Abstracting Over Datatypes

Due: 3/08/2011 10:00pm

Portfolio Problems

1. Finish Lab 7 and include all the work in your portfolio.

Pair Programming Assignment

7.1 Problem

This problem is written in the style of a lab tutorial. The goal is to help you understand how to design a more general (*generic*) programs by defining common behavior and structured data using parametrized data types (such as those for lists and/or binary trees).

Begin by downloading *Assignment7.zip* and building a project that contains all these files and the most recent version of the Tester Jar (`tester.jar`). Your project should have the following files:

- `BookBST.java`
- `AcctBST.java`

1. Each file represents a complete program that deals with binary search trees. Set up two run-configurations, one for each of them, and run the programs.
2. In Eclipse, `Window menu > New Window` will open a new window. Set up a new Eclipse window and make sure it is in the *Java Perspective* by selecting `Window menu > Open Perspective > Java`.
   Open the two files in the two windows, full size, side by side, and observe the differences and similarities.

3. Copy the file **AcctBST.java** and add it to the project with the name **BST.java**. We now have two copies of class and interface definitions.

   Comment out or delete the class definition for `Acct` from the new copy, `BST.java`. The one defined in the original file will be used instead.

4. Now replace `Acct` with `<T>` in all places that define classes or interfaces. So,

   - `ABSTAcct` becomes `ABST<T>`
   - `LeafAcct` becomes `Leaf<T>`
   - `NodeAcct` becomes `Node<T>`
   - `ICompAcct` becomes `IComp<T>`

   Rename the `AcctBSTExamples` class to `BSTExamples`.

5. What else needs to be done? In the classes `ABST`, `Leaf`, and `Node`, in every place where we refer to `Acct` replace this with `T`.

6. We are almost done. Look at what still needs to be done. How will we deal with the similarities between the definitions of `ICompAcct` and `ICompBook`? Figure out how to abstract these interfaces with your partner.

7. To complete the abstraction make the necessary changes in the `BSTExamples` class. Here we need to specify what type of data each binary search tree will contain. So, the type `ABSTAcct` becomes `ABST<Acct>` indicating that we are dealing with the abstract class `ABST`, with the type argument (parameter) of `Acct`. Finish the changes so there are no errors or warnings. Run the tests to be sure they pass.

8. Copy the data definitions and tests from the `BookBSTExamples` class, make the necessary changes as above, and run these tests. Make sure they pass.

7.2 Problem

Download the file *Expressions.java*. It includes the implementation and some sample tests of classes that represent arithmetic expression with integer values and binary addition.

1. Study the class diagram for this class hierarchy. Extend the classes so we can also represent multiplication expressions. *Hint:* add the class `Times`.

2. Design the method `asString` that produces a `String` representation of **`this`** expression with binary expressions in parentheses. Define examples that represent the following expressions and include tests that verify they are correctly converted to `String`s:

```
(2 + (3 + 4))
((3 + 5) * ((2 * 3) + 5))
```

3. We now want to represent expressions that compare two integer values (producing a boolean value) and boolean operators like *and* and *or* that combine two boolean values (producing a boolean value). We do this safely by parametrizing each expression over the kind of value it produces when it is evaluated.

   - The `IExp` interface becomes parametrized over the type of value it represents when evaluated.
   - The `BinOp` class needs to be parametrized over the type of operands it receives (assume they must be the same type), as well as the type of value it produces when evaluated.

   Make these changes and convert the rest of the hierarchy to use the new parametrized definitions.

4. Add/modify the necessary class/interface definitions so we can represent `Integer` and `Boolean` values, and *relational*, *boolean*, and *arithmetic* operators. To keep things simple, we limit our choices to *greater-than* (>) and *equal-to* (==). We also want to represent boolean expressions, *and* as well as *or*. Extend your `asString` method to all the necessary classes/interfaces.

   Make sure you have examples for each of them, as well as tests for the `eval` method for each case.

5. Now design two new classes `IntVar` and `BoolVar` that will represent a variable (of the appropriate type) with a `String` representing its name (e.g., `"x"`, `"width"`, etc.). Have each class implement the appropriote `IExp<...>` interface.

The classes require an `eval` method, but the implementation should **`throw`** an exception, indicating that the variable is undefined.

6. Design the method `noVars` for the expression class hierarchy that determins whether or not this expression contains no variables. *Hint*: think about the different cases, and structural recursion.

7. Design the methods `substInt` and `substBool` for the expression class hierarchy that produce a new `IExp` by replacing variable occurrences (of the correct type: `IntVar/BoolVar`) that match the given `String` with the given `Value` instance. Throw an exception if there is an attempt to substitute a `Boolean` value for a matching variable that represents an `Integer`, and vice versa.

   Hint: the signatures look something like: `substInt(String var, Value<Integer> val)`. The special cases are in the `IntVar` and `BoolVar` classes. Others are mostly structural recursion.

7.3 Problem

Rewrite your *ChickenWorld* game in the imperative style. For this you will need the most recent version of the `JavaWorld` library, which includes a class `VoidWorld` with **`void`** methods for most of the world interactions. Review the documentation and examples, available from the links on the assignment page.

You should meet with your TA and go over your previous game. Look over what your partner's group (if different) did, and decide which features to include. Think about the comments from the graders, and have fun.
Don't forget to Test!.