The University Of Sheffield.

ReMoDeL

Standard Library White Paper

Version: 0.5

Date: 12 April 2010

Anthony J H Simons
Department of Computer Science
University of Sheffield

ReMoDeL Standard Library

The following outlines the growing *standard library* for *ReMoDeL*. These standard classes and methods are assumed to exist in all implementations and may be provided in whatever way the target language finds easiest to support. That is, the *ReMoDeL* library classes could be defined from scratch, or they could be derived from similar classes in the native libraries of each target language.

The natural organisation of the classes here reflects the derivation of the *ReMoDeL* standard library for C++, which was developed first, due to the perceived difficulty of developing this from scratch, but it may turn out that this structure cannot be exactly replicated in other target languages, in which case it will probably change to accommodate what is most possible across all target languages. It is hoped that the declared compatibilities between standard classes and their superclasses, or between standard classes and standard interfaces, may nonetheless be preserved.

In the documentation provided below, classes and interfaces are described in a pseudo-code syntax that is close to Java, merely for the sake of convenience. This is not meant to discriminate against other target languages. The keywords *inherit* and *satisfy* are used in the *ReMoDeL* sense, in the derivation of subclasses and interfaces.

1. Package Organisation

It is assumed that the standard library classes are organised in a number of standard *ReMoDeL* packages, which are either *packages* (in Java), or *namespaces* (in C++), or *clusters* (in Eiffel), etc. and other languages are expected to follow one of these patterns. These are given standard package *names* (used as the namespace-names in languages like C++) or standard *locations* (pathnames used as directory structures in C++, or package names in Java, etc).

The following packages are assumed to exist (so far):

- package *Core*, location *lib.core*: contains the kernel ReMoDeL classes;
- package *Util*, location *lib.util*: contains utility classes such as the collections;
- package *InOut*, location *lib.io*: contains the input/output stream library.

Others may be added to this list. From this, it can be seen that the root directory structure starts with *lib* and then branches out under this. It is assumed that target languages will be able to load the object-code or bytecode for standard library classes from isomorphic directory structures.

2. The Core Package

The *Core* package contains the kernel classes. These include the official root class *Object*, the official root interface *Interface* and facilities to handle the boxing and unboxing of value types, in languages that need this explicitly. Frequently-used base classes such as *String* are expected to be found here. Also, the (small) hierarchy of *Exception* classes is defined here.

In certain target languages, there will be non-publicised implementation classes above *Object* in the class hierarchy. For example, in C++, the reference-counting behaviour is provided by classes called *Ref, Handle* and *Body*, where *Body* is the ultimate ancestor of all classes and interfaces. By contrast, it might make more sense in Java for *lib.core.Object* to be a child of the system root class *java.lang.Object*, and provide its facilities this way.

2.1 Root Interface and Object

The root interface *Interface* declares three methods that all classes are eventually supposed to implement:

```
public interface Interface {
  public Boolean equals(Object other);
  public Natural hashCode();
  public String toString();
}
```

where the types *Boolean* and *Natural* are *ReMoDeL* types that may be translated into built-in basic types in target languages (e.g. in C++ these respectively become *bool* and *unsigned*). All *ReMoDeL* interfaces are extensions of *Interface*. While many target languages do not need to declare a root *Interface*, it is provided to relate both concrete *Object* classes and abstract *Interfaces* to a secret reference-counted ancestor class in certain target implementations (see above).

The root class *Object* implements the basic *Interface* and has the structure:

```
public class Object satisfy Interface {
  public Object();
  public Boolean equals(Object other);
  public Natural hashCode();
  public String toString();
  protected Void assertInvariant();
  protected Void brokenContract(String message);
  protected Void systemError(String message);
}
```

The default implementation of *hashCode()* returns the memory address of the current object, cast to an unsigned natural number. The semantics of *equals(Object)* is meant to indicate deep equality, but may default to identity where objects have no state, as here. The method *toString()* is intended to return a printable representation of an object, by default “anObject”.

The secret method *assertInvariant()* is a placeholder for methods that assert the data type invariant. Each redefined version always invokes the supermethod (to combine invariants), hence this default nullop. The method *brokenContract(String)* is provided for convenience, so that classes may raise exceptions more succinctly. By default, this method creates and raises a *BrokenContract* exception, with the *message* as the construction argument. Similarly, *systemError(String)* creates and raises a *SystemError* exception, with the message as argument.

2.2 Standard Exceptions

There are very few types of exception in the *ReMoDeL* library. This is because standard assumptions are made in generated code about the kinds of faults that can arise. Basically, these are either system-related faults that cannot easily be repaired, or failures resulting from

broken contracts. In principle, the root *Exception* class is derived from *lib.core.Object*; although it may be useful to derive it instead from the target language's undeclared exception class (e.g. *java.lang.RuntimeException*) to avoid having to generate declarations about raised exceptions. The root *Exception* class has the structural interface:

```
public class Exception inherit Object {
  protected String message;
  public Exception();
  public Exception(String message);
  public String toString();
}
```

The default constructor creates an *Exception* with an empty *message*, whereas the standard constructor initialises the *message*. The *toString()* method should return an error string containing the *message*. Depending on how exceptions are naturally reported by the target language, the error string may also be prefixed by the header: "*Exception:* " to indicate the class of exception, if the language does not already do this automatically.

The two subclasses of *Exception* are provided merely to distinguish the two main types of exception and otherwise behave in the same way as *Exception*:

```
public class SystemError inherit Exception {
  public SystemError();
  public SystemError(String message);
  public String toString();  // changed prefix?
}

public class BrokenContract inherit Exception {
  public BrokenContract();
  public BrokenContract(String message);
  public String toString();  // changed prefix?
}
```

The implementations of these classes follow *Exception* and are based on it. The method *toString()* may be overridden if it is desired to add a different class prefix to the error message. Note that the convention for reporting *BrokenContract* exceptions is always to name the current method and the positively-asserted property that was broken, in the style:

```
BrokenContract: first: non-empty sequence
BrokenContract: get: valid index
```

2.3 Standard String

The standard *ReMoDeL String* class belongs to the *Core* package, because of its importance in the kernel, even though it bears a resemblance to the kinds of collection found in the *Util* package. A *String* is immutable and implements the immutable *Sequence* interface, described below. The structural interface of *String* is given, according to one possible derivation, by the declaration:

```
public class String inherit Object satisfy Sequence<Character> {
      // construction interface
  public String();
  public String(String other);
  public String(Collection<Character>);
      // override methods from Object
  public Boolean equals(Object);
```

```
  public Natural hashCode();
  public String toString();
      // implement Collection interface
  public Natural size();
  public Boolean isEmpty();
  public Boolean contains(Character item);
  public Iterator<Character> iterator();
      // implement Sequence interface
  public Sequence<Character> append(Sequence<Character> seq);
  public Sequence<Character> sublist(Integer start, Integer stop);
  public Character first();
  public Character last();
  public Integer firstIndex(Character item, Integer from);
  public Integer lastIndex(Character item, Integer from);
  public Character get(Integer index);
      // specific String operations
  public Integer firstOffset(String text, Integer from);
  public Integer lastOffset(String text, Integer from);
}
```

Alternatively, *lib.core.String* might be derived from *java.lang.String* in Java; or from a default implementation of the *Sequence* interface, such as *AbstractSequence*, which provides a common strategy for some sequence methods. It is likely that *String* will override a number of these, for the sake of efficient implementation.

The constructors must be able to build a legal empty *String* "", a copy of another *String* and a compact *String* copied from a *Collection<Character>* (where *Character* is translated as the base type *char* in some languages). If the target language does not treat primitive strings (viz. *char** in C++) in the same way as the *String* type, a copy constructor for primitive strings must also be provided; and possibly a method supporting assignment of a literal string to a *String* reference. Similarly, the target language must be able to convert a *String* reference into a printable character sequence automatically. (C++ may provide an *operator char*()* for example).

The *Object* methods: *equals, hashCode* and *toString* behave as follows. The *equals* method compares this *String* with the argument, and returns true only if the argument is convertible to a *String*, or to a *Sequence*, whose *Characters* appear in the same order as in this *String* (implementations may attempt to cast the argument). The *hashCode* should be computed according to the famous P J Weinberger algorithm in Aho, Sethi and Ullman, p436. The *toString* conversion should return this *String* object unchanged.

The *Collection* methods: *size, isEmpty, contains* and *iterator* behave as follows. The *size* method returns the length of this *String*. The *isEmpty* method returns true only if this *String* is "". The *contains* method returns true if this *String* contains the specified *Character*. The *iterator* method returns a *StringIterator*, which satisfies the interface *Iterator<Character>*. This iterator may be defined separately, or as an inner class in some target languages (see the collections library).

The *Sequence* methods: *append, sublist, first, last, firstIndex, lastIndex* and *get* behave as follows. The *append* method creates a new string appending the elements of this *String* and the argument *seq*. The *sublist* method creates a new string containing the elements of this *String* from the *start* index, up to but not including the *stop* index. These methods both return *Strings*, although the public result type is *Sequence<Character>,* to be compatible with the *Sequence* interface. Neither method modifies this *String*; in some target languages the result

may have to be constructed using private constructors). The *first* and *last* methods respectively return the first and last *Character* items of this *String*. The *get* method returns a *Character* at an *index*. The *firstIndex* method searches forward, starting *from* an index, and returns the index of the next occurrence of the sought *Character item*, or –1 if the *item* is not found. The *lastIndex* method searches symmetrically backwards.

The *String* methods: *firstOffset* and *lastOffset* perform similar kinds of searching for substrings embedded in this *String*. Note that these methods are differently named (in Java they are overloaded) and all searching methods require a starting index (again, Java provides overloaded versions with and without the index).

All indices to methods are checked, where possible using one-sided bounds checks afforded by unsigned conversion, viz. *((unsigned) index) < size*, which checks for underflow (negative index) and overflow. Methods raise *BrokenContract* exceptions if indices are out of range. Asserted preconditions are: *valid index* (or *indices*) and *non-empty sequence*.

3. The Util Package

The *Util* package contains utility classes, most importantly the collections and the iterators. The collections are organised as a set of abstract interfaces that are satisfied by classes representing alternative implementations. Likewise, the abstract *Iterator* interface is implemented by concrete iterators (possibly inner classes) specific to each collection class. Collections are externally accessible via indices, but are internally traversed via iterators. Likewise, the *foreach*-style of loop is converted to explicit iteration (either by the target language, or by the translator).

3.1 Standard Collection Interfaces

The *Collection* interface is generically-typed over its element-type *T* and has the following signature, extending the root *Interface*:

```
public interface Collection<T> satisfy Interface {
  public Natural size();
  public Boolean isEmpty();
  public Boolean contains(T item);
  public Iterator<T> iterator();
}
```

The outline behaviour of these methods has already been described above. It is possible to compare collections for equality, compute a hash code based on the elements and give a printed representation. All collections may be queried about their size and contents. The only mechanism for traversing a basic *Collection* is by using an *Iterator*, which has the same element-type *T*.

The *Sequence* interface represents immutable sequences, which are indexed and searchable, but cannot be modified. Sequences support constructive versions of the *append* and *sublist* operations. *Sequence* is derived from *Collection* and is also generically typed:

```
public interface Sequence<T> satisfy Collection<T> {
  public Sequence<T> append(Sequence<T> seq);
  public Sequence<T> sublist(Integer start, Integer end);
  public T first();
  public T last();
```

```
  public Integer firstIndex(T item, Integer from);
  public Integer lastIndex(T item, Integer from);
  public T get(Integer index);
}
```

The outline behaviour of these methods has been described above. The methods *append* and *sublist* construct and return new sequences of the same element type. The methods *first* and *last* select the elements at the front and back end of this *Sequence*. The methods *firstIndex* and *lastIndex* respectively search forwards, or backwards from an index for an element, returning the index, or –1 if not found. (Note that for some sequence implementations, *lastIndex* must logically search backwards, but physically search forwards for the last occurrence of the element before the starting index). All *Sequence* types support index-based searching (which should therefore be efficient).

The *List* interface extends the *Sequence* interface with mutating operations. These include adding and removing single elements at both ends, inserting, excising and replacing single elements at an index; and cognate methods (ending with the suffix *All*, e.g. *addAll*) to add, remove, insert and excise whole collections.

```
public interface List<T> satisfy Sequence<T> {
  public Void put(Integer index, T item);
  public Void addFirst(T item);
  public Void addLast(T item);
  public Void removeFirst();
  public Void removeLast();
  public Void remove(T item);
  public Void insert(Integer index, T item);
  public Void excise(Integer index);
  public Void addAll(Collection<T> other);
  public Void removeAll(Collection<T> other);
  public Void insertAll(Integer index, Collection<T> other);
  public Void exciseAll(Integer start, Integer stop);
}
```

The meanings of most of these are self-evident. The *remove(item)* method searches for the first occurrence of the *item* and excises it if found, otherwise does nothing. The *addAll(other)* method iterates over the elements of *other* and performs *addLast(item)* with each one. The *removeAll(other)* method iterates over the elements of other and performs *remove(item)* with each one.

The *Set* interface extends the *Collection* interface and declares the operations:

```
public interface Set<T> satisfy Collection<T> {
  public Void add(T item);
  public Void remove(T item);
  public Void include(Collection<T>);
  public Void exclude(Collection<T>);
  public Void retain(Collection<T>);
}
```

This is not yet the final form of *Set*; and further operations will be added. Also, there may well be some commonality among unordered collections (such as *Set* and *Bag*), which we might wish to capture. This is a work in progress.

3.2 Standard Iterators

All *ReMoDeL* collections may be traversed by iterators. An iterator is guaranteed to traverse a collection in one pass, visiting each element once. Modifying the collection during a traversal is not advised, and has undefined effects. The interface for *Iterator* distinguishes operations that move the cursor from operations that access the element under the cursor (unlike Java, which conflates both).

```
public interface Iterator<T> satisfy Interface {
  Boolean valid();
  T item();
  Void next();
}
```

The *valid* method returns *true* if the *Iterator* is still traversing elements, and *false* if the *Iterator* has passed the final element of the collection. The *item* method returns the current item under the cursor. The *next* method advances the cursor by one place.

Different collection classes provide their own versions of *Iterator*. For example, the *Vector* class returns a *BlockIterator*; the *LinkedList* returns a *LinkIterator*; the *String* returns a *StringIterator*; and so on. Each kind of iterator satisfies the *Iterator* interface, and some instantiate the element-type:

```
class BlockIterator<T> inherit Object satisfy Iterator<T> { … }

class LinkIterator<T> inherit Object satisfy Iterator<T> { … }

class StringIterator inherit Object satisfy Iterator<Character> { … }
```

These have constructors appropriate to the implementation strategy of the collection over which they iterate. For example, the *BlockIterator* may store raw pointers to a *Vector*'s memory block, whereas a *LinkIterator* may store a raw pointer to the current list link. *Iterators* are not used to insert or remove elements from a collection, but merely to traverse them, visiting each element exactly once. Translators may assume that the *foreach* style of loop:

```
foreach (X item : collX) { … }
```

translates into the following code:

```
for (Iterator<X> it = collX.iterator(); it.valid(); it.next()) {
  X item = it.item();
  …
}
```

Iterators are typically implemented in the same source files as the collections over which they iterate.

3.3 Default Collection Implementations

Where the target language permits this as the most obvious way of deriving concrete collections, the following default collection implementations may be provided. Two of *Collection*'s methods have default implementations:

```
public class AbstractCollection<T> inherit Object
                              satisfy Collection<T> {
  public Boolean isEmpty();
  public Boolean contains(T item);
}
```

The *isEmpty* method is implemented in terms of abstract *size*. The *contains* method is implemented by iterating until the iterator *item* matches the sought *item*, or the whole collection has been traversed. Both are efficient. Several of *Sequence*'s methods have default implementations:

```
public class AbstractSequence<T> inherit AbstractCollection<T>
                              satisfy Sequence<T> {
  public Boolean equals(Object other);
  public Natural hashCode();
  public T first();
  public T last();
  public Integer firstIndex(T item, Integer from);
  public Integer lastIndex(T item, Integer from);
}
```

It makes sense to introduce the first default implementations of *equals* and *hashCode* separately for the *Sequence* and *Set* collection types, since the former must take the order of elements into account. The *equals* method tries to convert its argument at least to a *Sequence*, then compares elements. If the argument is not a *Sequence*, false is returned. The *hashCode* method computes an incremental in-place product of the existing *hashCode*, a prime multiplier (31) and the hash code of the next element. The first and last methods are implemented in terms of get with suitable indices; and the *firstIndex* and *lastIndex* searching methods are both implemented in terms of the *iterator* searching from the front.

Note that since *String* is a kind of *Sequence*, it may inherit from *AbstractSequence* in some implementations. This affects any default implementation of *toString()* in the collection types, since the version for generic elements in *AbstractSequence* must be replaced by an identity operation in *String*.