A comparative study of the various styles of writing test-cases

[FunSuite](http://doc.scalatest.org/2.2.4/#org.scalatest.FunSuite)

For teams coming from xUnit, [FunSuite](http://doc.scalatest.org/2.2.4/" \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.

[FlatSpec](http://doc.scalatest.org/2.2.4/" \l "org.scalatest.FlatSpec)

A good first step for teams wishing to move from xUnit to BDD, [FlatSpec](http://doc.scalatest.org/2.2.4/" \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.

[FunSpec](http://doc.scalatest.org/2.2.4/" \l "org.scalatest.FunSpec)

For teams coming from Ruby's RSpec tool, [FunSpec](http://doc.scalatest.org/2.2.4/" \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 (withdescribe and it) provides an excellent general-purpose choice for writing specification-style tests.

[WordSpec](http://doc.scalatest.org/2.2.4/#org.scalatest.WordSpec)

For teams coming from specs or specs2, [WordSpec](http://doc.scalatest.org/2.2.4/" \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.

[FreeSpec](http://doc.scalatest.org/2.2.4/" \l "org.scalatest.FreeSpec)

Because it gives absolute freedom (and no guidance) on how specification text should be written, [FreeSpec](http://doc.scalatest.org/2.2.4/" \l "org.scalatest.FreeSpec) is a good choice for teams experienced with BDD and able to agree on how to structure the specification text.

[Spec](http://doc.scalatest.org/2.2.4/#org.scalatest.Spec)

[Spec](http://doc.scalatest.org/2.2.4/#org.scalatest.Spec) 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.

[PropSpec](http://doc.scalatest.org/2.2.4/#org.scalatest.PropSpec)

[PropSpec](http://doc.scalatest.org/2.2.4/#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.

[FeatureSpec](http://doc.scalatest.org/2.2.4/#org.scalatest.FeatureSpec)

Trait [FeatureSpec](http://doc.scalatest.org/2.2.4/" \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.

***Note: I have executed each of these styles and executed them successfully following sbt->clean->compile->test.***

***The screen-shots showing the output are attached in the mail.***