
Drive development with easyb

Embrace collaborative development with an intuitive domain-specific language

Skill Level: Intermediate

[Andrew Glover \(ajglover@gmail.com\)](mailto:ajglover@gmail.com)

Author and developer

05 Nov 2008

A disconnect between the stakeholders who define requirements and the developers who implement them has long plagued software development. In recent years, frameworks based on dynamic languages and domain-specific languages (DSLs) have tried to bridge the stakeholder-developer gap by making code read more like normal language. This tutorial shows how easyb — which provides a more natural DSL that is closely attuned to stakeholders — helps developers and stakeholders collaborate effectively.

Section 1. Before you start

About this tutorial

easyb is a behavior-driven development (BDD) framework for the Java™ platform. By using a specification-based DSL, easyb aims to enable executable yet readable documentation. You write easyb specifications in Groovy and execute them via a Java runner that can be invoked through the command line, Apache Maven 2, or Apache Ant. With easyb, you can verify the behavior of anything you write in Java code, in a more natural way.

Objectives

This tutorial guides you step-by-step through the fundamental concepts of using easyb and leveraging stories to collaborate with stakeholders. You'll learn how to:

- Define stories and scenarios using the stakeholder's own words
- Implement them with easyb
- Practice the true intentions of test-driven development (TDD) through BDD

When you are done with the tutorial, you'll understand the benefits of collaborative stories implemented with easyb and how this framework makes collaboration easy.

Prerequisites

To get the most from this tutorial, you should be familiar with Java syntax and the basic concepts of object-oriented development on the Java platform. You should also be familiar with refactoring and normal unit testing.

System requirements

To follow along and try out the code for this tutorial, you need a working installation of either:

- [Sun's JDK 1.5.0_09](#) (or later)
- [IBM® Developer Kit for Java technology 1.5.0 SR3](#)

You also need easyb and Apache Ant. The tutorial includes download links and installation instructions for easyb and Ant.

The recommended system configuration for this tutorial is:

- A system supporting either the Sun JDK 1.5.0_09 (or later) or the IBM JDK 1.5.0 SR3 with at least 500MB of main memory
- At least 20MB of disk space to install the software components and examples covered

The instructions and examples in the tutorial are based on a Microsoft® Windows® operating system. All the tools covered in the tutorial also work on Linux® and UNIX® systems.

Section 2. Impedance mismatch

There's long been a disconnect between the people who define software requirements (stakeholders) and those who implement them (developers). Most of the defects discovered in software, regardless of platform or application, relate to a misunderstanding about what was asked for. Most businesses struggle with software that doesn't meet their needs — even though business stakeholders think they articulated their requirements upfront just fine.

Commercial and open source solutions have been created over the years to help address this issue. On the open source side, the Fit and Fittesse frameworks come to mind. They try to bridge the gap by allowing stakeholders to write specifications (in tabular format), with developers implementing tests that essentially run the defined requirements. Yet, if you have spent any time with these frameworks and similar ones, you know a disconnect still exists. The problem lies with the medium in which requirements are defined.

We are all different

A popular book theorizes that (metaphorically) *Men Are from Mars, Women Are from Venus*. So, too, are stakeholders and developers different. You could easily say that stakeholders, who speak in normal everyday language, are from Earth, while developers — who can speak in code that is mysterious to everyone but those who have coded before — are probably from a different solar system altogether. An impedance mismatch occurs whenever requirements defined in normal language documents (be they written in English, Chinese, French, Farsi, or any other language) are translated into code (be it the Java language, Ruby, Python, or any other programming language). Consequently, our industry has spent years stressing testing, validation, and even traceability. But the simple fact remains that most stakeholders don't speak code, and developers think in code (and might not speak normal language much).

In recent years, however, with the advent of dynamic languages and the popularity of *domain specific languages* (DSLs), frameworks are coming into focus that try to bridge the stakeholder-developer gap. They do this by making code read more like normal language. This is not to say that stakeholders will start writing code. Rather, when developers start coding, the normal-language descriptions of what they should be coding means that they can more aptly collaborate with stakeholders. In fact, with easyb, the requirements (or specifications) are literally joined with the code that will implement them.

Using what they say as is

Consider this code statement:

```
assertEquals(42.50, order.price(), 0.0)
```

Without the context in which the statement appears, this code is somewhat incomprehensible. Now imagine you don't even read code; that is, you are a stakeholder asking (and probably paying) for new features. The code statement would be unreadable.

Now look at this statement:

```
order.price().shouldBe 42.50
```

Although the context in which this statement appears is still absent, this code is more coherent. In fact, it reads like a normal English sentence. It describes behavior in a more literal manner too: It uses a normal everyday phrase like `shouldBe`, which is distinctly different from the other version's `assertEquals`. Stakeholders, in this case, could understand this code if they chose to read it. It essentially matches what they asked for in the first place.

Both examples convey the same meaning and indeed *validate* the same requirement. Yet, `order.price().shouldBe 42.50` comes extremely close to leveraging the stakeholder's language. This is a fundamental point of the notion of BDD, which strives to validate a software system more appropriately by thinking in terms of the word *should* rather than *test*. In fact, by focusing on behavior and closely modeling behavior after what stakeholders ask for, BDD converges with the idea of executable documentation. Leveraging stakeholders' language decreases the impedance mismatch between what they want and what they ultimately receive. Moreover, using a stakeholder's language facilitates a deeper level of collaboration between all parties. Watch:

Stakeholder: For the next release of our online store, our Gold level customers should receive a discount when they make a purchase.

Developer: What kind of discount? What criteria do they have to meet in order to receive it?

Stakeholder: When they have at least \$50 in their shopping cart.

Developer: Does the discount increase based upon the amount or is it fixed regardless of the value of the shopping cart?

Stakeholder: Good question. The discount is fixed at 15 percent

regardless of price. So, given a Gold level customer, when their shopping cart totals \$50 or more, then they should receive a 15 percent discount off the total price.

The stakeholder's last statement is key. Note how it essentially specifies the requirement with a means for validating it. It narrates a specific scenario in a larger story related to discounts.

Making words real

Given a scenario, a developer can literally take the stakeholder's comments word for word and execute them. The easyb BDD framework facilitates system validation through a DSL that supports stories and scenarios. Listing 1 realizes requirements for the Gold level discount example:

Listing 1. Realizing the stakeholder's requirements

```
scenario "Gold level customer with $50 in shopping cart", {  
  given "a Gold level customer"  
  when "their shopping cart totals $50 or more"  
  then "they should receive a 15 percent discount off the total price"  
}
```

This particular scenario doesn't do anything other than capture the stakeholder's requirements (which is still quite important), so it is considered *pending*. This status alone conveys valuable information. First, and foremost, stakeholders can see their words as a means to validate their requests. Second, they can gauge if their requirement has been fulfilled. Once this scenario has been implemented, it can take on two other states, *success* or *failure*, which both convey further status information to interested parties.

With a collaborative scenario defined, development can proceed to the implementation. The beauty in this case is that developers can directly implement the desired behavior in line with the requirements, as in Listing 2:

Listing 2. Implementing requirements with easyb

```
scenario "Gold level customer with $50 in shopping cart", {  
  given "a Gold level customer", {  
    customer = new GoldCustomer()  
  }  
  when "their shopping cart totals $50 or more", {  
    customer.shoppingCart << new Item("widget", 50.00)  
  }  
  then "they should receive a 15% discount off the total price" , {  
    customer.orderPrice.shouldBe 42.50  
  }  
}
```

This scenario is now executable within the context of the application it serves to validate. The scenario leverages the stakeholder's exact words. And regardless of the stakeholder's ability to read code, the code itself leverages natural language: `customer.orderPrice.shouldBe 42.50`.

By leveraging the stakeholder's language, easyb makes it easier for the stakeholders to help collaboratively in validating the system they want built. It creates a direct link between what stakeholders ask for and what they receive.

Section 3. Getting started with easyb

Getting started with easyb is simple. The framework is bundled as a .tar.gz archive. [Download](#) the latest version (0.9 at the time of this writing) and unzip it (using your favorite utility) somewhere on your system, such as a temp directory.

No Groovy setup required

Even though the easyb framework is written in and leverages Groovy, this doesn't impose any additional setup requirements.

After you unzip the archive, you'll notice an easyb-0.9 folder containing three JAR files: `easyb.jar`, `commons-cli.jar`, and `groovy.jar`. You must add them to your classpath for easyb to work (just as you would for any other Java tool).

Stories in detail

Before I take you any further, you need to understand the notion of a *story*. Recall the [conversation](#) in which the stakeholder and developer considered a requirement. Note how two primary aspects were discovered from that conversation:

- A feature
 - Gold level customers should receive a discount.
- A means to validate the feature
 - If there is \$50 (or more) in their shopping cart, then they should receive a 15 percent discount. (That is, their total bill should be \$42.50 if they had exactly \$50 in their cart at the time of checkout.)

In the world of BDD, this is called a *scenario*. Scenarios are parts of stories. In this case, there's an overriding story regarding discounts in general. One particular

scenario focuses on discounts for Gold level customers with \$50 or more in their shopping carts. There could easily be other related scenarios, such as a customer with less than \$50 or a scenario regarding a sliding scale of discounts when a shopping cart exceeds \$200.

A scenario, at its essence, captures a feature and a means to validate the feature, and it uses specific language to describe both facets. Looking closely at the last statement by the stakeholder, notice that it uses three key words: *given*, *when*, and *then*.

"So, *given* a Gold level customer, *when* their shopping cart totals \$50 or more *then* they should receive a 15 percent discount off the total price."

These three words bind a feature together with a means to verify it, because the words describe a flow or sequence of events:

1. Given some context,
2. When something happens,
3. Something else should happen.

Moreover, these three words can easily be chained, because events can definitely become more complicated than in this example. And some sequences don't even require a when; that is, given some context, sometimes something should happen.

Project setup

You've already downloaded easyb and you've got a good idea of what stories are, so now it's time to set up a project. You'll create a project, configure it to work with easyb, and begin to write a story with a few scenarios to validate discounts with respect to customers.

Follow these steps:

1. Create a new directory called acme. (You could call it anything you'd like.) This directory will contain everything you need to build a software application: source code, libraries, build system, and easyb stories for validating that the application does what it is supposed to do. (See [Download](#) for the full sample code.)
2. Inside the acme directory, create three subdirectories:

- **lib** for holding binary dependencies (such as easyb-xx.jar)
 - **src** for storing source code
 - **stories** for putting holding easyb stories
3. Place all three libraries found in the easyb download (commons-cli, groovy, and easyb) into the lib directory.
 4. Create a file named build.xml in the root of the acme directory.
-

Section 4. Project automation

Before you can start to write stories, you need a way to run them. You can run easyb stories from the command line, but that would get tiresome quickly. It makes better sense to set up a build system for compiling your code and running your easyb stories automatically. You can also run any unit tests you might decide to write, deploy your application, and so on. By automating tasks, a build system (in this case, one based on Apache Ant) frees you to focus on creating your application.

Installing Ant

Your build.xml file is an Ant build file, so you need to have Ant installed, which is quite simple:

1. [Download](#) Ant and unzip it wherever you'd like. (For instance, I unzipped mine in /home/aglover/tools/ant.)
2. Add the bin directory found inside the Ant installation to your path (so, in my case, I'll add /home/aglover/tools/ant/apache-ant-1.7.0/bin).
3. Create a new environment variable named `ANT_HOME` and have it point to where you unzipped the Ant binary (for example, /home/aglover/tools/ant/apache-ant-1.7.0/).

Defining targets

Now that Ant is installed, you'll define a few targets that will serve you for the remainder of the tutorial. At a minimum, you'll need three targets:

- One to compile source code

- Another to run easyb stories
- One to clean up any generated files in order to enable a fresh run

Open the build.xml file and add the code in Listing 3:

Listing 3. A default Ant build file

```
<project name="acme" basedir="." default="run-stories">
</project>
```

Listing 3 is the basic definition of an Ant build file. Running Ant with this file won't do anything interesting, so add the XML in Listing 4 between the two lines in Listing 3:

Listing 4. Source code compilation in Ant

```
<path id="build.classpath">
  <fileset dir="lib">
    <include name="**/*.jar"/>
  </fileset>
</path>

<target name="compile-source" description="compiles source code">
  <mkdir dir="target/classes"/>
  <javac srcdir="src/" destdir="target/classes">
    <classpath>
      <path refid="build.classpath"/>
    </classpath>
  </javac>
</target>
```

The XML in Listing 4 defines:

- A task for compiling source code found in the src directory
- A path that includes the binary files found in the lib directory

Next, add this `clean` target:

```
<target name="clean"
  description="cleans out generated directory">
  <delete dir="target"/>
</target>
```

Next, you'll add a target that runs easyb stories.

Defining the easyb Ant task

Many ways to run a story

easyb provides a number of mechanisms for running stories, ranging from the command line to Ant to Maven to an IDE plug-in.

To run easyb stories from Ant, you must first configure the `easyb` Ant task via the `taskdef` task so that Ant can find and execute it. Then you then can use the `easyb` task. The task itself requires a few aspects:

- A classpath where to find dependent binaries and code
- Which type of report output you'd like
- A path to the stories you'd like to run

Luckily, you can do all of this in one snippet of XML, as shown in Listing 5:

Listing 5. Running easyb in Ant

```
<target name="run-stories" depends="compile-source">
  <taskdef name="easyb" classname="org.disco.easyb.ant.BehaviorRunnerTask">
    <classpath>
      <path refid="build.classpath"/>
    </classpath>
  </taskdef>

  <easyb failureProperty="easyb.failed">
    <classpath>
      <path refid="build.classpath"/>
      <pathelement path="target/classes"/>
    </classpath>

    <report location="target/stories.txt" format="txtstory"/>
    <behaviors dir="stories">
      <include name="**/*.story"/>
    </behaviors>
  </easyb>

  <fail if="easyb.failed" message="easyb reported a failure"/>
</target>
```

A number of things are going on in Listing 5, but it turns out to be rather intuitive. First, the `run-stories` target is defined. It depends on the `compile-source` target, which compiles your source code. The stories themselves will exercise your application code (much the same way that JUnit tests work with normal code).

Next, the `easyb` task is loaded via Ant's `taskdef` task. A `<classpath>` is provided, which essentially tells Ant to look in the `lib` directory because the code for the `easyb` Ant task is in the `easyb-xx.jar` file found in this directory.

The `easyb` task supports a failure property, which when defined enables you to fail a build should a story fail. This is a good idea because the whole idea of defining executable documentation is to prove an application is doing what it is supposed to

do. If a story fails, this is most likely an indication that things aren't going correctly.

After that, a report is defined. In this case, the target instructs `easyb` to create a text story report named `stories.txt` in the target directory. `easyb` supports other reporting formats too, including XML and HTML.

The `behaviors` element defines where `easyb` can find stories. In this case, `easyb` will look in the `stories` directory for any files ending in `.story`. The `.story` extension is an `easyb` convention; however, you are free to end story files with `.groovy`, *provided* you end the proper name of the file with `Story`. For example, both `Discounts.story` and `DiscountsStory.groovy` are valid `easyb` file names.

Finally, Ant's `fail` task is leveraged in cooperation with the `failure` property set earlier (`easyb.failed`). If this property is set, then Ant will trigger a failure condition with the message `easyb reported a failure`.

With these steps, your project can now run `easyb` stories automatically. Moreover, if a story fails, so will your build, providing valuable feedback about the status of your code base.

Section 5. Defining stories

As you've already seen, stories capture requirements and how to validate them in a standard format that both developers and stakeholders can read. And you've already seen one scenario elaborated in a conversation. Now that you have a better understanding of what stories and scenarios are, you can try your hand at implementing new ones.

A new conversation

Imagine that the earlier scenario in the discounts story has been implemented and your client was so happy with the way the feature turned out that they're requesting another feature. As before, the details can unfold in a conversation.

Stakeholder: Our Gold level customers are loving the fact that they're receiving a 15 percent discount on orders of \$50 or more. We think it would be a good idea to provide some incentives to non-Gold level customers.

Developer: What kind of incentives?

Stakeholder: When a non-Gold level customer makes a purchase of \$100 or more, they should receive a \$10 discount; plus, they should receive a coupon for discounts on future purchases.

Developer: The \$10 discount remains fixed regardless of the price (provided it is over \$100)?

Stakeholder: Yes, it doesn't matter. Our analysis of customer buying habits shows that rarely do non-Gold level customers buy more than \$250 worth of merchandise.

Developer: How will these customers receive a coupon?

Stakeholder: The system should e-mail it to them. We already have a coupon service that was implemented a few months ago, so we'll just reuse that.

Developer: OK, so given a non-Gold level customer, when they have \$100 or more in their shopping cart at the time of purchase, then they should receive a \$10 discount and they should be e-mailed a coupon.

Stakeholder: Exactly! When can you have this done?

As before, the stakeholder (or client) has tried to articulate clearly what he or she wants. Note that the stakeholder uses the word *should* rather naturally. The word *test* doesn't appear at all, and if it did, it would probably sound strange. The developer then took the client's words and used the key elements of a scenario to paint a more formal picture of what the client is asking for.

Given that a formalized structure has been agreed upon for defining stories and scenarios, a developer or even the stakeholder can then codify the conversation by leveraging easyb's DSL.

The first step is to create a story file. Given that the broader discussion is about the notion of discounts related to customers, create a new file in your project's stories directory called `CustomerDiscounts.story`.

Open that file in your favorite editor or IDE. If you are an Eclipse user, for instance, you can easily define story files, edit them, and even run them from within your environment. This process requires executing easyb's `BehaviorRunner`; see easyb's documentation for more details.

Inside the `CustomerDiscounts.story`, you'll define a scenario. Scenario definitions use the syntax shown in Listing 6:

Listing 6. Scenario syntax

```
scenario "description", {}
```

The last bit (the comma followed by two curly braces) is optional. If you leave them out, `easyb` treats the `scenario` you defined as pending.

Add the line in Listing 7 to your `CustomerDiscounts.story`:

Listing 7. Defining a scenario

```
scenario "Non-Gold level customer with \$100 or more"
```

Because the dollar sign (\$) in Groovy is essentially a delimiter, you need to escape it by preceding it with `\`.

Now, run your Ant build's `run-stories` task, as shown in Listing 8:

Listing 8. Running easyb via Ant

```
/home/aglover/Development/acme$ ant run-stories
Buildfile: build.xml

compile-source:

run-stories:
[easyb] easyb is preparing to process 1 file(s)
[easyb] Running customer discounts story (CustomerDiscounts.story)
[easyb] Scenarios run: 1, Failures: 0, Pending: 1, Time Elapsed: 0.534 sec
[easyb]
[easyb] 1 behavior run (including 1 pending behavior) with no failures
[easyb] easyb execution passed

BUILD SUCCESSFUL
Total time: 2 seconds
```

As you can see, `easyb` executed your story: one scenario was run, and it is in a pending state.

As you'll recall, when you configured the Ant task to run the `easyb` task, you also specified that you want report, called `stories.txt`, in a text-based format. You had `easyb` generate it in the target directory. Take a look at `stories.txt`, which should look something like Listing 9:

Listing 9. easyb report

```
1 scenario (including 1 pending) executed successfully
Story: customer discounts
    scenario Non-Gold level customer with $100 or more
```

```
given a non-Gold level customer
when they have $100 or more in their shopping cart
then they should receive a $10 discount
then they should be e-mailed a coupon [PENDING]
```

As you can see, this report serves as a nice view for stakeholders because it repeats the requirement(s) in normal language. Note too that your particular scenario is considered pending.

States in easyb

In easyb, scenarios — not stories — have states. A scenario can be in a *passing*, *failing*, or *pending* state. Pending is handy because it can indicate that a particular scenario is actively being worked on. Failing, obviously, indicates an error condition either occurring in the application being verified or possibly in how the scenario is defined.

Scenarios in easyb define steps using the keywords `given`, `when`, `then`, and optionally `and`. Each keyword definition in a scenario looks a lot like the `scenario` keyword definition, as Listing 10 shows:

Listing 10. DSL syntax of keywords

```
keyword "description", {}
```

As with `scenarios`, the comma and curly braces are optional. The curly braces are actually a *closure* in Groovy, so I'll use that term from here on out.

Collaborative stories

In the earlier conversation in this section between the stakeholder and the developer, a scenario was essentially defined when the developer stated:

"*Given* a non-Gold level customer, *when* they have \$100 or more in their shopping cart at the time of purchase, *then* they should receive a \$10 discount *and* they should be e-mailed a coupon."

Go back to the `CustomerDiscounts.story` file and codify the scenario details, as shown in Listing 11:

Listing 11. Codifying scenario details in easyb

```
scenario "Non-Gold level customer with \$100 or more" , {
  given "a non-Gold level customer"
```

```
when "they have \$100 or more in their shopping cart"  
  then "they should receive a $10 discount"  
  and "they should be emailed a coupon"  
}
```

Run the story again via Ant. You should see the same results as earlier; that is, the scenario is still in a pending state. Let's take a moment to understand what's going on. This scenario has four steps, which at a high level define:

1. Given some context,
2. When something happens,
3. Then something else should happen,
4. And another thing should happen.

As you can see, the last `and`, in this case, is literally a `then`. In `easyb`, an `and` can mean any of the other three keywords; in fact, it takes the meaning of the keyword it follows.

Now that the scenario has been codified, it becomes easier for both stakeholders and developers to collaborate because they have a common medium for discussion. Everyone is looking at the same thing. Both parties can discuss ambiguities or address questions regarding aspects; for example, when should the e-mail be sent to the customer? Immediately? A day later?

Developer: When should the e-mail be sent out? We could send it immediately upon successful purchase, or we could send it out within some specified amount of time.

Stakeholder: Ah, we didn't really consider that. Let's send it 24 hours later. That way, maybe the customer will come back and use it. If we send it immediately, it might be forgotten because they just made a purchase.

Developer: And they only get an e-mail if they spend \$100 or more?

Stakeholder: Correct.

Developer: The discount — is that applied for that order or is it a \$10 discount at some later date?

Stakeholder: It should be applied to the current order price, so they should receive \$10 off the total price.

This conversation provides details that clarify some aspects in the scenario that

were vague. Update the scenario to add these details, as shown in Listing 12:

Listing 12. Updating the scenario

```
scenario "Non-Gold level customer with \">$100 or more" , {  
  given "a non-Gold level customer"  
  when "they have \">$100 or more in their shopping cart"  
  then "at the time of checkout, they should receive \ $10 off the total price"  
  and "they should be emailed a coupon within 24 hours"  
}
```

Running this story with its one scenario still produces a pending result. It's probably time to start plugging in some behavior, right?

Section 6. Implementing stories

Although easyb stories are Groovy scripts, easyb's usefulness is not limited to Groovy alone. In fact, easyb stories can easily validate Java applications, because Groovy integrates seamlessly with Java code. And Groovy's syntax is quite flexible, so you can easily write normal Java code inside a story file and get away with it.

TDD, formally defined

TDD has many different definitions, so many people find themselves confused when asked if they practice TDD. Suffice it to say that one formal inference of TDD is that of *test-first development*. Test-first development essentially espouses writing a failing test first, followed by enough code to make that test pass, followed by another failing test, following by enough code to make that test pass. Repeat that process until the task at hand is complete, and you've just practiced a strict form of TDD. That said, few developers actually do this day in and day out.

BDD's focus on more natural ways of expressing specifications (via the focus on the word *should*) turns out to be a more effective means for driving development in the spirit of test-first development. So far in this tutorial, you've already taken that route. You first wrote a specification. It is now in a pending state. The next step is to put the scenario in a failing state. Then, you'll write enough code to make the scenario pass. Guess what you'll do after that? That's right: You'll write another scenario to make the code fail, and thus the process repeats itself until you've satisfied the requirements at hand.

As you'll see, it is quite natural to take this route — far more natural than focusing on the word *test* in combination with other frameworks such as JUnit.

Drive development easily

Open the `CustomerDiscounts.story` file with your favorite editor. The process of adding behavior to a story is a matter of adding the comma and corresponding closure to individual phrases.

Imagine that the application doesn't have any existing objects. You can flesh them out using BDD. For instance, the first phrase describes a non-Gold level customer. Add the code in Listing 13 to the story:

Listing 13. Implementing the given clause

```
given "a non-Gold level customer", {  
  customer = new Customer()  
  customer.setGoldLevel(false)  
}
```

As you can see in Listing 13, the code in the closure is quite similar to Java code — it's just lacking semicolons. You can make the code more Groovy by doing a few things, though. Groovy permits a more relaxed syntax that can drop parentheses from method calls. Property accessing is also relaxed: you don't need to call a setter or getter — you can access a property directly (even if it is privately declared in the corresponding object). So rewrite the `given` phrase as shown in Listing 14:

Listing 14. Refactoring the given clause

```
given "a non-Gold level customer", {  
  customer = new Customer()  
  customer.goldLevel = false  
}
```

Of course, you haven't defined a `Customer` object so far, but that's not a concern yet. Let's continue fleshing out the interactions between the various objects you'll need by focusing on the scenario.

Next, the `when` clause needs to link \$100 in a shopping cart with a customer. One possible way to code this is shown in Listing 15:

Listing 15. Implementing the when clause

```
when "they have $100 or more in their shopping cart", {  
  shoppingCart = new ShoppingCart()  
  shoppingCart.addItem(new Item("Foo", 101.00))  
  customer.shoppingCart = shoppingCart  
}
```

Again, the `ShoppingCart` object doesn't exist (nor does `Item`), but let's not worry about it. Because these objects don't exist yet, you're free to explore how they are interacting and change them as needed. Nothing will break (yet) because nothing exists.

Next, the `then` clause explicitly adds the verification of the new feature — that is, non-Gold level customers should receive \$10 off the total order price. Thus, the verification is that the order price should be \$91, right? `easyb` makes this aspect as intuitive as possible, as you can see in Listing 16:

Listing 16. Implementing the `then` clause

```
then "at the time of checkout, they should receive \$10 off the total price", {
  customer.checkOut()
  customer.orderPrice().shouldBe 91.00
}
```

The magic is in the last line of Listing 16. No more `asserts`! With `easyb`, you can use a more natural mechanism for checking things via the `shouldBe` clause. Regardless of what `orderPrice` returns, `shouldBe` is automatically available to that return type. (That is, `shouldBe` works on `Strings`, `Integers`, `BigDecimals`, and so on.)

`easyb` supports a host of similar checks, including:

- `shouldNotBe`
- `shouldEqual`
- `shouldNotEqual`
- `shouldBeGreaterThan`
- `shouldBeLessThan`

These are just a few of the verifications that are available to all objects within the context of an `easyb` story.

Finally, you need to handle the last phrase in [Listing 12](#). This one is interesting because it adds a high degree of complexity to verifying things. Verifying that an e-mail goes out within 24 hours might mean waiting to verify that it happened. Or, given that no objects exist yet, you could define an API that permits verifying things more easily, as shown in Listing 17:

Listing 17. Implementing the `and` clause

```
and "they should be emailed a coupon within 24 hours", {
  CouponService.scheduleDiscountEmail(customer)
```

```
CouponService.scheduledEmails.shouldHave customer
}
```

The `CouponService`, as currently coded, might need some more work; remember too, that the stakeholder said an existing service handles this. Perhaps your `CouponService` will act as an abstraction to the existing service. Again, it really doesn't matter at this point. Your focus isn't yet on the details of implementation but on the collaborative interaction of objects and the resulting APIs.

At this point, if you like what you've coded, you can try to run the story again. Of course, it'll fail, right? In fact, if you decide to run your Ant script, you should see something like Listing 18:

Listing 18. Things are still failing

```
run-stories:
[easyb] easyb is preparing to process 1 file(s)
[easyb] Running customer discounts story (CustomerDiscounts.story)
[easyb] There was an error running your easyb story or specification
[easyb] org.codehaus.groovy.control.MultipleCompilationErrorsException:
startup failed, Script1.groovy: 4: unable to resolve class Customer
[easyb] @ line 4, column 15.Script1.groovy: 8: unable to resolve class ShoppingCart
[easyb] @ line 8, column 19.Script1.groovy: 9: unable to resolve class Item
[easyb] @ line 9, column 25.
[easyb] 3 errors
[easyb]
[easyb] at org.codehaus.groovy.control.ErrorCollector.failIfErrors(
ErrorCollector.java:296)
[easyb] at org.codehaus.groovy.control.CompilationUnit.applyToSourceUnits(
CompilationUnit.java:787)
[easyb] at org.codehaus.groovy.control.CompilationUnit.compile(CompilationUnit.java:438)
[easyb] at groovy.lang.GroovyClassLoader.parseClass(GroovyClassLoader.java:277)
[easyb] at groovy.lang.GroovyShell.parseClass(GroovyShell.java:572)
[easyb] at groovy.lang.GroovyShell.parse(GroovyShell.java:584)
[easyb] at groovy.lang.GroovyShell.parse(GroovyShell.java:564)
[easyb] at groovy.lang.GroovyShell.evaluate(GroovyShell.java:542)
[easyb] at groovy.lang.GroovyShell.evaluate(GroovyShell.java:518)
[easyb] at org.disco.easyb.domain.Story.execute(Story.java:34)
[easyb] at org.disco.easyb.BehaviorRunner.runBehavior(BehaviorRunner.java:76)
[easyb] at org.disco.easyb.BehaviorRunner.main(BehaviorRunner.java:57)
[easyb] easyb execution FAILED

BUILD FAILED
```

The failures are due to the fact that you don't have any objects defined yet.

Application construction

In the `src` directory, create a package structure (for instance, `org.acme.app`). Accordingly, define four new objects:

- `Customer`
- `ShoppingCart`

- Item
- CouponService

Each object then needs the corresponding methods elaborated upon in the `CustomerDiscounts.story`.

Start with the `Item` class, shown in Listing 19:

Listing 19. Implementing the Item class

```
package org.acme.app;

import java.math.BigDecimal;

public class Item{
    public Item(String name, BigDecimal price){}
}
```

`Item`'s constructor takes a `String` and a `BigDecimal` because you defined it as such in [Listing 15](#) when you said:

```
shoppingCart.addItem(new Item("Foo", 101.00))
```

Next, code the `ShoppingCart` class. This class needs an `addItem` method, shown in Listing 20:

Listing 20. ShoppingCart implemented

```
package org.acme.app;

public class ShoppingCart{
    public void addItem(Item itm){}
}
```

Next up, the `Customer` object. This one is a bit more involved, but nothing too hard to code. `Customer` needs `goldLevel` and `shoppingCart` properties (and thus setters and getters, but recall that getters aren't really needed to make the scenario pass), a `checkout` method, and an `orderPrice` method. Listing 21 defines the `Customer` class:

Listing 21. The Customer class

```
package org.acme.app;

import java.math.BigDecimal;
```

```
public class Customer{
    private ShoppingCart shoppingCart;
    private boolean goldLevel;

    public Customer(){}

    public void setGoldLevel(boolean value){
        this.goldLevel = value;
    }

    public void setShoppingCart(ShoppingCart cart){
        this.shoppingCart = cart;
    }

    public boolean getGoldLevel(){
        return this.goldLevel;
    }

    public ShoppingCart getShoppingCart(){
        return this.shoppingCart;
    }

    public void checkOut(){}

    public BigDecimal orderPrice(){
        return null;
    }
}
```

Remember when I said that test-first development involves writing just enough code to make a test pass — or in our case, just enough code to make a scenario pass? Thus, the `Customer` class's `orderPrice` method isn't what we need now. To make the scenario pass, you need to make the `orderPrice` method return what? If your answer is `91.00`, as shown in Listing 22, then you're getting the hang of this:

Listing 22. `orderPrice` is hardcoded

```
public BigDecimal orderPrice(){
    return new BigDecimal("91.00");
}
```

If you're confused why you need to hardcode `91.00` as the return value, review my [description of a test-first development](#) at the start of this section. Don't get ahead of yourself though. You'll eventually code the required logic, but only after you've first made your scenario pass and written another failing scenario.

Next, you need to code the `CouponService`. Once again, you'll hardcode logic, as shown in Listing 23. You don't need to worry about details just yet.

Listing 23. The `CouponService`

```
package org.acme.app;

import java.util.ArrayList;

public class CouponService{
```

```

private static ArrayList<Customer> scheduledEmails = new ArrayList<Customer>();

public static ArrayList getScheduledEmails(){
    return scheduledEmails;
}

public static void scheduleDiscountEmail(Customer cust){
    scheduledEmails.add(cust);
}
}

```

Lastly, you need to update the story to import your newly defined objects, as shown in Listing 24:

Listing 24. Imported classes in your story

```

import org.acme.app.Customer
import org.acme.app.Item
import org.acme.app.ShoppingCart
import org.acme.app.CouponService

scenario "Non-Gold level customer with \$100 or more" , {
    given "a non-Gold level customer", {
        customer = new Customer()
        customer.goldLevel = false
    }
    when "they have \$100 or more in their shopping cart", {
        shoppingCart = new ShoppingCart()
        shoppingCart.addItem(new Item("Foo", 101.00))
        customer.shoppingCart = shoppingCart
    }
    then "at the time of checkout, they should receive \$10 off the total price", {
        customer.checkOut()
        customer.orderPrice().shouldBe 91.00
    }
    and "they should be emailed a coupon within 24 hours", {
        CouponService.scheduleDiscountEmail(customer)
        CouponService.scheduledEmails.shouldHave customer
    }
}

```

Rerun your stories via Ant, as shown in Listing 25:

Listing 25. Running your stories via Ant

```

/home/aglover/Development/acme$ ant clean run-stories

```

What do you see? That's right, everything passed, as shown in Listing 26:

Listing 26. Success

```

run-stories:
[easyb] easyb is preparing to process 1 file(s)
[easyb] Running customer discounts story (CustomerDiscounts.story)
[easyb] Scenarios run: 1, Failures: 0, Pending: 0, Time Elapsed: 0.454 sec

```

```
[easyb]
[easyb] 1 behavior run with no failures
[easyb] easyb execution passed
```

Listing 27 shows the resulting report:

Listing 27. What a nice report

```
1 scenario executed successfully

Story: customer discounts

scenario Non-Gold level customer with $100 or more
  given a non-Gold level customer
  when they have $100 or more in their shopping cart
  then at the time of checkout, they should receive $10 off the total price
  then they should be emailed a coupon within 24 hours
```

Not bad — but you're not done yet.

Section 7. Write, run, repeat

You've got a successful scenario in a larger story about customer discounts. However, you've written just about the least code you could write to make things work. The next step is to write another scenario that causes the code to fail. Then you'll write enough code to make that scenario pass. In the process of doing this, you'll actually end up writing the required logic to meet the stakeholders' requirements. And you'll have an extensive suite of scenarios and stories that validate that the application does what it is supposed to do.

Edge cases

As with all test cases, stories, scenarios, and requirements, there are edge cases. Think about what edge cases are related to the current scenario. What happens when non-Gold level customers purchase less than \$100 of items?

To define this edge case, open `CustomerDiscounts.story` and add the new scenario in Listing 28:

Listing 28. An edge-case scenario

```
scenario "Non-Gold level customer with less than \"$100", {
  given "a non-Gold level customer"
```

```

when "they have less than \$100 in their shopping cart"
then "at the time of checkout, they should not receive any
discounts"
and "they should not be emailed a coupon"
}

```

Rerun the Ant build. You'll see that a new scenario has been run and that it's in a pending state, as shown in Listing 29:

Listing 29. A pending scenario again

```

run-stories:
[easyb] easyb is preparing to process 1 file(s)
[easyb] Running customer discounts story (CustomerDiscounts.story)
[easyb] Scenarios run: 2, Failures: 0, Pending: 1, Time Elapsed: 0.479 sec
[easyb]
[easyb] 2 total behaviors run (including 1 pending behavior) with no failures
[easyb] easyb execution passed

```

The next step, as before, is to add behavior. Add the new logic to the new scenario, as shown in Listing 30:

Listing 30. New logic that should fail

```

scenario "Non-Gold level customer with less than \$100", {
  given "a non-Gold level customer", {
    customer = new Customer()
    customer.goldLevel = false
  }
  when "they have less than \$100 in their shopping cart", {
    shoppingCart = new ShoppingCart()
    shoppingCart.addItem(new Item("Foo", 99.00))
    customer.shoppingCart = shoppingCart
  }
  then "at the time of checkout, they should not receive any discounts", {
    customer.checkOut()
    customer.orderPrice().shouldBe 99.00
  }
  and "they should not be emailed a coupon", {
    CouponService.scheduledEmails.shouldNotHave customer
  }
}

```

Now, run Ant — first clean things up and then run the stories. You should see some intuitive text printing on your console, as shown in Listing 31:

Listing 31. Failed, once again

```

run-stories:
[easyb] easyb is preparing to process 1 file(s)
[easyb] Running customer discounts story (CustomerDiscounts.story)
[easyb] FAILURE Scenarios run: 2, Failures: 1, Pending: 0, Time Elapsed: 0.5 sec
[easyb] scenario "Non-Gold level customer with less than $100"
[easyb] step "at the time of checkout, they should not receive any discounts" --
expected 99.00 but was 91.00

```

```
[easyb]
[easyb] 2 total behaviors run with 1 failure
[easyb] easyb execution FAILED
```

```
BUILD FAILED
```

Sure enough, things failed because you hardcoded the 91.00, remember? Notice how easyb's error message is quite descriptive too, quickly alerting you to where any problems may lie.

You need to take some steps to make things work at this point. First, update the `ShoppingCart` object to keep track of `Items`, as shown in Listing 32:

Listing 32. Keeping track of Items

```
package org.acme.app;

import java.util.ArrayList;
import java.math.BigDecimal;

public class ShoppingCart{

    private ArrayList<Item> items;

    public ShoppingCart(){
        this.items = new ArrayList<Item>();
    }

    public void addItem(Item itm){
        this.items.add(itm);
    }

    public BigDecimal getTotalPrice(){
        BigDecimal total = new BigDecimal(0);
        for(Item item : this.items){
            total = total.add(item.getPrice());
        }
        return total;
    }
}
```

Note that Listing 32 adds a new `getTotalPrice()` method to the `ShoppingCart` object that permits understanding the total price of a cart. Discounts aren't applied to a cart because they are related to `Customers`. Accordingly, you'll add that logic to the `Customer` object in a moment. Now you need to update the `Item` object, as shown in Listing 33:

Listing 33. Updated Item class

```
package org.acme.app;

import java.math.BigDecimal;

public class Item{
    private BigDecimal price;
```

```
public Item(String name, BigDecimal price){
    this.price = price;
}

public BigDecimal getPrice(){
    return this.price;
}
}
```

You need the `Item` object to hold some state regarding its price because you'll be totaling up prices of a `ShoppingCart`.

Lastly, you need to update the `Customer` object to apply the discount based upon the total price. Consequently, the only change is to the `orderPrice` method, as shown in Listing 34:

Listing 34. Updating the `orderPrice` method

```
public BigDecimal orderPrice(){
    if(!this.goldLevel){
        BigDecimal total = this.shoppingCart.getTotalPrice();
        if(total.compareTo(new BigDecimal("100.00")) >= 0 ){
            return this.shoppingCart.getTotalPrice().subtract(new BigDecimal("10.00"));
        }else{
            return total;
        }
    }else{
        return this.shoppingCart.getTotalPrice();
    }
}
```

This method might not be perfect just yet; however, you have enough to make the new scenario pass, so give Ant a run. You should see the results in Listing 35:

Listing 35. Rerunning Ant

```
run-stories:
[easyb] easyb is preparing to process 1 file(s)
[easyb] Running customer discounts story (CustomerDiscounts.story)
[easyb] Scenarios run: 2, Failures: 0, Pending: 0, Time Elapsed: 0.646 sec
[easyb]
[easyb] 2 total behaviors run with no failures
[easyb] easyb execution passed
```

Open up the report to see some nice readable text, as shown in Listing 36:

Listing 36. Nice readable story text

```
2 scenarios executed successfully

Story: customer discounts

scenario Non-Gold level customer with $100 or more
given a non-Gold level customer
```

```
when they have $100 or more in their shopping cart
then at the time of checkout, they should receive $10 off the total price
then they should be emailed a coupon within 24 hours

scenario Non-Gold level customer with less than $100
  given a non-Gold level customer
  when they have less than $100 in their shopping cart
  then at the time of checkout, they should not receive any discounts
  then they should not be emailed a coupon
```

Notice how this exercise improved the code while capturing edge cases the client might have not considered.

The job is never done

At this point, you've got a working story with two scenarios. The next step is to write another scenario (ideally, collaboratively with a stakeholder) that causes things to fail. Then write enough code to make the new scenario pass. Finally, repeat.

Other easyb constructs

easyb supports some other constructs as well. For instance, as the story is currently written, the `given` phrase is the same in both scenarios. If you want to write this phrase once (rather than twice), you can pull the phrase out of both scenarios and place it into a fixture-like construct known as `before_each`. This, like all other constructs in easyb, has the syntax in Listing 37:

Listing 37. `before_each` specification

```
before_each "description" , {}
```

Using this syntax, you can then place the `given` phrase in the `before_each` closure as shown in Listing 38:

Listing 38. Using `before_each` as a fixture

```
before_each "initialize a Non-Gold level customer", {
  given "a non-Gold level customer", {
    customer = new Customer()
    customer.goldLevel = false
  }
}
```

And your entire story looks like Listing 39:

Listing 39. Refactored story using `before_each`

```
import org.acme.app.Customer
import org.acme.app.Item
import org.acme.app.ShoppingCart
import org.acme.app.CouponService

before_each "initialize a Non-Gold level customer", {
  given "a non-Gold level customer", {
    customer = new Customer()
    customer.goldLevel = false
  }
}

scenario "Non-Gold level customer with \$100 or more" , {
  when "they have \$100 or more in their shopping cart", {
    shoppingCart = new ShoppingCart()
    shoppingCart.addItem(new Item("Foo", 101.00))
    customer.shoppingCart = shoppingCart
  }
  then "at the time of checkout, they should receive \$10 off the total price", {
    customer.checkOut()
    customer.orderPrice().shouldBe 91.00
  }
  and "they should be emailed a coupon within 24 hours", {
    CouponService.scheduleDiscountEmail(customer)
    CouponService.scheduledEmails.shouldHave customer
  }
}

scenario "Non-Gold level customer with less than \$100", {
  when "they have less than \$100 in their shopping cart", {
    shoppingCart = new ShoppingCart()
    shoppingCart.addItem(new Item("Foo", 99.00))
    customer.shoppingCart = shoppingCart
  }
  then "at the time of checkout, they should not receive any discounts", {
    customer.checkOut()
    customer.orderPrice().shouldBe 99.00
  }
  and "they should not be emailed a coupon", {
    CouponService.scheduledEmails.shouldNotHave customer
  }
}
```

In addition to a `before_each` construct, `easyb` supports an `after_each` (which could be a series of `then` phrases to be executed for each scenario). What's more, if you'd rather have some one-time logic (that perhaps doesn't naturally fit with phrases, such as setting up a database connection), `easyb` also permits `before` and `after` clauses. Unlike their respective cousins (`before_each` and `after_each`), `before` and `after` run only once per story.

As you can see, `easyb`, while quite different-looking than normal xUnit frameworks, is similar in spirit: you can verify code (albeit more naturally) and you can reuse logic in fixtures.

Section 8. BDD: Customer-focused TDD

In this tutorial, you learned that by using a more natural language that is closely in tune with stakeholders, a more collaborative platform unfolds that permits developers and stakeholders to talk on the same level. With such a platform, the traditional ambiguities and misunderstandings that have plagued software development for years have a real chance of being overcome. Of course, there are no silver bullets in software development, and easyb isn't positioned to be one; however, it represents an evolutionary step toward the goal of more human-centered software development.

BDD with easyb embraces the original goals of test-first development. By using a stakeholder's words as is, the design of the code unfolds naturally without constraints. The steps you follow are what test-first development originally conceived too: you write a failing scenario, followed by enough code to make it pass, followed by a failing scenario, followed by enough code to make it pass, repeat.

easyb aims to be easy — the syntax tries not to get in your way — but of course it's up to you to write appropriate stories and scenarios and suitable code that meets stakeholders' demands.

Downloads

Description	Name	Size	Download method
Sample project code	j-easyb.zip	3.9MB	HTTP

[Information about download methods](#)

Resources

Learn

- [easyb](#): easyb is a behavior-driven development framework for the Java platform. By using a specification-based DSL, easyb aims to enable executable, yet readable, documentation.
- "[Adventures in behavior-driven development](#)" (Andrew Glover, developerWorks, September 2007): This article helps you take the leap from TDD to BDD using the JBehave framework.
- "[Behavior-driven development with easyb](#)" (Rod Coffin, JavaWorld, September 2008): Read an overview of BDD and easyb.
- "[Lost in translation? Stop using different languages](#)" (Andrew Glover, thediscoblog.com, October 2008): Andrew Glover suggests tools like easyb can bridge the communication gap between stakeholders and developers.
- "[Fluently Groovy](#)" (Andrew Glover, developerWorks, March 2008): Get started in this tutorial with Groovy's simplified variation of the Java syntax and learn about essential features like native collections, built-in regular expressions, and closures.
- "[easyb: Introducing Conversational Unit Tests for Java](#)" (Steven Devijver, JavaLobby, July 2008): An interview with Andrew Glover about easyb.
- "[An introduction to easyb: presentation slides](#)" (John Ferguson Smart's Blog, September 2008): The author of *Java Power Tools* shares his slides, which present an overview of easyb.
- "[Behavior-Driven Development for Everyone](#)" (Craig Wickesser, InfoQ, September 2008): InfoQ interviews the easyb team to see what's next.
- "[Is easyb Easy?](#)" (Meera Subbarao, JavaLobby, September 2008): Subbarao explores using easyb to verify a Java EE application and answers the title question in the affirmative.
- "[Stories are easy, man](#)" (Andrew Glover, thediscoblog.com, January 2008): The author explores using stories to convey requirements via easyb.
- [In pursuit of code quality](#): (Andrew Glover, developerWorks): This series explores techniques, tools, and methods for ensuring and measuring software quality.
- Browse the [technology bookstore](#) for books on these and other technical topics.
- [developerWorks Java technology zone](#): Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- [easyb](#): Download easyb.
- [Ant](#): Download Apache Ant.
- [Sun JDK 1.5 or later](#): You'll need at least version 1.5.0_09 to follow the examples in this tutorial.
- [IBM developer kits](#): IBM Java developer kits are available for AIX and Linux.
- Download [IBM product evaluation versions](#) and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

Discuss

- [The Groovy Zone](#): The DZone community for Groovy and Grails developers.
- [Improve your code quality](#): Andrew Glover's developerWorks discussion for developers focused on test-driven development, code quality, and reducing risk.
- Check out [developerWorks blogs](#) and get involved in the [developerWorks community](#).

About the author

Andrew Glover

Andrew Glover is a developer, author, speaker, and entrepreneur with a passion for behavior-driven development, Continuous Integration, and Agile software development. You can keep up with him at his [blog](#).

Trademarks

DB2, IBM, Lotus, Rational, Tivoli, and WebSphere are trademarks of IBM Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.