L14: Writing a Program with Abstract Data Types

Today

- Recipe for programming
- Review of datatype

Introduction

Today’s lecture will develop an abstract datatype to solve a simple problem, as an example.

Recipes for Programming

Recall the test-first programming approach for Writing a procedure (a static method):

1. **Spec.** Write the spec, including the method signature (name, argument types, return types, exceptions), and the precondition and the postcondition as a Javadoc comment.
2. **Test.** Create systematic test cases and put them in a JUnit class so you can run them automatically.
   a. You may have to go back and change your spec when you start to write test cases. Just the process of writing test cases puts pressure on your spec, because you’re thinking about how a client would call the method. So steps 1 and 2 iterate until you’ve got a better spec and some good test cases.
   b. Make sure at least some of your tests are failing at first. A test suite that passes all tests even when you haven’t implemented the method is not a good test suite for finding bugs.
3. **Implement.** Write the body of the method. You’re done when the tests are all green in JUnit.
   a. Implementing the method puts pressure on both the tests and the specs, and you may find bugs in them that you have to go back and fix. So finishing the job may require changing the implementation, the tests, and the specs, and bouncing back and forth among them.

Let’s broaden this to a recipe for Writing an abstract data type:

1. **Spec.** Write specs for the operations of the datatype, including method signatures, preconditions, and postconditions.
2. **Test.** Write test cases for the ADT’s operations.
   a. Again, this puts pressure on the spec. You may discover that you need operations you hadn’t anticipated, so you’ll have to add them to the spec.
3. **Implement.** For an ADT, this part expands to:
   a. **Choose rep.** Write down the private fields of a class, or the variants of a recursive datatype. Write down the rep invariant as a comment.
   b. **Assert rep.** Implement a checkRep() method that enforces the rep invariant. This is critically important if the rep invariant is nontrivial, because it will catch bugs much earlier.
   c. **Implement operations.** Write the method bodies of the operations, making sure...
to call checkRep() in them. You’re done when the tests are all green in JUnit.

And let’s broaden it further to a recipe for

Writing a program (consisting of ADTs and procedures):

1. Choose datatypes. Decide which ones will be mutable and which immutable.
2. Choose procedures. Write your top-level procedure and break it down into smaller steps.
3. Spec. Spec out the ADTs and procedures. Keep the ADT operations simple and few at first. Only add complex operations as you need them.
4. Test. Write test cases for each unit (ADT or procedure).
5. Implement simply first. Choose simple, brute-force representations. The point here is to put pressure on the specs and the tests, and try to pull your whole program together as soon as possible. Make the whole program work correctly first. Skip the advanced features for now. Skip performance optimization. Skip corner cases. Keep a to-do list of what you have to revisit.
6. Reimplement and iterate and optimize. Now that it’s all working, make it work better.

Problem: Matrix Multiplication

Suppose we want to compute matrix multiplications and we want to do them faster.

For example, if a, b are scalar constants, and X is a big matrix, then

\[(aX)b\]

is slow to compute because it loops over the matrix X twice: once to multiply it by a, and then again to multiply it by b. It’d be equivalent and cheaper to do:

\[(ab)X\]

That’s just one example of an optimization we could make by rearranging the products in a matrix multiplication. (Remember however that matrix multiplication is associative but not commutative; only scalars commute.)

Choose Datatypes

Let’s call this a Mat.

Let’s define some operations:

make: double -> Mat
make: double[][] -> Mat
  requires: array.length > 0, and array[i].lengths are equal and > 0
  effects: returns a
times: Mat, Mat -> Mat
  requires: m1 and m2 are compatible for multiplication
  effects: returns m1*m2
isIdentity: Mat -> boolean
And the one we really want:

optimize: Mat -> Mat

effects: returns a matrix product with the same value, but which may be faster to compute

Test

Let’s test optimize(). partitions:

ask class for partitions

#scalars in expression: 0, 1, 2, >2

position of scalar in tree: immediate left, immediate right, left-left, left-right, right-left, right-right

Test cases:

X => X covers 0

aX => aX covers 1, imm left

a(Xb) => (ab)X covers 2, imm left, right-right

(aX)b => (ab)X covers 2, imm right, left-left

(Xa)(bY) => (((ab)X)Y) covers 2, left-right, right-left

Choose a Rep

This problem is natural for a recursive datatype.

Mat = Identity + Scalar(double) + Array(double[][])) + Product(Mat, Mat)

```java
private static class Identity implements Mat {
    public Identity() {}
}
```

```java
private static class Scalar implements Mat {
    final double value;
    public Scalar(double value) {
        this.value = value;
    }
}
```

```java
private static class Array implements Mat {
    final double[][] array;
    public Array(double[][] array) {
        this.array = array;
    }
}
```
private static class Product implements Mat {
    final Matrix m1;
    final Matrix m2;
    public Product(Matrix m1, Matrix m2) {
        this.m1 = m1;
        this.m2 = m2;
    }
}

Choose an identity

It’s always good to have a value in the datatype that represents nothing, so that we can avoid using null. For a matrix product, the natural choice is the identity matrix – an empty product expression is just the identity anyway. So let’s define that:

    /** identity for all matrix computation */
    public static final Mat I = new Identity();

Implementing make(): use factory methods

Let’s start implementing, starting with the make() creators. We can’t put a constructor in the interface. And we don’t want to expose our concrete rep classes Scalar, Array, and Product, so that clients won’t depend on them and we’ll be able to change them later (being ready for change).

So we’re going to create static methods to implement make():

    /** @return a matrix product consisting of just the scalar value */
    public static Mat make(double value) {
        return new Scalar(value);
    }

    /** @return a matrix product consisting of just the matrix given */
    public static Mat make(double[][] array) {
        return new Array(array);
    }

These are called factory methods – static methods that play the role of constructors. The factory-method pattern is a common design pattern that you’ll see throughout object-oriented programming, in many languages.

Implementing isIdentity: don’t use instanceof

Now let’s implement the isIdentity observer. Here’s a bad way to do it:

    // don’t do this
    public static boolean isIdentity(Mat m) {
        if (m instanceof Scalar) {
            return ((Scalar)m).value == 1;
        } else if (m instanceof Array) {
            // ... check for 1s on the diagonal and 0s everywhere else
        } else ... // do the right thing for other variant classes
    }
In general, using instanceof in object-oriented programming is a bad smell. Break the operation down into pieces that are appropriate for each class, and write instance methods instead:

```java
private static class Identity implements Mat {
    public boolean isIdentity() { return true; }
}

private static class Scalar implements Mat {
    public boolean isIdentity() { return value == 1; }
}

private static class Array implements Mat {
    public boolean isIdentity() {
        for (int row = 0; row < array.length; row++) {
            for (int col = 0; col < array[row].length; ++col) {
                double expected = (row == col) ? 1 : 0;
                if (array[row][col] != expected) return false;
            }
        }
        return true;
    }
}

private static class Product implements Mat {
    public boolean isIdentity() {
        return m1.isIdentity() && m2.isIdentity();
    }
}
```

Implementing optimize without using instanceof

Let's implement optimize(). Again, here's a bad way to do it, which will quickly get us mired in weeds:

```java
public static Mat optimize(Mat m) {
    if (m instanceof Product) {
        Product p = (Product) m;
        if (p.m1 instanceof Scalar) {
            ...
        } else if (p.m2 instanceof Scalar) {
            ...
        }
    }
}
```

If you find yourself writing code with instanceof checks all over the place, you need to take a step back and rethink the problem.

In particular, to optimize the scalars, we're going to need two recursive helper operations, so we'll add them to our abstract datatype:

```
scalars: Mat -> Mat
  effects: returns a Mat with no matrices in it, only the scalars
```
matrices: Mat -> Mat

effects: returns a Mat with no scalars in it, only matrices in the same order they appear in the input expression

These expressions will allow us to pull the scalars out of an expression and move them together in a single multiplication expression.

```java
/** Mat represents an immutable matrix product. */
public interface Mat {
...
/** @return the product of all the scalars in this expression */
public Mat scalars();

/** @return the product of all the matrices in this expression.
 * times(scalars(), matrices()) is equivalent to this expression. */
public Mat matrices();
}
```

Now we'll implement it as expected:

```java
private static class Identity implements Mat {
    public Mat scalars() { return this; }
    public Mat matrices() { return this; }
}

private static class Scalar implements Mat {
    public Mat scalars() { return this; }
    public Mat matrices() { return I; }
}

private static class Array implements Mat {
    public Mat scalars() { return I; }
    public Mat matrices() { return this; }
}

private static class Product implements Mat {
    public Mat scalars() {
        return times(m1.scalars(), m2.scalars());
    }
    public Mat matrices() {
        return times(m1.matrices(), m2.matrices());
    }
}
```

With these helper functions, optimize() can just separate the scalars and the matrices:

```java
private static class Identity implements Mat {
    public Mat optimize() { return this; }
}

private static class Scalar implements Mat {
    public Mat optimize() { return this; }
}```
private static class Array implements Mat {
    public Mat optimize() { return this; }
}

private static class Product implements Mat {
    public Mat optimize() {
        return times(scalars(), matrices());
    }
}