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

Durée: 18h00 – 21h00 (3 heures) **Documentation autorisée:** Toute documentation personnelle.

Nom: _____

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

Directives:

a. Répondez aux questions directement sur le questionnaire. Pour les questions bonus, vous devez utiliser le verso de la feuille pour votre réponse.

b. **Il n'est pas nécessaire de répéter la signature des opérations ou fonctions que vous devez spécifier.**

c. Utilisez des **noms significatifs** pour les identificateurs de vos pré/post-conditions.

1	/15
2	/5
3	/10
4	/10
5	/10
Bonus	/10
Total	/50

1. Type mutable pour des objets `StringBuffer` de style Java (15 pts)

Le module OCL 2 (p. 9) présente une spécification *partielle* d'un **type mutable** pour des objets `StringBuffer` *semblables* (donc pas complètement identiques) à ceux qu'on retrouve en Java.

Deux (2) types auxiliaires (immuables) sont introduits :

- `JString` : Un type pour des chaînes de caractères de style Java (d'où le `J` au début du nom). Si nécessaire, vous devez utiliser ce type et ses opérations plutôt que le type `String` d'OCL. (Notez toutefois que le modèle abstrait pour le `StringBufer` est de type `Sequence(Char)`.)
- `Char` : Un type pour des caractères de style Java, type qui n'existe pas en OCL et qu'on modélise ici par des `String` OCL de longueur 1 (ne pas confondre avec des `JString`).

Spécifiez des contrats avec pré/post-conditions, dans la notation OCL, pour les opérations indiquées ci-bas — utilisez «`true`» comme pré-condition si aucune condition particulière n'est requise.

a. `appendChar( c: Char )`

Si de l'espace est disponible dans le tampon (selon capacité et le nombre de caractères déjà présents), ajoute `c` à la fin du tampon (après les caractères déjà présents).

b. `setCharAt( index: Integer, c: Char )`

Remplace le caractère à la position indiquée par le caractère `c` — le position doit être valide, et ce en fonction des caractères déjà présents et non en fonction de la capacité.

c. `reverse()`

Renverse le contenu du tampon, donc le premier et le dernier caractères sont intervertis, le deuxième et l'avant-dernier, etc. Exemple : Si le tampon contient «`abcde`» avant l'appel, alors il contiendra «`edcba`» après l'appel.

Questions bonus (Répondez au verso)

a. `clone(): StringBuffer` : Crée une copie du `StringBuffer`, i.e., un *nouveau* `StringBuffer` dont le contenu est identique à celui de départ.

b. `indexOf( s: JString ): Integer` : Retourne l'index de la première occurrence de `s` dans le tampon. Retourne 0 si pas présent.

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

Les mises en oeuvre de deux des méthodes associées à la classe `AbstractSequence` sont présentées plus bas. Identifiez de quelles méthodes il s'agit — voir le Code Java 1 (p. 10) pour la signature des différentes méthodes sur les séquences.

a.
```
public Sequence<T> m1( final T elem ) {

   Expression<T,Boolean> estElem = new Expression<T,Boolean>() {
       public Boolean eval( T e ) {
          return elem.equals(e);
       } };

   return this.reject( estElem );
}
```

b.
```
public Sequence<T> m2( Expression<T,Boolean> p ) {
   Sequence<T> res = Collections.emptySequence();
   for ( T e : this ) {
       if ( p.eval(e) ) {
          res = res.append( e );
       }
   }
   return res;
}
```

3. Type immuable pour des `SacBornes` (10 pts)

Le module OCL 1 (p. 8) définit un **type immuable** pour des `SacBorne`s. Il s'agit de «sacs» (*bags*) au sens OCL, c.-à-d. que l'ordre des éléments n'est pas significatif alors que le nombre d'occurrences l'est.

Les valeurs de type `SacBorne` sont des collections *bornées*, et ce de deux façons :

- Le nombre maximum d'éléments distincts que peut contenir le sac est borné — attribut `maxElems`.
- Le nombre maximum d'occurrences d'un élément donné est lui aussi borné — attribut `maxOccurs`.

Exemple : soit un `SacBorne` avec `maxElems=3` et `maxOccurs=2`. Alors, le champ `elems` (de type `Bag(T)`) pourra contenir au plus trois (3) éléments *distincts* et chaque élément ne pourra être présent qu'au plus deux (2) fois. Ces contraintes sont spécifiées dans les invariants à la fin de la spécification du module.

Spécifiez des contrats avec pré/post-conditions, dans la notation OCL, pour les opérations indiquées ci-bas — utilisez `true` comme pré-condition si aucune condition particulière n'est requise.

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

Retourne un sac contenant tous les éléments du sac de départ en plus de l'élément `x` — mais uniquement si les bornes associées au sac le permettent.

b. `supprimerUneOccurrence( x: T ): SacBorne`

Retourne un sac qui contient les mêmes éléments que le sac de départ, sauf pour `x` qui, s'il était présent, apparaît une fois de moins. Si `x` n'est pas présent, le résultat est identique au sac de départ.
Rappel : L'opération `excluding` supprime *toutes* les occurrences d'un élément!

4. Classe d'objets pour comptes bancaires (10 pts)

Spécifiez les contrats OCL des deux opérations suivantes sur les comptes bancaires du Devoir 3.

a. `augmenterMarge( montant: Real )`

Augmente la marge de crédit du compte d'un certain montant (positif!). Une transaction — de sorte `#AUGM_MARGE` — est ajoutée à la fin des transactions courantes. Des frais de 10.00 $ sont associés à cette opération, *payables de façon immédiate* — donc le solde est immédiatement réduit de 10.00 $.

b. `nbAugmentationsDeMarge(): Integer`

Détermine le nombre d'augmentations de marge de crédit qui ont été effectuées depuis l'ouverture du compte. Peut être appelée à tout moment (⇒ analyse des transactions courantes **et** anciennes).

5. Fonctions Java utilisant la bibliothèque `OclCollections` (10 pts)

Complétez la mise en oeuvre des deux fonctions (méthodes statiques) Java manipulant des séquences (de la bibliothèque `OclCollections`) dont les signatures sont présentées plus bas — avec un cas de test JUnit illustrant leur utilisation.

Votre mise en oeuvre, comme dans le devoir 2, doit *utiliser* les opérations de la bibliothèque `OclCollections`, et doit être écrite **dans le «style OCL»**. En d'autres mots, vous ne devez pas écrire du code Java impératif (avec variables et boucles `for`) ; vous devez plutôt utiliser des expressions Java semblables à ce qu'on écrirait en OCL. De plus, vous devez aussi spécifier, à l'aide d'une instruction `assert`, une pré-condition appropriée — utilisez «`assert true;`» si aucune condition particulière n'est nécessaire.

Vous pouvez évidemment, si nécessaire, introduire des méthodes (privées) auxiliaires. Vous pouvez aussi utiliser la méthode `indices()` suivante — qu'on suppose définie dans la classe `AbstractSequence` :

```
public Set<Integer> indices() {
   Set<Integer> s = emptySet();
   for( int k = 1; k <= size(); k++ ) {
       s = s.including( k );
   }
   return s;
}
```

a. `static public <T> Set<Integer> indicesDesElementsMultiples( final Sequence<T> seq )`

Cette fonction retourne l'ensemble des indices des éléments de la séquence qui apparaissent *plusieurs fois* dans la séquence — donc les éléments multiples.

```
@Test public void indexDesElementsMultiples_exemple() {
   assertEquals( mkSet(1,2,3,4,6),
                 indexDesElementsMultiples( mkSequence(10,10,20,10,30,20,50) ) );
}
```

b. `static public <T> T unElementDIndicePair( final Sequence<T> seq )`

Cette fonction retourne un élément arbitraire de la séquence dont l'indice associé (la position dans la séquence) est pair (divisible par 2) — les indices, en OCL, débutent à 1!

```
@Test public void unElementDIndicePair_exemple() {
   int res = unElementDIndicePair( mkSequence(10, 20, 30, 40) );
   assertTrue( res == 20 || res == 40 );
}
```

Modules OCL et code Java

```
class SacBorne

attributes
  maxElems: Integer     -- Nombre maximum d'elements *distincts*.
  maxOccurs: Integer    -- Nombre maximum d'occurrences d'un element donne.
  elems: Bag(T)         -- Les elements.

operations
  estVide(): Boolean =
    elems->isEmpty()

  nbElementsDistincts(): Integer =
    elems->asSet()->size()

  nbOccurrences( x: T ): Integer =
    elems->count(x)


  ajouter( x: T ): SacBorne

  union( autre: SacBorne ): SacBorne

  supprimerTous( x: T ): SacBorne             -- Supprime toutes les occurrences de x.

  supprimerUneOccurrence( x: T ): SacBorne    -- Supprime une seule occurrence de x.

constraints
  inv MaxElementsPositif:
    0 < maxElems

  inv MaxOccurrencesPositif:
    0 < maxOccurs

  inv MaxElementsDistinctsPasDepasse:
    nbElementsDistincts() <= maxElems

  inv MaxOccurrencesPasDepasse:
    elems->forAll( e | elems->count(e) <= maxOccurs )
end
```

Module OCL 1: Un type immuable définissant des sacs bornés.

```
class StringBuffer
attributes
  capacity: Integer            -- Capacite maximale du tampon.
  buffer: Sequence(Char)       -- Contenu effectif du tampon.

operations
  -- Requetes.
  capacity(): Integer = capacity

  length(): Integer   = buffer->size()

  isFull(): Boolean   = (length() = capacity())

  toString(): JString
    post ChaineResultanteOk:
      result.cars = buffer

  indexOf( s: JString ): Integer

  substring( i: Integer, j: Integer ): JString

  clone(): StringBuffer

  -- Commandes.
  ensureCapacity( newCapacity: Integer )

  appendChar( c: Char )

  appendString( s: JString )

  setCharAt( index: Integer, c: Char )

  deleteCharAt( index: Integer )

  reverse()

constraints
  inv CapacityPositive:
    0 < capacity
  inv CapacityJamaisDepassee:
    length() <= capacity()
end

class JString
attributes
  cars : Sequence(Char)
operations
  length(): Integer = cars->size()
  charAt( i: Integer ): Char = cars->at(i)
  concat( js: JString ): JString
    post ConcatenationDesCars:
      result.cars = cars->union( js.cars )
end

class Char
attributes
  value: String
constraints
  inv UnSeulCaractere:
     value.size() = 1
end
```

Module OCL 2: Un type mutable OCL spécifiant une classe `StringBuffer` style Java.

```
public interface Collection<T> extends Iterable<T> {
    int size();

    boolean isEmpty();
    boolean notEmpty();

    int count( T elem ); // Nombre d'occurrences de elem.

    boolean includes( T elem );
    boolean includesAll( Collection<T> col );

    boolean excludes( T elem );
    boolean excludesAll( Collection<T> col );

    boolean exists( Expression<T,Boolean> p );  // Au moins un element a la propriete p.
    boolean forAll( Expression<T,Boolean> p );  // Tous les elements ont la propriete p.
    boolean one( Expression<T,Boolean> p );     // Un seul element a la propriete p.
    boolean isUnique( Expression<T,?> expr );

    T any( Expression<T,Boolean> p );   // Retourne un element arbitraire.

    // Methodes specifiques a Java.
    Iterator<T> iterator();             // Pour obtenir les divers elements de la collection.
    boolean equals( Object s2 );
    String toString();
}


public interface Sequence<T> extends Collection<T> {
    Sequence<T> including( T elem );  // Sequence avec elem ajoute.
    Sequence<T> excluding( T elem );  // Sequence avec elem retire.

    Sequence<T> append( T elem );
    Sequence<T> prepend( T elem );

    T at( int i );
    T first();
    T last();

    Sequence<T> subSequence( int lower, int upper );
    Sequence<T> insertAt( int index, T elem );
    Sequence<T> union( Sequence<T> s2 );  // Concatenation.

    Set<T> asSet();
    Bag<T> asBag();

    Sequence<T> select( Expression<T,Boolean> p );
    Sequence<T> reject( Expression<T,Boolean> p );

    <R> Sequence<R> collect( Expression<T,R> expr );
}
```

Code Java 1: Aide-mémoire pour les opérations sur les séquences de la bibliothèque `OclCollections`.