CSE341, Fall 2011, Lecture 23 Summary

Standard Disclaimer: This lecture summary is not necessarily a complete substitute for attending class, reading the associated code, etc. It is designed to be a useful resource for students who attended class and are later reviewing the material.

This lecture *compares procedural (functional) decomposition and object-oriented decomposition* using the classic example of implementing operations for a small expression language. It shows that the two approaches largely lay out the same ideas in exactly opposite ways, and which way is "better" is either a matter of taste or depends on how software might be changed or *extended* in the future. We then consider how both approaches deal with operations over multiple arguments, which in many object-oriented languages requires a technique called *double (multiple) dispatch* in order to stick with an object-oriented style.

The Basic Set-Up

The following problem is the canonical example of a common programming pattern, and, not coincidentally, is a problem we have already considered a couple times in the course. Suppose we have:

- Expressions for a small "language" such as for arithmetic
- Different *variants* of expressions, such as integer values, negation expressions, and addition expressions
- Different *operations* over expressions, such as evaluating them, converting them to strings, or determining if they contain the constant zero in them

This problem leads to a conceptual *matrix* (two-dimensional grid) with one entry for each combination of variant and operation:

	eval	toString	hasZero
Int			
Add			
Negate			

No matter what programming language you use or how you approach solving this programming problem, you need to indicate what the proper behavior is for each entry in the grid. Certain approaches or languages might make it easier to specify defaults, but you are still deciding something for every entry.

The Functional Approach

In functional languages, the standard style is to do the following:

- Define a *datatype* for expressions, with one *constructor* for each variant. (In a dynamically typed language, we might not give the datatype a name in our program, but we are still thinking in terms of the concept. Similarly, in a language without direct support for constructors, we might use something like lists, but we are still thinking in terms of defining a way to construct each variant of data.)
- Define a *function* for each operation.
- In each function, have a branch (e.g., via pattern-matching) for each variant of data. If there is a default for many variants, we can use something like a wildcard pattern to avoid enumerating all the branches.

Note this approach is really just procedural decomposition: breaking the problem down into procedures corresponding to each operation.

This ML code shows the approach for our example: Notice how we define all the kinds of data in one place and then the nine entries in the table are implemented "by column" with one function for each column:

```
exception BadResult of string

datatype exp =
    Int    of int
  | Negate of exp
  | Add    of exp * exp

fun eval e = (* no environment because we don't have variables *)
    case e of
        Int _      => e
      | Negate e1  => (case eval e1 of
                           Int i => Int (~i)
                         | _ => raise BadResult "non-int in negation")
      | Add(e1,e2) => (case (eval e1, eval e2) of
                           (Int i, Int j) => Int (i+j)
                         | _ => raise BadResult "non-ints in addition")

fun toString e =
    case e of
        Int i      => Int.toString i
      | Negate e1  => "-(" ^ (toString e1) ^ ")"
      | Add(e1,e2) => "("  ^ (toString e1) ^ " + " ^ (toString e2) ^ ")"

fun hasZero e =
    case e of
        Int i      => i=0
      | Negate e1  => hasZero e1
      | Add(e1,e2) => (hasZero e1) orelse (hasZero e2)
```

The Object-Oriented Approach

In object-oriented languages, the standard style is to do the following:

- Define a *class* for expressions, with one *abstract method* for each operation. (In a dynamically typed language, we might not actually list the abstract methods in our program, but we are still thinking in terms of the concept. Similarly, in a language with duck typing, we might not actually use a superclass, but we are still thinking in terms of defining what operations we need to support.)
- Define a *subclass* for each variant of data.
- In each subclass, have a method definition for each operation. If there is a default for many variants, we can use a method definition in the superclass so that via inheritance we can avoid enumerating all the branches.

Note this approach is a data-oriented decomposition: breaking the problem down into classes corresponding to each data variant.

This Java code[1] shows the approach for our example: Notice how we define all the operations in one place

[1] We use the name `toStrng` instead of `toString` to avoid overriding the `toString` method in class `Object`.

(the abstract class) and then the nine entries in the table are implemented "by row" with one class for each row.

```
abstract class Exp {
    abstract Value eval();
    abstract String toStrng();
    abstract boolean hasZero();
}
class Int extends Exp {
    public int i;
    Int(int i) {
        this.i = i;
    }
    Value eval() {
        return this;
    }
    String toStrng() {
        return "" + i;
    }
    boolean hasZero() {
        return i==0;
    }
}
class Negate extends Exp {
    public Exp e;
    Negate(Exp e) {
        this.e = e;
    }
    Value eval() {
        // we downcast from Exp to Int, which will raise a run-time error
        // if the subexpression does not evaluate to an Int
        return new Int(- ((Int)(e.eval())).i);
    }
    String toStrng() {
        return "-(" + e.toStrng() + ")";
    }
    boolean hasZero() {
        return e.hasZero();
    }
}
class Add extends Exp {
    Exp e1;
    Exp e2;
    Add(Exp e1, Exp e2) {
        this.e1 = e1;
        this.e2 = e2;
    }
    Value eval() {
        // we downcast from Exp to Int, which will raise a run-time error
        // if either subexpression does not evaluate to an Int
        return new Int(((Int)(e1.eval())).i + ((Int)(e2.eval())).i);
```

```
    }
    String toStrng() {
        return "(" + e1.toStrng() + " + " + e2.toStrng() + ")";
    }
    boolean hasZero() {
        return e1.hasZero() || e2.hasZero();
    }
}
```

This Ruby code is similar, though we do not need to define abstract methods or insert type casts. (Even the superclass is unnecessary, but we leave it in for clarity and since in more complicated examples it is a good place to put helper methods.)

```
class Exp
  # could put default implementations or helper methods here
end
class Int < Exp
  attr_reader :i
  def initialize i
    @i = i
  end
  def eval
    self
  end
  def toString
    @i.to_s
  end
  def hasZero
    i==0
  end
end
class Negate < Exp
  attr_reader :e
  def initialize e
    @e = e
  end
  def eval
    Int.new(-e.eval.i) # error if e.eval has no i method (not an Int)
  end
  def toString
    "-(" + e.toString + ")"
  end
  def hasZero
    e.hasZero
  end
end
class Add < Exp
  attr_reader :e1, :e2
  def initialize(e1,e2)
    @e1 = e1
    @e2 = e2
```

```
  end
  def eval
    Int.new(e1.eval.i + e2.eval.i) # error if e1.eval or e2.eval have no i method
  end
  def toString
    "(" + e1.toString + " + " + e2.toString + ")"
  end
  def hasZero
    e1.hasZero || e2.hasZero
  end
end
```

The Punch-Line

So we have seen that functional decomposition breaks programs down into functions that perform some operation and object-oriented decomposition breaks programs down into classes that give behavior to some kind of data. These are so exactly opposite that they are the same — just deciding whether to lay out our program "by column" or "by row." Understanding this symmetry is invaluable in conceptualizing software or deciding how to decompose a problem. Moreover, various software tools and IDEs can help you view a program in a different way than the source code is decomposed. For example, a tool for an OOP language that shows you all methods `foo` that override some superclass' `foo` is showing you a column even though the code is organized by rows.

So, which is better? It is often a matter of personal preference whether it seems "more natural" to lay out the concepts by row or by column, so you are entitled to your opinion. What opinion is most common can depend on what the software is about. For our expression problem, the functional approach is probably more popular: it is "more natural" to have the cases for `eval` together rather than the operations for `Negate` together. For problems like implementing graphical user interfaces, the object-oriented approach is probably more popular: it is "more natural" to have the operations for a kind of data (like a `MenuBar`) together (such as `backGroundColor`, `height`, and `doIfMouseIsClicked` rather than have the cases for `doIfMouseIsClicked` together (for `MenuBar`, `TextBox`, `SliderBar`, etc.). The choice can also depend on what programming language you are using, how useful libraries are organized, etc.

Extending the Code

The choice between "rows" and "columns" becomes less subjective if we later extend our program by adding new data variants or new operations.

Consider the functional approach. Adding a new operation is easy: we can implement a new function without editing any existing code. For example, this function creates a new expression that evaluates to the same result as its argument but has no negative constants:

```
fun noNegConstants e =
    case e of
        Int i       => if i < 0 then Negate (Int(~i)) else e
      | Negate e1   => Negate(noNegConstants e1)
      | Add(e1,e2)  => Add(noNegConstants e1, noNegConstants e2)
```

On the other hand, adding a new data variant, such as `Mult of exp * exp` is less pleasant. We need to go back and change all our functions to add a new case. In a statically typed language, we do get some help: after adding the `Mult` constructor, *if* our original code did not use wildcard patterns, then the type-checker will give a non-exhaustive pattern-match warning everywhere we need to add a case for `Mult`.

Again the object-oriented approach is exactly the opposite. Adding a new variant is easy: we can implement a

new subclass without editing any existing code. For example, this (Java) class adds multiplication expressions to our language:

```
class Mult extends Exp {
    Exp e1;
    Exp e2;
    Mult(Exp e1, Exp e2) {
        this.e1 = e1;
        this.e2 = e2;
    }
    Value eval() {
        return new Int(((Int)(e1.eval())).i * ((Int)(e2.eval())).i);
    }
    String toStrng() {
        return "(" + e1.toStrng() + " * " + e2.toStrng() + ")";
    }
    boolean hasZero() {
        return e1.hasZero() || e2.hasZero();
    }
}
```

On the other hand, adding a new operation, such as `noNegConstants`, is less pleasant. We need to go back and change all our classes to add a new method. In a statically typed language, we do get some help: after adding the abstract method `abstract Exp noNegConstants();` to the `Exp` class, the type-checker will give an error for any non-abstract class that needs to implement the method.

Planning for extensibility

As seen above, functional decomposition allows new operations and object-oriented decomposition allows new variants without modifying existing code and without explicitly planning for it — the programming styles "just work that way." It is possible for functional decomposition to support new variants or object-oriented decomposition to support new operations *if you plan ahead* and use somewhat awkward programming techniques (that seem less awkward over time if you use them often).

We do not consider these techniques in detail here. For object-oriented programming, "the visitor pattern" is a common approach. This pattern often is implemented using double dispatch, which is covered for other purposes below. For functional programming, we can define our datatypes to have an "other" possibility and our operations to take in a function that can process the "other data." Here is the idea in SML:

```
datatype 'a ext_exp =
    Int    of int
  | Negate of 'a ext_exp
  | Add    of 'a ext_exp * 'a ext_exp
  | OtherExtExp  of 'a

fun eval_ext (f,e) = (* notice we pass a function to handle extensions *)
  case e of
    Int i       => i
  | Negate e1   => 0 - (eval_ext (f,e1))
  | Add(e1,e2)  => (eval_ext (f,e1)) + (eval_ext (f,e2))
  | OtherExtExp e => f e
```

With this approach, we could create an extension supporting multiplication by instantiating `'a` with `exp * exp`,

passing `eval_ext` the function `(fn (x,y) => eval_ext(f,e1) * eval_ext(f,e2))`, and using `OtherExtExp(e1,e2)` for multiplying `e1` and `e2`. This approach can support different extensions, but it does not support well combining two separately created extensions.

Notice that it does *not* work to wrap the original datatype in a new datatype like this:

```
datatype myexp_wrong =
         OldExp of exp
       | MyMult of myexp_wrong * myexp_wrong
```

This approach does not allow, for example, a subexpression of an `Add` to be a `MyMult`.

Thoughts on Extensibility

It seems clear that if you expect new operations, then a functional approach is more natural and if you expect new data variants, then an object-oriented approach is more natural. The problems are (1) the future is often difficult to predict; we may not know what extensions are likely, and (2) both forms of extension may be likely. Newer languages like Scala aim to support both forms of extension well; we are still gaining practical experience on how well it works as it is a fundamentally difficult issue.

More generally, making software that is both robust and extensible is valuable but difficult. Extensibility can make the original code more work to develop, harder to reason about locally, and harder to change (without breaking extensions). In fact, languages often provide constructs exactly to *prevent* extensibility. ML's modules can hide datatypes, which prevents defining new operations over them outside the module. Java's `final` modifier on a class prevents subclasses.

Binary (N-Ary) Operations and Double Dispatch

The operations we have considered so far used only one value of a type with multiple data variants: `eval`, `toString`, `hasZero`, and `noNegConstants` all operated on one expression. When we have operations that take two (binary) or more (n-ary) variants as arguments, we often have many more cases, and it is more difficult in many object-oriented languages to maintain an object-oriented style.

For sake of example, suppose we add string values and rational-number values to our expression language. Further suppose we change the meaning of `Add` expressions to the following:

- If the arguments are ints or rationals, do the appropriate arithmetic.
- If either argument is a string, convert the other argument to a string (unless it already is one) and return the concatenation of the strings.

(For simplicity, the SML/Ruby/Java code associated with this summary does not change the meaning of `Negate` or `Mult`; evaluation of these expressions raises a run-time error if either subexpression evaluates to a value that is not an `Int`.)

The interesting change to the SML code is in the `Add` case of `eval`. We now have to consider 9 (i.e., $3 * 3$) subcases, one for each combination of values produced by evaluating the subexpressions. To make this explicit and more like the object-oriented version considered below, we move these cases out into a helper function `add_values` as follows:

```
fun eval e =
   case e of
      ...
     | Add(e1,e2)  => add_values (eval e1, eval e2)
      ...
```

```
fun add_values (v1,v2) =
    case (v1,v2) of
        (Int i, Int j) => Int (i+j)
      | (Int i, String s) => String(Int.toString i ^ s)
      | (Int i, Rational(j,k)) => Rational(i*k+j,k)
      | (String s, Int i) => String(s ^ Int.toString i) (* not commutative *)
      | (String s1, String s2) => String(s1 ^ s2)
      | (String s, Rational(i,j)) => String(s ^ Int.toString i ^ "/" ^ Int.toString j)
      | (Rational _, Int _) => add_values(v2,v1) (* commutative: avoid duplication *)
      | (Rational(i,j), String s) => String(Int.toString i ^ "/" ^ Int.toString j ^ s)
      | (Rational(a,b), Rational(c,d)) => Rational(a*d+b*c,b*d)
      | _ => raise BadResult "non-values passed to add_values"
```

Notice our functional decomposition generalizes in a straightforward way to binary (or in general, n-ary) operations: we can use pattern-matching over tuples to enumerate all the cases.

While the number of cases may be large, that is inherent to the problem. If many cases work the same way, we can use wildcard patterns and/or helper functions to avoid redundancy. One common source of redundancy is *commutativity*, i.e., the order of values not mattering. In the example above, there is only one such case: adding a rational and an int is the same as adding an int and a rational. Notice how we exploit this redundancy by having one case use the other with the call `add_values(v2,v1)`.

We now turn to supporting the same addition in an object-oriented style, as in either our Java or Ruby code. The obvious first step is to have all the classes for values in our language, i.e., `Int`, `Rational`, and `Strng`, define an `add_values` method that takes one argument, the "other" value to be added. Then the `eval` method of the `Add` class would be as follows in Ruby:

```
  def eval
    e1.eval.add_values e2.eval
  end
```

In Java, for type-checking purposes, we can create an abstract class `Value` (since non-value expression classes do not have an `add_values` method) and arrange things as follows:

```
abstract class Exp {
    abstract Value eval();
    abstract String toStrng();
    abstract boolean hasZero();
}
abstract class Value extends Exp {
    abstract Value add_values(Value other);
}
class Add extends Exp {
   ...
    Value eval() {
        return e1.eval().add_values(e2.eval());
    }
   ...
}
```

By putting `add_values` methods in the `Int`, `Strng`, and `Rational` classes, we nicely divide our work into three pieces using dynamic dispatch depending on the class of the object that `e1.eval()` returns, i.e., the

receiver of the `add_values` call in the `eval` method in `Add`. But then each of these three needs to handle three of the nine cases, based on the class of the second argument. One approach would be to, in these methods, abandon object-oriented style and use run-time tests of the classes to include the three cases. The Ruby code would look like this and the Java code would be similar using `instanceof`:

```
class Int
   ...
   def add_values other
      if other.is_a? Int
          ...
      elsif other.is_a? Rational
          ...
      else
          ...
      end
   end
end
class Rational
   ...
   def add_values other
      if other.is_a? Int
          ...
      elsif other.is_a? Rational
          ...
      else
          ...
      end
   end
end
class Strng
   ...
   def add_values other
      if other.is_a? Int
          ...
      elsif other.is_a? Rational
          ...
      else
          ...
      end
   end
end
```

While this approach works, it is really not object-oriented programming. Rather, it is a mix of object-oriented decomposition (dynamic dispatch on the first argument) and functional decomposition (using `is_a?` to figure out the cases in each method). There is not necessarily anything wrong with that: it is probably simpler to understand than what we are about to demonstrate.

A "full" object-oriented solution would use just dynamic dispatch to choose among the nine cases, not `is_a?` tests. Some languages have *multimethods* to support this sort of operation. A multimethod allows multiple implementations (just like regular methods support implementations in different classes) and chooses which method is dispatched to using the classes of multiple arguments instead of just the receiver. If we had multimethods, we could define one called `add_values`, provide nine implementations, and rely on dynamic

dispatch to pick the right one.

But neither Ruby nor Java has multimethods. In such languages, there is a trick to coding up the same idea manually. This trick is called *double-dispatch* when the total number of arguments is 2 (the receiver plus 1 other), like with `add_values`. With additional work, the technique extends to any number of arguments.

We will demonstrate double dispatch by using it to complete `add_values` in object-oriented style. The relevant Ruby and Java code appears at the end. The idea is to have all our value classes implement three more methods:

- `addInt` takes an argument `other` and produces the result of addition where the left argument is an `Int other` and the right argument is `self` / `this`.
- `addString` takes an argument `other` and produces the result of addition where the left argument is a `Strng other` and the right argument is `self` / `this`.
- `addRational` takes an argument `other` and produces the result of addition where the left argument is a `Rational other` and the right argument is `self` / `this`.

So with three classes each implementing these three methods, we have the nine cases we need. Now we just need to dispatch to the correct one. Our `eval` method in `Add` already does the "first dispatch" to pick the `add_values` method of its receiver. We then implement these methods to do the "second dispatch" by having `add_values` in `Int` call `other.addInt self`, `add_values` in `Strng` call `other.addString self`, and `add_values` in `Rational` call `other.addRational self`.

```
class Int < Exp
  ...
  def add_values v # first dispatch
    v.addInt self
  end
  def addInt v # second dispatch: other is Int
    Int.new(v.i + i)
  end
  def addString v # second dispatch: other is Strng (notice order flipped)
    Strng.new(v.s + i.to_s)
  end
  def addRational v # second dispatch: other is Rational
    Rational.new(v.i+v.j*i,v.j)
  end
end
class Strng
  ...
  def add_values v # first dispatch
    v.addString self
  end
  def addInt v # second dispatch: other is Int (notice order is flipped)
    Strng.new(v.i.to_s + s)
  end
  def addString v # second dispatch: other is Strng (notice order flipped)
    Strng.new(v.s + s)
  end
  def addRational v # second dispatch: other is Strng (notice order flipped)
    Strng.new(v.i.to_s + "/" + v.j.to_s + s)
```

```
  end
end
class Rational < Exp
  ...
  def add_values v # first dispatch
    v.addRational self
  end
  def addInt v # second dispatch: reuse computation of commutative operation
    v.addRational self
  end
  def addString v # second dispatch: other is Strng (notice order flipped)
    Strng.new(v.s + i.to_s + "/" + j.to_s)
  end
  def addRational v # second dispatch: other is Strng (notice order flipped)
    a,b,c,d = i,j,v.i,v.j
    Rational.new(a*d+b*c,b*d)
  end
end
class Add < Exp
  ...
  def eval
    e1.eval.add_values e2.eval # start off the double-dispatch
  end
end
```

```
abstract class Value extends Exp {
    abstract Value add_values(Value other); // first dispatch
    abstract Value addInt(Int other); // second dispatch
    abstract Value addString(Strng other); // second dispatch
    abstract Value addRational(Rational other); // second dispatch
}
class Int extends Value {
    ...
    Value add_values(Value other) {
        return other.addInt(this);
    }
    Value addInt(Int other) {
        return new Int(other.i + i);
    }
    Value addString(Strng other) {
        return new Strng(other.s + i);
    }
    Value addRational(Rational other) {
        return new Rational(other.i+other.j*i,other.j);
    }
}
class Strng extends Value {
    ...
    Value add_values(Value other) {
        return other.addString(this);
    }
```

```
    Value addInt(Int other) {
        return new Strng("" + other.i + s);
    }
    Value addString(Strng other) {
        return new Strng(other.s + s);
    }
    Value addRational(Rational other) {
        return new Strng("" + other.i + "/" + other.j + s);
    }
}
class Rational extends Value {
    ...
     Value add_values(Value other) {
        return other.addRational(this);
    }
    Value addInt(Int other) {
        // reuse computation of commutative operation
        return other.addRational(this);
    }
    Value addString(Strng other) {
        return new Strng(other.s + i + "/" + j);
    }
    Value addRational(Rational other) {
        int a = i;
        int b = j;
        int c = other.i;
        int d = other.j;
        return new Rational(a*d+b*c,b*d);
    }
}
class Add extends Exp {
    ...
    Value eval() {
        return e1.eval().add_values(e2.eval());
    }
}
```