# Note On Developing

**Objects**: Analysis (IDF, ModelFunctions), Classify, Token(Stopwords), DocGathering

**Help libraries custom:**

**Classes:** IDFValue, Dictionary, DocumentVector, NormalisedVector, TokenizedText <= Tokenizer, DocumentFinder

**Workflow:** One or more directories can be entered, and they can be explored *recursively* to gather every document which has mime type of the like *docx, doc, txt, pdf (work in progress), md (work in progress (my need a library and to exclude hash symbol).* Sub-directories set to be excluded are excluded in the process (helper method *recursiveTreeDocsSearch* to gather all docs in a folder).

**Attempt OOP**

Creation of classes representing vectors, depending on the model wanted they would implement different traits, visibly divided such tf, idf or bag of words.

They contain functions that model the vector upon certain criteria. They always ultimately return (1st iteration of code) a map value Double assign to an index String(Token), which is respectively the weight of the term (tf: importance based on frequency, bag-of-words: frequency) and the term itself. Idf are values brought from outside (they are calculated separately using all documents being analysed as parameters for construction) although they have the same type. (2nd iteration of the code) a tuple consisting of the term and the weight. They will be converted to a map on a second moment.

In the second attempt there would be only one vector class which would be fed into a NormalisedVector class to normalised its length in relation to all other documents, letting normalisation easing up the process of comparison. This normalised vector would be then specialised over specific type depending on the model sought, which would be achieved with subtypes implementing different functionalities (different traits).

**Attempt Functional**

As second OOP attempt, removing any subtype and let the modelling being fed into the NormalisedVector as a function. This could be any process as long as from a tokenised document and a term would produce a tuple of a term and a decimal value.

This could give more flexibility over the model that can be achieved, and the user/future developer could feed in any function with those criteria to achieve the wanted modelled vector.

The vector would still be an entity per-se to group within it more information such the size of it, and the path to the original document.

## Clusterisation

## Test

**Test styles from http://www.scalatest.org/user\_guide/selecting\_a\_style**

# Selecting testing styles for your project

ScalaTest supports different styles of testing, each designed to address a particular set of needs. To help you find the best-fit styles for your project, this page will describe the intended use cases for each option.

We recommend you choose a set of testing styles for each project, then encourage everyone working on the project use the chosen styles. This allows the testing styles to fit the team while maintaining uniformity in the project code base. We recommend you select one main style for unit testing and another for acceptance testing. Using a different style for unit and acceptance testing can help developers "switch gears" between low-level unit testing to high-level acceptance testing. You may also want to select particular styles to be used in special situations, such as using [PropSpec for test matrixes](http://doc.scalatest.org/3.0.1/" \l "org.scalatest.PropSpec@testMatrix). We usually write integration tests—tests that involve subsystems such as a database—in the same style as the unit tests.

In short, ScalaTest's flexibility is not intended to enable individual developers to use different testing styles on the same project. Rather, it is intended to enable project leaders to select a best-fit style or styles for the team. If you have trouble enforcing chosen styles, you can [specify the chosen styles](http://doc.scalatest.org/3.0.1/#org.scalatest.tools.Runner$@specifyingChosenStyles) in your build.

The style you choose dictates only how the declarations of your tests will look. Everything else in ScalaTest—assertions, matchers, mixin traits, etc.—works consistently the same way no matter what style you chose.

## If you don't enjoy shopping

If you would rather be told which approach to take rather than pick one yourself, we recommend you use [FlatSpec](http://doc.scalatest.org/3.0.1/" \l "org.scalatest.FlatSpec) for unit and integration testing and [FeatureSpec](http://doc.scalatest.org/3.0.1/" \l "org.scalatest.FeatureSpec) for acceptance testing. We recommend FlatSpec as the default choice, because it is flat (unnested) like the XUnit tests familiar to most developers, but guides you into writing focused tests with descriptive, specification-style names.

## Style trait use cases

If you would rather make your own choices, this table gives a quick overview of the advantages and disadvantages of each style trait. For more information and examples, click on the links:

|  |
| --- |
| **Style Trait Descriptions and Examples** |
| FunSuite For teams coming from xUnit, [FunSuite](http://doc.scalatest.org/3.0.1/" \l "org.scalatest.FunSuite) feels comfortable and familiar while still giving some of the benefits of BDD: FunSuite makes it easy to write descriptive test names, natural to write focused tests, and generates specification-like output that can facilitate communication among stakeholders.  import org.scalatest.FunSuite  class SetSuite extends FunSuite {  test("An empty Set should have size 0") {  assert(Set.empty.size == 0)  }  test("Invoking head on an empty Set should produce NoSuchElementException") {  assertThrows[NoSuchElementException] {  Set.empty.head  }  }  } |
| FlatSpec A good first step for teams wishing to move from xUnit to BDD, [FlatSpec](http://doc.scalatest.org/3.0.1/" \l "org.scalatest.FlatSpec)'s structure is flat like xUnit, so simple and familiar, but the test names must be written in a specification style: "X should Y," "A must B," etc.  import org.scalatest.FlatSpec  class SetSpec extends FlatSpec {  "An empty Set" should "have size 0" in {  assert(Set.empty.size == 0)  }  it should "produce NoSuchElementException when head is invoked" in {  assertThrows[NoSuchElementException] {  Set.empty.head  }  }  } |
| FunSpec For teams coming from Ruby's RSpec tool, [FunSpec](http://doc.scalatest.org/3.0.1/" \l "org.scalatest.FunSpec) will feel very familiar; More generally, for any team that prefers BDD, FunSpec's nesting and gentle guide to structuring text (with describeand it) provides an excellent general-purpose choice for writing specification-style tests.  import org.scalatest.FunSpec  class SetSpec extends FunSpec {  describe("A Set") {  describe("when empty") {  it("should have size 0") {  assert(Set.empty.size == 0)  }  it("should produce NoSuchElementException when head is invoked") {  assertThrows[NoSuchElementException] {  Set.empty.head  }  }  }  }  } |
| WordSpec For teams coming from specs or specs2, [WordSpec](http://doc.scalatest.org/3.0.1/" \l "org.scalatest.WordSpec) will feel familiar, and is often the most natural way to port specsN tests to ScalaTest. WordSpec is very prescriptive in how text must be written, so a good fit for teams who want a high degree of discipline enforced upon their specification text.  import org.scalatest.WordSpec  class SetSpec extends WordSpec {  "A Set" when {  "empty" should {  "have size 0" in {  assert(Set.empty.size == 0)  }  "produce NoSuchElementException when head is invoked" in {  assertThrows[NoSuchElementException] {  Set.empty.head  }  }  }  }  } |
| FreeSpec Because it gives absolute freedom (and no guidance) on how specification text should be written, [FreeSpec](http://doc.scalatest.org/3.0.1/" \l "org.scalatest.FreeSpec) is a good choice for teams experienced with BDD and able to agree on how to structure the specification text.  import org.scalatest.FreeSpec  class SetSpec extends FreeSpec {  "A Set" - {  "when empty" - {  "should have size 0" in {  assert(Set.empty.size == 0)  }  "should produce NoSuchElementException when head is invoked" in {  assertThrows[NoSuchElementException] {  Set.empty.head  }  }  }  }  } |
| PropSpec [PropSpec](http://doc.scalatest.org/3.0.1/#org.scalatest.PropSpec) is perfect for teams that want to write tests exclusively in terms of property checks; also a good choice for writing the occasional test matrix when a different style trait is chosen as the main unit testing style.  import org.scalatest.\_  import prop.\_  import scala.collection.immutable.\_  class SetSpec extends PropSpec with TableDrivenPropertyChecks with Matchers {  val examples =  Table(  "set",  BitSet.empty,  HashSet.empty[Int],  TreeSet.empty[Int]  )  property("an empty Set should have size 0") {  forAll(examples) { set =>  set.size should be (0)  }  }  property("invoking head on an empty set should produce NoSuchElementException") {  forAll(examples) { set =>  a [NoSuchElementException] should be thrownBy { set.head }  }  }  } |
| FeatureSpec Trait [FeatureSpec](http://doc.scalatest.org/3.0.1/" \l "org.scalatest.FeatureSpec) is primarily intended for acceptance testing, including facilitating the process of programmers working alongside non-programmers to define the acceptance requirements.  import org.scalatest.\_  class TVSet {  private var on: Boolean = false  def isOn: Boolean = on  def pressPowerButton() {  on = !on  }  }  class TVSetSpec extends FeatureSpec with GivenWhenThen {  info("As a TV set owner")  info("I want to be able to turn the TV on and off")  info("So I can watch TV when I want")  info("And save energy when I'm not watching TV")  feature("TV power button") {  scenario("User presses power button when TV is off") {  Given("a TV set that is switched off")  val tv = new TVSet  assert(!tv.isOn)  When("the power button is pressed")  tv.pressPowerButton()  Then("the TV should switch on")  assert(tv.isOn)  }  scenario("User presses power button when TV is on") {  Given("a TV set that is switched on")  val tv = new TVSet  tv.pressPowerButton()  assert(tv.isOn)  When("the power button is pressed")  tv.pressPowerButton()  Then("the TV should switch off")  assert(!tv.isOn)  }  }  } |
| RefSpec (JVM only) [RefSpec](http://doc.scalatest.org/3.0.1/#org.scalatest.refspec.RefSpec) allows you to define tests as methods, which saves one function literal per test compared to style classes that represent tests as functions. Fewer function literals translates into faster compile times and fewer generated class files, which can help minimize build times. As a result, using Spec can be a good choice in large projects where build times are a concern as well as when generating large numbers of tests programatically via static code generators.  import org.scalatest.refspec.RefSpec  class SetSpec extends RefSpec {  object `A Set` {  object `when empty` {  def `should have size 0` {  assert(Set.empty.size == 0)  }  def `should produce NoSuchElementException when head is invoked` {  assertThrows[NoSuchElementException] {  Set.empty.head  }  }  }  }  }  Note: The "Ref" in RefSpec stands for reflection, which RefSpec uses to discover tests. As reflection is not availble in Scala.js, this class is not available on Scala.js. |