TRAVAUX PRATIQUES: série n°2

Implémentation du type abstrait des comptes bancaires

JANKOWIAK-Matthias / CHRISTOPH-Samuel

I-Problématique:

L'objectif du TP est L'implémentation du type abstrait Compte, La spécification nous est ici déjà donné.L'objectif est aussi de comprendre Cette même spécification, en décryptant le code CASL puis par la suite en distinguant la signature et la sémantique.

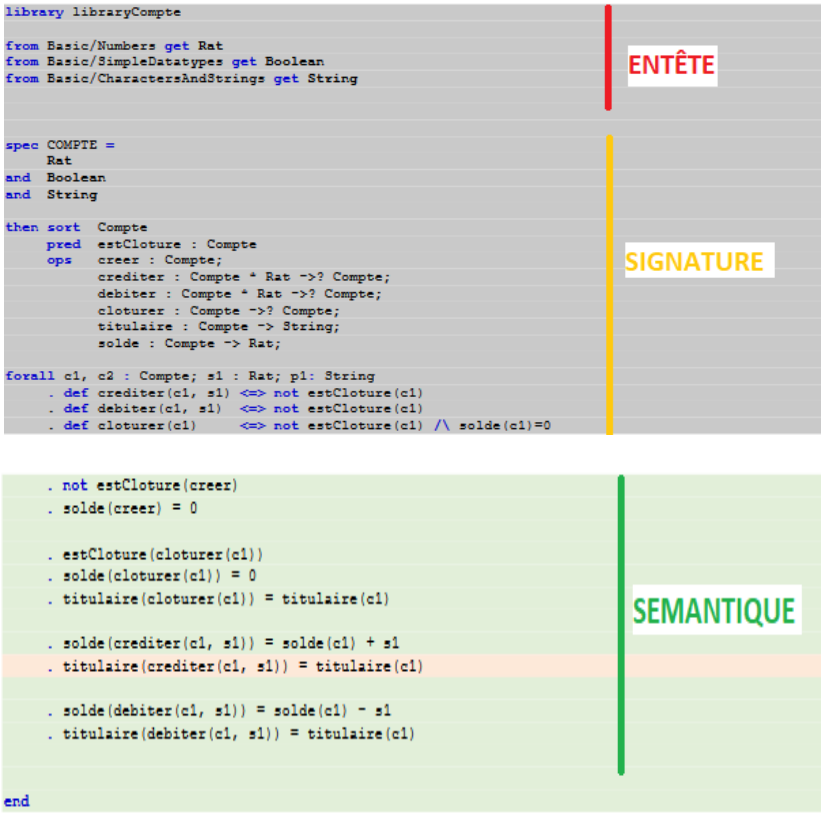
Grace a cela, nous pouvons mieux connaître la spécification ainsi que la sémantique et la signature. Ceci est essentiel dans le développement de programmes d'application.

II-Réalisation :

Étape 1: Spécification du type abstrait

La signature est la partie visible ou interface d'une spécification et la sémantique est un ensemble de propriétés. La sémantique d'une spécification consiste à énoncer les propriétés des opérations du type abstrait.

Spécification en CASL :



Dans une spécification, une opération peut avoir deux statuts possibles:

- les constructeurs
- les accesseurs

Une opération est un constructeur si elle retourne un résultat, dont le type est défini par la spécification en cours. Ici les constructeurs sont : **créer(compte), créditer(compte,s), débiter(compte,s)** et **cloturer(compte).**

Un accesseur du type est une opération qui permet d'observer l'état d'un objet du type et sans y apporter la moindre modification. Mais cette opération est un accesseur si et seulement si elle a au moins un argument de type défini, elle rend un résultat de type importée. Ici les accesseurs sont : **titulaire(compte), cloture(compte), solde(compte).**

On peut voir ci-dessus, dans la spécification, les mots clés du langage CASL, en bleu : pour mieux comprendre nous allons les nommer ainsi que dire ce qu'ils représentent :

«library» → librairie/bibliothèque,
«from» → (module)depuis,
«get» → prend le type (booléen, rationnel...),
«spec» → spécification du type abstrait,
«and» → et (signifiant les types abstrait à nouveau utilisé),

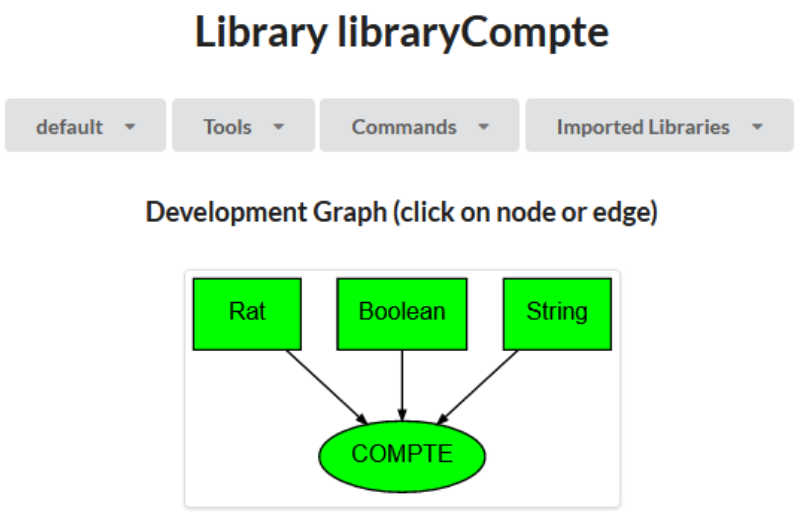
«then sort» → type abstrait,
«pred» → prédicat,
«ops» → opération à qui on donne un nom (signature),
«forall» → pour tout (défini les axiomes),
«def» → définition du domaine,
«<=>» → si et seulement si,
«not» → inverse(non),
«/\» → et,
«end» → fin.

Pour dérouler le cycle de développement du type abstrait, il faut d'abord étudier le cahier des charges fourni par le responsable du secteur gestion clients d'une banque afin de respecter sa demande. Nous devons donc respecter les fonctionnalités suivantes :

- création d'un compte bancaire avec un solde nul et non clôturé,
- débiter un compte / créditer un compte,
- clôturer un compte si son solde est nul,
- connaître le nom de son titulaire, (un titulaire ne peut pas posséder plusieurs comptes),
- chercher à savoir son solde,
- *tester si un compte est clôturé,*

Étape 2: Validation de la spécification du type abstrait

Cette étape permet de prouver que la spécification satisfait la consistante (pas d'axiomes exprimant des propriétés contradictoires) et la complétude (suffisamment d'axiomes pour savoir si une certaine propriété est vraie ou fausse). Dans notre cas, la spécification est validée.



Résultat à partir de l'analyseur HETS:

L'un des outils les plus puissants permettant de valider une spécification est hets (Heterogeneous ToolSet). Il est développé par CoFi (Universität Magdeburg) Il est possible d'appeler l'analyseur HETS en ligne à partir de DOLiator (comme ci-dessus).

Étape 3: Implémentation de la spécification

Pour cela on réalise un fichier.h et un fichier.c : *Le fichier.h est un fichier d'interface qui est consultable par le futur utilisateur du type. Ce fichier est ici appelé libraryCompte.h* .

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#define FAUX 0
#define VRAI 1

typedef int boolean;

/* Proposer un type concret pour implémenter le type abstrait des polynomes */
typedef struct un_compte{
  char* titulaireDuCompte;
  float SoldeDuCompte;
  boolean IndicateurCloture;
}compte;

/* Définition du type des comptes : un type pointeur vers un objet de type compte */
typedef struct un_compte * COMPTE ;
```

```c
/* Créer un compte avec un solde nul et non clôturé */
COMPTE creer();

/* Crédite un compte */
COMPTE crediter(COMPTE c, float s);

/* Débite un compte */
COMPTE debiter(COMPTE c, float s);

/* Clôture un compte si son solde est nul */
COMPTE cloturer(COMPTE c);

/* Permet de savoir le nom du titulaire du compte */
char* titulaire(COMPTE c);

/* Teste si un compte est clôturé */
boolean cloture(COMPTE c);

/* Permet de savoir le solde d'un compte */
float solde(COMPTE c);
```

Voici notre fichier.h : libraryCompte.h

Le fichier.c lui est un fichier d'implémentation, non consultable pour implémenter les corps de toutes les opérations déclarées dans libraryCompte.h, ce *fichier est ici appelé libraryCompte.c .*

```c
#include <stdio.h>
#include <stdlib.h>
#include "libraryCompte.h"

COMPTE creer (){
 COMPTE c ;
 c = malloc(sizeof(struct un_compte));
 if(c == NULL){
   fprintf(stderr,"Allocation impossible \n");
   exit(EXIT_FAILURE);
 }

 else{
   scanf( "Entrer le nom du titulaire %s",c->titulaireDuCompte) ;
   c -> SoldeDuCompte = 0.;
   c -> IndicateurCloture = FAUX;
 }
 return c;
 }

```

```
21  COMPTE crediter (COMPTE c, float s){
22    if(!cloture(c)){
23      COMPTE c1;
24      c1 = creer();
25      c1 -> SoldeDuCompte = c-> SoldeDuCompte + s;
26      return c1;
27    }
28    else{
29      exit(EXIT_FAILURE);
30    }
31  }
32
33  COMPTE debiter (COMPTE c, float s){
34    if(!cloture(c)){
35      COMPTE c1;
36      c1 = creer();
37      c1 -> SoldeDuCompte = c->SoldeDuCompte - s;
38      return c1;
39    }
40    else{
41      exit(EXIT_FAILURE);
42    }
43  }
44
```

```
45  COMPTE cloturer (COMPTE c){
46    if(!cloture(c) && solde(c) == 0){
47      COMPTE c1;
48      c1 = creer();
49      c1 -> IndicateurCloture = VRAI;
50      return c1;
51    }
52    else{
53      exit(EXIT_FAILURE);
54    }
55  }
56
57  char* titulaire (COMPTE c){
58    return c -> titulaireDuCompte;
59  }
60
61  boolean cloture (COMPTE c){
62    return c -> IndicateurCloture;
63  }
64
65  float solde (COMPTE c){
66    return c -> SoldeDuCompte;
67  }
```

Voici notre fichier.c : libraryCompte.c

Étape 4: Vérification de l'implémentation

Afin de vérifier si l'implémentation est correcte vis à vis de sa spécification, nous allons créer un fichier PreuveLibraryCompte.c (processus de la vérification formelle) incluant libraryCompte.c (permettant d'afficher les erreurs demandées en cas d'échec de compilation). Le principe de mise en œuvre consiste à vérifier que les constructeurs du type construisent correctement les objets du type.Cela signifie que ces constructeurs doivent satisfaire tous les axiomes énoncés en spécification.

La fonction main() va permettre de vérifier que:

- chacun des constructeurs implémentés
- *satisfait tous les axiomes des accesseurs*

```c
#include <stdio.h>
#include <stdlib.h>
#include "libraryCompte.c"

int main(int argc, char*argv[]){
  COMPTE c,c1;
  int success;
  float s;

  /* Allocation mémoire et vérification */
  c = malloc(sizeof(struct un_compte));
  c1 = malloc(sizeof(struct un_compte));
  if(c == NULL){
    fprintf(stderr,"Allocation impossible \n");
    exit(EXIT_FAILURE);
  }

  /* Vérifier l'implémentation du constructeur creer() */
  c = creer();
  /* Initialiser l'indice success */
  success = 0;
  /* Vérification avec l'accesseur estCloture */
  /* Vérifier la propriété : estCloture(c)= FALSE */
  if (cloture(c)) success=success+1;
  /* Vérification avec l'accesseur solde */
  /* Vérifier la propriété : solde(c) = 0 */
  if (solde(c) != 0)   success=success+1;
```

```c

  /* Bilan de la vérification */
  if (success != 0){
      printf ("\n Implémentation incorrecte de cloturer(solde)");
      printf("Interruption de la vérification: revoir l'implémentation du type abstrait \n");
      exit(EXIT_FAILURE);
  };
  printf("test" );

  /* Réinitialiser la variable solde */
  success=0 ;
  /* Vérification avec l'accesseur titulaire */
  if (titulaire(c)!=titulaire(c1)) success=success+1;

  /* Bilan de la vérification */
  if (success != 0){
        printf ("\n Implémentation incorrecte de cloturer(titulaire)");
        printf("Interruption de la vérification: revoir l'implémentation du type abstrait \n");
        exit(EXIT_FAILURE);
  };
  printf("L'implementation du type abstrait est vérifiée\n");
  printf("Fin normale de la vérification de l'implémentation du type abstrait\n");
  return EXIT_SUCCESS;
}

```

Voici deux extraits (début et fin) de notre fichier PreuveLibraryCompte.c

III- Bilan/Conclusion:

Sur le plan théorique, réaliser ce TP nous a permis de comprendre le cycle de développement afin d'implémenter un type abstrait en lien avec notre problématique. En effet, l'utilisateur implémente des types abstraits afin de vérifier un compte.

La définition du type abstrait est ici avantageuse par rapport à la définition d'un type concret car elle est indépendante du langage de codage utilisé. Donc tout changement de langage ne remet pas cause cette définition. Il en résulte une grande stabilité des logiciels développés.