare une classe fille qui redéfinit la méthode `add`. La nouvelle version de cette méthode accepte les chaînes non initialisées et leur associe systématiquement l'entier 0.



Quelle doit être la précondition de cette méthode ?

Analogie entre typage et spécification



- ▶ Pour les mêmes raisons qui imposent une **contra-variance** des arguments des méthodes dans les sous-classes, la précondition de redéfinition d'un méthode doit être plus faible que la précondition initiale.
- ▶ En d'autres termes, les préconditions des méthodes de la classe fille **doivent impliquer logiquement** les préconditions des méthodes de la classe mère.

Retour sur notre exemple

```
class NUMBIS
  inherit
    NUM redefine make, add, upd, get end
```

```
creation
  make
```

```
feature {ANY}
```

```
  make is
    do
      Precursor
      – 0 is reserved for Void
      last_i := 1
    end
```

Retour sur notre exemple (suite)

```
add (s : STRING) : INTEGER is
  require else
    True
  do
    if s = Void then
      Result := 0
    else
      Result := Precursor (s)
    end
  ensure then
    s = Void implies Result = 0
  end
```

Retour sur notre exemple (suite)

```
get (x : INTEGER) : STRING is
  require else
    x = 0
  do
    if x = 0 then
      Result := Void
    else
      Result := Precursor (x)
    end
  ensure then
    x = 0 implies Result = Void
  end
```

Retour sur notre exemple (suite)

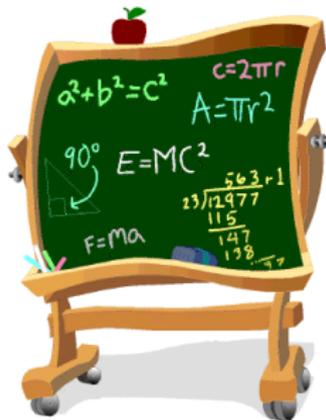
```
upd (x : INTEGER; s : STRING) is
  require else
    unused (x)
    s = Void implies x = 0
  do
    if s /= Void then
      Precursor (x, s)
    end
  end
end
```

Vérification partielle des contrats

- ▶ Une fois le développement du programme et de sa spécification achevé, on peut **tester** la conformité du programme à sa spécification à l'aide :
 - ▶ de **tests unitaires** : on isole un objet et on vérifie que les invariants et les postconditions sont valides pour des entrées conformes aux préconditions.
 - ▶ de **tests d'intégration** : on isole des groupes d'objets et on vérifie que les préconditions des méthodes sont validées.
- ▶ Il existe des outils pour gérer et automatiser les jeux de tests (JUnit, Quickcheck, ...)
- ▶ mais ...

Un test n'est pas une preuve !

Vérification totale des contrats



- ▶ Des outils modernes (majoritairement issus de la recherche) s'attaquent à la **preuve formelle** de la correction des programmes vis-à-vis de leur spécification.
- ▶ Tous s'appuient sur une formalisation mathématique de la **sémantique** des langages de programmation et sur une **mécanisation du raisonnement logique**.

Preuve formelle sur machine

Goal reduction phase

$$\frac{}{\Gamma; \Delta \vdash \top} \top R \quad \frac{\Gamma; \Delta \vdash B \quad \Gamma; \Delta \vdash C}{\Gamma; \Delta \vdash B \& C} \&R$$
$$\frac{\Gamma; \Delta, B \vdash C}{\Gamma; \Delta \vdash B \multimap C} \multimap R \quad \frac{\Gamma, B; \Delta \vdash C}{\Gamma; \Delta \vdash B \Rightarrow C} \Rightarrow R \quad \frac{\Gamma; \Delta \vdash B[y/x]}{\Gamma; \Delta \vdash \forall x. B} \forall R$$

Backchaining phase

$$\frac{\Gamma; \Delta \vdash^{D_1} A}{\Gamma; \Delta \vdash^{D_1 \& D_2} A} \&L \quad \frac{\Gamma; \Delta \vdash^{D_2} A}{\Gamma; \Delta \vdash^{D_1 \& D_2} A} \&L \quad \frac{\Gamma; \Delta \vdash^{D[y/x]} A}{\Gamma; \Delta \vdash^{\forall x. D} A} \forall L$$
$$\frac{\Gamma; \Delta_1 \vdash G \quad \Gamma; \Delta_2 \vdash^D A}{\Gamma; \Delta_1, \Delta_2 \vdash^{G \multimap D} A} \multimap L \quad \frac{\Gamma; \cdot \vdash G \quad \Gamma; \Delta \vdash^D A}{\Gamma; \Delta \vdash^{G \Rightarrow D} A} \Rightarrow L$$

Identity and Decide rules

$$\frac{\Gamma, D; \Delta \vdash^D A}{\Gamma, D; \Delta \vdash A} \text{decide!} \quad \frac{\Gamma; \Delta \vdash^D A}{\Gamma; \Delta, D \vdash A} \text{decide} \quad \frac{}{\Gamma; \cdot \vdash A} \text{init}$$

Un ordinateur peut manipuler des preuves.

- ▶ Certains formalismes permettent d'exprimer les propriétés des programmes.
- ▶ Des outils les implémentent et permettent de vérifier des preuves sur ordinateur (COQ, PVS, ISABELLE/HOL, ...).
- ▶ D'autres outils permettent même d'effectuer des **recherches automatiques de preuve** (en général dans des logiques moins riches, comme la logique du premier ordre).

Preuve formelle de programmes sur machine

1. Écrire un programme annoté par sa spécification formelle.
2. Produire des obligations de preuve.
3. Prouver les obligations.

La méthode de Hoare



- ▶ La phase 1 est du ressort du programmeur.
- ▶ La phase 2 est un algorithme automatique.
- ▶ La phase 3 peut être traitée par :
 - ▶ Des prouveurs automatiques pour les preuves « faciles » ;
 - ▶ Le programmeur dans un assistant de preuve pour les démonstrations « subtiles ».

Vérification statique d'une classe JAVA

- ▶ L'outil `KRAKATOA`, intégré à la plateforme `WHY`, est développé au LRI de l'Université Paris 11.
- ▶ Il permet de prouver de nombreuses propriétés des programmes `JAVA` en utilisant la méthode de Hoare.
- ▶ D'autres outils sont actuellement développés un peu partout dans le monde : `ESC/JAVA`, `SPEC#`, `FOCAL`, ...
- ▶ [Démonstration]

Des problèmes encore ouverts

- ▶ Ces systèmes ont des difficultés à traiter le **partage de références entre objets**.
- ▶ Il est aussi très difficile de savoir comment **réutiliser les preuves** à travers l'héritage.
- ▶ La notion d'invariant de classe semble aujourd'hui trop rigide pour spécifier des programmes réalistes : il arrive qu'un objet brise l'invariant d'autres objets temporairement, sans mettre en cause la correction d'un programme.

MCours.com