INF3140 — Modélisation et spécification formelles de logiciels
Examen final (Automne 2009)

Durée: 13h30 – 16h30 (3 heures) **Documentation autorisée:** Toute documentation personnelle.

Nom: _____

Code permanent: | | | | | | | | | | | | |

Directives:

a. Répondez aux questions directement sur le questionnaire.

b. Vous pouvez détacher la dernière page pour consultation.

c. Pour les contrats, vous devez toujours spécifier **au moins** une précondition et **au moins** une postcondition. Si aucune précondition particulière ne s'applique, utilisez `pre: true`.

1	/10
2	/5
3	/10
4	/10
5	/10
Bonus	/5
Total	/45

1. Expressions régulières (10 pts)

[5] 1.1) Écrivez une expression régulière qui permet de valider partiellement (niveau **lexical**) une date écrite sous la forme «*j/m/aa*» ou «*j/m/aaaa*». Plus précisément, pour l'année, il ne faut accepter que les années 1900–2099 (20–21^{ième} siècle). De plus, bien qu'il puisse y avoir des blancs avant ou après la date dans son ensemble, il ne doit pas y en avoir entre les éléments de la date, ni non plus aucun autre caractère.

Par exemple, pour le 5 juillet 2009, on veut accepter des dates telles que «5/7/09», «05/07/09», «5/07/2009 », etc. Par contre, on n'accepte pas «5/7/9», «5/7/009», «5/7/ 2009», «5/7/09 ab».

[5] 1.2) Dans le devoir 2, la classe `AnalyseurXMLAvecRegex` contenait une méthode ayant la signature suivante : `public void analyser( final Sequence<String> s );`

Dans la solution fournie sur le site Web, cette méthode `analyser` appelait la méthode `analyserLigne`, laquelle identifiait une sous-chaine dans la ligne formant une balise complète puis appelait la méthode `analyserBalise` pour en faire l'analyse. Supposons que cette dernière méthode soit définie telle qu'indiqué dans le Code Java 1 — **qui n'est pas** la méthode appropriée pour le devoir 2 mais une **variante**!

Le Code Java 2 présente une méthode de test utilisée pour le devoir 2. Indiquez (directement dans le code) **quelles modifications devraient être apportées aux assertions** pour que le test réussisse à s'exécuter correctement si on utilise la méthode `analyserBalise` indiquée dans le Code Java 1.

```
  private void analyserBalise( String balise, String chainePourBalise ) {
    assert Pattern.compile( "^</?\\w+(\\s+\\w+\\s*=\\s*\"\\w+\"\\s*)*\\s*/?>$" )
           .matcher(chainePourBalise).find();

    InfoBalise ib;
    if ( baliseDefinie(balise) ) {
        ib = info( balise );             // La balise a deja ete vue.
    } else {
        ib = fabriqueIB.creer( balise ); // La balise n'a encore jamais ete vue.
        balises = balises.including( ib );
    }

    Pattern patronAttribut = Pattern.compile( "(\\w+)\\s*=\\s*\"(\\w+)\"" );
    Matcher matcherAttribut = patronAttribut.matcher( chainePourBalise );
    while( matcherAttribut.find() ) {
        ib.ajouterAttribut( matcherAttribut.group(1) );
        if ( ! matcherAttribut.group(1).equals( matcherAttribut.group(2) ) ) {
          ib.ajouterAttribut( matcherAttribut.group(2) );
        }
    }
  }
```

Code Java 1: Version révisée de la méthode `analyserBalise`.

```
  @Test public void avecAttributsMultiples() {
     String l1 = "<TAG1 A1=\"v1\">";
     String l2 = "<TAG2></TAG2></TAG1>";
     String l3 = "<TAG1 A1=\"v1\" A3=\"v3\"></TAG1>";
     String l4 = "<TAG1 A2=\"A2\">";
     String l5 = "</TAG1>";
     anl.analyser( mkSequence( l1, l2, l3, l4, l5 ) );

    assertEquals( mkSet( "TAG1", "TAG2" ), anl.balises() );


    assertEquals( mkSet( "A1", "A2", "A3" ), anl.info( "TAG1" ).attributs() );


    assertEquals( (Integer) 0, anl.info( "TAG1" ).nbOccurrences( "v1" ) );


    assertEquals( (Integer) 2, anl.info( "TAG1" ).nbOccurrences( "A1" ) );


    assertEquals( (Integer) 1, anl.info( "TAG1" ).nbOccurrences( "A2" ) );


    assertEquals( (Integer) 1, anl.info( "TAG1" ).nbOccurrences( "A3" ) );


    assertEquals( emptySet(), anl.info( "TAG2" ).attributs() );
  }
```

Code Java 2: Méthode de test pour la classe `AnalyseurXMLAvecRegex`.

2. Mise en oeuvre d'opérations de la classe `AbstractBag` (5 pts)

Des mises en oeuvre de deux méthodes définies dans la classe `AbstractBag` sont présentées plus bas. Identifiez de quelles méthodes il s'agit.

a.
```
@Override
public int m1( T elem ) {
    final T elem_ = elem;
    return
        this
        .select( new Expression<T,Boolean>() {
            public Boolean eval( T e ) {
                return elem_.equals( e );
            } } )
        .size();
}
```

b.
```
public Bag<T> m2( Expression<T,Boolean> p ) {
    Bag<T> res = this;
    for ( T e : this ) {
        if ( !p.eval(e) ) {
            res = res.excluding( e );
        }
    }
    return res;
}
```

3. Contrats OCL pour des fonctions sur des séquences d'entiers (10 pts)

Spécifiez des contrats OCL pour les fonctions (pures) indiquées ci-bas.

a. Description : Reçoit en entrée une séquence d'entiers et retourne en résultat une version triée (en ordre croissant) de la séquence, mais où chaque élément n'apparait qu'une seule fois.

Exemple : `trierUnique( Sequence{10,8,20,10,8,14} ) = Sequence{8,10,14,20}`

Contrainte : Vous n'avez pas le droit d'utiliser `sortedBy`!

`trierUnique( seq: Sequence(Integer) ): Sequence(Integer)`

b. Description : Reçoit en entrée une séquence d'entiers et un entier, et retourne l'index où apparait la première occurrence de l'élément.

Exemple : `indexPremiereOccurrence( Sequence{4,3,2,2,1}, 2 ) = 3`

`indexPremiereOccurrence( seq: Sequence(Integer), elem: Integer ): Integer`

Contrainte : Vous n'avez pas le droit d'utiliser `indexOf`!

4. Contrats sur modèle pour des programmes d'études universitaires (10 pts)

Spécifiez des contrats OCL pour les deux opérations suivantes sur les entités du modèle conceptuel du Devoir 3.

a. `creerSigle` : Étant donné un département, cette méthode crée un nouveau sigle de cours qui est rattaché au département, sigle ayant le `code` indiqué.

Note : Les invariants suivants sont associés à la classe `Sigle` :

```
inv TroisLettres: code.size() = 3

inv CodeTousDistincts:  Sigle.allInstances->isUnique( code )
```

`Departement::creerSigle( code: String ): Sigle`

b. `retirerPrealable` : Étant donné un cours, cette méthode supprime le cours `c` (indiqué en argument) des préalables requis pour le cours.

`Cours::retirerPrealable( c: Cours )`

5. Contrats pour un type mutable `FilePriorite` (10 pts)

Une pile est une collection dont les accès se font de façon LIFO (*Last-In/First-Out*) : le prochain élément retiré est le dernier ajouté. Une file ordinaire (on dit aussi une *queue*) est une collection dont les accès se font de façon FIFO (*First-In/First-Out*) : le prochain élément retiré est le premier qui a été ajouté (le plus ancien). Quant à une **file de priorité**, le prochain élément retiré *est celui dont le niveau de priorité est le plus élevé*. Cette notion de «niveau de priorité» peut être modélisée par une méthode `priorite`, qui retourne un entier : plus l'entier est grand, plus le niveau de priorité est élevé.

Le module OCL 1 (p. 8) spécifie partiellement (modèle abstrait et signatures des méthodes) un **type mutable** pour des objets `FilePriorite`.

Spécifiez des contrats OCL pour les opérations indiquées ci-bas. Vous pouvez — en fait, vous devez! — utiliser l'opération auxiliaire privée qui est indiquée.

a. `ajouter( x: T ): FilePriorite`

Ajoute l'élément indiqué dans la file de priorité, si le nombre d'éléments déjà présents le permet. Retourne l'objet lui-même (style Smalltalk).

b. `retirerLeMax(): FilePriorite`

Retire l'élément de priorité maximum de la file. Si plusieurs éléments de même priorité sont présents, n'importe lequel est choisi. Retourne l'objet lui-même (style Smalltalk).

Bonus (5 pts)

Soit les classes et les méthodes décrites ci-bas à l'aide de contrats OCL. Certaines de ces classes et méthodes sont des spécifications OCL de concepts Java.

Pour chacune des méthodes `m1`, `m2`, `m3` :

a. Indiquez si cette méthode spécifie une **requête** ou une **commande**.

b. Dans le cas d'une requête, indiquez si cette méthode définit une fonction **pure** ou **impure**.

c. Suggérez un **nom plus significatif**, qui réflète le plus possible ce qui existe en Java.

```
class T end

--

class C
attributes
  elems: Set(T)

operations
  m1(): Boolean
    post:
      result = elems->notEmpty()


  m2(): T
    pre:
      elems->notEmpty()

    post:
      elems@pre->includes(result)
    post:
      elems = elems@pre->excluding(result)
end


--


class FC

operations
  m3( elems: Set(T) ): C
    post:
      result.elems = elems
    post:
      result.oclIsNew()
end
```

```
--
-- Equivalent OCL d'une interface Java: Un objet est de type AvecPriorite
-- s'il possede une methode qui permet de determiner son niveau de priorite.
--
abstract class AvecPriorite
operations
  priorite(): Integer
end



--
-- Type pour argument generique du type FilePriorite.
--
class T < AvecPriorite
end



--
-- Type abstrait (pseudo-)generique pour des files de priorite.
--
class FilePriorite
attributes
  elems: Bag(T)            -- Les objets dans la file.
  nbMaxElems: Integer      -- Le nombre maximum d'elements (collection bornee).

operations
  estVide(): Boolean


  estPleine(): Boolean


  ajouter( x: T ): FilePriorite


  leMax(): T
    pre PasVide: not estVide()
    post ResultEstLeMax: result = max( elems )


  retirerLeMax(): FilePriorite



  --
  -- Operation privee auxiliaire.
  --
  max( elems: Bag(T) ): T =
       elems->any( m |
                   elems->forAll( y | y.priorite() <= m.priorite() ) )
end
```

Module OCL 1: Un type mutable (classe d'objets) pour des files de priorité.