easyb User Guide (this is a work in progress!)

Publish Date 04/09/2010

Version 0.9.7

Author easyb Development Team

Table of Contents

1. Overview 1
1.1. History 1
1.2. Goals 1
2. Intro to BDD 2
3. BDD the easyb way 4
4. Usage 8
4.1. ...Ways to run it 8
4.2. Reporting 8
5. Syntax 9
5.1. Stories 9
5.1.1. scenario 9
5.1.2. given, andGiven 9
5.1.3. when 9
5.1.4. then, andThen 10
5.1.5. narrative 10
5.1.6. as_a 10
5.1.7. i_want 10
5.1.8. so_that 10
5.2. Specifications 11
5.2.1. it 11
5.3. Common 11
5.3.1. and (syntax replacement) 11
5.3.2. and (assertion chain) 11
5.3.3. before 12
5.3.4. before_each 12
5.3.5. after 12
5.3.6. after_each 12
5.4. Assertions 13
5.4.1. should 13
5.4.1.1. shouldBe 13
5.4.1.2. shouldEqual 13
5.4.1.3. shouldBeEqual 13
5.4.1.4. shouldBeEqualTo 13
5.4.1.5. shouldNotBe 13
5.4.1.6. shouldNotEqual 14
5.4.1.7. shouldNotBeEqual 14
5.4.1.8. shouldNotBeEqualTo 14
5.4.1.9. shouldBeA 14
5.4.1.10. shouldBeA 14
5.4.1.11. shouldBeAn 14
5.4.1.12. shouldNotBeA 14
5.4.1.13. shouldNotBeAn 15
5.4.1.14. shouldBeGreaterThan 15
5.4.1.15. shouldBeLessThan 15
5.4.1.16. shouldHave 15
5.4.1.17. shouldNotHave 15
5.4.2. ensure 15
5.4.2.1. isNull 16

5.4.2.2. isNotNull 16
5.4.2.3. isA<class type> 16
5.4.2.4. isEqualTo(value) 16
5.4.2.5. isEqualTo<value> 16
5.4.2.6. isNotEqualTo(value) 16
5.4.2.7. isNotEqualTo<value> 16
5.4.2.8. isTrue 17
5.4.2.9. isFalse 17
5.4.2.10. ensureThrows 17
5.4.3. common 17
5.4.3.1. has 17
5.4.3.2. contains 17
5.4.3.3. startsWith 18
5.4.3.4. endsWith 18
6. Appendix 19
6.1. Glossary 19
Index 20

Chapter 1. Overview

[Please note, this is an unfinished document and is a work in progress. Feedback and/or help is most welcome!]

easyb is a behavior driven development framework for the Java platform. By using a Domain Specific Language, easyb aims to enable executable, yet readable documentation.

1.1. History

Behavior driven development (or BDD) isn't anything new or revolutionary-- it's just an evolutionary offshoot of test driven development, in which the word test is replaced by the word should. Semantics aside, a lot of people have found that the concept of should is a much more natural development driver than the concept of testing. In fact, when you think in terms of behavior (i.e. shoulds) you'll find that writing specifications is easier to do first, which is the intent of test driven development in the first place.

What came next was a natural progression from idea to implementation. The idea of TDD had become widely accepted, primarily because of its freely available tooling in the form of JUnit. BDD is following the same path and pulling it to the forefront are a new crop of tools to support it. One of the first was JBehave by Dan North. Dan is regarded as one of the fathers of BDD thru his role in defining it and evangelizing those concepts. JBehave implemented many of the ideas and terms of BDD and did so in code which is the real reference for most developers.

There was still a significant problem. BDD by its nature is a very Stakeholder centric process and trying to marry up Non-Technical folks with Java syntax is just a recipe for failure or frustration at best. Around the same time, ruby had taken off thanks to Rails and as such DomainSpecificLanguages were becoming a hotbed of discussion among developers. It was only a matter of time before all of the above met at the crossroads. RSpec was first on the scene and provided an early form of BDD that used 'it should' as its primary means of describing behavior. It was a fantastic spark to the community and most assuredly a step in the right direction but it too had its drawbacks. RSpec ran on the Ruby VM and this left the Java world out in the cold. Running RSpec via JRuby was possible but JRuby was a fledgeling project and also came with its own baggage, not the least of which was that Enterprise organizations weren't ready to embrace Ruby or even JRuby.

Enter groovy and easyb. Groovy runs natively on the JVM with nothing more than a jar file in the classpath. This is key to lowering the barriers of adoption by that Enterprise segment of the population. What Groovy also provides is a dynamic language with all the power and syntatic sugar opportunities for creating a rich DSL. With all the pieces now available, easyb was primed to step in and put them all in place.

1.2. Goals

- Easy
- Natual Language DSL for capturing Behaviors
- Bridge gap between Stakeholders and Development

Easyb accomplishes all of these goals in spades as you'll see throughout this document. Goal #1 has always been, and continues to be, that it must be easy. To do so, easyb provides out of the box a slew of great features and the easyb team has bundled along with (and built on top of them) a suite of products and plugins.

Chapter 2. Intro to BDD

Test Driven Development has proved itself useful for a number of reasons. Improving code quality up front, documenting the intentions of the code its testing (to a small extent), and even assure that bugs are not reintroduced during refactoring. It does not, however, work well to capture the requirements for which the application code is supposed to be implementing. What often happens is that a separate set of requirements are created and maintained. Even worse is that the interpretation of those requirements is most likely different between those that created them and those that are writing code to implement them. This can be directly linked to cost of a project because work has to be redone to make the code reflect what the requirements creators had in mind.

Behavior Driven Development takes the approach that Stakeholders and Developers should be on the same page when it comes to interpretation of requirements. It is accomplished by changing the verbiage used when defining requirements. This is a key factor because Stakeholders are most often not technical, however, the developers that they employ to implement their requirements primarily know how to communicate in very technical terms. So a common pattern of capturing requirements was created. It is very specific about capturing exactly the context, events and outcomes that make up a requirement, or in BDD terms, a scenario. The scenario is captured in this form:

```
given an initial context
when an event occurs
then ensure the outcome
```

Writing scenarios in this fashion removes the ambiguity and allows both sides to focus on properly capturing, and implementing, the requirements, knowing they are on the same page.

Lets look at an example requirement and how it fits into the BDD model. The ubiquitous Bank Account example should do just fine.

```
given "an account with 100 dollars"
when "account holder withdrawls 20 dollars"
then "account should have 80 dollars left"
```

This is clearly capturing the requirement, and not a single line of code was written. What was just defined was a scenario consisting of the context, events and expected outcomes. Since we now have a complete scenario, we'll go ahead and explicitly state that as well.

```
scenario "money is withdrawn from an account containing sufficient funds"
  given "an account with 100 dollars"
  when "account holder withdrawls 20 dollars"
  then "account should have 80 dollars left"
```

Multiple scenarios that are members of a logical theme can be grouped together in a story.

```
AccountWithdrawlStory

scenario "money is withdrawn from an account containing sufficient funds"
  given "an account with 100 dollars"
  when "account holder withdrawls 20 dollars"
  then "account should have 80 dollars left"

scenario "money is withdrawn from an account with insufficient funds"
  given "an account with 10 dollars"
  when "account holder withdrawls 20 dollars"
  then "account holder should be notified about the insufficient funds"
  and "then account should still have 10 dollars"
```

Additionally, further clarity around exactly what viewpoint this is being written from can be added via a narrat-

ive.

```
AccountWithdrawlStory

narrative "scenarios written from the viewpoint of bank"
  as a "bank"
  i want "to verify that withdrawls work on accounts with sufficient and insufficient funds"
  so that "i do not lose money or confuse the customers"


scenario "money is withdrawn from an account containing sufficient funds"
  given "an account with 100 dollars"
  when "account holder withdrawls 20 dollars"
  then "account should have 80 dollars left"

scenario "money is withdrawn from an account with insufficient funds"
  given "an account with 10 dollars"
  when "account holder withdrawls 20 dollars"
  then "account holder should be notified about the insufficient funds"
  and "then account should still have 10 dollars"
```

What we have accomplished here is to establish a conversation between stakeholder and developer in common terms. Effectively we are capturing the behavior of a system and from that we can derive our specific development tasks.

Chapter 3. BDD the easyb way

"Ok, great. Looks like just another way for the business to bury me in more requirements documents.", you say. What if we can actually make that documentation executable. Thats right, take it almost verbatim and run it. Easyb allows exactly that. First we'll focus on just one of the bank account scenarios and sprinkle in some very minor syntax additions.

```
scenario "money is withdrawn from an account containing sufficient funds", {
  given "an account with 100 dollars"
  when "account holder withdrawls 20 dollars"
  then "account should have 80 dollars left"
}
```

That should do it. A comma and two braces. That wasn't too painful was it? Now lets run it. Save that text into a file named AccountWithdrawl.story and then tell easyb where to find it.

```
java -cp easyb-0.9.3.jar:commons-cli-1.1.jar:groovy-1.5.4.jar \
>  org.disco.easyb.BehaviorRunner \
>  ./behavior/groovy/org/disco/bdd/story/account/AccountWithdrawl.story
```

It worked and we even get some feedback to boot. It tells us that we have run the Account Withdrawl story which contains one scenario, none of which failed and one is pending. Pending? Interesting. Out of the box if you don't provide any actual code to verify objects and behaviors the scenario is marked as pending. This is an undeniably useful way to bridge the gap between a scenario the Stakeholder hands you and one that you've since had time and domain understanding to write the code to verify the behavior.

```
Running account withdrawl story (AccountWithdrawl.story)
Scenarios run: 1, Failures: 0, Pending: 1, Time Elapsed: 0.453 sec

1 behavior run (including 1 pending behavior) with no failures
```

We shouldn't forget about the other scenario given to us, so we'll add it to the same story now.

```
scenario "money is withdrawn from an account containing sufficient funds", {
  given "an account with 100 dollars"
  when "account holder withdrawls 20 dollars"
  then "account should have 80 dollars left"
}

scenario "money is withdrawn from an account with insufficient funds", {
  given "an account with 10 dollars"
  when "account holder withdrawls 20 dollars"
  then "account holder should be notified about the insufficient funds"
  and "then account should still have 10 dollars"
}
```

Run it again and we see the same story now has two behaviors, no failures and 2 pending.

```
Running account withdrawl story (AccountWithdrawl.story)
Scenarios run: 2, Failures: 0, Pending: 2, Time Elapsed: 0.495 sec

2 total behaviors run (including 2 pending behaviors) with no failures
```

Enough hocus pocus, on to actually writing some code. We are given a disambiguated set of requirements and it is now far easier for Developers to think in terms of 'should' instead of 'test'. Its just more natural as a human (which developers have sometimes been accused of being) to think in terms of 'it should do xyz'. Again we'll focus on a single scenario, so let's start with the sufficient funds behavior.

```
scenario "money is withdrawn from an account containing sufficient funds", {
```

```
  given "an account with 100 dollars", {
    account = new Account(100);
  }

  when "account holder withdrawls 20 dollars", {
    account.withdrawl 20
  }

  then "account should have 80 dollars left", {
    account.balance.shouldBe 80
  }
}

...
```

Still human readable, yet fully functional validation of behavior. What are we waiting for, run it!

```
Running account withdrawl story (AccountWithdrawl.story)
There was an error running your easyb story or specification
org.codehaus.groovy.control.MultipleCompilationErrorsException: startup failed, Script1.groovy: \
        25: unable to resolve class Account
 @ line 25, column 15.
1 error

    at org.codehaus.groovy.control.ErrorCollector.failIfErrors(ErrorCollector.java:296)
    at org.codehaus.groovy.control.CompilationUnit.applyToSourceUnits(CompilationUnit.java:787)
    at org.codehaus.groovy.control.CompilationUnit.compile(CompilationUnit.java:438)
    at groovy.lang.GroovyClassLoader.parseClass(GroovyClassLoader.java:277)
    at groovy.lang.GroovyShell.parseClass(GroovyShell.java:572)
    at groovy.lang.GroovyShell.parse(GroovyShell.java:584)
    at groovy.lang.GroovyShell.parse(GroovyShell.java:564)
    at groovy.lang.GroovyShell.evaluate(GroovyShell.java:542)
    at groovy.lang.GroovyShell.evaluate(GroovyShell.java:518)
    at org.disco.easyb.domain.Story.execute(Story.java:34)
    at org.disco.easyb.BehaviorRunner.runBehavior(BehaviorRunner.java:78)
    at org.disco.easyb.BehaviorRunner.main(BehaviorRunner.java:59)
```

Uh-oh. We forgot to add the Account class itself. Create an Account class with a balance member variable and a withdrawl method. It can be in pure Java, or groovy, we'll stick with Java for now.

```
package org.disco.bdd.account;

public class Account {
    private int balance;

    public Account(int initialBalance) {
        balance = initialBalance;
    }

    public int getBalance() {
        return balance;
    }

    public void withdrawl(int withdrawlAmount) {
        balance = balance - withdrawlAmount;
    }
}
```

Import that into the story.

```
import org.disco.bdd.account.Account

scenario "money is withdrawn from an account containing sufficient funds", {
  given "an account with 100 dollars", {
    account = new Account(100);
  }
```

```
  when "account holder withdrawls 20 dollars", {
    account.withdrawl 20
  }

  then "account should have 80 dollars left", {
    account.balance.shouldBe 80
  }
}

...
```

Add the location of the Account.class to the classpath and kick it off again.

```
java -cp easyb-0.9.3.jar:commons-cli-1.1.jar:groovy-1.5.4.jar:target/behavior-classes \
> org.disco.easyb.BehaviorRunner \
> ./behavior/groovy/org/disco/bdd/story/account/AccountWithdrawl.story

Running account withdrawl story (AccountWithdrawl.story)
Scenarios run: 2, Failures: 0, Pending: 1, Time Elapsed: 0.538 sec

2 total behaviors run (including 1 pending behavior) with no failures
```

Only one behavior is pending now. Lets review what we did in the snippet above. We created an instance of an account, with a balance of 100 dollars, because that is what the 'given' told us we should do. Twenty dollars was then withdrawn because that is what the 'when' told us we should be doing. It may look a little different than standard Java but that is just normal groovy syntax. Finally, we verified that 80 dollars still exists in the account because, yes you got it, the 'then' told us that is what was expected. Here is where you encounter some of the syntatic sugar of easyb. The assertion is done in a very readable fashion via the 'should' enhancements available on objects. More focus is given to the 'ensure' and 'should' syntax in the Syntax chapter.

So far we have successful as well as pending scenarios. If we were all perfect we could stop right there. We're human so we know better, thats why we'll now see what happens when we expect a particular outcome but end up with something different.

Finishing the second scenario and running it will give us exactly that opportunity.

```
...

scenario "money is withdrawn from an account with insufficient funds", {
  given "an account with 10 dollars", {
   account = new Account(10);
  }
  when "account holder withdrawls 20 dollars", {
    account.withdrawl 20
  }
  then "account holder should be notified about the insufficient funds", {
    // will cover in the next step
  }
  and "then account should still have 10 dollars", {
    account.balance.shouldBe 10
  }
}


...
```

```
java -cp easyb-0.9.3.jar:commons-cli-1.1.jar:groovy-1.5.4.jar:target/behavior-classes \
> org.disco.easyb.BehaviorRunner \
> ./behavior/groovy/org/disco/bdd/story/account/AccountWithdrawl.story

Running account withdrawl story (AccountWithdrawl.story)
FAILURE Scenarios run: 2, Failures: 1, Pending: 0, Time Elapsed: 0.656 sec
        scenario "money is withdrawn from an account with insufficient funds"
        step "then account should still have 10 dollars" -- expected 10 but was -10
```

```
2 total behaviors run with 1 failure
```

According to our requirements it appears that our Account class needs a little better logic.

Shown below is just the change necessary to the Account class. A minimal amount of logic to validate that we cannot overdraw the account.

```
...

    public void withdrawl(int withdrawlAmount) {
        if(balance - withdrawlAmount < 0) {
            throw new RuntimeException("This should really be a checked exception, but keeping the lis
        } else {
            balance = balance - withdrawlAmount;
        }
    }

...
```

A small change to the story itself is in order as well. This is so we can ensure that an exception is thrown when we try to withdrawl more than allowed.

```
...

  then "account holder should be notified about the insufficient funds", {
    ensureThrows(RuntimeException){
      withdrawl()
     }
  }


...
```

Run it and we find that all is indeed good again. What we did was to create a closure to hold the execution of the withdrawl and then call it within the scope of an ensureThrows. That allows easyb to validate that the specified exception was indeed thrown. Since the logic was changed to prevent an overdraw you'll notice that the other condition (of the account balance still being 10 dollars) was satisfied.

```
java -cp easyb-0.9.3.jar:commons-cli-1.1.jar:groovy-1.5.4.jar:target/behavior-classes \
> org.disco.easyb.BehaviorRunner \
> ./behavior/groovy/org/disco/bdd/story/account/AccountWithdrawl.story

Running account withdrawl story (AccountWithdrawl.story)
Scenarios run: 2, Failures: 0, Pending: 0, Time Elapsed: 0.596 sec

2 total behaviors run with no failures
```

Chapter 4. Usage

Some stuff about running easyb

4.1. ...Ways to run it

4.2. Reporting

Chapter 5. Syntax

easyb contains a rich language for capturing behaviors as executable documentation.

5.1. Stories

These entries are normally found within a Story file.

5.1.1. scenario

Illustrate a specific aspect of behavior of the application

Appears. Inside story file. Multiple may be included in a single story file.

```
scenario "description here", {
  ...
}
```

5.1.2. given, andGiven

Context specific to this scenario. Object and Data setup will often happen in a given.

Appears. Inside scenario. Multiple may appear in each scenario. Also appears directly in a story file outside of a scenario (not recommended).

```
scenario "scenario description here", {
  given "given description here", {
    // given impl here
  }

  andGiven "andGiven description here", {
    // andGiven impl here
  }

  ...
}
```

5.1.3. when

An action to be taken.

Appears. Inside scenario. Follows any givens. Multiple may appear in each scenario. Also appears directly in a story file outside of a scenario (not recommended).

```
scenario "scenario description here", {
  ...

  when "when description here", {
    // when impl here
  }

  ...
}
```

5.1.4. then, andThen

Validation of expected outcome.

Appears. Inside scenario. Follows any whens. Multiple may appear in each scenario. Also appears directly in a story file outside of a scenario (not recommended).

```
scenario "scenario description here", {
  ...

  then "then description here", {
    // then impl here
  }

  andThen "andThen description here", {
    // andThen impl here
  }

  ...
}
```

5.1.5. narrative

Gives a bried description of what is to be delivered. Provides a container for other narrative components.

Appears. Inside a story, preferably at the top. Only one can appear in each story.

```
narrative "narrative description here", {
  ...
}

scenario "scenario description here"  ...
```

5.1.6. as_a

An aspect that describes the person, or thing, that will benefit from the feature.

Appears. Inside narrative, prior to i_want.

```
narrative "narrative description here", {
  as_a "role description here"
  ...
}
```

5.1.7. i_want

Describes something that the system should do or a feature in business terms and not in terms of technology

Appears. Inside narrative, following as_a.

```
narrative "narrative description here", {
  ...
  i_want "feature description here"
  ...
}
```

5.1.8. so_that

Describes the business value accrued from this feature.

Appears. Inside narrative, following i_want.

```
narrative "narrative description here", {
  ...
  so_that "benefit description here"
}
```

5.2. Specifications

The entries below would normally be found exclusively in a Specification file.

5.2.1. it

Captures a specification. Usually described in terms of should.

Appears. At the root of a Specification file.

```
it "should have an it description here", {
  // it impl here
}
```

5.3. Common

The following can be placed in both Story and Specification files.

5.3.1. and (syntax replacement)

Used as replacement syntax for story and specification contstructs. Carries the connotation that it is equivalent to whatever the previous story or specification construct was.

Appears. between, or in place of, story and specification constructs (given, then, etc)

```
...
given "a given description here", {
  // given impl here
}

and "equivalent to another given", {
  // another given impl here
}
...
```

5.3.2. and (assertion chain)

Used to logically tie together assertion statements.

Appears. within a story or specification construct, placed between assertions to give more natural language readability

```
true.shouldBe true
and
```

```
false.shouldBe false
```

5.3.3. before

Actions to be taken before any Story or Scenario components are run. Usually used for boilerplate setup common to all scenarios or specifications in that file. Placement should be prior to any scenarios (in a story) or specifications.

Appears. At the top of a story or specification, prior to any other easyb components.

```
...
before "a before description here", {
  // before impl here
}
...
```

5.3.4. before_each

Actions to be taken before every Story or Scenario component is run. Usually used for boilerplate setup common to all scenarios or specifications in that file. Placement should be prior to any scenarios (in a story) or specifications.

Appears. At the top of a story or specification, prior to any other easyb components.

```
...
before_each "a before_each description here", {
  // before_each impl here
}
...
```

5.3.5. after

Actions to be taken after any Story or Scenario components are run. Usually used for boilerplate setup common to all scenarios or specifications in that file. Placement should be prior to any scenarios (in a story) or specifications.

Appears. At the top of a story or specification. Follows any before or before_each, prior to any other easyb components.

```
...
after "an after description here", {
  // after impl here
}
...
```

5.3.6. after_each

Actions to be taken after every Story or Scenario component is run. Placement should be prior to any scenarios (in a story) or specifications.

Appears. At the top of a story or specification. Follows any before or before_each, prior to any other easyb components.

```
...
```

```
after_each "an after_each description here", {
  // after_each impl here
}
...
```

5.4. Assertions

The meat of the syntactic sugar easyb provides for making assertions very readable.

5.4.1. should

All of the should assertions work from left to right. The object on the left having the attributes that will be validated by what is on the right.

Appears. Inside any of the story or scenario components, typically as part of an 'it' or a 'then'.

5.4.1.1. shouldBe

Equality of target checked against value passed in.

```
...
true.shouldBe true
...
```

5.4.1.2. shouldEqual

Synonym for shouldBe. Equality of target checked against value passed in.

```
...
true.shouldEqual true
...
```

5.4.1.3. shouldBeEqual

Synonym for shouldBe. Equality of target checked against value passed in.

```
...
true.shouldBeEqual true
...
```

5.4.1.4. shouldBeEqualTo

Synonym for shouldBe. Equality of target checked against value passed in.

```
...
true.shouldBeEqualTo true
...
```

5.4.1.5. shouldNotBe

Inequality of target checked against value passed in.

```
...
true.shouldNotBe false
```

```
...
```

5.4.1.6. shouldNotEqual

Synonym for shouldNotBe. Inequality of target checked against value passed in.

```
...
true.shouldNotEqual false
...
```

5.4.1.7. shouldNotBeEqual

Synonym for shouldNotBe. Inequality of target checked against value passed in.

```
...
true.shouldNotBeEqual false
...
```

5.4.1.8. shouldNotBeEqualTo

Synonym for shouldNotBe. Inequality of target object checked against value passed in.

```
...
true.shouldNotBeEqualTo false
...
```

5.4.1.9. shouldBeA

Verifies the target's type matches that which is passed in.

```
...
"StringValue".shouldBeA(String)
...
```

5.4.1.10. shouldBeA

Verifies the target's type matches that which is passed in.

```
...
"StringValue".shouldBeA String
...
```

5.4.1.11. shouldBeAn

Synonym for shouldBeA. Verifies the target's type matches that which is passed in.

```
...
1.shouldBeAn Integer
...
```

5.4.1.12. shouldNotBeA

Verifies the target's type does not match that which is passed in.

```
...
1.shouldNotBeA String
```

```
...
```

5.4.1.13. shouldNotBeAn

Synonym for shouldNotBeA. Verifies the target's type does not match that which is passed in.

```
...
1.shouldBeAn Integer
...
```

5.4.1.14. shouldBeGreaterThan

Verifies the target is greater than the value passed in using groovy comparison.

```
...
2.shouldBeGreaterThan 1
...
```

5.4.1.15. shouldBeLessThan

Verifies the target is less than the value passed in using groovy comparison.

```
...
1.shouldBeLessThan 2
...
```

5.4.1.16. shouldHave

Verifies the target contains the value passed in. Collections and Strings are handled in the value positions. Collections, Strings and Object Fields are handled in the target position.

```
...
"test".shouldHave("est")

myMap = [value: "Andy", another: 34, 55: "test"]
myMap.shouldHave("Andy")
myMap.shouldHave(55: "test")
myMap.shouldHave([value: "Andy", 55: "test"])
...
```

5.4.1.17. shouldNotHave

Verifies the target contains the value passed in. Collections and Strings are handled in the value positions. Collections, Strings and Object Fields are handled in the target position.

```
...
myMap = [value: "Andy", another: 34, 55: "test"]
myMap.shouldNotHave("Mervin") // covers a value in a map
myMap.shouldNotHave([value: "Mervin", 55: "somethingnottest"]) // covers a map in a map where neither
myMap.shouldNotHave([value: "Andy", 55: "somethingnottest"])  // covers a map in a map where at least
myMap.shouldNotHave(55: "foobar") // covers map where key exists but value doesn't
...
```

5.4.2. ensure

All of the ensure assertions validate the target (specified as the argument to ensure()) matches the assertion in

the term or the argument to the right of the term.

Appears. Ensure normally appears inside an 'it' or 'then. All of the assertion terms listed below will appear inside the block associated with the ensure itself.

5.4.2.1. isNull

Value of target must be null.

```
ensure(someNullTargetValue) {
 isNull
}
```

5.4.2.2. isNotNull

Value of target must not be null.

```
ensure(someNonNullTargetValue) {
 isNotNull
}
```

5.4.2.3. isA<class type>

Value of target must be the same class as the <class type> at the end of the term.

```
ensure(aStringTarget) {
 isAString
}
```

5.4.2.4. isEqualTo(value)

Value of target must equal the value of the argument to isEqualTo.

```
ensure(true) {
 isEqualTo(true)
}
```

5.4.2.5. isEqualTo<value>

Value of target must equal the value at the end of the term.

```
ensure("Test") {
 isEqualToTest
}
```

5.4.2.6. isNotEqualTo(value)

Value of target must not equal the value of the argument to isNotEqualTo.

```
ensure(false) {
 isNotEqualTo(true)
}
```

5.4.2.7. isNotEqualTo<value>

Value of target must not equal the value at the end of the term.

```
ensure("Foo") {
 isNotEqualToBar
}
```

5.4.2.8. isTrue

Value of target must be true.

```
ensure(true) {
 isTrue
}
```

5.4.2.9. isFalse

Value of target must be false.

```
ensure(false) {
 isFalse
}
```

5.4.2.10. ensureThrows

This is a special variant of the ensure clause. It is used to verify that a block of code must throw an exception of the type specified.

```
ensureThrows(RuntimeException.class) {
  String tst = null
  tst.toUpperCase()
}
```

5.4.3. common

The remaining assertions are common to both stories and specifications. They can also be used standalone (inside of an ensure) as well as a magic method (similar to the should syntax).

5.4.3.1. has

Target's contents must consist of (at minimum) the contents of the argument to has.

```
ensure("Test") {
 has("es")
}
```

```
...
 "Test".has("es") // Preferable to use shouldHave instead for readability
...
```

5.4.3.2. contains

Target's contents must consist of (at minimum) the contents of the argument to contains.

```
ensure("Test") {
 contains("es")
}
```

```
...
 "Test".contains("es") // Preferable to use shouldHave instead for readability
...
```

5.4.3.3. startsWith

Target's contents must begin with the contents of the argument to startsWith.

```
ensure("Test") {
  startsWith("Te")
}
```

```
... // Issue 110. shouldStartWith doesn't exist in the dsl yet
 "Test".shouldStartWith("Te") // Preferable to use shouldHave instead for readability
...
```

5.4.3.4. endsWith

Target's contents must end with the contents of the argument to endsWith.

```
ensure("Test") {
  endsWith("st")
}
```

```
... // Issue 111. shouldEndWith doesn't exist in the dsl yet
 "Test".shouldEndWith("st") // Preferable to use shouldHave instead for readability
...
```


Chapter 6. Appendix

6.1. Glossary

BDD	See Behavior Driven Development
Behavior Driven Development	Agile software development technique that encourages collaboration between developers, QA and non-technical or business participants in a software project. It was originally conceived in 2003 by Dan North [1] as a response to Test Driven Development, and has evolved over the last few years. The focus of BDD is the language and interactions used in the process of software development. Behavior-driven developers use their native language in combination with the ubiquitous language of Domain Driven Design to describe the purpose and benefit of their code. This allows the developers to focus on why the code should be created, rather than the technical details, and minimizes translation between the technical language in which the code is written and the domain language spoken by the business, users, stakeholders, project management etc. (Courtesy: http://en.wikipedia.org/wiki/Behavior_driven_development)
Story	Description of a requirement, which has an associated benefit and criteria for validation. Contains any of the following components: scenario, given, when then. It also may contain a narrative with the following components: narrative, as_a, i_want, so_that.

Index

A

after, 12
after_each, 12
and (assertion chain), 11
and (syntax replacement), 11
andGiven (see given)
andThen (see then)
as_a, 10

B

before, 12
before_each, 12

C

contains, 17

E

endsWith, 18
ensureThrows, 17

G

given, 9

H

has, 17

I

i_want, 10
isA, 16
isEqualTo(value), 16
isEqualTo<value>, 16
isFalse, 17
isNotEqualTo(value), 16
isNotEqualTo<value>, 16
isNotNull, 16
isNull, 16
isTrue, 17
it, 11

N

narrative, 10

S

scenario, 9
shouldBe, 13
shouldBeA, 14, 14
shouldBeAn, 14
  (see also shouldBeA)
shouldBeEqual, 13
  (see also shouldBe)
shouldBeEqualTo, 13
  (see also shouldBe)
shouldBeGreaterThan, 15
shouldBeLessThan, 15
shouldEqual, 13
  (see also shouldBe)
shouldHave, 15
shouldNotBe, 13
shouldNotBeA, 14
shouldNotBeAn, 15
  (see also shouldNotBeA)
shouldNotBeEqual, 14
  (see also shouldNotBe)
shouldNotBeEqualTo, 14
  (see also shouldNotBe)
shouldNotEqual, 14
  (see also shouldNotBe)
shouldNotHave, 15
so_that, 10
startsWith, 18
Syntax Specification, 9-18
  Common, 11-13
  common assertions, 17-18
  ensure, 16-17
  Should Assertions, 13-15
  Specifications, 11-11
  Stories, 9-11

T

then, 10

W

when, 9