9 Understanding Libraries

9.1 Reading JavaDocs

The documentation for Java projects can be quite extensive. Reading just the comments in the code is difficult. Furthermore, when a library is distributed in the compiled compressed from of an archive (.jar file) you can no longer read the source code to find out what classes and methods the library provides.

To make it possible to generate readable and searchable documentation for Java programs the programmers write the source code comments formatted in a special way *the Javadoc style*. Java then provides a program that reads the documentation and generates nicely formatted web pages that contain all well-formatted comments provided by the programmer.

On the main web page find the link to the documentation for the *tester* package. You see right away that it consists of three *interface*s, six regular *class*es and two *Exception* classes. There is a comment next to each of these. On the left is a list of all classes, interfaces, and exceptions and each name is a link to the detailed description of that particular item.

Follow the link to the *interface ISame*. The code for this interface has been written as follows:

```
package tester;

/**
 * An interface to represent a method that compares
 * two objects for user-defined equality.
 *
 * @author Viera K. Proulx
 * @since 30 May 2007
 */
public interface ISame<T>{

  /**
   * Is this object the same as that?
   * @param that object
   * @return true is the two objects are the same
   *    (by our definition)
   */
  public boolean same(T that);
}
```

We will implement or use this interface later on. Now look at the *class IllegalUseOfTraversalException*. It shows you that programmers can define a new class of exceptions, specific to the situations that may be encountered in their programs. The content of a class that extends *java.lang.RuntimeException* is quite standard and there is not much to see there.

We will now look at where this exception is needed. Follow the link to the *interface Traversal*. Did you notice that the names of *interface*s on the left hand side bar are written in *italics*?

Here is the code for the *interface Traversal*:

```
package tester;

/**
 *  An interface that defines a functional iterator
 *  for traversing datasets
 *
 * @author Viera K. Proulx
 * @since 30 May 2007
 */
public interface Traversal<T> {

  /**
   * Produce true if this
   * <CODE>{@link Traversal Traversal}</CODE>
   * represents an empty dataset
   *
   * @return true if the dataset is empty
   */
   public boolean isEmpty();

   /**
     * <P>Produce the first element in the dataset represented
     * by this <CODE>{@link Traversal Traversal}</CODE> </P>
     * <P>Throws <code>IllegalUseOfTraversalException</code>
     * if the dataset is empty.</P>
     *
     * @return the first element if available -- otherwise
     * throws <code>IllegalUseOfTraversalException</code>
     */
   public T getFirst();

    /**
     * <P>Produce a <CODE>{@link Traversal Traversal}</CODE>
```

```
     * for the rest of the dataset </P>
     * <P>Throws <code>IllegalUseOfTraversalException</code>
     * if the dataset is empty.</P>
     *
     * @return the <CODE>{@link Traversal Traversal}</CODE>
     * for the rest of this dataset if available - otherwise
     * throws <code>IllegalUseOfTraversalException</code>
     */
    public Traversal<T> getRest();
}
```

Next week you can use this as a guide for writing your own *JavaDoc* documentation.

9.2 Implementing Traversals

Create a new project in Eclipse called *Lab9*. Add to it an interface we used before, the *ISelect* interface. Add also the *tester* library.

In the past we have designed classes that represent recursively constructed lists of arbitrary items. However, every time we wanted to add some functionality to these classes, we had to modify all three classes. This works well when we are the sole users of our program. If we want to distribute our program as a library, we need to equip the classes with methods that will allow the users that come later on to manipulate the data contained in this list.

The *Traversal* interface has been designed to supply the methods we may need for any program that needs to look in some orderly manner at the data contained in a list.

We would like to - again - implement our *filter* method. We will need again the *ISelect* interface:

```
// Our usual Selector interface
interface ISelect<T>{
  boolean select(T t);
}
```

We can now design the classes that represent lists of data:

```
//Generic List Union
interface AList<T> extends Traversal<T>{
  // Note that isEmpty(), getFirst() and getRest()
  //   are also added (abstractly) to this class by the
```

```
  //      'implementation' of Traversal
}

//Represents an Empty List of T
class MtList<T> implements AList<T>{
  // Basic Constructor
  public MtList(){}

  // Traversal functions so that things like 'filter' can be
  //   written without disturbing the list classes
  public boolean isEmpty(){ return true; }

  public T getFirst(){
    throw new IllegalUseOfTraversalException(
    "No first element in an empty list"); }

  public Traversal<T> getRest(){
    throw new IllegalUseOfTraversalException(
    "No remaining elements in an empty list"); }

  public String toString(){ return "Mt()"; }
}


//Represents a Non-empty List of T
class ConsList<T> implements AList<T>{
  T first;
  AList<T> rest;

  // Basic Constructor
  public ConsList(T first, AList<T> rest){
    this.first = first;
    this.rest = rest;
  }

  // Traversal functions so that things like 'filter'
  // can also be written without disturbing the list classes
  public boolean isEmpty(){ return false; }

  public T getFirst(){ return this.first; }

  public Traversal<T> getRest(){ return this.rest; }

  public String toString(){
```

```
    return "Cons( "+ first + ", " + rest + ")"; }
}

//First attempt at a generic filter algorithm
class Algorithms1{
  // Filter the Traversal based on the given Selector
  public <T> AList<T> filter(Traversal<T> tr,
                             ISelect<T> pick){

    if(tr.isEmpty())
      return new MtList<T>();
    else
      if(pick.select(tr.getFirst()))
        return new ConsList<T>(tr.getFirst(),
                               filter(tr.getRest(), pick));
      else
        return filter(tr.getRest(), pick);
  }
}
```

Add these classes and interfaces to your project. Make examples of lists of *String*s and design a couple of test cases for these methods. You do not have to complete all tests, but make sure you understand what is going on and how the method in the *Algorithms1* class can be used.

9.3 Designing Datasets

Throughout the rest of the lab you should implement the classes and interfaces given here, make examples of data for each, and run the programs to see the behavior. We suggest that you use a simple class of data, such as a *CartPt* or a *Book* for which you can implement the *ISame* interface, as well as some other interfaces.

One thing to notice about the filter function above is that it can only produce a *List*. We can change/fix this by creating a new interface for *DataSets* where new items can be added to our existing data.

```
// Interface for a collection of data which can be added to
// and later  traversed
interface DataSet<T>{
  DataSet<T> add(T t);
  Traversal<T> getTraversal();
}
```

The reason we need the *getTraversal*() method is because once we have a *DataSet*, we don't want to just keep adding things to it, so we can get a *Traversal* and do some interesting stuff.

Now we have to create some *DataSet*s, and a few methods which can use them. What we can do is *wrap* existing data structures so the interface can be implemented without changing the earlier code:

```
// Wraps an immutable List as a DataSet
class ListWrapper<T> implements DataSet<T>{
    AList<T> list;

    // Clients can only create an empty ListWrapper
    public ListWrapper(){ this(new MtList<T>()); }

    // Private constructor gets passed a List
    private ListWrapper(AList<T> list){ this.list = list; }

    // DataSet Function... add an element to the internal list
    public ListWrapper<T> add(T t){
        return new ListWrapper<T>(new ConsList<T>(t, list));
    }

    // Return the List as a Traversal
    public Traversal<T> getTraversal(){ return list; }
}
```

9.4 Binary Search Trees

Well, here is where we win. Think about the binary search trees. There, too, we add data items, get the first item, remove the first item and get the rest, etc. So, we are doing the same operations as we have been doing with the lists. We should be able to implement the same interfaces as before.

However, before we go on, we need one more interface in order to build a *Binary Search Tree* (*BST*) - a method that allows us to decide the ordering of the data items in the *BST*. Here is a possibility:

```
// Interface for less than comparison...
// ... a little bit like ISame
interface LessThan<T>{
  boolean lessThan(T t);
}
```

Of course, any class whose data we want to store in the *BST* structure must then implement the *LessThan* interface.

Here is what the original implementation of a *BST* may have looked like:

```
//The straight forward BST interface
interface BST<T extends LessThan<T>>{
  // Insert the given T into this BST
  BST<T> insert(T t);

  // Return the Smallest
  T smallest();

  // Chop off the smallest
  BST<T> withoutSmallest();

  // Is this BST a Leaf
  boolean isLeaf();
}

// Represents a non-empty BST
class Node<T extends LessThan<T>> implements BST<T>{
  T data;
  BST<T> left, right;

  // Simple Constructor
  public Node(T d, BST<T> lft, BST<T> rght){
    this.data = d;
    this.left = lft;
    this.right = rght;
  }

  // The usual Insert method...
  // Note: we can have repeated data,
  // it will just go to the right
  public BST<T> insert(T t){
    if(t.lessThan(data))
      return new Node<T>(data, left.insert(t), right);
    else
      return new Node<T>(data, left, right.insert(t));
  }

  // Return the smallest element == farthest Left
  public T smallest(){
```

```
    if(left.isLeaf())
      return data;
    else
      return left.smallest();
  }

  // Remove the smallest element
  public BST<T> withoutSmallest(){
    if(left.isLeaf())
      return right;
    else
      return new Node<T>(data,
                         left.withoutSmallest(), right);
    }

  // Definitely not a Leaf!
  public boolean isLeaf(){ return false; }
}

//Represents the empty BST
class Leaf<T extends LessThan<T>> implements BST<T>{
  // The Default Constructor is just fine

  // Insert a T into this Leaf
  public BST<T> insert(T t){
    return new Node<T>(t, this, this); }

  // No smallest, so we say Error
  public T smallest(){
    throw new IllegalUseOfTraversalException(
    "No smallest element in an empty BST"); }

  // No without smallest, so we say Error
  public BST<T> withoutSmallest(){
    throw new IllegalUseOfTraversalException(
    "No more elements in an empty BST"); }

  // It's a Leaf!
  public boolean isLeaf(){ return true; }
}
```

Well, this does not help much, as all the method names are different, and we do not have the full functionality of the original *DataSet*. So now

how can we use it in some more general algorithms? Well, as you may have guessed... we wrap it!

Here is the code:

```
// Wraps a BST so we can use it
// as a DataSet and/or a Traversal
class BSTWrapper<T extends LessThan<T>>
            implements DataSet<T>, Traversal<T>{
  BST<T> bst;

  // Public Default Constructor, starts empty
  public BSTWrapper(){ this(new Leaf<T>()); }

  // Public, Wraps a given BST
  public BSTWrapper(BST<T> b){ bst = b; }

  // Add a given T to this BST, and Wrap the result
  public BSTWrapper<T> add(T t){
    return new BSTWrapper<T>(bst.insert(t));
  }

  // Return 'this' as a Traversal
  public Traversal<T> getTraversal(){ return this; }

  // Translation of the BST functions
  // to our Traversal interface
  public boolean isEmpty(){ return bst.isLeaf(); }
  public T getFirst(){ return bst.smallest(); }

  // Here we need to wrap the result again
  public Traversal<T> getRest(){
    return new BSTWrapper<T>(bst.withoutSmallest());
  }
}
```

So, now we can implement a filter without using any constructors:

```
//Use our new DataSets to build a completely general
// filter function
class Alg2{
  // Filter the Traversal into the DataSet
  <T> DataSet<T> filterAcc(Traversal<T> tr,
                           ISelect<T> pick,
                           DataSet<T> acc){
```

```
    if(tr.isEmpty()) return acc;
    else
      return filterAcc(tr.getRest(), pick,
                       updateAcc(pick, tr.getFirst(), acc));
  }

  // Update the accumulator if the given T is to be selected
  <T> DataSet<T> updateAcc(ISelect<T> pick,
                           T that,
                           DataSet<T> acc){
    if(pick.select(that))
      return acc.add(that);
    else
      return acc;
  }
}
```

9.5 Javadocs

If you have some time left, convert all documentation for the classes you designed into the *Javadoc* style and generate the web pages of documentation. In the *Project* menu select *Generate Javadoc* and then select which files should be used to generate the documentation. See where you have warnings and fix the problems.