Decomposition

Principles of Functional Programming

Decomposition

Suppose you want to write a small interpreter for arithmetic expressions.

To keep it simple, let's restrict ourselves to numbers and additions.

Expressions can be represented as a class hierarchy, with a base trait `Expr` and two subclasses, `Number` and `Sum`.

To treat an expression, it's necessary to know the expression's shape and its components.

This brings us to the following implementation.

Expressions

```
trait Expr:
  def isNumber: Boolean
  def isSum: Boolean
  def numValue: Int
  def leftOp: Expr
  def rightOp: Expr

class Number(n: Int) extends Expr:
  def isNumber = true
  def isSum = false
  def numValue = n
  def leftOp = throw Error("Number.leftOp")
  def rightOp = throw Error("Number.rightOp")
```

Expressions (2)

```
class Sum(e1: Expr, e2: Expr) extends Expr:
  def isNumber = false
  def isSum = true
  def numValue = throw Error("Sum.numValue")
  def leftOp = e1
  def rightOp = e2
```

Evaluation of Expressions

You can now write an evaluation function as follows.

```
def eval(e: Expr): Int =
  if e.isNumber then e.numValue
  else if e.isSum then eval(e.leftOp) + eval(e.rightOp)
  else throw Error("Unknown expression " + e)
```

Problem: Writing all these classification and accessor functions quickly becomes tedious!

Problem: There's no static guarantee you use the right accessor functions. You might hit an `Error` case if you are not careful.

Adding New Forms of Expressions

So, what happens if you want to add new expression forms, say

```
class Prod(e1: Expr, e2: Expr) extends Expr   // e1 * e2
class Var(x: String) extends Expr             // Variable ‘x’
```

You need to add methods for classification and access to all classes defined above.

Question

To integrate `Prod` and `Var` into the hierarchy, how many new method definitions do you need?

(including method definitions in `Prod` and `Var` themselves, but not counting methods that were already given on the slides)

Possible Answers

0	9
0	10
0	19
0	25
0	35
0	40

Question

To integrate `Prod` and `Var` into the hierarchy, how many new method definitions do you need?

(including method definitions in `Prod` and `Var` themselves, but not counting methods that were already given on the slides)

Possible Answers

0	9
0	10
0	19
0	25
0	35
0	40

Non-Solution: Type Tests and Type Casts

A "hacky" solution could use type tests and type casts.

Scala let's you do these using methods defined in class `Any`:

```
def isInstanceOf[T]: Boolean  // checks whether this object's type conforms to
def asInstanceOf[T]: T        // treats this object as an instance of type 'T'
                              // throws 'ClassCastException' if it isn't.
```

These correspond to Java's type tests and casts

Scala	Java
`x.isInstanceOf[T]`	`x instanceof T`
`x.asInstanceOf[T]`	`(T) x`

But their use in Scala is discouraged, because there are better alternatives.

Eval with Type Tests and Type Casts

Here's a formulation of the `eval` method using type tests and casts:

```
def eval(e: Expr): Int =
  if e.isInstanceOf[Number] then
    e.asInstanceOf[Number].numValue
  else if e.isInstanceOf[Sum] then
    eval(e.asInstanceOf[Sum].leftOp)
    + eval(e.asInstanceOf[Sum].rightOp)
  else throw Error("Unknown expression " + e)
```

This is ugly and potentially unsafe.

Solution 1: Object-Oriented Decomposition

For example, suppose that all you want to do is *evaluate* expressions.

You could then define:

```
trait Expr:
  def eval: Int

class Number(n: Int) extends Expr:
  def eval: Int = n

class Sum(e1: Expr, e2: Expr) extends Expr:
  def eval: Int = e1.eval + e2.eval
```

But what happens if you'd like to display expressions now?

You have to define new methods in all the subclasses.

Assessment of OO Decomposition

- OO decomposition mixes *data* with *operations* on the data.
- This can be the right thing if there's a need for encapsulation and data abstraction.
- On the other hand, it increases complexity(*) and adds new dependencies to classes.
- It makes it easy to add new kinds of data but hard to add new kinds of operations.

(*) In the literal sense of the word:
complex = plaited, woven together

Thus, complexity arises from mixing several things together.

Limitations of OO Decomposition

OO decomposition only works well if operations are on a *single* object.

What if you want to simplify expressions, say using the rule:

```
a * b + a * c   ->   a * (b + c)
```

Problem: This is a non-local simplification. It cannot be encapsulated in the method of a single object.

You are back to square one; you need test and access methods for all the different subclasses.