data & behavior

Algorithms + Data Structures = Programs

NIKLAUS WIRTH

lucid, systematic, and penetrating treatment of basic and dynamic data structures, sorting, recursive algorithms, language structures, and compiling

PRENTICE-HALL
SERIES IN AUTOMATIC COMPUTATION
Is this a DSL?
The Expression Problem

There are multiple formulations. This is one of them.

• \( \exists \) a library \( L \) of nouns (data) and verbs (behaviors)
• Person A extends \( L \) to add a new noun (data)
• Person B extends \( L \) to add a new verb (behavior)
• Person C wants to \textit{safely} combine A and B’s extensions

\textit{User-defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction}, John Reynolds, 1975


\textit{The Expression Problem}, Philip Wadler, 1998

\textit{Independently Extensible Solutions to the Expression Problem}, Matthias Zenger & Martin Odersky, 2005

\textit{Data types à la carte}, Wouter Swierstra, 2008
CS 111 Spring ’16: The Playing Wildebeests
DSL takeaways

1. We can use traits to mix in syntax!
import org.scalatest.FunSuite
import scala.collection.mutable.Stack

class ExampleFunSuite extends FunSuite {

    test("pop is invoked on a non-empty stack") {

        val stack = new Stack[Int]
        stack.push(1)
        stack.push(2)
        val oldSize = stack.size
        val result = stack.pop()
        assert(result === 2)
        assert(stack.size === oldSize - 1)
    }
}
import org.scalatest.FunSpec
import scala.collection.mutable.Stack

class ExampleFunSpec extends FunSpec {

describe("A Stack") {

  it("should pop values in last-in-first-out order") {
    val stack = new Stack[Int]
    stack.push(1)
    stack.push(2)
    assert(stack.pop() === 2)
    assert(stack.pop() === 1)
  }

}
}
Dialect: functional testing

ScalaTest = Traits for DSLs

```scala
import org.scalatest.FeatureSpec
import org.scalatest.GivenWhenThen
import scala.collection.mutable.Stack

class ExampleFeatureSpec extends FeatureSpec with GivenWhenThen {

  feature("The user can pop an element off the top of the stack") {
    info("As a programmer")
    info("I want to be able to pop items off the stack")
    info("So that I can get them in last-in-first-out order")

    scenario("pop is invoked on a non-empty stack") {
      given("a non-empty stack")
      val stack = new Stack[Int]
      stack.push(1)
      stack.push(2)
      val oldSize = stack.size

      when("when pop is invoked on the stack")
      val result = stack.pop()

      then("the most recently pushed element should be returned")
      assert(result === 2)
      and("the stack should have one less item than before")
      assert(stack.size === oldSize - 1)
    }
  }
}
```

examples taken from scalatest.org
DSL takeaways

1. We can use traits to mix in syntax!

2. We prefer case classes over classes.