Un aperçu des méthodes formelles

Guy Tremblay
Dépt. d'informatique, UQAM

INF3140 Modélisation et spécifications formelle de logiciels
UQAM
Automne 2009

Plan de la présentation :

- Qu'est-ce qu'une spécification?
- Qu'est-ce qu'une méthode formelle?
- Qu'est-ce qu'un langage formel de spécification?
- À quelles étapes les méthodes formelles peuvent-elles être utilisées?
- Pourquoi y-a-t-il plusieurs notations et méthodes?
- Quels sont les principaux bénéfices des méthodes formelles?
- Sur quels aspects le cours INF3140 portera-t-il?

1 Prélude : Analyse, spécification et conception

Plusieurs niveaux de description d'un système :

(a) Les besoins et exigences des usagers ;
(b) L'ensemble (l'espace) des solutions possibles ;
(c) Une solution particulière ;
(d) La structure interne de la solution choisie ;
(e) La description des divers modules (interfaces) ;
(f) Les algorithmes utilisés par chaque module ;
(g) Le code final.

Analyse et spécification

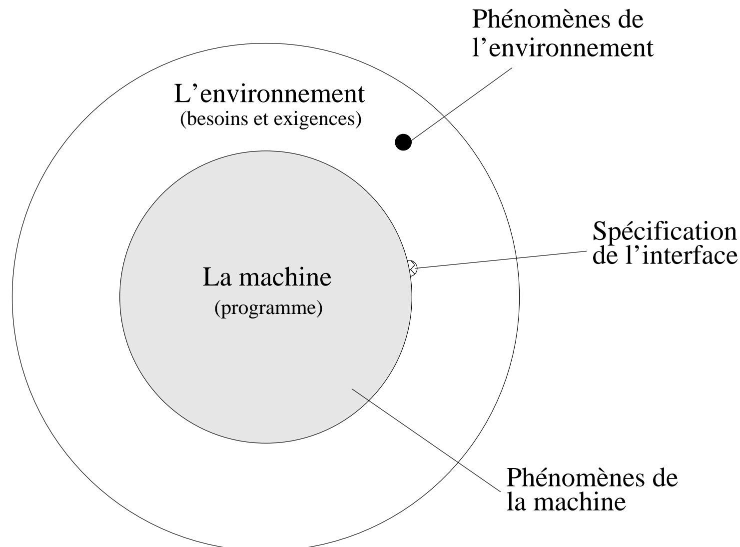
L'étape d'**analyse et spécification** porte sur :

1. Analyse du problème = Analyser et comprendre le problème à résoudre, les besoins et exigences des usagers, les contraintes à respecter = (a)+(b)
   (a) Besoins et exigences
   (b) Espace des solutions possibles

2. Spécification du produit = Décrire le logiciel qui va satisfaire aux besoins et exigences et qui va respecter les contraintes = (c)
   (c) Description d'une solution spécifique

Exigences et besoins vs. spécifications (selon M. Jackson) :

- Exigences et besoins = phénomènes de l'environnement
- Programmes = phénomènes internes à la machine
- Spécifications = phénomènes à l'interface environnement/machine



Conception

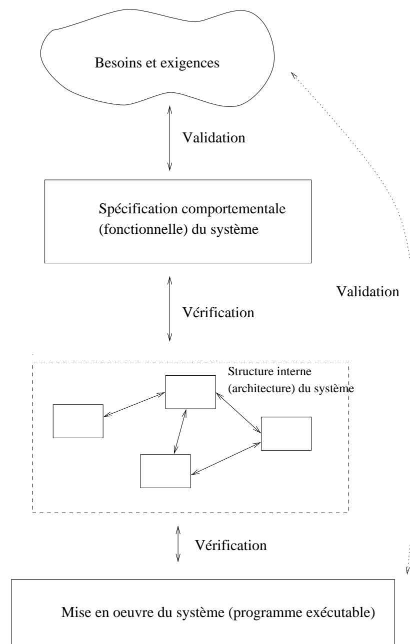
- Conception architecturale :
  - Quels sont les principaux modules?
  - Comment sont-ils organisés et structurés?
  - Quelles sont les interfaces entre les modules?
- Conception détaillée (procédurale) :
  - Que fait précisément chacun des modules?

Analyse et spécification vs. conception

Donc, analyse et spécification vs. conception =>

- Analyse et spécification : Concepts et interfaces pour utiliser le système (point de vue de l'*usager*) ;
- Conception : Concepts et interfaces pour construire le système (point de vue du *constructeur*).

Pourquoi une spécification est-elle importante?



Spécification = *Contrat à satisfaire*

- Validation : *Are we building the right product?* = Construit-on le bon produit?
- Vérification : *Are we building the product right?* = Construit-on le produit correctement?

2 Qu'est-ce qu'une "méthode formelle"?

> *"[Les méthodes formelles sont des] techniques basées sur les mathématiques pour décrire des propriétés [de] systèmes [informatiques]. [Elles] fournissent un cadre [pour] spécifier, développer, et vérifier des systèmes d'une façon systématique, plutôt que d'une façon* ad hoc.*"* [Wing90]

Éléments clés d'une méthode formelle :

- Langage *formel* pour l'écriture de spécifications
- Règles pour évaluer la validité/qualité des spécifications
- Stratégies et règles pour raffiner (mettre en oeuvre) les spécifications et vérifier ces raffinements

Fondation sur laquelle *tout* repose = <u>spécification formelle</u>

3 Qu'est-ce qu'un langage *formel* de *spécification?*

Langage avec syntaxe et sémantique bien définies :

- Syntaxe = EBNF ; diagrammes syntaxiques ; etc.
- Sémantique = algèbres ; automates et systèmes de transitions ; fonctions, relations et prédicats ; etc.

Doit permettre de décrire le comportement d'un composant logiciel

- en décrivant ses propriétés importantes
- de façon abstraite, sans détails inutiles
- sans dire comment il est réalisé (non-algorithmique)

Un langage de programmation *n'est pas* un langage de spécification :

- Trop algorithmique (opérationnelle, i.e., *procédurale*)
- Pas assez abstrait (tableaux, pointeurs, etc.)

La langue naturelle est trop floue, trop ambigue, trop imprécise

- Que signifie "Le voltage V doit toujours être compris entre 100 et 120 volts?

$$100.0 \leq V \leq 120.0$$

$$100.0 < V < 120.0$$

Les notions mathématiques pour les spécifications sont relativement simples :

- Logique propositionnelle :

```
p implies q  =  (not p) or q
not (p and q) =  (not p) or (not q)
```

- Ensembles = collections *non-ordonnées* d'éléments

```
e1: Set(Integer)
e1 = Set{20, 30}
e1->including(20) = e1
e1->union(e1)->size() = 2
e1->union( Set{30, 31, 32} ) = Set{20, 30, 31, 32}
```

- Logique des prédicats :

```
Set{0,2,4,6,8}->exists( i | 2*i = i )
not Set{0,2,4,6,8}->exists( i | i <> 0 and 2*i = i )
Set{0,2,4,6,8}->forAll( i | i < i+1 )
Set{0,2,4,6,8}->forAll( i, j | i < j implies i+1 < j+1 )
```

- Séquences = suites ordonnées d'éléments :

```
s1: Sequence(Integer)
s1 = Sequence{20, 30}
s1->size() = 2
s1->union(s1)->size() = 4
s1->prepend(10) = Sequence{10, 20, 30}
s1->append( Sequence{30, 31, 32} ) = Sequence{20, 30, 30, 31, 32}
```

Partie vraiment difficile des spécifications

$\neq$ mathématiques (ensembles, fonctions)
$=$ modélisation (comprendre le problème et modéliser une solution)

4 À quelles étapes les spécifications et les méthodes formelles peuvent-elles être utilisées?

- Analyse et spécification :
  - Description des concepts clés du problème (domaine d'application)
  - Spécification de la solution (comportement du système)
- Conception architecturale et détaillée :
  - Spécification de l'interface des composants
    - Avec des contrats (pré/post-conditions)
    - Avec des tests
- Construction de logiciels :
  - Vérification dynamique d'assertions
  - Tests unitaires

5 Pourquoi y-a-t-il plusieurs notations et méthodes différentes?

Il existe plusieurs styles différents de spécification :

- Modélisation abstraite
- Approche algébrique (types abstraits)
- Algèbre de processus
- Logique temporelle
- ...

Situation semblable à celle des langages de programmation :

- Domaines d'applications différents
- Styles (paradigmes) d'utilisation différents
- Pouvoirs expressifs différents

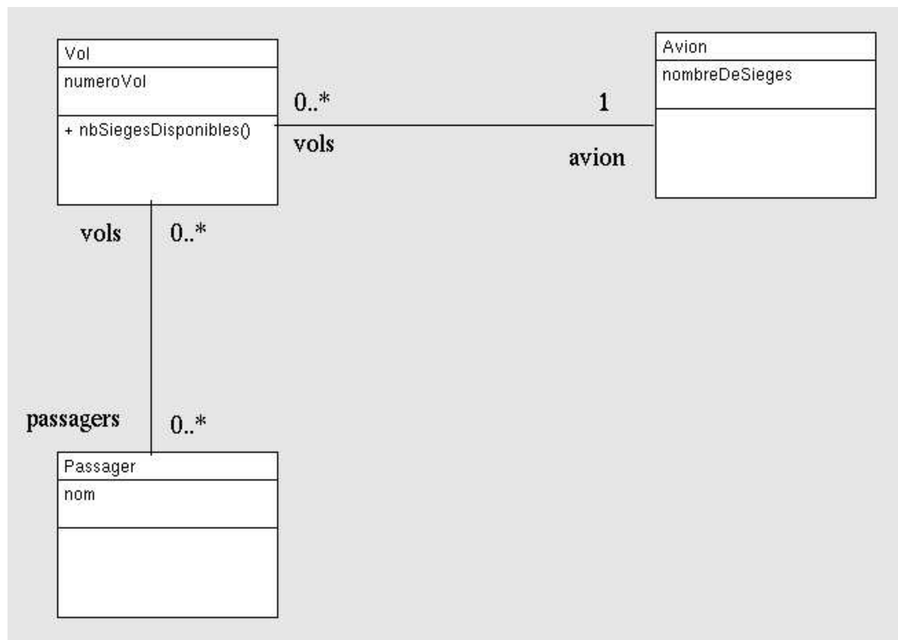
6 Des exemples

• Exemple = formalisation pour rendre *explicite* certaines propriétés *implicites*

Soit les deux affiches suivantes au pied d'un escalier mobile :

```
forAll( p: personne |
     veut_prendre_escalier(p)
     implies
     exists( s: souliers | porte_sur_ses_pieds(p, s) ) )

forAll( p: personne, c: chien |
     veut_prendre_escalier_avec(p, c)
     implies
     porte_dans_ses_bras(p, c) )
```



Contrainte additionnelle (ne pouvant pas être formulée graphiquement) :

```
context Vol
  inv: passagers->size() <= avion.nombreDeSieges
```

• Exemple = Fonction pour calcul de racine carrée d'un nombre réel :

Spécification JML avec exception :

```
    public final static double PRECISION = 0.1;

    /*@ public normal_behavior
      @   requires x >= 0.0;
      @   ensures approximatelyEqualTo( x, \result * \result, PRECISION );
      @ also
      @ public exceptional_behavior
      @   requires x < 0.0;
      @   signals (Exception e) e instanceof RacineImaginaireException;
      @*/
    public static double sqrt( double x ) {
       return ...;
    }
```

Résultats acceptables/permis pour `x = 4.0` et `PRECISION = 0.1` :

```
  1.9748417 < sqrt(4.0) < 2.0248456
```

Question : Est-ce que ce sont les seuls résultats possibles/permis?

Tests JUnit avec *assertions* :

```
import org.junit.*;
import static org.junit.Assert.*;

public class TesterRacineCarree {

    @Test public void testerExact() {
       assertEquals( 2.0, sqrt(4.0), 0.1 );
    }

    @Test public void testerInexact() {
       assertEquals( 1.41421356237, sqrt(2.0), 0.1 );
    }

    @Test(expected=RacineImaginaireException.class)
    public void testerException() {
       sqrt(-12.0);
    }

}
```

Autre spécification JML, avec pré-condition forte :

```
public final static double PRECISION = 0.1;

/*@ requires x >= 0.0;
  @   ensures approximatelyEqualTo( x, \result * \result, PRECISION );
  @*/
public static double sqrt( double x ) {
   assert x >= 0.0;  // Pre-condition forte!

   return ...;
}
```

- Exemple = Une *classe d'objets* pour des `Piles` (en OCL)

```
context Pile
inv tailleBornee:  0 < taille_max and elems->size() <= taille_max

context Pile::Pile()
post:  result.est_vide()

context Pile::est_vide(): Boolean
post:  result = elems->isEmpty()

context Pile::empiler( Object o )
post:  elems = elems@pre->prepend(o)

context Pile::depiler()
pre:  elems->notEmpty()
post:  elems = elems@pre->subSequence(2, elems@pre->size())

context Pile::sommet(): Object
pre:  elems->notEmpty()
post:  result = elems->first()
```

• Exemple = Une *classe d'objets* pour des `Piles` (en Eiffel)

```
class interface
   STACK[T]
feature
   nb_elems: INTEGER

   elem( i: INTEGER ): T
     require
       index_valide: 1 <= i and i <= nb_elems

   initialiser
     ensure
      est_vide: nb_elems = 0

   sommet: T
     require
       pile_non_vide: nb_elems > 0
     ensure
       dernier_arrive: Result = elem(nb_elems)
```

```
   empiler( x: T )
     ensure
       nb_elems_augmente: nb_elems = old nb_elems + 1
       x_au_sommet: sommet = x

   depiler
      require
         pile_non_vide: nb_elems > 0
      ensure
         nb_elems_diminue: nb_elems = old nb_elems - 1

invariant
  nb_elems_jamais_negatif: nb_elems >= 0

end
```

7 Quels sont les principaux bénéfices d'une spécification formelle?

"Devoir en arriver à une meilleure compréhension de l'élément à spécifier en forçant l'analyste à être abstrait tout en étant précis à propos des propriétés du système peut être plus gratifiant que d'avoir le document de spécification lui-même." [Wing90]

- Impact *positif* de l'*effort de formalisation.*
- Spécifications explicites, précises, non-ambiguës.
- Base pour la mise en oeuvre et pour des vérifications formelles.
- Outils (manipulation, analyse, simulation, génération de tests).
- Base pour le développement des tests.

8 Principaux aspects du cours INF3140

En trois mots :

- **Assertions**
- **Contraintes**
- **Contrats**

En quelques mots :

- Assertions
  - Utilisation pour les tests (unitaires)
  - Utilisation pour le débogage
- Contraintes
  - Formalisation de contraintes sur des diagrammes UML
- Contrats
  - Spécification de pré/post-conditions

Voir le “*mind map*” (“carte heuristique”) à la page suivante (produit avec l’outil *FreeMind*)