9 Abstracting over Data Type and over Methods

9.1 Introduction

Until now every time we have designed a list of data we needed to specify the type of data the list holds. The amount of repeated code has been tremendous. In the lectures we have seen how we can abstract over the type of data contained in a list using two different techniques, first by construction a list of `Objects`, then by using the type parameter `<T>`.

In this lab we will learn how to abstract over the type of data contained in any class hierarchy that represents a collection of data. We will first explore the techniques seen in the lectures by abstracting over the data contained in binary search trees — a data structure used to store data for which we can define ordering.

This will lead us to abstractions that represent methods (ensuring the desired behavior for specific objects) and abstractions that allow us to access datasets without modifying the classes that define the datasets.

The goal is both to eliminate code duplication and to design programs that can be reused as libraries.

Work though the exercises carefully. We expect that you will finish the work at home as the part of the next assignment.

9.2 Binary Search Trees: Abstracting with Object

A *Binary Search Tree* or *BST* is one of

- *Leaf*
- *Node* that consists of *Data*, a left *BST* and a right *BST*.

A. The code in the files **bookstore-BST.java** and **bankaccts-BST.java** defines two examples of a *binary search tree*. Look at the printout of the two programs and observe the similarities. Run each program as an Eclipse project. Add one more example of a *binary search tree* to each program.

B. Both classes, `Book` and `Acct` define the method `compareTo` that compares `this` instance with the given instance of the same type. Java libraries define the following interface:

```
interface Comparable{
/** compare this item with the given one
  * return 0 if they are the same
  * return <0 if the first one is smaller
  * return >0 if the first one is larger
  */
int compareTo(Object o);
}
```

We can modify the class `Book` so that it implements the `Comparable` interface as follows:

```
// was this book published before the given book?
public int compareTo(Object o){
  Book that = (Book)o;
  return this.year - that.year;
}
```

Modify the `Acct` class in a similar way. Make sure you run the tests after you have made the changes.

C. Create an new Eclipse project BST-object that abstracts over the code that defined the *binary search tree* by using data of the type *Comparable*. Add to the project the classes `Book` and `Account` that you have modified to implement the `Comparable` interface. Run all examples from the previous two programs.

D. The program works, but you can see a number of yellow warning marks along the code. Run the following experiment:

   In the `Examples` class create a BST that contains both instances of `Acct` and instances of `Book`.

   Run the program and observe what happens. The program should run with no error detected.

E. Now build another *mixed* tree in a method `init` as follows. First build a tree with two nodes:

```
// construct a tree with two books
public IBSTComparable init(){
  IBSTComparable tree = new LeafComparable();
  return tree.add(eos).add(oms);
}

IBSTComparable badtree = init();
```

Run the program. It should work fine.

Now change the method `init` to be:

```
// construct a tree with two books and two accounts
public IBSTComparable init(){
  IBSTComparable tree = new LeafComparable();
  return tree.add(eos).add(oms).add(chk1);
}

IBSTComparable badtree = init();
```

Run the program again. It will blow up.

We see that this approach allows us to introduce serious errors into the program that will not be caught until the program runs.

9.3 Binary Search Trees: Abstracting with Type Parameters

The DESIGN RECIPE FOR ABSTRACTIONS consists of the following steps:

A. Observe the similarities between two or more parts of the code. Highlight the differences.

B. Replace the differences by parameters.

C. Implement the original solution or examples using the parametrized version of the code.

D. Re-run the tests.

Following this guidelines, we want to replace the `Acct`] and `Book` in the names of the classes that define the *binary search trees* by a parameter. The parameter will represent the type of data the *binary search tree* will contain.

A. Follow the examples from the lectures and define the class hierarchy for the *binary search tree* parametrized over the type of data, i.e., `interface IBST<T>`, `class Leaf<T>`, and class `Node<T>`, starting from the original definitions.

   Notice the error in the `add` method. For now, eliminate the problem by casting the field `this.data` to `Comparable` as follows:

```
    // add the given object to this binary search tree
     public IBST<T> add(T o){
      if (((Comparable)this.data).compareTo(o) < 0){
    ...
   }
```

B. Add the code for the classes `Book` and `Acct`. Modify these classes so that they implement the `Comparable<Book>` or `Comparable<Acct>` interfaces. Run the tests.

C. Try again to construct a binary search tree that contains both `Books` and `Accts`. You should fail.

9.4 Binary Search Trees with Traversal

The *tester* package defines the following `interface`:

```
// An interface that defines a functional iterator
// for traversing datasets
public interface Traversal<T> {

  // Produce true if this Traversal represents an empty dataset
  public boolean isEmpty();

  // Produce the first element in the dataset
  // represented by this Traversal
  // Throw IllegalUseOfTraversalException if the dataset is empty.
  public T getFirst();

  // Produce a Traversal for the rest of the dataset
  // Throw IllegalUseOfTraversalException if the dataset is empty
  public Traversal<T> getRest();
}
```

We would like to be able to see all data in a *binary search tree* one at a time, in the order from the smallest to the largest. To do so, we can implement the `Traversal` interface.

A. Start a new Eclipse project *BST-generic-traversal* and import into it your solution to the previous problem.

B. Add the methods needed to implement the `Traversal` interface to the class hierarchy that represents the *binary search trees*. The header for the interface `IBST<T>` will be:

```
interface IBST<T> extends Traversal<T>{
...
}
```

and we give you the implementation of the needed methods for the class `Leaf<T>`:

```
  // is this an empty dataset?
  public boolean isEmpty(){
   return true;
  }

  // produce the first element in this dataset
  public T getFirst(){
    throw new IllegalUseOfTraversalException(
    "Cannot access the first element of an empty data set");
  }

  // produce a traversal for the rest of this dataset
  public Traversal<T> getRest(){
    throw new IllegalUseOfTraversalException(
    "Cannot advance to the rest of an empty data set");
}

  // is this the same BST as the given one?
  public boolean same(IBST<T> t){
   return t.isEmpty();
  }
}
```

Remember that the smallest data is in the leftmost node that has no left children, and to remove that node all we need is to move its right child up.

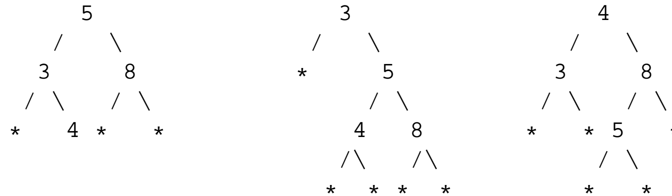
Follow the Design Recipe

C. Implement the method `same` that implements the `ISame` interface for the binary search trees, so that two trees are considered to be the same if they contain the same data.

   The `ISame` interface is defined in the *tester* package as follows:

```
interface ISame<T>{
 boolean same(T t);
}
```

With this comparison the following three trees would be considered to be the same:



9.5 Binary Search Trees with Comparator

There is still one problem with the program we have designed. The class `Book` may have a subclass that implements a different way for comparing two books, but as the things stand now, we could add this book to the *binary search tree* and mix the two different methods for comparing books. To assure that the binary search tree is never built incorrectly, we can do the following:

- Allow the user to construct only a leaf and add all modes using the *add* method.
- Make sure the method for adding new data is determined when the leaf is constructed and is never modified.

The solution to this problem will be a part of the Assignment 9.