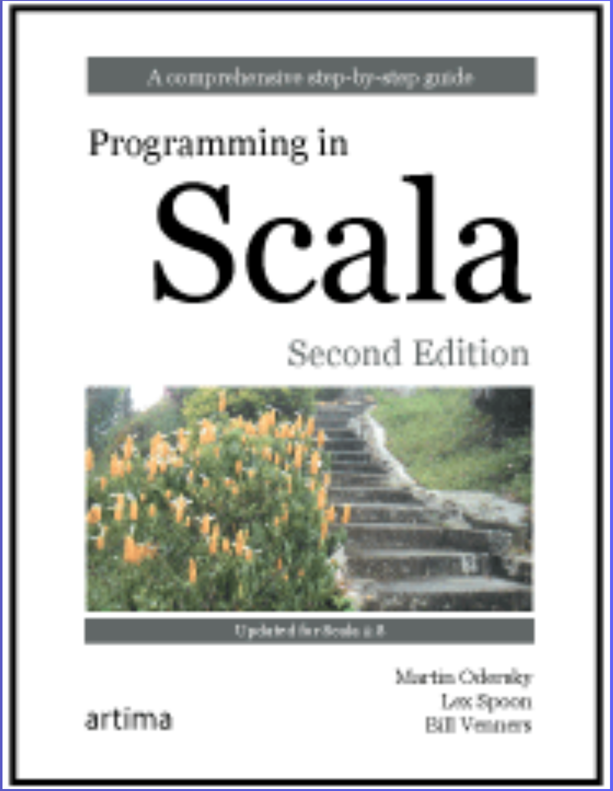
artima

Stairway to Scala - Flight 11

Assertions and Unit Testing

Bill Venners
Dick Wall

www.artima.com



Flight 11 goal

Look at Scala's assertions mechanism and testing tools. Discuss Design-by-Contract, TDD, BDD, and other techniques for quality.

Assertions

```
def above(that: Element): Element = {
  val this1 = this widen that.width
  val that1 = that widen this.width
  assert(this1.width == that1.width)
  elem(this1.contents ++ that1.contents)
}
```

Design by contract

- Preconditions
- Postconditions
- Invariants

Checking preconditions

```
class Rational(n: Int, d: Int) {
  require(d != 0)
  // ...
```

- Can also use "assume".

Ensuring postconditions

```
private def widen(w: Int): Element =
  if (w <= width)
    this
  else {
    val left = elem(' ', (w - width) / 2, height)
    var right = elem(' ', w - width - left.width, height)
    left beside this beside right
  } ensuring (w <= _.width)
```

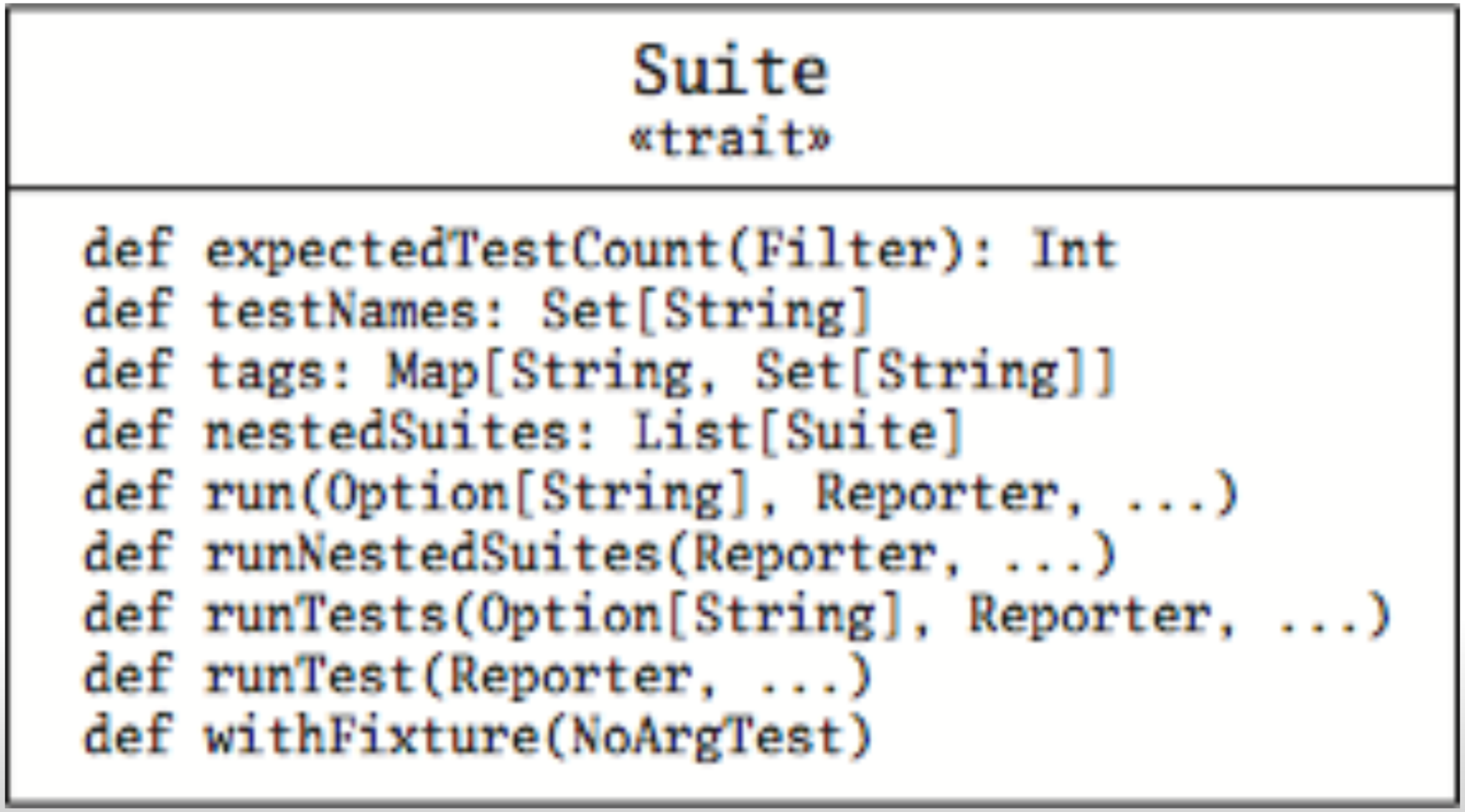
Scala testing tools

- JUnit
- TestNG
- specs
- ScalaTest
- ScalaCheck

specs - behavior driven design

```
import org.specs._

object ElementSpecification extends Specification {
 "A UniformElement" should {
  "have a width equal to the passed value" in {
    val ele = elem('x', 2, 3)
    ele.width must be_==(2)
  }
  "have a height equal to the passed value" in {
    val ele = elem('x', 2, 3)
    ele.height must be_==(3)
  }
  "throw an IAE if passed a negative width" in {
    elem('x', -2, 3) must
      throwA[IllegalArgumentException]
  }
 }
}
```

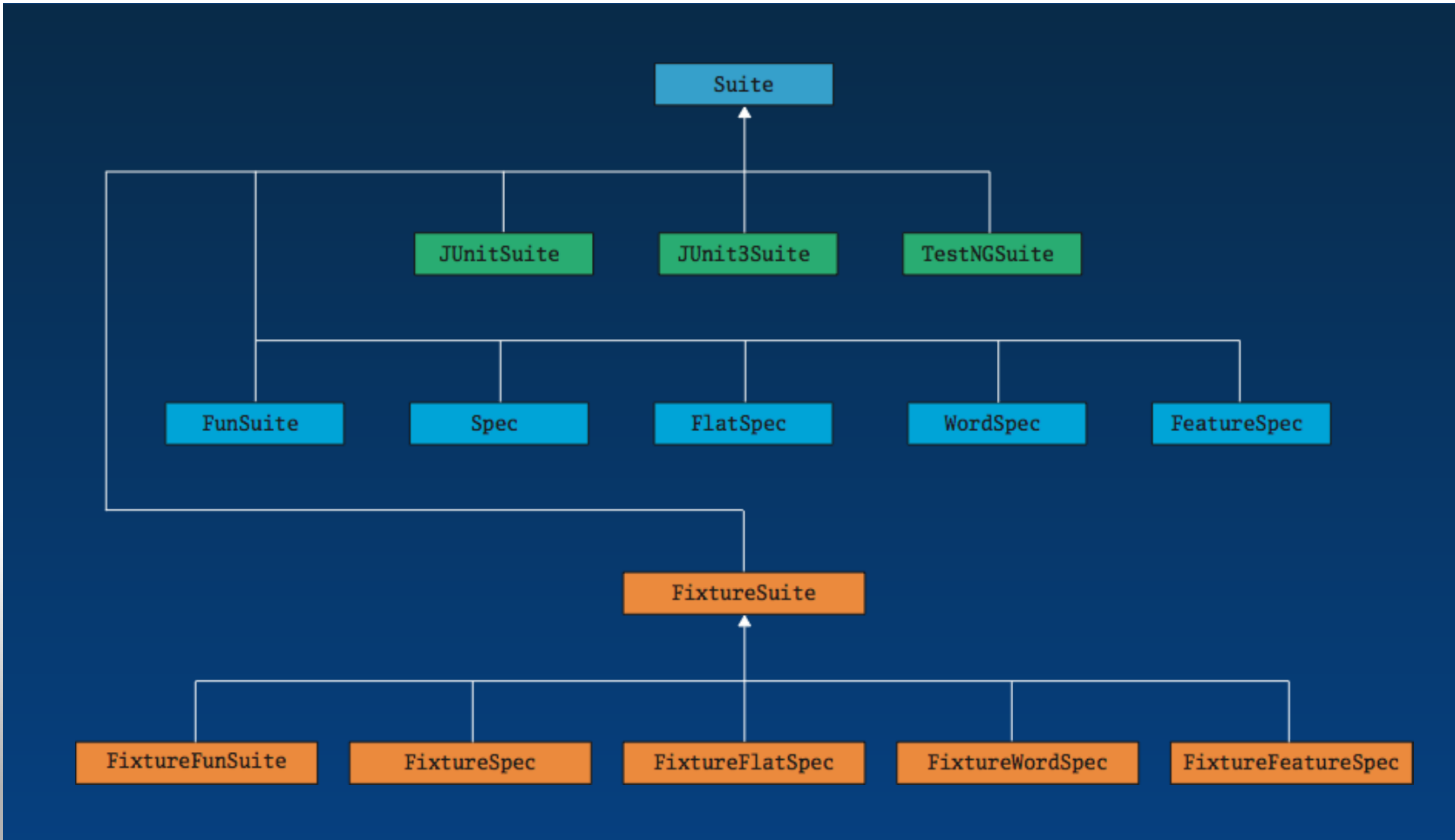
ScalaTest is customizable



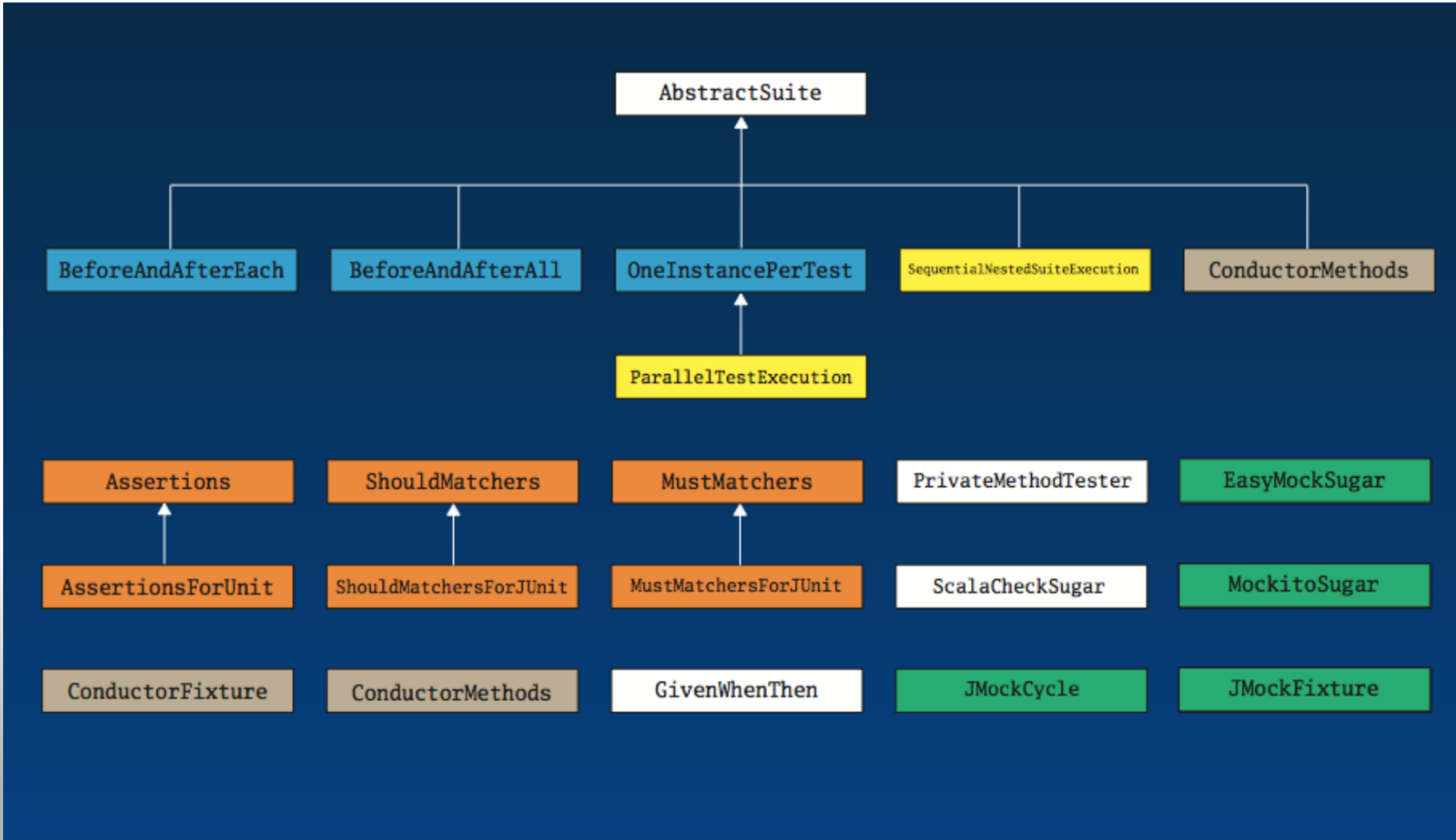
```
Suite
«trait»

def expectedTestCount(Filter): Int
def testNames: Set[String]
def tags: Map[String, Set[String]]
def nestedSuites: List[Suite]
def run(Option[String], Reporter, ...)
def runNestedSuites(Reporter, ...)
def runTests(Option[String], Reporter, ...)
def runTest(Reporter, ...)
def withFixture(NoArgTest)
```

ScalaTest is like self-serve frozen yogurt

1. Pick a core Suite trait



2. Mix in some other traits



3. Enjoy

```
class MySuite extends ProjectSuite
    with UserBobInDatabase {

  test("Bob posts to his blog") {
    // ...
  }
}
```

FunSuite

```
import org.scalatest.FunSuite

class ExampleSuite extends FunSuite {

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

  test("pop is invoked on an empty stack") (pending)
}
```

FunSuite filled in

```
import org.scalatest.FunSuite
import scala.collection.mutable.Stack

class ExampleSuite 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)
  }

  test("pop is invoked on an empty stack") {

    val emptyStack = new Stack[String]
    intercept[NoSuchElementException] {
      emptyStack.pop()
    }
    assert(emptyStack.isEmpty)
  }
}
```

FunSuite with ShouldMatchers

```
import org.scalatest.FunSuite
import org.scalatest.matchers.ShouldMatchers
import scala.collection.mutable.Stack

class ExampleSuite extends FunSuite with ShouldMatchers {

  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()
    result should equal (2)
    stack.size should equal (oldSize - 1)
  }

  test("pop is invoked on an empty stack") {

    val emptyStack = new Stack[String]
    evaluating { emptyStack.pop() } should produce [NoSuchElementException]
    emptyStack should be ('empty)
  }
}
```

ScalaTest matchers

```
map should contain key ('a')

collection should contain (1.0)

collection should have size (17)

collection should be ('empty)

string must equal ("done")

array must have length (9)
```

Test-driven development (TDD)

- write a test
- run it to see red
- implement code
- run to see green
- refactor and repeat

Behavior-driven development (BDD)

- not tests, *specifications*
- use the right words
- spec-like output

Spec

```
import org.scalatest.Spec
import org.scalatest.matchers.ShouldMatchers
import scala.collection.mutable.Stack

class StackSpec extends Spec with ShouldMatchers {

  describe("A Stack") {

    describe("(when empty)") {

      val stack = new Stack[Int]

      it("should be empty") {
        stack should be ('empty)
      }

      it("should complain when popped") {
        evaluating { stack.pop() } should produce [NoSuchElementException]
      }
    }
  }
}
```

Spec-like output

```
ExampleSpec:
A Stack (when empty)
- should be empty
- should complain when popped
```

WordSpec

```
import org.scalatest.WordSpec
import org.scalatest.matchers.ShouldMatchers
import scala.collection.mutable.Stack

class StackSpec extends WordSpec with ShouldMatchers {

  "A Stack" when {

    "empty" should {

      val stack = new Stack[Int]

      "be empty" in {
        stack should be ('empty)
      }

      "complain when popped" in {
        evaluating { stack.pop() } should produce [NoSuchElementException]
      }
    }
  }
}
```

FlatSpec

```
import org.scalatest.FlatSpec
import org.scalatest.matchers.ShouldMatchers
import scala.collection.mutable.Stack

class StackSpec extends FlatSpec with ShouldMatchers {

  val stack = new Stack[Int]

  "A Stack (when empty)" should "be empty" in {
    stack should be ('empty)
  }

  it should "complain when popped" in {
    evaluating { stack.pop() } should produce [NoSuchElementException]
  }
}
```

FeatureSpec

```
import org.scalatest.FeatureSpec
import org.scalatest.GivenWhenThen

class ExampleSpec 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")
      when("when pop is invoked on the stack")
      then("the most recently pushed element should be returned")
      and("the stack should have one less item than before")
      pending
    }

    scenario("pop is invoked on an empty stack") {

      given("an empty stack")
      when("when pop is invoked on the stack")
      then("NoSuchElementException should be thrown")
      and("the stack should still be empty")
      pending
    }
  }
```

FeatureSpec output

```
ExampleSpec:
Feature: The user can pop an element off the top of the stack
  As a programmer
  I want to be able to pop items off the stack
  So that I can get them in last-in-first-out order
  Scenario: pop is invoked on a non-empty stack (pending)
    Given a non-empty stack
    When when pop is invoked on the stack
    Then the most recently pushed element should be returned
    And the stack should have one less item than before
  Scenario: pop is invoked on an empty stack (pending)
    Given an empty stack
    When when pop is invoked on the stack
    Then NoSuchElementException should be thrown
    And the stack should still be empty
```