Decomposition

Expressions

- 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 *Expression* 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:

```
trait Expression:
   def isNumber: Boolean
   def isSum: Boolean
   def numericValue: Int
   def leftOperand: Expression
   def rightOperand: Expression

class Number(n: Int) extends Expression:
   def isNumber: Boolean = true
   def isSum: Boolean = false
   def numericValue: Int = n
   def leftOperand: Expression = throw Error(“Number.leftOperand”)
   def rightOperand: Expression = throw Error(“Number.rightOperand”)

class Sum(e1: Expression, e2: Expression) extends Expression:
   def isNumber: Boolean = false
   def isSum: Boolean = true
   def numericValue: Int = throw Error(“Sum.numericValue”)
   def leftOperand: Expression = e1
   def rightOperand: Expression = e2
```

Evaluation of Expressions

- You can write an evaluation function as follows:

```
def evaluate(e: Expression): Int =
   if e.isNumber then e.numericValue
   else if e.isSum then evaluate(e.leftOperand) + evaluate(e.rightOperand)
   else throw Error(“Unknown expression ” + e)
```

- **Problem**: There is no static guarantee you can use the right accessor functions. You might hit an Error case if you are not careful. Many of the functions throw exceptions when they are called, which means you don’t have a compile-time guarantee that your program will be executed without exceptions being thrown
- **Problem**: If you want to add new expression forms, such as product or a variable, you need to add methods for classification and access to all classes defined above.

Non-Solution: Type Tests and Type Casts

- A “hacky” solution could use types tests and type casts
- Scala lets you do these using methods defined in class Any:
    - def isInstanceOf[T]: Boolean -> checks whether this object’s type conforms to T
    - 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 casts and tests
- But their use in Scala is discouraged, because there are better alternatives

Solution 1: Object-Oriented Decomposition

- For example, suppose that all you want to do is evaluate expressions. You could then define:

```
trait Expression:
  def evaluate: Int

class Number(n: Int) extends Expression:
  def evaluate: Int = n

class Sum(e1: Expression, e2: Expression) extends Expression:
  def evaluate: Int = evaluate(e1) + evaluate(e2)
```

- But if you want to display the expression, you have to define new methods in all subclasses

Assessment of Object-Oriented Decomposition

- Object-oriented 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(arises from mixing several things together) and adds new dependencies to classes
- It makes it easy to add new kinds of data but hard to add new kinds of operations

Limitations of Object-Oriented Decomposition

- Object-oriented decomposition only works well if operations are on 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.
- So you are back to square one; you need test and access methods for all different subclasses.