6 Abstracting with Function Objects

Portfolio Problems

1. Include the solution of the Lab 6 in your portfolio.
2. Add a few more examples of tests in the `Examples` class.
3. Define a class `ImageSmallerThanAndGivenKind` that implements the `ISelectImageFile` interface with a method that selects image files that are small and of the given kind. Allow the user to decide how small should the images be (measuring the size as the number of pixels in the image) and allowing the user to choose the `kind` of images that should be selected. (All selected images must be of the same `kind`.)

   Test your class definition on several examples before you use it in your `allSuch`, `anySuch`, and `filter` methods.
4. Add test cases that will test the methods `allSuch`, `anySuch`, and `filter` with several instances of the `ImageSmallerThanAndGivenKind` predicate.

Pair Programming Assignment

6.1 Problem

During the lectures we have designed the method `filter` for a list of `Book`s, selecting books written by the given author and books published in the given year.

A. Define the class `Book` and the classes that represent a list of `Book`s. You can start with the classes posted on the lecture notes.

B. Define the following interface in your project:

```
// interface to represent a method compare for books
public interface ICompareBooks{

  // does b1 come before b2 in this ordering?
  public boolean compare(Book b1, Book b2);
}
```

C. Define three classes that implement this interface, ordering the books by the length of the book title (class `BookOrderByTitleLength`), author's names (class `BookOrderByAutor`), and one more criterion of your choice.

D. Design the method `sort` for the classes that represent a list of `Book`s that uses an instance of a class that implements the `ICompareBooks` interface to define the appropriate ordering of the books and produces the list in the correctly sorted order.

E. Design the method `isSorted` for the classes that represent a list of `Book`s that uses an instance of a class that implements the `ICompareBooks` interface to determine whether a list of `Book`s is sorted correctly.

   Make sure your tests use all three ways of comparing books.

   Note: Remember the one task one method rule.

   Note: This is just a slight modification of the methods you have already designed.

6.2 Problem

You will work with a binary search tree that represents a collection of `Book` objects. It should be very similar to the binary search trees that had only integer data.

The class diagram on the next page should help you.

A. Define the classes that represent a binary search tree of `Book` objects as shown in the class diagram above.

B. Define the method `insert` that produces a new binary search tree by inserting a new `Book` into the binary search tree, using the `ICompareBooks` already defined for this tree.

```
                 +---------------------+
                 | abstract class ABST |
                 +---------------------+
      +----------| ICompareBooks order |
      |          +---------------------+
      |                 / \
      |                 ---
      |                  |
      |      -----------------
      |      |               |
      |   +------+   +------------+
      |   | Leaf |   | Node       |
      |   +------+   +------------+
      |              | Book data  |--------+
      |              | ABST left  |        |
      |              | ABST right |        |
      |              +------------+        |
      |                                    v
      v                                 +---------------+
+------------------------------------+  | Book          |
| ICompareBooks                      |  +---------------+
+------------------------------------+  | String title  |
| boolean isBefore(Book b1, Book b2) |  | String author |
+------------------------------------+  | int price     |
                                        +---------------+
```

C. Design the method `getFirst` that produces the first `Book` in the binary search tree (as given by the appropriate `ICompareBooks`.

   In the `Leaf` class this method should have the following body:

```
throw new RuntimeException("No first in an empty tree");
```

D. Design the method `getRest` that produces a new binary search tree with the first `Book` removed.

   In the `Leaf` class this method should have the following body:

```
throw new RuntimeException("No rest of an empty tree");
```

E. Design the method `sameTree` that determines whether this binary search tree is the same as the given one (i.e., has matching structure and matching data in all nodes).

F. Design the method `sameData` that determines whether this binary search tree contains the same books as the given tree.

   Note: Given the following three trees:

```
bstA:       bstB:       bstC:       bstD:
     b3          b3          b2          b3
    /  \        /  \        /  \        /  \
   b2  b4      b2  b4      b1   b4     b1   b4
  /           /                /             \
b1           b1               b3              b5
```

- `bstA` is the same tree as `bstB`
- `bstA` is not the same tree as `bstC`
- `bstA` is not the same tree as `bstD`
- `bstA` has the same data as `bstB`
- `bstA` has the same data as `bstC`
- `bstA` does not have the same data as `bstD`

G. We would like to know whether a binary search tree of books contains the same data as a list of books. Design the method that allows us to make this comparison.

   Write a short explanation of your design.

   Note: You are allowed to introduce new classes or interfaces to solve this problem.

H. Design a new `bstSort` method for the classes that represent a list of books, that first builds a binary search tree from the data in this list, then converts the binary search tree into a sorted list.

   Write a short explanation of your design.

6.3 Problem

In this problem you will start with your solution to Problem 5.4 in the previous assignment, and use some of the code you have written for the Problem 3.1.

A. Start with a new project that includes your solution to the Problem 5.4.

B. Design a class `Capitol` that represents a capitol of one of the 48 US states (not Alaska or Hawaii). This class should extend the class `Place`. This class should have the following features:

   - The field `name` should be the state abbreviation.

- There should be a constructor that consumes the same data in the same order as you have done for the class `City` in Problem 3.1, with the list of neighbors as an additional last argument.
- The conversion from the latitude and longitude data to `CartPt` should emulate the method `toPosn` in the Problem 3.1. If you wish, you can change the size of the Canvas to be 300*x*300.

C. Make sure all methods you have designed for the class `StateMap` including the animation of the routing works correctly.