The Secret Ninja Cucumber Scrolls

David de Florinier
Gojko Adzic

STRICTLY CONFIDENTIAL

The Secret Ninja Cucumber Scrolls

Strictly Confidential

David de Florinier

Gojko Adzic

The Secret Ninja Cucumber Scrolls: Strictly Confidential

David de Florinier
Gojko Adzic
Cover design: Annette de Florinier

Publication date 2011 03 16

Neuri Limited
25 Southampton Buildings
London WC2A 1AL
United Kingdom

About this document

This document is a step-by-step guide for Cucumber, a tool that is quickly becoming the weapon of choice for many agile teams when it comes to functional test automation, creating executable specifications and building a living documentation. We believe in iterative development so we write and publish iteratively.

What's new in this version

The changes in this version are:

- **Introduction cleaned up, added a section on how Cucumber fits into BDD, moved terminology to the Gherkin part**
- **Fixed PDF inclusion of some missing .NET sources**
- **and a lot of minor fixes and tweaks**

Online

To get notified when we publish an update, grab the code for the examples we used in the guide, provide feedback or ask a question, do one or more of the following:

- **Point your browser to http://cuke4ninja.com periodically**
- **Follow @cuke4ninja on Twitter for updates and news**
- **Follow @davedf and @gojkoadzic on Twitter to keep in touch with the authors or drop us a message there**

Become a contribuninja

Do you have an interesting story to tell about cucumbers? It is related to the right kind of cucumbers? Is it morally acceptable? Do you have a different opinion about any of the things we wrote here? Please get engaged and help us make this document better.

See http://cuke4ninja.com/contribuninja for more information on how to contribute.

About the authors

David de Florinier is the world renowned sportsman, ninja-assassin and raconteur who has brought the world joy with works such as "How to bring about world peace using fried potato chips and HP sauce", "My amazing adventures with dolphins, and how they taught me to beat Wall St" and "How I taught Chuck Norris all he knows, in one morning, while I was solving the Riemann hypothesis at the same time". Read David's blog at http://deflorinier.blogspot.com or follow him on Twitter as @davedf.

Still active as a Bond-villain, Gojko Adzic gave up a career in the Soviet spy service to pursue tougher challenges in software development. All that remains of his past life now is the stupid accent. Gojko's goal with this document is to send subliminal messages to SMERSH sleepers around the world and thereby bring about the final rise to power of the Illuminati. Read Gojko's blog at http://gojko.net or follow him on Twitter as @gojkoadzic.

Ninja training

David and Gojko run public and on-site training workshops for aspiring Cucumber teams. Our workshops are designed for business users, analysts, testers and developers. Through facilitated exercises and discussion, these workshops provide teams with practical experience of agile acceptance testing, behaviour-driven development and specification by example. The participants learn how to collaborate on specifications and tests, how to design good Cucumber tests and how to automate them efficiently.

On-site workshops are typically customised to fit the business domains and particular needs of teams, so that the participants get real-world experience and instant benefits.

More than 90% of previous attendees said that the workshops improved their performance at work.

For more information on our workshops, see http://cuke4ninja.com/training, drop us an e-mail on training@cuke4ninja.com or give us a call on +442079932982.

Chapter 1

Why should you care about Cucumber?

Ninja is one of the most popular types of cucumbers in Thailand and South-Eastern Asia. Ninja plants start to set fruits 35 days after sowing and continue to produce cucumbers for a long time, making Ninjas excellent for both home gardens and commercial growing.[1]

Oh, no... sorry about that, wrong type of cucumber. Let's try again.

Cucumber is a functional test automation tool for lean and agile teams. It supports behaviour-driven development, specification by example and agile acceptance testing. You can use it to automate functional validation in a form that is easily readable and understandable to business users, developers and testers. This helps teams create *executable specifications*, that are a goal for development, acceptance criteria and functional regression checks for future changes. In this way, Cucumber allows teams to create *living documentation*, a single authoritative source of information on system functionality that is always up-to-date.

Executable what?

Yes, executable specifications. You can use Cucumber to execute checks based on your specifications against the software automatically, when you use specification by example. To keep this document short we won't go into specification by example, how behaviour-driven development works or how best to apply agile acceptance testing. There are some nice references in Appendix A for further research.

Cucumber is quickly becoming as popular with teams that develop and maintain web-based systems as the shuriken is with *Steve Jobs*[2]. Teams love Cucumber because of its flexibility and support for web automation frame-

[1] http://www.evergreenseeds.com/orcuhyni.html
[2] http://www.crazyapplerumors.com/?p=1273

works. It started as a tool for Ruby, but now supports a wide variety of platforms. This makes it appealing to teams that work with JVM-based languages and even those shinobi that use the dark magic of .NET.

Why use Cucumber?

Cucumber is one of the rare tools that try very hard to stay out of your way, to let you do your work without making you worry about the tool itself too much. It helps teams automate their specifications efficiently in several ways:

- It is relatively easy to set up.
- It supports multiple report formats, including HTML and PDF.
- It supports writing specifications in about 40 spoken languages, making it easy for teams outside of English-speaking territories or those working on internationally targeted software to engage their business users better.
- It supports different ways of describing executable specifications — including story-like prose, lists and tabular data. You can unleash your inner writer, or more likely an inner accountant.
- It allows scripting, abstraction and component reuse on several levels, allowing both technical and non-technical users to efficiently describe specifications and tests.
- It generates the tricky parts of the code so that you don't have to write most of the boiler-plate automation or make mistakes doing it.
- It integrates nicely with the rest of the development ecosystem. It does not try to impose a version control system, but works off plain-text files that can be stored in any version control system. For continuous build integration, it emulates JUnit. JUnit XML format is, of course, the equivalent of Star Trek transporters when it comes to testing — use it and you can beam your test results anywhere.
- Although Cucumber is a Ruby tool, people who work on other platforms do not have to learn Ruby to use it. You can use Cucumber with .NET or JVM languages almost natively.
- It's integrated with all the most popular web testing libraries.
- It allows you to cross reference and index tests using tags, so that you can quickly find all tests related to a function or run a group of related tests (eg quick tests, slow tests, integration tests, accounting tests).

Although you can automate almost anything with it,[3] the killer feature of Cucumber is the ease of web workflow testing.

How does Cucumber compare to other tools?

Cucumber is like that brave Athenian soldier who centuries ago decided to test the resilience of humans by running from the Marathon field. It is essentially a test runner. It does a pretty good job of executing automated tests and spitting out reports in lots of various formats, including HTML, PDF, JUnit (for integration with continuous build tools) and colour text on the screen. On the other hand, it doesn't try to do anything before test execution in the process. So there is no Cucumber-IDE or anything similar. There are some external programs for that, mostly of beta quality and in development:

- **http://github.com/henritersteeg/cuke4vs does syntax highlighting and intellisense in Visual Studio.**
- **http://github.com/QuBiT/cucumber-eclipse-plugin is a plugin for Eclipse**
- **http://github.com/cs3b/cucumber_fm/ is a website system for Cucumber file management**
- **http://relishapp.com/ is a web product in development to publish Cucumber files as a nice web site**
- **http://21croissants.github.com/courgette/ another web product in development to show files as web pages**
- **https://github.com/asterite/cukecooker/ is a web based IDE for Cucumber feature files when working in Ruby**

Cucumber files (explained later in this chapter) are plain-text which makes it very simple to edit them but does not allow any clever formatting. You can put a short description in a file header, but apart from that everything else in the file is related to scenarios and test cases. That means that there is no way to include images, hyperlinks, rich-text descriptions and similar.

Cucumber's step definition (automation layer, integration to domain API, explained in later chapters) model is very simple, which is both a strength

[3] **See the Getting Started section of http://wiki.github.com/aslakhellesoy/cucumber/ for an impressive list of platforms and tools Cucumber integrates with**

and a weakness in different contexts. A simple API makes it very easy to understand and learn what you can do with Cucumber, so you won't spend three months getting your head around all the possible ways to use tables. On the other hand, it doesn't have the smart domain-object mapping features that some other tools have, which allow you to expose your domain objects and services directly for testing and provide lots of flexibility in how you can manage complex scenarios.

What does Cucumber have to do with BDD

Aspiring Cucumber ninjas are well advised that Cucumber is frequently mentioned in the context of Behaviour driven Development (BDD). In fact, Cucumber is as closely tied to BDD as Chuck Norris is to his beard. Dan North, the central authority on BDD, summarised BDD as:[4]

> *A second generation, outside-in, pull based, multiple stakeholder, multiple scale, high automation, agile methodology*

Second Generation

BDD came out of merging ideas of Extreme Programming, Lean software development and Domain Driven Design, as a second-generation agile process that helps teams deliver high quality software and answers many of the most confusing questions of early agile processes, such as the ones dealing with documentation and testing.

Outside-in and pull-based

BDD takes the ideas of value chains and pulling features from Lean manufacturing, ensuring that the right software is built by focusing on the expected outputs of the system and ensuring that these outputs are achieved. Teams achieve this by collaborating on specifications and illustrating the specifications with key examples of expected outcomes. These specifications are created just in time when they are needed, from user stories and use cases that get pulled into development.

[4] See http://skillsmatter.com/podcast/java-jee/how-to-sell-bdd-to-the-business

Multiple-stakeholder

BDD dispenses with the idea that there is a single amorphous 'user' of the system, and recognises that different groups of people will be affected with the software in different ways. In Quality Software Management, Gerald Weinberg wrote that 'Quality is value to some person'. BDD forces the delivery teams to understand and define quality, by specifying *who* are the people that the software will bring some value to and *what* will be valuable to them.

Multiple-scale

A core idea of BDD is to clearly specify the expected outputs with realistic examples and then develop the software to achieve those outputs. This works on multiple levels. On the top level, we can work with high-level examples of how a feature will be useful in a holistic way. Below that, we can look at the impact of that on individual functional areas of the system. Below that we can work on technical implementation units. The process is the same, regardless of the level.

Agile

BDD works best with short iterations or flow-based work, where teams specify, implement and deliver relatively small chunks of functionality. It requires cross-functional teams to collaborate on specifications and tests.

High-automation

Once the expected quality is defined, the team will need to check the system functionality frequently and compare it to the expected results. To make this check efficient, it has to be automated. BDD relies heavily on automation of executable specifications.

Cucumber takes care of the 'high-automation' part of the BDD definition. It allows teams to describe features in a business language, with key examples, and automate the validation of those features against system functionality.

Part I. Getting Started

In this part we cover how to get started using Cucumber, including:

- prerequisites for using Cucumber with Ruby, Java and .NET
- setting up Cucumber and working through a simple Hello-World example
- using Cucumber in continuous integration

We keep things fairly basic for now and introduce more complex features in later parts. You can work through all the chapters in sequence or just skip to the one that interests you. There are three chapters in this part:

- Chapter 2 covers Ruby
- Chapter 4 covers Java
- Chapter 3 covers .NET

Chapter 2

Cucumber and Ruby

In this chapter, we cover setting up Cucumber for a Ruby project.

> Do you have Ruby installed?
>
> **In this chapter, we assume you have already installed ruby on your system. If this is not the case, there is more information on the *Ruby site*[1].**
>
> **If you are running Windows there are some notes on installing Ruby in Chapter 3**

Installing Cucumber

To automate feature files using Ruby you need to install the `cucumber` gem. Do that with the following command:

```
gem install cucumber
```

> Installing Gems behind a firewall
>
> If you do not have direct access to the internet, but instead go through a proxy server, you will have to set the HTTP_PROXY environment variable. Information about environment variables is available on the RubyGems site.[2]

Check that Cucumber is installed by running the following command in the terminal window:

```
cucumber --version
```

Cucumber should print out the version (0.9.4 at the time when we wrote this).

[1] **http://www.ruby-lang.org/en/**
[2] **http://docs.rubygems.org/read/chapter/12**

Cucumber uses RSpec for assertions. If you do not already have RSpec, run the following command to install it:

```
gem install rspec
```

Cucumber won't install!

Cucumber depends on a particular gherkin version. It will try to download it automatically, but if you already have a later version then RubyGems gets confused and won't download the version required by Cucumber. As a result you might get an error message similar to this one:

```
ERROR:  Error installing cucumber:
        cucumber requires gherkin (~> 2.2.9, runtime)
```

To resolve this, install the correct gherkin gem version first:

```
gem install gherkin --version 2.2.9
```

Or just uninstall the previous version of gherkin if you do not need it for other reasons, and then install the latest version:

```
gem uninstall gherkin
gem install gherkin
```

Hello World from Ruby

Let's go through a quick sample project to verify that the installation works. Create a directory for the project, for example `HelloCucumber`. Create a `features` directory in it. This is where the feature files go. Create a new file in it, called `basic.feature`, with the following content:

```
Feature: Hello World Feature
  In order to ensure that my installation works
  As a Developer
  I want to run a quick Cucumber test
```

```
  Scenario: Hello World Scenario
    Given The Action is Hello
    When The Subject is World
    Then The Greeting is Hello, World
```

Run `cucumber` from the command line in the main project directory. When it cannot match steps to step definitions, Cucumber prints out what it thinks would be a good default setup for the steps. As we have not implemented any steps yet, Cucumber should print a suggestion similar to the one in Figure 2.1.

Figure 2.1. Cucumber suggests a default implementation for step definitions

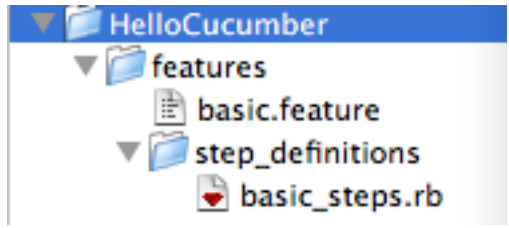
```
Feature: Hello World Feature
  In order to ensure that my installation works
  As a Developer
  I want to run a quick Cucumber test

  Scenario: Hello World Scenario      # features/basic.feature:6
    Given The Action is Hello         # features/basic.feature:7
    When The Subject is World         # features/basic.feature:8
    Then The Greeting is Hello, World # features/basic.feature:9

1 scenario (1 undefined)
3 steps (3 undefined)
0m0.002s

You can implement step definitions for undefined steps with these snippets:

Given /^The Action is Hello$/ do
  pending # express the regexp above with the code you wish you had
end

When /^The Subject is World$/ do
  pending # express the regexp above with the code you wish you had
end

Then /^The Greeting is Hello, World$/ do
  pending # express the regexp above with the code you wish you had
end

If you want snippets in a different programming language, just make sure a file
with the appropriate file extension exists where cucumber looks for step definitions.
```

Now create a sub-directory called `step_definitions` in the `features` directory. This is where the automation layer between the feature files and the domain code goes. Create a new file called `basic_steps.rb` in the `step_definitions` directory. You project structure should end up looking something like in Figure 2.2. Paste the following content into `basic_steps.rb`:

```
require 'rspec/expectations'

Given /The Action is ([A-z]*)/ do |action|
  @action = action
end

When /The Subject is ([A-z]*)/ do |subject|
  @subject = subject
end

Then /The Greeting is (.*)/ do |greeting|
  greeting.should == "#{@action}, #{@subject}"
end
```

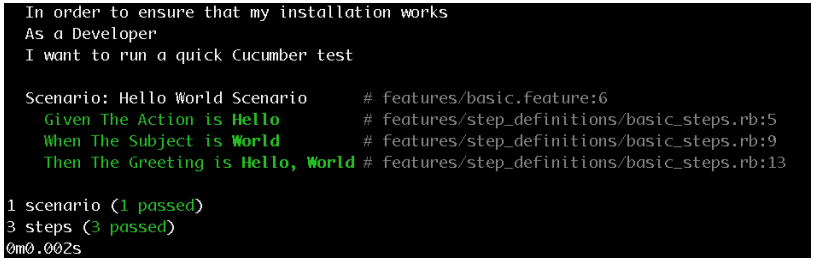
This is a very simplistic implementation for all the steps, that does not actually connect to any domain code. We go through all the details of the feature file and step definitions in Chapter 5. For now, you can probably guess that Cucumber uses regular expression after the keywords `Given`, `When` and `Then` to map lines in a feature file to step definitions.

Figure 2.2. Ruby project folders



Now we are ready to run the features. Run `cucumber` again from the command line in your main project directory, and you should get a result similar to the one in Figure 2.3. This tells us the number of features and steps executed, the lines of the step definitions, the steps executed and reports successful and failed test executions.

Figure 2.3. Our feature works in Ruby



Chapter 3

Cucumber and .NET

In this chapter, we cover setting up Cucumber for a .NET project, debugging Cucumber tests in .NET and integrating with a continuous build system.

Setting up Ruby

Don't worry, you won't have to know Ruby to work with .NET

Cucumber is written in Ruby and so you need it installed to run Cucumber tests. You do not need to know Ruby to write or run .NET tests.

Download and run the one-click installer from the RubyInstaller web site.[1] Although the latest version of Ruby at time of writing is 1.9.1, download the version 1.8.6. The Cucumber .NET integration, called Cuke4Nuke, does not work out of the box with the latest version of Ruby.

Next, add the location of the Ruby `bin` directory, by default `c:\Ruby\bin`, to your PATH environment variable. If you you are not sure how to do this, there are some good instructions in the Microsoft knowledge base. [2]

Checking if Ruby is installed correctly

Check your installation by running the following command from a command window:

```
ruby -v
```

If you set up everything properly and added the `bin` directory to your executable path, that command should print the version of Ruby you just installed, similar to the following:

[1] (http://rubyinstaller.org/)
[2] http://support.microsoft.com/kb/310519

```
ruby 1.8.6 (2010-02-04 patchlevel 398) [i386-mingw32]
```

For full instructions on installing a different version of Ruby, troubleshooting and advanced installation options, see the Ruby language web site.[3]

Installing Cucumber and Cuke4Nuke

Now that Ruby is installed, you can set up the Cucumber .NET integration. Cucumber talks to .NET projects using a gem called Cuke4Nuke.[4] (See the sidebar *Ruby Gems* on page 17 to learn about gems). Cuke4Nuke sets up a TCP server which will accept Cucumber commands and execute them against .NET code. In the Cucumber jargon, this way of executing is called the wire protocol. Cuke4Nuke allows you to use Cucumber for .NET projects without having any knowledge of Ruby.

To install Cuke4Nuke, add GemCutter.org to the list of allowed Ruby Gem sources by executing the following command:

```
gem sources -a http://gemcutter.org/
```

Then install the Cuke4Nuke gem by executing the following command from the terminal window:

```
gem install cuke4nuke
```

Finally, install the Win32Console gem to get colour reports, by using the following command:

```
gem install win32console
```

Cucumber colours do not show up correctly on some versions of Windows. *WAC*[5] is a workaround that enables colour reporting by using ANSI colours. Download it from the following URL:

```
http://github.com/aslakhellesoy/wac/raw/master/wac.exe
```

Put it somewhere on the disk, ideally in the executable path.

[3] http://www.ruby-lang.org/en/downloads
[4] http://github.com/richardlawrence/Cuke4Nuke
[5] http://github.com/aslakhellesoy/wac

Ruby Gems

Gems are Ruby packages for applications and libraries. You can use the RubyGems package manager (the `gem` command) to download, install, update and uninstall gems. To install a gem, use the following command:

```
gem install gem_name
```

To download and install gems, RubyGems needs to know the addresses of one or more gem repositories, called sources. You can view the currently configured sources using the following command:

```
gem sources -l
```

You should get a result like the following:

```
*** CURRENT SOURCES ***

http://gems.rubyforge.org/
```

The gem manager downloads and installs gems in a path relative to the main Ruby installation. For a standard windows installation, gems will be in the following folder:

```
C:\Ruby\lib\ruby\gems\1.8\gems\
```

For more information on the gem package manager, see the list of options using the following command:

```
gem --help
```

Installing Gems behind a firewall

If you do not have direct access to the internet, but instead go through a proxy server, you will have to set the HTTP_PROXY environment variable. Information about environment variables is available on the RubyGems site.[6]

[6] **http://docs.rubygems.org/read/chapter/12**

Verifying that Cuke4Nuke is installed correctly

To verify that the system is set up, execute `Cuke4Nuke` from the command line. You should see a help screen with Cuke4Nuke options. If you see an error that the command Cuke4Nuke isn't found, check the following:

- **There should be a `Cuke4Nuke.bat` executable in your Ruby `bin` folder. The path for this folder will be `C:\Ruby\bin` if you accepted the default installation options for Ruby.**
- **The Ruby `bin` folder should be on on your executable path. You can check this by using the command `echo %PATH%`.**
- **The Cuke4Nuke gem should exist in the gems folder. The default location is `C:\Ruby\lib\ruby\gems\1.8\gems\cuke4nuke-0.3.0`.**

Cuke4Nuke uses NUnit assertions

Cuke4Nuke uses NUnit for assertions. Most .NET developers will have some version of NUnit installed, so we won't go into the details of that here. If you do not have it, go to http://nunit.org and follow the installation instructions from that web site.

Hello World from .NET

Cuke4Nuke does not like the 4.0 framework

Your project needs to set the target framework as .NET Framework 3.5 if you are using VS2010. If you get an error message like the one below, then this is probably worth checking.

```
Unable to contact the wire server at :3901. Is it up?
```

Let's go through a quick sample project to verify that the installation works. Create a normal C# class library project (we called our example project `Cuke4NukeExample`). Add a `Features` folder to it. This is where your Cucumber

feature files will go. Add a `step_definitions` subfolder to that and create a file called `cucumber.wire` in it, with the following content:

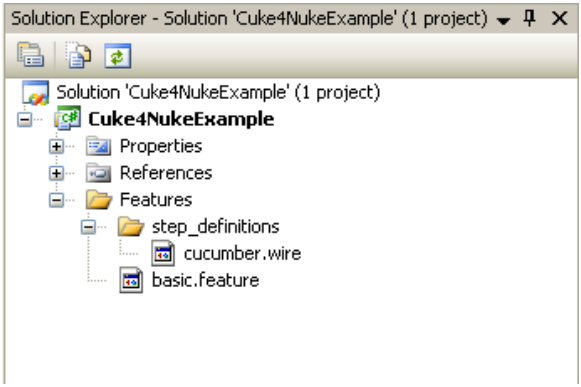
```
host: localhost
port: 3901
```

The `cucumber.wire` file tells Cucumber to use the wire protocol instead of executing steps directly. Cuke4Nuke starts on port 3901 by default, so this file makes Cucumber talk to Cuke4Nuke. Now let's add a test scenario to make sure that everything is working correctly. Create a file called `basic.feature` in the `Features` folder, with the following content:

```
Feature: Hello World
In order to ensure that my installation works
As a Developer
I want to run a quick Cuke4Nuke test

Scenario: Hello World
  Given The Action is Hello
  When The Subject is Ninja
  Then The Greeting is Hello, Ninja
```

At this point, your project structure should look similar to the one in Figure 3.1.

Figure 3.1. Cuke4Nuke Project structure



Now let's run Cuke4Nuke for the first time. Build the project, open a console window, go to your project folder (in this case `Cuke4NukeExample`) and run the following command:

```
Cuke4Nuke bin\debug\Cuke4NukeExample.dll
```

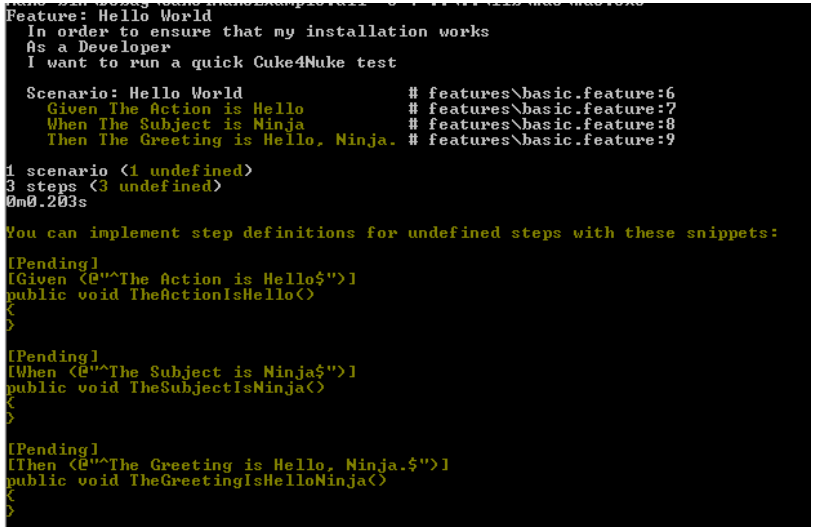
Of course, if you named your project differently, replace the DLL path accordingly.

You should see the Cucumber output telling you that there is one scenario with three steps, all of which are undefined. If you do not see the colours then add the `-c` flag and pipe the output to WAC.exe, making the command similar to the following (replace the path to wac.exe with the folder where you saved it):

```
Cuke4Nuke bin\debug\Cuke4NukeExample.dll -c | d:\apps\wac.exe
```

The result should be similar to one in Figure 3.2. As we still have not implemented any step definitions, Cuke4Nuke will not be able to match the feature file to any code. It therefore suggests some default step definitions. If these are in Ruby rather than C#, Cucumber cannot find the `cucumber.wire` file. Make sure that it is in `Features\step_definitions`.

Figure 3.2. Cuke4Nuke suggests steps



Now let's add the step definitions that help Cucumber talk to our project code. Create a C# class file called `HelloWorldSteps.cs` and edit it so it looks like the example below:

```
using System;
using Cuke4Nuke.Framework;
using NUnit.Framework;

namespace Cuke4NukeExample
{
    public class HelloWorldSteps
    {
        private string _action;
        private string _subject;

        [Given("^The Action is (.*)$") ]
        public void ActionIs(string action)
        {
            _action = action;
        }

        [When("^The Subject is (.*)$")]
        public void SubjectIs(string subject)
        {
            _subject = subject;
        }

        [Then("^The Greeting is (.*)$")]
        public void CheckGreeting(string greeting)
        {
            Assert.AreEqual(greeting,
              String.Format("{0}, {1}", _action, _subject));
        }
    }
}
```

Figure 3.3. Cuke4Nuke executes our test

```
Feature: Hello World
  In order to ensure that my installation works
  As a Developer
  I want to run a quick Cuke4Nuke test

  Scenario: Hello World                # features\basic.feature:6
    Given The Action is Hello          # Unknown
    When The Subject is Ninja          # Unknown
    Then The Greeting is Hello, Ninja # Unknown

1 scenario (1 passed)
3 steps (3 passed)
0m0.188s
```

Add a reference to NUnit.Framework.dll (from your NUnit installation) and Cuke4Nuke.Framework.dll (by default, it is installed into

C:\Ruby\lib\ruby\gems\1.8\gems\cuke4nuke-0.3.0\dotnet\). **Now build the project again, go to the console and re-run the Cuke4Nuke command. The output should now say that the steps passed, similar to the result in Figure 3.3.**

This is a very simplistic implementation for all the steps, that does not actually connect to any domain code. We go through all the details of the feature file and step definitions in Chapter 5. For now, you can probably guess that Cucumber uses regular expression in the attributes `Given`, `When` **and** `Then` **to map lines in a feature file to step definitions.**

Our simple script will set the subject and the action and then verify the greeting using the standard NUnit assertion `Assert.AreEqual`**. Just to see it when it fails, modify the** `CheckGreeting` **method or feature source so that they don't match and re-run Cuke4Nuke.**

Cuke4Nuke keeps timing out

Sometimes you might think that Cuke4Nuke has some special ninja invisibility skills, it just disappears. The reason for that is most likely that a step is taking too long to execute. Cucumber and Cuke4Nuke communicate using TCP and Cucumber only allows steps a very short time to execute by default. To increase the timeout, add a timeout section to your `cucumber.wire` file and an `invoke` subitem. Set the number of seconds allowed for a step to execute there. For example, the following file sets that to 200 seconds:

```
host: localhost
port: 3901
timeout:
  invoke: 200
```

Build integration

We can add Cuke4Nuke as a post-build step so that all the specifications get executed after every build. Open your project .csproj file and add this just above the closing tag:

```
<PropertyGroup>
  <PostBuildEvent>
       Cuke4Nuke $(TargetPath) $(ProjectDir)features -q
      </PostBuildEvent>
</PropertyGroup>
```

When you build the project, the output window should show that Cucumber tests are executed as well. You can now inspect the results of Cuke4Nuke in your output and problem windows every time a project is built.

Continuous Integration

To complete the project setup, let's run Cucumber tests within a continuous integration environment and store test outputs next to project build results. We'll use TeamCity in this example.

Cucumber and TeamCity

Setting up TeamCity is outside the scope of this tutorial, but it is fairly easy to do. Grab it from http://jetbrains.com. TeamCity 5 should support Cucumber out of the box but we have not been able to make it work. TeamCity documentation suggests that a few environment variables should do the trick for ANT/Java builds, but this does not work for Cuke4Nuke. The Cucumber Teamcity template uses a deprecated API and fails to build with the latest Cucumber version.

Cucumber can export a JUnit XML test report file, which is more than enough to get it nicely integrated with TeamCity. We'll make Cucumber save the test results into the test subfolder of our project folder, and tell TeamCity to monitor that. First, let's change the .csproj project file. Modify the `PropertyGroup` block you just added to the following:

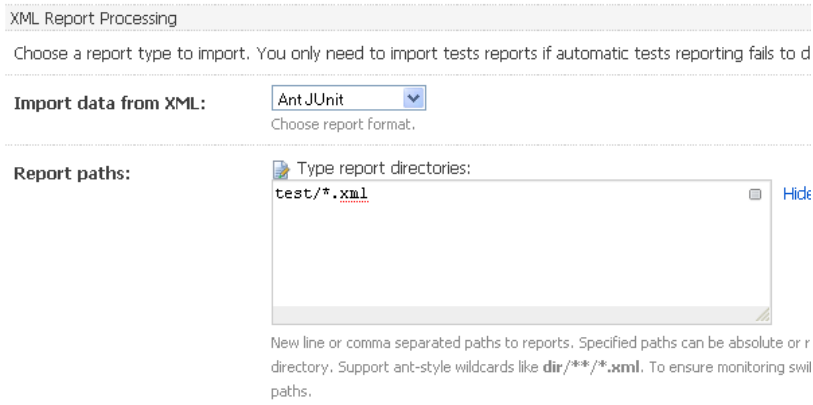
```
<PropertyGroup>
    <PostBuildEvent>cuke4nuke $(TargetPath) $(ProjectDir)features -q
     -f junit -o$(ProjectDir)test</PostBuildEvent>
    <PreBuildEvent>del $(ProjectDir)test\*.xml</PreBuildEvent>
</PropertyGroup>
```

This `-f junit` option will tell Cucumber to save test results in the JUnit format, and `-o$(ProjectDir)test` tells it to save the files in the `test` subfolder

of our project directory. The `PreBuildEvent` will delete any previous test results from that folder when the build starts.

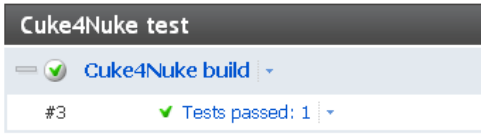
Now set up your TeamCity project as normal. On the Build Runner configuration screen, find the 'XML Report Processing' section and choose `ANT Junit` from the 'Import data from XML' list. Set `test/*xml` as the report path. (See Figure 3.4.)

Figure 3.4. Setting up XML Report Processing



Everything should now be set up. You should be able to run the TeamCity build and get the 'Tests passed:1' message in TeamCity, similar to the result show in Figure 3.5.

Figure 3.5. TeamCity reporting test passes



Debugging Cuke4Nuke steps

Because of the way that Cuke4Nuke and Cucumber run, it is virtually impossible to debug Cuke4Nuke steps properly directly from Visual Studio,

even if you set up the project to launch an external process for debugging. There are two workarounds that we use to debug steps:

1. **Call the step from a unit test, or even convert the step definition class to a unit test fixture class (Cuke4Nuke can use NUnit test methods as step definitions as long as they have the correct attributes). This allows you to execute individual steps in isolation. If you find yourself doing this too much, this means that there is a lot of logic in the steps which probably hints at the need to push some of that logic into the domain code.**
2. **The unit test approach only works for troubleshooting the logic inside a step, but it doesn't really allow you to debug a running feature file. For that, put `System.Diagnostics.Debugger.Break()` where you want to set a breakpoint and run Cuke4Nuke normally. When the CLR hits the breakpoint, it will offer to open up another Visual Studio instance and debug it. You will then be able to inspect any variables and step through the execution. Remember to increase the timeout for the wire protocol otherwise Cucumber will kill Cuke4Nuke before the debugger launches. Also remember to delete or comment out this line before committing the step definition files to the version control system, otherwise the continuous build process will start blocking and offering to debug the code as well.**

Chapter 4

Cucumber and Java

In this chapter, we cover setting up Cucumber for a Java project and integrating with a continuous build system. Although the examples are in Java, you can use absolutely the same approach to integrate Cucumber with any JVM-based language, including Groovy, Scala and Clojure.

Thanks to JRuby,[1] Cucumber can talk directly to JVM code, without a wire protocol. This simplifies the Java project setup and also means that you do not need a stand-alone Ruby installation.

What is JRuby?

JRuby is a version of the Ruby runtime environment which runs in the JVM. It differs from other Ruby like JVM based languages such as Groovy in that it aims for complete compatibility with Ruby. At the time of writing JRuby is compatible with Ruby version 1.8.7.

A ruby gem called *Cuke4Duke*[2] enables us to write step definitions in Java and other JVM languages. (See the sidebar *Ruby Gems* on page 17 to learn about gems). Cuke4Duke uses JUnit for assertions. It also relies on an inversion-of-control container for application wiring. By default, it uses *PicoContainer*[3] to inject dependencies. If you already use Spring for application wiring on your project, you can use Spring instead of PicoContainer Cucumber automation as well. Add the following JVM argument when you run tests to tell Cuke4Duke to use Spring:[4]

```
-Dcuke4duke.objectFactory=cuke4duke.internal.jvmclass.SpringFactory
```

In the rest of this chapter we are going to cover the three most common ways of using Cuke4Duke.

[1] **http://jruby.org/**
[2] **http://wiki.github.com/aslakhellesoy/cuke4duke/**
[3] **http://www.picocontainer.org/**
[4] **For more information on the Spring integration, see http://wiki.github.com/aslakhellesoy/cuke4duke/spring**

- **the section *Using JRuby directly* on page 28 covers installing JRuby and running Cuke4Duke from the terminal**
- **the section *Using ANT and Ivy* on page 31 covers using Cuke4Duke with Ant**
- **the section *Using Maven* on page 34 covers using Cuke4Duke with Maven**

Then we will create a simple Hello-World project, and finally take a look at how to include Cucumber tests into a continuous build process.

Using JRuby directly

To run Cucumber features from the console without a build system such as Ant or Maven, you'll need JRuby. If you do not already have JRuby installed, download the binary zip or tarball from the JRuby download page,[5] extract the contents into a directory and add the `bin` sub-directory of the JRuby installation to your PATH environment variable.

Maven (of course) downloads the Internet separately

Don't install JRuby yourself if you intend to use Maven, jump to the section Using Maven on page 34 and continue reading that. Maven will install JRuby in its own repository separately.

Checking whether JRuby is installed correctly

Check if JRuby is installed correctly by running the following command from a terminal window:

```
jruby -v
```

If you set up JRuby correctly and added the `bin` folder to the executable path, you should see a message similar to the following:

```
jruby 1.4.0 (ruby 1.8.7 patchlevel 174) (2009-11-02 69fbfa3)
(Java HotSpot(TM) Client VM 1.6.0) [i386-java]
```

[5] **http://jruby.org/download**

Installing the Cuke4Duke gem

Installing Gems behind a firewall

If you do not have direct access to the internet, but instead go through a proxy server, you will have to set the HTTP_PROXY environment variable. Information about environment variables is available on the RubyGems site.[6]

Install the Cuke4Duke gem for JRuby using the following command:

```
jruby -S gem install cuke4duke
```

Even if you have Cucumber installed in your stand-alone Ruby gem repository, you still need to install Cuke4Duke from JRuby. To test the installation, run the following command from a terminal window:

```
cuke4duke --help
```

This should print the available Cuke4Duke options.

Running Cuke4Duke from JRuby

Cuke4Duke has two additional options compared to Cucumber:

- **The first is `--jars` which allows you to specify the location of library classes to include. Confusingly, this option does not accept a jar file, but instead expects a directory. You can specify this option several times to include multiple directories containing dependencies.**
- **The second additional option is `--require`, which you can use to specify the root directory for your compiled classes for non-dynamic languages. For dynamic languages, put the step definitions in the `features/step_definitions` folder and they will be included automatically.**

For example, the following command will run all the feature files in the `features` folder, including all the JAR archives from the `lib` folder using the step definitions from the `target/test-classes` folder:

```
cuke4duke --jars lib --require target/test-classes features
```

[6] **http://docs.rubygems.org/read/chapter/12**

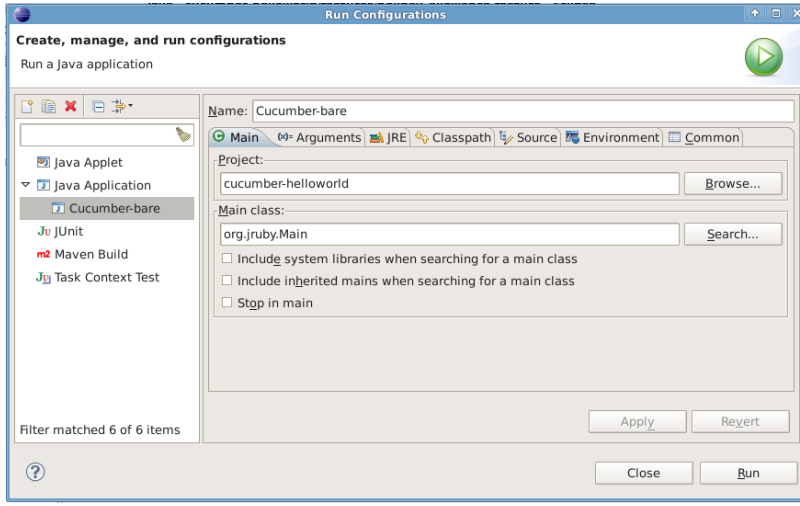
If you write the step definitions in JRuby, then you can get the same result with the following command:

```
cuke4duke --jars lib features
```

Debugging from a Java IDE

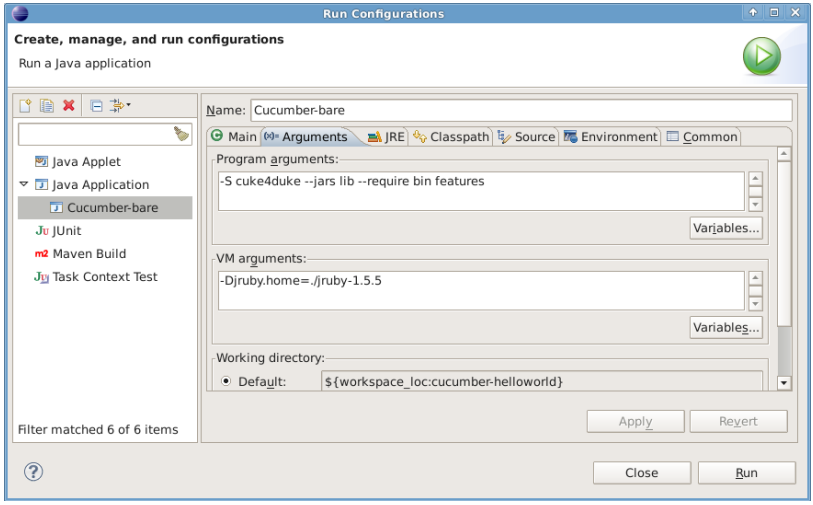
The rest of this chapter describes how to run Cuke4Duke from ANT and Maven, which makes it also easy to debug. If you decide to use JRuby to run Cucumber features directly, debugging is a bit more complex. JRuby is a Java program, so instead of launching Cuke4Duke directly, you can launch JRuby and pass the arguments you would pass from the command line. The main JRuby class is `org.jruby.Main`. For JRuby to run correctly, you should also define the JRuby root directory using the `jruby.home` JVM variable. See Figure 4.1 and Figure 4.2 for an example of how to do that in Eclipse.

Figure 4.1. Running JRuby from Eclipse



You can execute or debug this run configuration now as you would do any other Java program. Don't forget to add PicoContainer and JUnit dependencies to your classpath for Cuke4Duke to work correctly.

Figure 4.2. Setting Cuke4Duke JRuby parameters in Eclipse



Using ANT and Ivy

If you want to run Cuke4Duke via Ant, you do not have to install JRuby manually. We recommend using *Apache Ivy*[7] for dependency management with Ant (although you can still install everything by hand).

To install all the required dependencies with Ivy, you need to create the following three files in your project:

- `build.xml` is the ANT build file that specifies how you want to build and package the project
- `ivy.xml` specifies the required dependencies for the project.
- `ivysettings.xml` tells Ivy where download the dependencies from.

Project build file

Create a new ANT build file in your project folder, call it `build.xml` and paste the following content into it:

[7] http://ant.apache.org/ivy/

```
<project xmlns:ivy="antlib:org.apache.ivy.ant" name="HelloCucumber"
  default="cucumber" basedir=".">
  <property name="jruby.home" value="${basedir}/lib/.jruby"/>

  <target name="-compile-steps" depends="-define-paths">
    <mkdir dir="target/test-classes"/>
    <javac srcdir="src/test/java" destdir="target/test-classes"
      classpathref="compile.classpath" encoding="UTF-8"/>
  </target>

  <target name="cucumber" depends="-compile-steps,-install-gems"
      description="Run Cucumber">
    <mkdir dir="target/junit-report"/>
    <taskdef name="cucumber" classname="cuke4duke.ant.CucumberTask"
      classpathref="compile.classpath"/>
    <cucumber
        args="--verbose --require target/test-classes --color
      --format pretty --format junit
      --out target/junit-report features"
        objectFactory="pico">
      <classpath>
        <pathelement location="target/test-classes"/>
      </classpath>
    </cucumber>
  </target>

  <target name="-download-jars" if="ivy">
    <ivy:retrieve/>
  </target>

  <target name="-install-gems" depends="-define-paths" if="gems">
    <taskdef name="gem" classname="cuke4duke.ant.GemTask"
       classpathref="compile.classpath"/>
    <gem args="install cuke4duke --version 0.3.2
         --source http://gemcutter.org/"/>
  </target>

  <target name="-define-paths" depends="-download-jars">
    <path id="jruby.classpath">
      <fileset dir="lib">
        <include name="**/*.jar"/>
      </fileset>
    </path>

    <path id="compile.classpath">
      <fileset dir="lib">
```

```
        <include name="**/*.jar"/>
      </fileset>
    </path>
  </target>

  <target name="clean" description="Delete all generated artifacts">
    <delete dir="${basedir}/target"/>
  </target>

  <target name="clean-deps" description="Delete all dependencies">
    <delete dir="${basedir}/lib/.jruby"/>
    <delete>
      <fileset dir="${basedir}/lib" includes=".jar"/>
    </delete>
  </target>

</project>
```

There are two tasks specific to Cuke4Duke in this build file example:

- **The `<gem>` tag downloads the Cuke4Duke ruby gem**
- **The `<cucumber>` tag executes the tests to validate Cucumber feature files**

Project dependencies

Create a file called `ivy.xml` in your project folder which instructs Ivy to download Cuke4Duke and its dependencies. It should have the following content:

```
<ivy-module version="2.0">
  <info organisation="cukes.info" module="java-example"/>
  <dependencies>
    <dependency org="cuke4duke" name="cuke4duke" rev="0.3.2"/>
    <dependency org="org.jruby" name="jruby-complete"
       rev="1.5.1" transitive="false"/>
    <dependency org="org.picocontainer" name="picocontainer" rev="2.10.2"/>
    <dependency org="junit" name="junit" rev="4.8.1"/>
  </dependencies>
</ivy-module>
```

Ivy settings

Create a file called `ivysettings.xml` to tell Ivy how to download the dependencies:

```
<ivysettings>
  <settings defaultResolver="all-repos"/>
  <resolvers>
    <chain name="all-repos">
      <url name="cukes" m2compatible="true">
        <artifact pattern=" http://cukes.info/maven/[organisation]/
 [module]/[revision]/[artifact]-[revision].[ext]"/>
      </url>
      <ibiblio name="ibiblio" m2compatible="true" usepoms="false"/>
    </chain>
  </resolvers>
</ivysettings>
```

Running Cucumber through ANT

To compile and run the cucumber feature use the following command:

```
ant -Divy=true -Dgems=true
```

Using Maven

If you are using Maven, you will need to set up the `pom.xml` project file to include Cuke4Duke and install the required gems. and java dependencies. Setting up Maven is outside the scope of this tutorial, but you will find the full working example of the POM file on the companion web site of this book.[8]

Setting up the POM file

Add the following repositories to the POM file:

```
    <repositories>
        <repository>
            <id>codehaus</id>
            <url>http://repository.codehaus.org</url>
        </repository>
        <repository>
            <id>cukes</id>
            <url>http://cukes.info/maven</url>
        </repository>
    </repositories>
```

[8] **http://cuke4ninja.com**

Next, add a plugin repository, so that Maven can get hold of the cuke4duke-maven-plugin.

```
    <pluginRepositories>
        <pluginRepository>
            <id>cukes</id>
            <url>http://cukes.info/maven</url>
        </pluginRepository>
    </pluginRepositories>
```

Add the following dependencies to the `dependencies` **block:**

```
    <dependencies>
        <dependency>
            <groupId>org.picocontainer</groupId>
            <artifactId>picocontainer</artifactId>
            <version>2.10.2</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>cuke4duke</groupId>
            <artifactId>cuke4duke</artifactId>
            <version>0.3.2</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.8.1</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
```

Finally, configure the Cuke4Duke plugin:

```
<plugins>
 <plugin>
  <groupId>cuke4duke</groupId>
  <artifactId>cuke4duke-maven-plugin</artifactId>
  <configuration>
    <jvmArgs>
      <jvmArg>
      -Dcuke4duke.objectFactory=cuke4duke.internal.jvmclass.PicoFactory
      </jvmArg>
      <jvmArg>-Dfile.encoding=UTF-8</jvmArg>
    </jvmArgs>
```

```
    <!-- You may not need all of these arguments in your
      own project. We have a lot here for testing purposes... -->
    <cucumberArgs>
        <cucumberArg>--backtrace</cucumberArg>
        <cucumberArg>--color</cucumberArg>
        <cucumberArg>--verbose</cucumberArg>
        <cucumberArg>--format</cucumberArg>
        <cucumberArg>pretty</cucumberArg>
        <cucumberArg>--format</cucumberArg>
        <cucumberArg>junit</cucumberArg>
        <cucumberArg>--out</cucumberArg>
        <cucumberArg>${project.build.directory}/cucumber-reports</cucumberArg>
        <cucumberArg>--require</cucumberArg>
        <cucumberArg>${basedir}/target/test-classes</cucumberArg>
    </cucumberArgs>
    <gems>
        <gem>install cuke4duke --version 0.3.2</gem>
    </gems>
  </configuration>
  <executions>
    <execution>
      <id>run-features</id>
      <phase>integration-test</phase>
      <goals>
          <goal>cucumber</goal>
      </goals>
    </execution>
  </executions>
 </plugin>
</plugins>
```

You can control how Cuke4Duke runs and saves its results with the arguments in the `cucumberArg` **tags. For example, the following arguments:**

```
<cucumberArg>--out</cucumberArg>>
<cucumberArg>${project.build.directory}/cucumber-reports</cucumberArg>
```

are equivalent to the following command line:

```
cuke4duke --out target/cucumber-reports
```

Running Cuke4Duke from Maven

The example setup from the previous section binds Cuke4Duke to the `integration-test` Maven lifecycle. To run Cucumber tests using Maven, execute the following command line:

```
mvn clean integration-test
```

That command cleans the output folder, re-compiles the projects and then reruns the test. While this is the best way to ensure the tests are running against the latest code base, it will take longer to run. If you are not making changes to classes, you can run the test using the following command:

```
mvn cuke4duke:cucumber
```

Installing gems

The first time you run Cucumber with Maven, you will have to specify one additional arguments so that JRuby downloads and installs the required gems. Run Maven as:

```
mvn integration-test -Dcucumber.installGems=true
```

You only need to do this once on any particular system.

Hello world from Java

Let's go through a quick sample project to verify that the installation works. We will use Maven to run the project. If you would prefer to use Ant or JRuby directly rather than Maven, then set up the project as described in the previous section. You can download the full project code for both Ant and Maven from the *companion web site*[9] of this book.

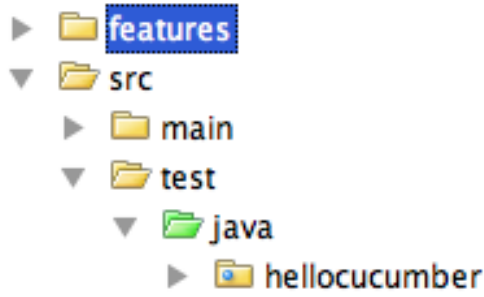
Project Setup

Set up a new project using the standard project layout and add a `features` sub-directory to the project directory. This is where your feature files will

[9] **http://cuke4ninja.com**

go. For a Maven project, the final layout will be as in Figure 4.3. Make sure that `src/test/java` is marked as a test source directory in your IDE.

Figure 4.3. Maven project directory structure



Adding the feature file

Create a file in the `features` folder in your project called `basic.feature` and add the following content:

```
Feature: Hello World Feature
        In order to ensure that my installation works
        As a Developer
        I want to run a quick Cuke4Duke test

        Scenario: Hello World Scenario
                Given The Action is Hello
                When The Subject is Ninja
                Then The Greeting is Hello, Ninja.
```

Now run Cucumber. Open a terminal window and execute the following command from the project folder:

```
mvn integration-test -Dcucumber.installGems=true
```

Of course, if you decided to use Ant or JRuby directly, use the commands described in the previous sections to execute the tests.

Near the end of the output of the maven build report in the terminal window you should see something like the result in Figure 4.4. Below that, Maven build result shows a suggested implementation for the step definitions, such

as in Figure 4.5. This shows that Cuke4Duke is running and has identified the feature file. As we have not yet created any step definitions, the test execution will fail. It will however do three three important things:

- install the cuke4duke gem and its dependencies
- confirm that cuke4duke is installed correctly and that it can find the feature file as expected
- provides an example feature step definitions in Java

Figure 4.4. Cuke4Duke finds the feature file

```
[INFO] Feature: Hello World Feature
[INFO]   In order to ensure that my installation works
[INFO]   As a Developer
[INFO]   I want to run a quick Cuke4Duke test
[INFO]
[INFO]   Scenario: Hello World Scenario       # features/basic.feature:6
[INFO]     Given The Action is Hello          # features/basic.feature:7
[INFO]     When The Subject is Ninja          # features/basic.feature:8
[INFO]     Then The Greeting is Hello, Ninja. # features/basic.feature:9
[INFO]
[INFO] 1 scenario (1 undefined)
[INFO] 3 steps (3 undefined)
```

Figure 4.5. Cuke4Duke suggests step definitions

```
[INFO] You can implement step definitions for undefined steps with these snippets:
[INFO]
[INFO] @Given ("^The Action is Hello$")
[INFO] @Pending
[INFO] public void theActionIsHello() {
[INFO] }
[INFO]
[INFO] @When ("^The Subject is Ninja$")
[INFO] @Pending
[INFO] public void theSubjectIsNinja() {
[INFO] }
[INFO]
[INFO] @Then ("^The Greeting is Hello, Ninja\.$")
[INFO] @Pending
[INFO] public void theGreetingIsHello,Ninja.() {
[INFO] }
```

Writing step definitions

Now let's add the step definitions that help Cucumber talk to our project code. Add a class called `BasicFeature` into the `test/java/hellocucumber` directory, with the following content:

```
package hellocucumber;

import cuke4duke.annotation.I18n.EN.Given;
import cuke4duke.annotation.I18n.EN.Then;
import cuke4duke.annotation.I18n.EN.When;
import static junit.framework.Assert.assertEquals;

@SuppressWarnings({"UnusedDeclaration"})
public class BasicFeature {
    private String action;
    private String subject;
    @Given("^The Action is ([A-z]*)$")
    public void theActionIs(String action) {
        this.action = action;
    }
    @When("^The Subject is ([A-z]*)$")
    public void theSubjectIs(String subject) {
        this.subject = subject;
    }
    @Then("^The Greeting is ([^\\.]*).$")
    public void theGreetingIs(String greeting) {
        assertEquals(String.format("%s, %s", action, subject), greeting);
    }
}
```

Now build the project again and re-run Cuke4Duke, using the following command:

```
mvn clean integration-test
```

Note that we don't need the `-D` any more because the gems should now be installed. The output should now say that the steps passed, similar to the result in Figure 4.6.

This is a very simplistic implementation for all the steps, that does not actually connect to any domain code. We go through all the details of the feature file and step definitions in Chapter 5. For now, you can probably

guess that Cucumber uses regular expression in the attributes `Given`, `When` and `Then` to map lines in a feature file to step definitions.

Figure 4.6. The feature passes

```
[INFO]
[INFO] Feature: Hello World Feature
[INFO]   In order to ensure that my installation works
[INFO]   As a Developer
[INFO]   I want to run a quick Cuke4Duke test
[INFO]
[INFO]   Scenario: Hello World Scenario       # features/basic.feature:6
[INFO]     Given The Action is Hello          # BasicFeature.theActionIs(String)
[INFO]     When The Subject is Ninja          # BasicFeature.theSubjectIs(String)
[INFO]     Then The Greeting is Hello, Ninja. # BasicFeature.theGreetingIs(String)
[INFO]
[INFO] 1 scenario (1 passed)
[INFO] 3 steps (3 passed)
[INFO] 0m0.160s
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 15 seconds
[INFO] Finished at: Wed Sep 08 10:21:35 BST 2010
[INFO] Final Memory: 28M/81M
[INFO] ------------------------------------------------------------------------
```

Maven may need some TLC

Depending on the IDE you are using, you may need to process the `pom.xml` file we updated earlier to import the dependencies.

Continuous integration

We are only going to cover setting up TeamCity with a Maven base project in detail here, but you should be able to configure other CI systems similarly.

Cucumber can write out the results of the tests in a format that your CI system can process. In the section *Using Maven* on page 34 we used `cucumberArgs` to set configure the output format:

```
<cucumberArg>--format</cucumberArg>
<cucumberArg>junit</cucumberArg>
<cucumberArg>--out</cucumberArg>
<cucumberArg>target/cucumber-reports</cucumberArg>
```

These arguments instructed Cucumber to output the test results in JUnit format to the `target/cucumber-reports` directory. When we ran Cucumber, it created a file like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuite errors="0" failures="0"
   name="Hello World Feature" tests="1" time="0.025000">
<testcase
  classname="Hello World Feature.Hello World Scenario"
  name="Hello World Scenario" time="0.025000">
</testcase>
</testsuite>
```

Most CI servers should be able to parse these results and display them. As an example, we are going to set up a TeamCity job to run the `integration-test` target and report the results.

Setting up CI using TeamCity

Setting up TeamCity is outside the scope of this tutorial, but it is fairly easy to do. Grab it from http://jetbrains.com. It is free to use for up to 20 build configurations.

Add the project you created in sec_java_helloworld to TeamCity as a Maven 2 project. On page that configures the Build Runner, select 'Maven 2' from the dropdown and enter `clean integration-test` as goals, as in Figure 4.7. Further down the page, set the command line parameters to `-Dcucumber.installGems=true` as in Figure 4.8.

Figure 4.7. Selecting Maven2 as the build runner

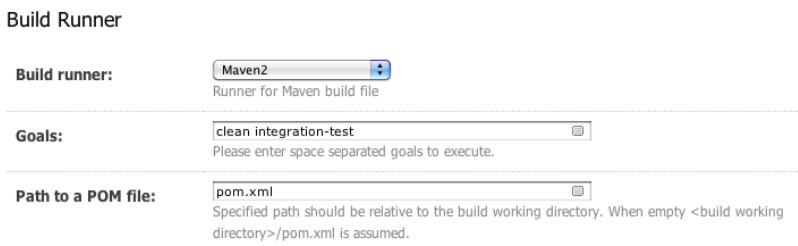
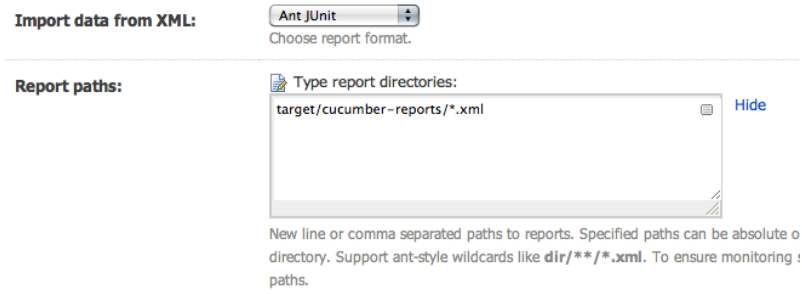


Figure 4.8. Setting command line parameters so gems are downloaded

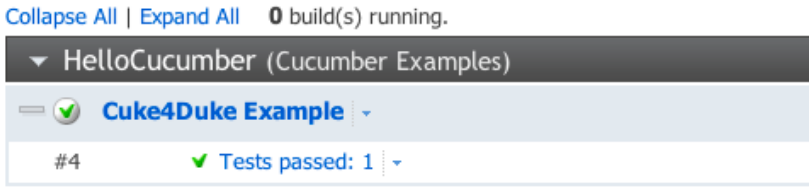
At the bottom of the page, in the 'Import data from XML' section, select 'Ant JUnit' as a report type to import and enter `target/cucumber-reports/*.xml` as the report path, as in Figure 4.9. That is about it! Save the page and scroll to the top, where you should see a run button. Once the project is running, if you select the projects tab, you should see the build happening. This may take a while the first time, since maven in its usual fashion will insist on downloading many libraries.

Figure 4.9. Setting up build report type



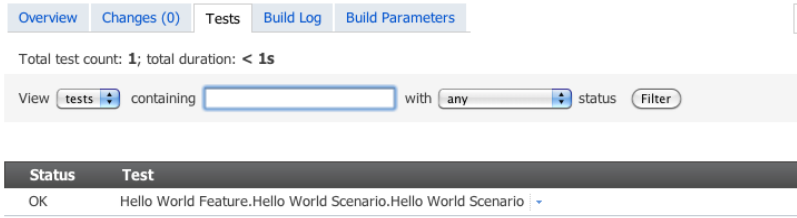
Once the build is complete it should show that one test passed, as in Figure 4.10.

Figure 4.10. Test results in TeamCity



You can drill down into the result to see the test name, as with a normal JUnit test. For an example, see Figure 4.11.

Figure 4.11. Test names in TeamCity



Part II. Gherkin

In this part, we go through the Gherkin language in more detail. We explain:

- **How Cucumber interprets feature files**
- **How to write good feature files**
- **How to implement step definitions in Java, .NET and Ruby**
- **How to handle complex scenarios efficiently**

Chapter 5

Cucumber feature files

In this chapter, we look into the basic structure of a Cucumber feature file. You will learn:

- The basic elements of feature files
- How to write some basic scenarios so that Cucumber can understand them
- What makes a good feature file

Cucumber jargon

We all know that Ninjas all speak Japanese fluently, but this book is in English. To ensure that nobody gets killed by mistake by saying a wrong word, here are some of the basic phrases that you need to memorise:

Stakeholder

A Stakeholder is a person who gets some value out of the system. Typical stakeholders are representative of groups of users, for example a trader or an administrator. Some teams describe stakeholders with user personae (explained nicely in User Stories Applied[7]). Personae are particularly useful for web based projects as they encourage you to think about how a typical user is likely to interact with the system— for example look at a monitor in despair.

Feature

A Feature is a piece of system functionality that delivers value to one or more stakeholders. Executable specifications work best when described from the perspective of stakeholders, not technical infrastructure, since doing so enables business users to understand them better and engage more in discussions about what makes a particular feature complete.

Features are normally vertical functional slices of the system, for example credit card processing. Horizontal infrastructural slices, for example database caching, are not good as features. They do not deliver value directly to any stakeholders. Stakeholders will not be able to engage properly in specifying the acceptance criteria for such functionality, pretty much defeating the point of specification by example.

User story

A User story is a rough description of scope for future work, used as a planning tool. Stories are vertical slices of system functionality that are deliverable independently. Each story should specify the value it brings to one or more stakeholders, the stakeholders that care about the story and the scope of the intended work. A common format for the stories is:

In order to..., as a ..., I want ...

There are alternative formats, such as:

As a ..., I want..., So that...

Many a ninja has died in the futile format wars and some silly shoguns will try to convince you that one way of telling stories is significantly better than the other one. Ninja cucumber disagrees! For the purposes of this document all these formats are equivalent and we won't enter into a discussion on which one is better.

Stories often impact several features — a payment story might have an impact on card processing, fraud control and back-office reporting. Stories are often relatively small chunks of work to support frequent delivery. A single feature might be delivered through a large number of stories.

Stories are more than just the format

Many junior ninjas confuse stories with the cards that they are written on. Here is an example of a good user story:

> In order to reduce fraud, as a financial controller I want the system to automatically send Chuck Norris to beat up suspected fraudsters.

This story clearly states who cares about some functinality, why it is important and what it delivers. This provides enough information for a meaningful discussion on the specifications when the time comes to implement it. Here is an example of a bad user story:

> As a trader I want to enter a trade because I want to trade

Although this story obeys the form, it is too broad, too generic and it does not state a clear benefit. It would be much better to specify a full flow of activities that provides some end-to-end value to a stakeholder. We would normally look for thin slices of functionality, for example not supporting all possible trade types or trade workflows from the start but only one type. In this way, once a story is delivered some stakeholders can actually start getting the value from it.

Feature file

A Feature file describes a feature or a part of a feature with representative examples of expected outcomes. Cucumber uses these files to validate some system functionality against its specifications. Feature files are plain-text files, ideally stored in the same version control system as the related project.

Here is an example of a Cucumber feature file:

```
Feature: Fight or flight
  In order to increase the ninja survival rate,
  As a ninja commander
  I want my ninjas to decide whether to take on an
  opponent based on their skill levels

  Scenario: Weaker opponent
    Given the ninja has a third level black-belt
    When attacked by a samurai
    Then the ninja should engage the opponent

  Scenario: Stronger opponent
    Given the ninja has a third level black-belt
```

```
When attacked by Chuck Norris
Then the ninja should run for his life
```

Later in this chapter we cover how Cucumber interprets this file.

Feature file extension

Feature files should have a `.feature` extension

Key Examples

Each feature should be illustrated with Key Examples. The examples show the expected outcomes in specific representative cases with very precise information. This makes it absolutely clear what the system is expected to do and helps to avoid any misunderstanding and ambiguity. It also makes it possible for an automation tool, such as Cucumber, to check whether the system works according to the specification.

Scenario

A Scenario captures one key example for a feature in the feature file. It represents a way that the system delivers some value to a stakeholder. Scenarios are units of specifications for Cucumber. They allow us to chop up feature functionality like Sushi, into chunks that can be separately consumed. Examples of good scenarios for credit card processing might be successful transaction authorisation, transaction failure due to lack of funds, transaction failure due to wrong verification code and so on.

Step

Steps are domain language phrases that we can combine to write scenarios. They can either refer to the context of a scenario, an action that the scenario describes or a way to validate the action. The steps that define the context of a scenario generally begin with the Given keyword, for example:

Given the ninja has a third level black-belt

The steps that define an action generally begin with the When keyword, for example:

When attacked by a samurai

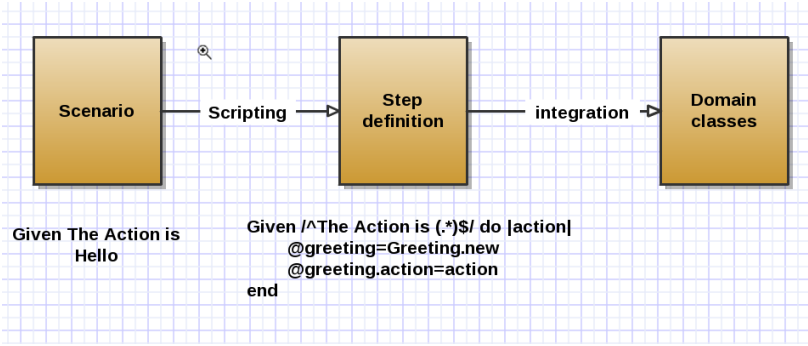
The steps that define an expected outcome generally start with the Then keyword, for example:

Then the ninja should engage the opponent

Step Definition

Step Definitions are reusable automation components that execute individual steps. They tell Cucumber what to do when interpreting a particular step (see Figure 5.1). Programmers or automation specialists write step definitions where necessary when implementing scenarios to validate that they work correctly. Once a step definition is implemented, the step can be reused in other scenarios. We cover how to write step definitions in Ruby, .NET and Java in the following chapters.

Figure 5.1. Cucumber step definitions automate validation



Gherkin

Gherkin is the language for describing Cucumber feature files. It is also the name of a separate piece of software that interprets that language.

Gherkin, the language, defines the structure of a feature file and the keywords that can be used to describe features. Gherkin, the software, parses files for Cucumber and related projects, such as Specflow.[1] You can use it to implement your own version of Cucumber for a platform that is not currently supported, if you can find one.

[1] http://specflow.org a native .NET implementation of a Gherkin-compliant automation tool

How Cucumber interprets feature files

Let's look into how Cucumber interprets files. Here is the previous example again, with italics added to point out the keywords (Cucumber files are normally just plain text):

```
Feature: Fight or flight
  In order to increase the ninja survival rate,
  As a ninja commander
  I want my ninjas to decide whether to take on an
  opponent based on their skill levels

Scenario: Weaker opponent
    Given the ninja has a third level black-belt
    When attacked by a samurai
    Then the ninja should engage the opponent


Scenario: Stronger opponent
    Given the ninja has a third level black-belt
    When  attacked by Chuck Norris
    Then  the ninja should run for his life
```

Each feature file starts with the `Feature` keyword (if in English, we'll deal with other languages later). The keyword is followed by a colon and the feature name. In this case, the feature name is:

> ***Fight or flight***

The feature name helps a reader understand what the file is about in general.

A good technique for deciding on a title is to think about what you would type into Google to search for this file if it were online.

The paragraph which follows the feature title is a free-form text that describes the intention of the feature. In this case, the feature description is:

> ***In order to increase the ninja survival rate, As a ninja commander I want my ninjas to decide whether to take on an opponent based on their skill levels***

This paragraph should give the reader a context to understand the rest of the file. Describing features in the user story format is a common practice, although they do not necessarily map directly to user stories. To describe a feature, name the stakeholder, the benefit and the intended solution. A description in the user story format also helps people decide later whether a new scenario belongs to a particular feature file or not.

A good strategy for describing a feature is to write the scenarios first, then summarise them and explain to someone who hasn't seen them before. Capture the explanation you used and put it into the header file. This will ensure that the feature file is self-explanatory.

One or more scenarios follow the introductory paragraph, each starting with the `Scenario` keyword, followed by a colon and a scenario title on the first line. In this example, the scenarios are:

Weaker opponent

and

Stronger opponent

A scenario title provides a reader with the context required to understand the key example described by the scenario. A good strategy to use when deciding on a scenario title is to try to summarise the intent of an example covered by the scenario to another person, and capture the phrase you used to explain it.

Each scenario is described with one or more steps. In this example, the steps of the second scenario are:

Given the ninja has a third level black-belt

When attacked by Chuck Norris

Then the ninja should run for his life

Feature files are your katanas

Ninja always knows where his katana is. You should know the same for your feature files. They

form a living documentation of a software system. They allow developers to understand what needs to be implemented, testers to understand what is being automatically checked and business users and support staff to quickly discover exactly what the system does. They are not just test scripts. Feature files should be kept safe and secure as much as the system code. When the code is branched, tagged or merged, these specifications should follow. The easiest way to do that is to keep them in the same version control system as the code.

What makes a good feature file

The long term benefit of using Cucumber comes from living documentation, a source of information on system functionality that is easily accessible to everyone and always current. To achieve that, a feature file should be:

- **Easy to understand**
- **Consistent**
- **Easy to access**

To serve as a target for development that prevents misunderstanding and ambiguities, scenarios in a feature file should be:

- **Precise and testable**
- **Specifications, not scripts**
- **About business functionality, not about software design**

Once a feature is implemented, the related file becomes a living document for the feature. To allow the software to evolve and provide functional regression checks, the scenarios should be:

- **Self explanatory**
- **Focused**
- **In domain language**

Write your feature files as if they will be read by an angry Chuck Norris looking for an excuse to beat you up if they are hard to understand. Think

about them as documentation that will be read much more than it will be changed. Consider that other people will need to read and understand it months or years after you write it. Here are some things to watch out for:

- Very long files or scenarios: these are hard to understand, so break them apart or try to describe the feature from a higher level of abstraction
- Technical automation concepts: these belong in step definitions (code), not scenarios (human-readable text). Mixing the two makes the scenarios harder to understand and change.
- Generic values in examples: For example 'Given a card with a valid number' where valid is not precisely defined somewhere else, or 'Then the system should work correctly'. Generic values can be ambiguous and cause misunderstanding. Examples should be realistic and very precise. Use an example of a valid number (and specify another scenario that illustrates an invalid number). Define what 'correctly' means by illustrating it with the precise output.
- Inconsistent language: As the project evolves, the domain language might change. Cucumber feature files are your documentation, update them to be consistent with the new knowledge or people won't be able to understand them later.
- Several actions or mixed validations and actions: although a scenario that checks several cases makes a perfectly good regression check, it is really hard to understand. Each scenario should be independent, focused on a single example and perform a single action. That will make it easier to understand the scenarios later. It also enables developers and testers to get quick feedback from individual checks.
- Generic feature explanations or titles of features and scenarios that do not explain the content of the file completely: Feature files should be self-explanatory. A good litmus-test for this is to show the feature file to someone who hasn't seen it and ask them to explain it back to you. If they understand it differently than you, then the descriptions are wrong.

Feature files should be written collaboratively

Many beginners fail with specification by example by expecting the business analysts or business users to write feature files. You can see people often complaining how Cucumber doesn't work for them because their business analysts don't want to write feature files. This is a huge misunderstanding

of the process and even if you can get them to write the files on their own, avoid doing that. Feature files should be a universally accepted definition of what done means for a particular story, from all the perspectives. That is why they need to be specified and written collaboratively. Everyone should contribute. A good feature file contains the following key examples:

- A representative example illustrating each important aspect of business functionality. Business users, analysts or customers will typically define these.
- An example illustrating each important technical edge case, such as technical boundary conditions. Developers will typically suggest such examples when they are concerned about functional gaps or inconsistencies. Business users, analysts or customers will define the correct expected behavior.
- An example illustrating each particularly troublesome area of the expected implementation, such as cases that caused bugs in the past and boundary conditions that might not be explicitly illustrated by previous examples. Testers will typically suggest these and business users, analysts or customers will define the correct behavior.

Structuring scenarios

Although there is no syntax check to prevent misuse, the steps in a scenario should follow the Given-When-Then flow, which means that the entire context comes first, then an action, and then the expected outcome of the action. Structuring scenarios like this ensures that they are focused and makes them easier to understand.

Given

A `Given` step defines the preconditions for a scenario, similar to how Chunk Norris showing up is a precondition for a good fight. Technically, the step definition that automates it should put the system into a known state for an automated test.

`Given` steps should describe the state of the world before an interesting action. Avoid writing them as actions, but instead explain the state. For example, it is much better to write:

```
Given a ninja is in the room
```

than

```
Given a ninja enters a room
```

This distinction in the language might not seem so important initially, but it helps to focus readers' attention on the important activity of the scenario and avoid falling into the trap of scripting.

When

A `When` step describes the primary action of a scenario — in case of Chuck Norris almost guaranteed to involve some kicking. Some good examples of the primary action are:

- **An activity of a user**
- **A domain event**
- **A system function**

Try to write `When` steps as actions, something that happens. This will focus the attention of the reader on that. It will also help developers understand where to put the business logic and how to call the appropriate code modules to implement the functionality.

Then

A `Then` step specifies tears, black eyes and such. They are post-conditions, expected observable outcomes of the primary action in the given context. It is important that these are observable outcomes, something that the business users can actually understand and see. Otherwise, they won't be able to engage in specifying it. Internal state changes, though they may be easy to test, do not guarantee that the user of the system benefits in any way from the feature.

But I want to verify internal changes

Verifying internal changes might be very important from a technical perspective, but the business users often couldn't care less about that. Use a technical testing tool (such as your favourite unit testing tool) to specify and verify such things. You'll get a much faster turnaround from a technical perspective. You don't really need a test that is readable and accessible by business users for

that, so Cucumber won't give you any real benefits for those checks.

You can use three double-quote symbols (`"""`) to start and end a multi-line block. For example: `Given a ninja has the following hit-list """ 5 Samurai 10 Peasants 1 Shogun 0 Chucks """`

The difference between When and Then

Spoken languages have quite a lot of ambiguity and flexibility. The same thing can be said in lots of different ways, which often confuses people when they start with Gherkin. For example, we can say:

> ***Given the opponent is Chuck Norris, then the ninja should run for his life***

Using `When` instead of `Given` doesn't change the meaning at all:

> ***When the opponent is Chuck Norris, then the ninja should run for his life***

Syntax error in the previous example

Those among you who read this carefully will surely know that it is a syntax error to use Chuck Norris in a `Given` step, as he does not take orders from anyone. The current version of Cucumber has a bug and does not report this as a syntax error but this will probably be fixed in the future.

In Gherkin, however, there is a huge difference. `Given` steps describe the world as it is before an action happens, and `When` describes the action that is played out in a scenario. Think of it this way: each scenario explains an action. A context for the action (`Given`) is optional, but the action (`When`) is mandatory.

Internationalisation

One of the biggest advantages of Cucumber over other literal automation tools is that it fully supports internationalisation. You do not have to use keywords in English. You can write the feature file in around 40 languages, including some very obscure such as LOLCATS and artificial languages such as Welsh. Here is, for example, the feature file from the introduction in French:

```
Fonctionnalité: Lutte ou de fuite
    Afin d'augmenter le taux de survie ninja,
    Tant que commandant ninja
    Je veux que mon ninjas de décider de prendre un adversaire
      en fonction de leurs niveaux de compétences

Plan du Scénario: ennemi plus faible
    Soit le Ninja a un troisième niveau ceinture noire
    Lorsqu'il est attaqué par un samouraï
    Alors le ninja doit engager l'adversaire

Plan du Scénario: ennemi plus fort
    Soit le Ninja a un troisième niveau ceinture noire
    Lorsqu'il est attaqué par Jean-Claude Van Damme
    Alors le ninja doit fuir pour sauver sa vie
```

See the *Cucumber wiki examples*[2] for more examples in all the supported languages.

Tagging for fun and profit

After a few months, scenarios and features in a project can become unwieldy. With hundreds of feature files, we might not want to run everything all the time. It is often useful to be able to quickly find or check just a subset of the features, for example skip running one or two very slow tests to get faster feedback.

Cucumber allows us to manage large groups of scenarios easier with tags. Tags are free-form text comments assigned to a scenario. They provide meta-

[2] **http://github.com/aslakhellesoy/cucumber/blob/master/examples/i18n**

data about the type, nature or group of that scenario. You can assign tags to a scenario by putting it before the scenario header, adding an 'at' sign (@) before the tag name. For example:

```
@slowtest
Scenario: end-to-end signup
...
```

You can execute only the scenarios with a particular tag using the `--tags` option, for example:

```
cucumber --tags @fasttest
```

Another option is to skip only the scenarios without a particular tag. For that, prefix the tag with a tilde (~). Here is an example:

```
cucumber --tags ~@slowtest
```

Here are some useful things you can do with tags:

- **Identify work in progress to avoid breaking the build when things like that fail.**
- **Separate quick and slow checks and execute as cascaded builds on the build server to speed up feedback.**
- **Execute some checks only overnight.**
- **Cross-reference stories and features/scenarios.**
- **Cross-reference entire feature areas and scenarios with cross-cutting concerns.**

You can tag a feature or a scenario (or outline). Feature tags are automatically inherited by all the enclosed scenarios and scenario outlines. Cucumber also supports some more advanced tag features, such as enforcing the limit on the number of scenarios with a particular tag (useful for in-progress work) and running scenarios that satisfy a combination of tags. For information on those options, have a look at the *tags page on the Cucumber wiki*[3].

[3] **https://github.com/aslakhellesoy/cucumber/wiki/tags**

Remember

- **Describe features from the perspective of users and stakeholders, not technical tasks**
- **Think of feature files as documentation when you write them.**
- **Define the examples for feature files collaboratively.**
- **Given steps describe the world as it was before an action. When steps describe activities that cause the world to change. Then steps specify how the world shuld change.**

Chapter 6

Implementing the steps

To give aspiring Cucumber ninjas more examples to copy and paste while they are practicing their new skills, we now implement step definitions from Chapter 5. We present the most important snippets of the code to help you focus on the key parts. You can download the entire source code with all the additional project files from http://cuke4ninja.com.

The basic feature file — again

Create a new directory for the project, and add two sub-directories to it: `features` **and** `src`**. The feature files, unsurprisingly, go to the** `features` **subdirectory. We use the** `src` **subdirectory for the implementation.**

Create a new file in the `features` **(for example,** `ninja_survival_rate.feature`**) with the following content:**

```
Feature: Fight or flight
  In order to increase the ninja survival rate,
  As a ninja commander
  I want my ninjas to decide whether to take on an
  opponent based on their skill levels

  Scenario: Weaker opponent
    Given the ninja has a third level black-belt
    When attacked by a samurai
    Then the ninja should engage the opponent

  Scenario: Stronger opponent
    Given the ninja has a third level black-belt
    When attacked by Chuck Norris
    Then the ninja should run for his life
```

Steps and regular expressions

Cucumber uses regular expressions to match steps in the feature file to step definitions in code. We really need only three step definitions — one that sets the ninja belt level at the start, one that captures what the ninja does when attacked by an opponent and one that checks if the specified action was in that list of activities.

A very quick introduction to regular expressions

Regular expressions are a way to specify patterns of text. If you have never used them before, here are just a few basic rules to follow.

- **Letters and numbers have literal values. The `ninja` pattern only matches the exact lowercase sequence n,i,n,j,a.**
- **Square brackets define groups of interchangeable elements. The whole group will match a single character. So `[Nn]inja` matches either n,i,n,j,a or N,i,n,j,a but not N,n,i,n,j,a.**
- **An asterisk turns the character or group immediately before it to a sequence that repeats zero or more times. So `Nin*ja` will match N,i,n,j,a but also N,i,j,a and N,i,n,n,n,n,j,a.**
- **A question mark turns the character or group immediately before it into a sequence that appears zero or one time. So `N?inja` will match N,i,n,j,a and i,n,j,a, but not N,N,i,n,j,a**
- **A dash defines a range. For example, `ninja1-9` will match n,i,n,j,a,1 but also n,i,n,j,a,2 and n,i,n,j,a,3 to n,i,n,j,a,9.**
- **A dot (`.`) matches any character. So `Ninj.` will match any sequence of five characters starting with N,i,n,j.**
- **A backslash `\` turns a special character into a literal value. For example, `1\-9` matches only the sequence 1,-,9, not a number between 1 and 9.**
- **`\s` matches a whitespace (blank).**
- **You can mix and match ranges, asterisks, letters...**
- **A caret (^) matches the beginning of the line. A dollar sign ($) matches the end of the line.**
- **Parenthesis () mark parts of an expression that is captured for later processing.**

Initialising the context

To set the black belt level, we create a step definition which matches the lines such as:

Given the ninja has a third level black-belt

and capture the word between 'a' and 'level' as an argument. We can use a dash to define a range of characters. So `a-z` will match any lowercase letter. An asterisk after any character says that we expect a sequence. We use square brackets to apply an asterisk to the entire range, not just the letter `z`. We add brackets around the expression to tell Cucumber to capture the word as an argument and pass it on. So the full expression we want to use for the match is:

```
^the ninja has a ([a-z]*) level black\-belt$
```

Double-escape in Java

The backslash `\` in a regular expression escapes the following character. In this case, we use a backslash to tell Cucumber that a dash is literally a dash, not a special range definition symbol such as in `a-z`. Ruby is OK with the expression as it is. In Java, backslash is also used in strings to escape the following character, so you'll have to use two backslashes `\\` to write it. So the correct way to specify this in Java and .NET is:

```
^the ninja has a ([a-z]*) level black\\-belt$
```

Triggering the action

To trigger the action in both scenarios, we create a step definition which matches the lines such as:

When attacked by a samurai

and

When attacked by Chuck Norris

To do that, we want to capture everything after 'attacked by' as the opponent, ignoring 'a' and a space if they follow straight after that. The expression `[a\s]` will match the lowercase letter `a` or a space (matched by a special sequence `\s`). We again add an asterisk to say that we expect any sequence of spaces or letter `a`, including an empty one. We don't add brackets around this expression because we want Cucumber to just ignore it. After that we want to capture anything till the end of the line. A dot matches any character, an asterisk turns this into a sequence and we add a dollar sign at the end of the expression `$`, telling Cucumber that the sequence should extend till the end of the current line. We want Cucumber to capture the sequence but not the line-end character, so we close the bracket before the dollar sign. So the full regular expression for the action step is:

```
^attacked by [a\s]*(.*)$
```

Validating the result

To validate the result, we create a step definition which matches the lines such as:

Then the ninja should engage the opponent

We capture anything that follows 'should engage' using the same expression as in the previous step, matching everything from a certain position till the end of the line `(.*)$`. The full expression in this case is:

```
^the ninja should (.*)$
```

Now we can wire these expressions into code. Continue reading the section relevant to your technology platform.

- For the Ruby implementation, see the section *Implementing in Ruby* on page 67
- For the Java implementation, see the section *Implementing in Java* on page 69
- For the .NET implementation, see the section *Implementing in .NET* on page 73

Implementing in Ruby

Create a new sub-directory called `step_definitions` inside `features`. You can run `cucumber` from the main project directory and it will print out an example of how the steps could be implemented. Take that and put it into a new file called `ninja_steps.rb`, and replace the regular expressions with the ones we suggested in the previous section.

Do I always create a steps file for a feature file?
Not necessarily. Cucumber will try to load steps from any step definition file when it executes a feature file. For convenience, it is often useful to keep related steps together, but you can even share state across different step definition files. See the section Sharing context across step definition files on page 75 for more information.

Initializing the context

We just need pass the argument captured by the step information to the initialiser of the `Ninja` class, and store the resulting `Ninja` object as a local variable. In Ruby, one way to do it is this:

```
Given /^the ninja has a ([a-z]*) level black\-belt$/ do |belt_level|
  @ninja=Ninja.new belt_level
end
```

Wow — where did that class and initialiser come from?

If you're reading this carefully, you probably noticed that this is the first time we used a `Ninja` class and presumed what the arguments of its initializer are. Don't scroll up looking for the class definition, because it doesn't exist yet. We will drive the class interface from its usage, so we first declare what we need that class to do in the step definitions and then implement the class to match the required interface and behaviour. BDD takes the test-first ideas of TDD to the business requirements level.

Triggering the action

To trigger the action, we take the `Ninja` object initialised in the first step and pass the opponent name/type to its `attacked_by` method. To be able to check the resulting actions, we'll expect the `attacked_by` method to return the actions a ninja should take when attacked by a particular opponent and store them in a local variable.

```
When /^attacked by [a\s]*(.*)$/ do |opponent|
  @actions=@ninja.attacked_by(opponent)
end
```

Validating the result

To validate the results, we just need to check whether the action expected in the step is included in the list of actions we saved previously.

```
Then /^the ninja should (.*)$/ do |expected_action|
  @actions.should include expected_action
end
```

What is that 'should include' syntax?

Cucumber in Ruby relies on RSpec expectations to perform validations of expected outcomes. `should include` is a way to specify that a collection should contain an element with RSpec. See http://rspec.info/ or the RSpec Book [8] for more information on the RSpec syntax.

Implementing the domain code

We still don't have a Ninja class to handle the calls from the steps, so let's add it. Create a file called `ninja.rb` in the `src` directory with the following content:

```
# ninja!
class Ninja
  def initialize (belt_level)
    @belt=belt_level
  end
  def attacked_by (opponent)
```

```
    if (opponent=="Chuck Norris")
      ["run for his life"]
    else
      ["engage the opponent"]
    end
  end
  def belt?
    @belt
  end
end
```

Now run `cucumber` again. You should get a nice green report, as in Figure 6.1

Figure 6.1. Ninjas survive in Ruby

```
$cucumber
Feature: Fight or flight
       In order to increase the ninja survival rate,
       As a ninja commander
       I want my ninjas to decide whether to take on an
       opponent based on their skill levels

  Scenario: Weaker opponent                      # features/ninja_survival_rate.feature:7
    Given the ninja has a third level black-belt # features/step_definitions/ninja_steps.rb:8
    When attacked by a samurai                   # features/step_definitions/ninja_steps.rb:12
    Then the ninja should engage the opponent    # features/step_definitions/ninja_steps.rb:16

  Scenario: Stronger opponent                    # features/ninja_survival_rate.feature:12
    Given the ninja has a third level black-belt # features/step_definitions/ninja_steps.rb:8
    When attacked by Chuck Norris                # features/step_definitions/ninja_steps.rb:12
    Then the ninja should run for his life       # features/step_definitions/ninja_steps.rb:16

2 scenarios (2 passed)
6 steps (6 passed)
0m0.093s
```

Implementing in Java

In this example, we use Maven to manage the dependencies, as SMERSH controls a stake in the world's biggest internet providers and Maven finances our drug-smuggling operations by downloading unnecessary libraries every time you run it. Create a pom.xml for your project as explained in the section *Using Maven* on page 34, and run the following command to get a suggested structure for step definitions:

```
mvn clean compile integration-test
```

Remember to download the gems the first time you run Cucumber from Maven

When you run Cucumber with Maven for the first time, remember to add `-Dcucumber.installGems=true`. If you do not do that, Maven won't install cuke4duke and all the dependent gems in the local repository, and we won't get our money from force-feeding bandwidth.

Put the suggested expressions into a new class in the `src/test` branch. In the example, we use `src/test/java/ninjasurvivalrate/NinjaSurvivalSteps.java`. Now let's change the expressions to the one we defined earlier.

Do I always create a step class for a feature file?
Not necessarily. Cucumber will try to load steps from any step definition class when it executes a feature file. For convenience, it is often useful to keep related steps together, but you can even share state across different step definition classes. See the section Sharing context across step definition files on page 75 for more information.

Initializing the context

We just need pass the argument captured by the step information to the constructor of the `Ninja` class, and store the resulting `Ninja` object as a local variable. In Java, one way to do it is this:

```
@Given ("^the ninja has a ([a-z]*) level black\\-belt$")
public void theNinjaHasABlackbelt(String level) {
 ninja=new Ninja(level);
}
```

Wow — where did that class and constructor come from?

If you're reading this carefully, you probably noticed that this is the first time we used a `Ninja` class and presumed what the arguments of its constructor are. Don't scroll up looking for the

class definition, because it doesn't exist yet. We will drive the class interface from its usage, so we first declare what we need that class to do in the step definitions and then implement the class to match the required interface and behaviour. BDD takes the test-first ideas of TDD to the business requirements level.

Triggering the action

To trigger the action, we take the `Ninja` object initialised in the first step and pass the opponent name/type to its `attackedBy` method. To be able to check the resulting actions, we'll expect the `attackedBy` method to return the actions a ninja should take when attacked by a particular opponent. We'll store them in a local variable for later inspection.

Remember the double-escape

To specify a backslash in Java, you have to write it twice (`\\`).

```
@When ("^attacked by [a\\s]*(.*)$")
public void attackedBy(String opponent) {
 actions=ninja.attackedBy(opponent);
}
```

Validating the result

To validate the results, we just need to check whether the action expected in the step is included in the list of actions we saved previously. Cuke4Duke uses the normal JUnit assertions.

```
@Then ("^the ninja should (.*)$")
public void theNinjaShould(String action) {
 assertTrue(actions.contains(action));
}
```

Implementing the domain code

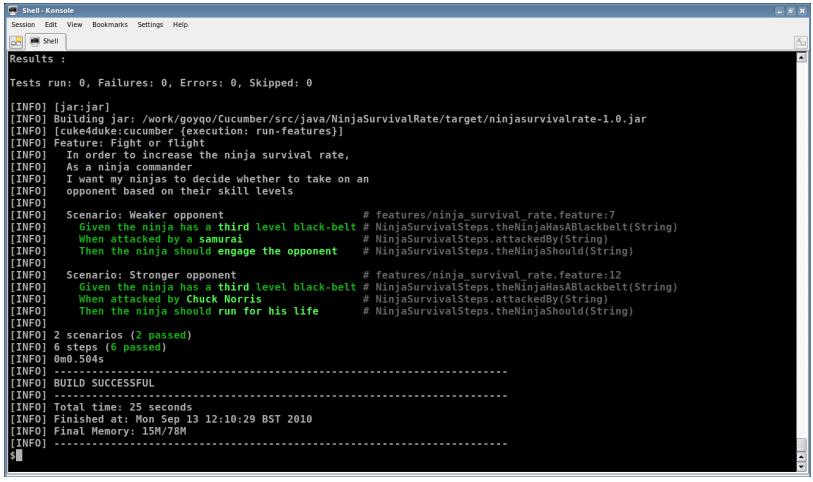
We still don't have a Ninja class to handle the calls from the steps, so let's add it. Create it in the `src/main` branch. In this example, we use

`src/main/java/ninjasurvivalrate/Ninja.java`. **The content is relatively straightforward:**

```
package ninjasurvivalrate;
import java.util.Arrays;
import java.util.Collection;
public class Ninja {
  private String belt;
  public Ninja (String belt){
    this.belt=belt;
  }
  public Collection<String> attackedBy(String opponent){
    if ("Chuck Norris".equals(opponent))
      return Arrays.asList(new String[]{"run for his life"});
    else
      return Arrays.asList(new String[]{"engage the opponent"});
  }

}
```

Now run `mvn clean compile integration-test` **again. You should get a nice green report, as in Figure 6.2**

Figure 6.2. Ninjas survive in Java



Implementing in .NET

Create a .NET project and put the feature file into it, in the `features` folder. Create a new sub-directory called `step_definitions` inside `features` and add a `cucumber.wire` file with the following content:

```
host: localhost
port: 3901
```

This will set up the Cuke4Nuke integration with Cucumber.

You can now run `cuke4nuke` from the main project directory and it will print out an example of how the steps could be implemented. Take that and put it into a new source file called `NinjaSteps.cs`. Replace the regular expressions with the ones we suggested in the previous section.

Do I always create a step class for a feature file?
Not necessarily. Cucumber will try to load steps from any step definition class when it executes a feature file. For convenience, it is often useful to keep related steps together, but you can even share state across different step definition classes. See the section Sharing context across step definition files on page 75 for more information.

Initializing the context

We just need pass the argument captured by the step information to the initialiser of the `Ninja` class, and store the resulting `Ninja` object as a local variable. In .NET, one way to do it is this:

```
[Given(@"^the ninja has a ([a-z]*) level black-belt$")]
public void TheNinjaHasABlackBelt(String level)
{
  ninja = new Ninja(level);
}
```

Wow — where did that class and constructor come from?

If you're reading this carefully, you probably noticed that this is the first time we used a `Ninja` class and presumed what the arguments of its constructor are. Don't scroll up looking for the class definition, because it doesn't exist yet. We will drive the class interface from its usage, so we first declare what we need that class to do in the step definitions and then implement the class to match the required interface and behaviour. BDD takes the test-first ideas of TDD to the business requirements level.

Triggering the action

To trigger the action, we take the `Ninja` object initialised in the first step and pass the opponent name/type to its `AttackedBy` method. To be able to check the resulting actions, we'll expect the `AttackedBy` method to return the actions a ninja should take when attacked by a particular opponent and store them in a local variable.

```
[When(@"^attacked by [a\s]*(.*)$")]
public void AttackedBy(String opponent)
{
  actions = ninja.AttackedBy(opponent);
}
```

Validating the result

To validate the results, we just need to check whether the action expected in the step is included in the list of actions we saved previously. For validations, we use standard NUnit assertions.

```
[Then("^the ninja should (.*)$")]
public void TheNinjaShould(String action)
{
  Assert.IsTrue(actions.Contains(action));
}
```

Implementing the domain code

We still don't have a Ninja class to handle the calls from the steps, so let's add it. Create a `Ninja.cs` file in the project with the following content:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace NinjaSurvivalRate
{
  public class Ninja
  {
    public Ninja(String beltLevel)
    {

    }
    public List<String> AttackedBy(String opponent)
    {
      if ("Chuck Norris" == opponent)
        return new List<string>(
          new String[] { "run for his life" });
      else
        return new List<string>(
          new String[] { "engage the opponent" });
    }
  }
}
```

Now run `Cuke4Nuke` again. You should get a nice green report.

Sharing context across step definition files

Putting all our ninjas in a single file is a good trick, made popular by the Black Star Ninja in the famous documentary about the life of Ninjas on Philippines from the 80's, wrongly titled The American Ninja. However, sometimes it is useful to break our step definitions into several files, and then share the context between them. Cucumber supports this with all the platforms, but in different ways.

So far, all our steps have been in a single file, corresponding to a feature file. As things get more complex, you'll want to write feature files that reuse some of the steps from other feature files and combine them with other steps, so the mapping between a step definition and a feature file is not necessarily the same as between Subzero and Scorpion. In this section, we'll show aspiring Ninja Cucumberists how to split these apart.

To demonstrate that, let's add a new feature file that reuses some of the steps that we already defined and adds some new steps. Paste the following content in `features/split.feature`**:**

```
Feature: Split Brains
  Ninjas can receive training to withstand a direct hit on the head

  Scenario: Samurai katana is useless against a ninja >= 3rd level
    Given the ninja has a third level black-belt
    When hit on the head by a samurai with a katana
    Then the ninja's brains should not be harmed

  Scenario: Ninja training is useless against Chuck Norris
    Given the ninja has a third level black-belt
    When hit on the head by Chuck Norris with a fist
    Then the ninja's brains should split
```

The initial step `Given the ninja has a third level black-belt` **will be already implemented. This step previously created a** `Ninja` **object that we'll share with another step definition file.**

Sharing context in Ruby

Ruby loads step definitions from any files in the `step_definitions` **directory. They do not have to all be in a single file. To see that, run the following command line:**

```
cucumber features/split.feature
```

Cucumber will execute just the new feature and report that it found the first step in `ninja_steps.rb` **but that the other steps are still undefined (see Figure 6.3).**

Figure 6.3. Cucumber finds some split ninja steps

```
$ cucumber features/split.feature
Feature: Split Brains
  Ninjas can receive training to withstand a direct hit on the head

  Scenario: Samurai katana is useless against a ninja >= 3rd level # features/split.feature:4
    Given the ninja has a third level black-belt                   # features/step_definitions/ninja_steps.rb:9
    When hit on the head by a samurai with a katana                # features/split.feature:6
    Then the ninja's brains should not be harmed                   # features/split.feature:7

  Scenario: Ninja training is useless against Chuck Norris # features/split.feature:9
    Given the ninja has a third level black-belt           # features/step_definitions/ninja_steps.rb:9
    When hit on the head by Chuck Norris with a fist       # features/split.feature:11
    Then the ninja's brains should split                   # features/split.feature:12

2 scenarios (2 undefined)
6 steps (4 undefined, 2 passed)
0m0.009s
```

Now we can add another step definition that completes the impact analysis required to decide whether the brains of a ninja should not be harmed or be split after a direct hit. Just to demonstrate that things can be a bit more complex, we'll put this logic into an impact calculator. The step definition file just needs to read the ninja created by the step in `ninja_steps.rb` using the same class variable name `@ninja`. When Cucumber compiles the steps, both references to `@ninja` will map to the same value.

```
require 'rspec/expectations'
require 'cucumber/formatter/unicode'

$:.unshift(File.dirname(__FILE__) + '/../../src')
require 'ninja'
require 'impact_calculator'

When /^hit on the head by [a ]*([A-z ]*) with a ([A-z]*)$/ do |opponent, weapon|
  @opponent=opponent
  @weapon=weapon
end

Then /^the ninja's ([A-z]*) should ([A-z]*)$/ do |target,expected_impact|
  impact_c=ImpactCalculator.new
  actual_impact=impact_c.impact @ninja,@opponent,@weapon,@target
  actual_impact.should==expected_impact
end

Then /^the ninja's ([A-z]*) should not be harmed$/ do |target|
  impact_c=ImpactCalculator.new
  actual_impact=impact_c.impact @ninja,@opponent,@weapon,@target
  actual_impact.should=="not harmed"
end
```

We'll leave the ImpactCalculator class to you for homework, or if you want to cheat go and get it from the *Cuke4Ninja github repository*[1] - look for a file called `impact_calculator.rb`.

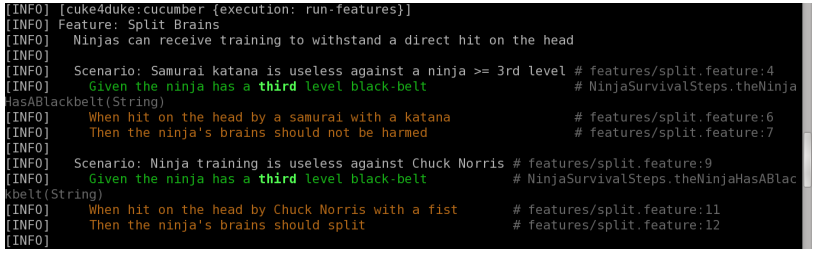
Sharing context in Java

Cuke4Duke loads step definitions from any classes annotated with Cucumber annotations such as `@Given`, they to not have to be all in the same file. To see that, run your tests again after adding a new feature file. For example, if running with maven, run the following command line:

```
mvn integration-test
```

Cucumber will execute the new feature file and find some of the steps in the old `NinjaSkillSteps.java` step definition class, but report other steps as still undefined (see Figure 6.4).

Figure 6.4. Cuke4Duke finds some split ninja steps



The first step, from `NinjaSkillSteps.java`, creates an internal Ninja object. Because we want to be able to reuse that object in two different step definition classes, we have to enable another step definition class to somehow see that state. Before you even think about saying 'static' know that Chuck Norris moves static objects. Remember that you had to add PicoContainer or Spring to the classpath? Now you'll finally understand why. Cuke4Duke shares state between step definitions using dependency injection. We can just create a class to hold the context required for our steps and add an instance of that class as a constructor argument for step definition classes. Cuke4Duke will ensure that both classes get the same context. Let's do that.

[1] http://github.com/davedf/cuke4ninja

First, let's define a context class. We just need to share the `Ninja` object so far, so let's create a container for a ninja:

```
package ninjasurvivalrate;

public class NinjaContext{
  private Ninja ninja;
  public void setNinja(Ninja ninja){
    this.ninja=ninja;
  }
  public Ninja getNinja(){
    return this.ninja;
  }
}
```

Now let's change the existing `NinjaSurvivalSteps.java` to expect an instance of this new class in the constructor:

```
public class NinjaSurvivalSteps {
 private NinjaContext ninjaContext;
 public NinjaSurvivalSteps(NinjaContext ninjaContext){
   this.ninjaContext=ninjaContext;
 }
```

At this point, you should be able to run the tests again and not notice any difference in the old functionality. Although the old step definition class did not have a constructor and this one does, and it requires a context parameter, Cuke4Duke wires that in automatically using your selected dependency injection container

Now we can add another step definition class with a similar constructor to handle the missing steps:

```
package ninjasurvivalrate;

import cuke4duke.annotation.I18n.EN.Given;
import cuke4duke.annotation.I18n.EN.Then;
import cuke4duke.annotation.I18n.EN.When;
import java.util.Map;
import java.util.List;
import java.util.Collection;

import static junit.framework.Assert.assertEquals;

public class SplitSteps {
 private NinjaContext ninjaContext;
```

```
 private String opponent;
 private String weapon;
 public SplitSteps(NinjaContext ninjaContext){
   this.ninjaContext=ninjaContext;
 }
 @When ("^hit on the head by [a ]*([A-z ]*) with a ([A-z]*)$")
 public void hitOnTheHead(String opponent, String weapon) {
  this.opponent=opponent;
  this.weapon=weapon;
 }
 @Then ("^the ninja's ([A-z]*) should not be harmed$")
 public void shouldNotBeHarmed(String target) {
   expectImpact(target,"not harmed");
 }
 @Then ("^the ninja's ([A-z]*) should ([A-z]*)$")
 public void expectImpact(String target, String expectedImpact) {
  String actualImpact=ninjaContext.getNinja().impact(target,opponent,weapon);
  assertEquals(expectedImpact,actualImpact);
 }
}
```

We'll leave it to you for homework to implement the missing function for impact analysis in the `Ninja` class and to change the rest of the `NinjaSurvivalSteps.java` to use the context Ninja instead of the private one. If you want to cheat or admit that you are lazy, look for a folder called `NinjaSurvivalRateWithSharedState` in the *Cuke4Ninja code repository*[2].

Sharing context in .NET

Cuke4Nuke loads step definitions from any classes annotated with Cucumber annotations such as `[Given]`, they to not have to be all in the same file. To see that, run your tests again after adding a new feature file.

Cucumber will execute the new feature file and find some of the steps in the old `NinjaSteps.cs` step definition class, but report other steps as still undefined.

The first step, from `NinjaSteps.cs`, creates an internal Ninja object. Because we want to be able to reuse that object in two different step definition classes, we have to enable another step definition class to somehow see that state. Before you even think about saying 'static' know that Chuck Norris moves static objects. Cuke4Nuke shares state between step definitions using

[2] **http://github.com/davedf/cuke4ninja**

dependency injection. We can just create a class to hold the context required for our steps and add an instance of that class as a constructor argument for step definition classes. Cuke4Nuke will ensure that both classes get the same context. Let's do that.

First, let's define a context class. We just need to share the `Ninja` object so far, so let's create a container for a ninja:

```
Ôªønamespace NinjaSurvivalRate
{
    public class NinjaContext
    {
        public Ninja Ninja{get; set;}
    }
}
```

Now let's change the existing `NinjaSteps.cs` to expect an instance of this new class in the constructor:

```
        public NinjaSteps(NinjaContext ninjaContext)
        {
            _ninjaContext = ninjaContext;
        }
```

At this point, you should be able to run the tests again and not notice any difference in the old functionality. Although the old step definition class did not have a constructor and this one does, and it requires a context parameter, Cuke4Nuke wires that in automatically using your selected dependency injection container

Now we can add another step definition class with a similar constructor to handle the missing steps:

```
Ôªøusing Cuke4Nuke.Framework;
using NUnit.Framework;

namespace NinjaSurvivalRate
{
    public class SplitSteps
    {
        private string _opponent;
        private string _weapon;
        private readonly NinjaContext _ninjaContext;

        public SplitSteps(NinjaContext ninjaContext)
```

```
        {
            _ninjaContext = ninjaContext;
        }

        [When(@"^hit on the head by [a ]*([A-z ]*) with a ([A-z]*)$")]
        public void HitOnHead(string opponent, string weapon)
        {
            _opponent = opponent;
            _weapon = weapon;
        }

        [Then(@"^the ninja's ([A-z]*) should ([A-z]*)$")]
        public void ExpectImpact(string target, string expectedmpact)
        {
            string actualImpact = _ninjaContext.Ninja.CalculateImpact(_opponent)
            Assert.AreEqual(expectedmpact, actualImpact);
        }

        [Then(@"^the ninja's ([A-z]*) should not be harmed$")]
        public void NinjaNotHarmed(string target)
        {
            ExpectImpact(target, "not harmed");
        }
    }
}
```

We'll leave it to you for homework to implement the missing function for impact analysis in the `Ninja` class and to change the rest of the `NinjaSteps.cs` to use the context Ninja, not the private one. If you want to cheat or admit that you are lazy, look for a folder called `NinjaSurvivalRateWithSharedState` in the *Cuke4Ninja code repository*[3].

[3] **http://github.com/davedf/cuke4ninja**

Chapter 7

Managing complex scenarios

The BDD Given-When-Then scenario structure is too verbose when a specification includes lots of similar cases or when complicated objects come into play. One of the strengths of Cucumber is handling complex scenarios and groups of scenarios efficiently. It has several features that help developers and testers write and manage complex specifications yet still keep them in a form that is self-explanatory. In this chapter, we explore several such features.

- **Scenario outlines allow us to group related scenarios and add examples with no loss of clarity**
- **Tables allow us to define or validate related attributes as a group and handle lists of objects easily**
- **Backgrounds allow us to group related pre-conditions for a set of scenarios and manage them in one place**
- **Hooks allow us to programmatically manage cross-cutting technical activities such as transactions, security or external resources.**

Complex setups and validations

Steps can also start with keywords `And` and `But`. They continue the current section and allow us to use multiple steps to set up the context or validate several outcomes. Using these keywords can improve readability. For example, the following scenario uses two validations:

```
Scenario: Angry Chuck
 Given the ninja has a third level black-belt
 Given the ninja has never fought Chuck Norris before
 When attacked by Chuck Norris
 Then the ninja should apologise
 Then the ninja should run away
```

It reads better when it is written in the following way:

```
Scenario: Angry Chuck
 Given the ninja has a third level black-belt
 But the ninja has never fought Chuck Norris before
 When attacked by Chuck Norris
 Then the ninja should apologise
 And the ninja should run away
```

Technically, it isn't really important whether a step starts with `Given`, `And` or `But` (or `Then` and `When`, actually). Cucumber will match the step using a regular expression and execute it. Using `And` and `But` makes it easier to read the specification.

This also means that the syntax allows you to chain `When` steps by using `And` or `But` steps. Though the Gherkin syntax won't stop you, it will send a secret message to Chuck Norris and he will come to your house and burn it to the ground with you in it.[1] A scenario should have one and only one action.

But I need to execute several actions

If you want to put more than one `When` step, then you have a complex multi-step action that means something to the business. That flow should be captured by your domain model. Talk to the business users, decide on the name for that flow and specify it using a single step, which invokes the appropriate domain model procedure. Don't use feature files for scripting the flow.

Managing groups of related scenarios

The feature file we used in the Chapter 5 has 13 lines for two scenarios. To specify a business rule properly we would most likely have to add many more scenarios, that illustrate what happens when a ninja is not experienced enough to engage a samurai, when it becomes experienced enough to engage Chuck Norris, not to mention other ninjas and types of opponents. If we

[1] Legal Note: Chuck reserves the right to sub-contract to the Deadly Viper Assassination Squad if he is busy that day.

had to add four lines for every new scenario, the file would quickly become too big to be readable.

To address this, Gherkin supports scenario outlines. Scenario outlines are essentially template scenarios, allowing us to provide the scenario structure once and then illustrate the behaviour with different combinations of parameters. Scenario outlines are a great way to consolidate and simplify related scenarios and present them in a way that is easier to understand. Here is an example:

```
Scenario Outline: third-level ninjas engage samurai
  Given the ninja has a <belt level> level black-belt
  When attacked by <opponent>
  Then the ninja should <expected action>

  Examples:
  |belt level |opponent     |expected action  |
  |third      |a samurai    |engage the opponent |
  |third      |Chuck Norris |run for his life  |
```

The interesting parts are in bold:

- Instead of `Scenario`, this block starts with `Scenario Outline`
- The steps have placeholders enclosed into less-than and greater-than (< and >).
- There is a new block after the steps, starting with a line that contains the `Examples` keyword and a colon.
- A table in the normal wiki format (pipes separate cells, one row per line) defines the examples and expected outcomes. The first row of the table contains placeholder names.

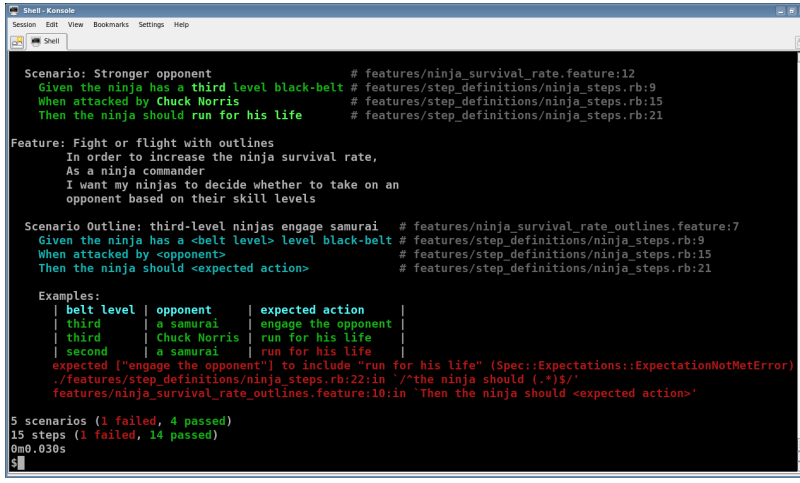
This feature file is functionally equivalent to the one we used in the previous chapter. The same step definitions will be used to execute it. In fact, try that yourself. Delete the two step definitions we used earlier, or create a new feature file and run it using Cucumber. You'll see the same results, just formatted differently.

What's nice about this format is that we can easily add more examples. To make the specification stronger, let's add another example that shows that second level ninjas don't even fight against samurai. We add just one line to the table: `|second |a samurai |run for his life |` The result is easier to

read and understand than if we added another four-line scenario. With more complex scenarios, the result is even better.

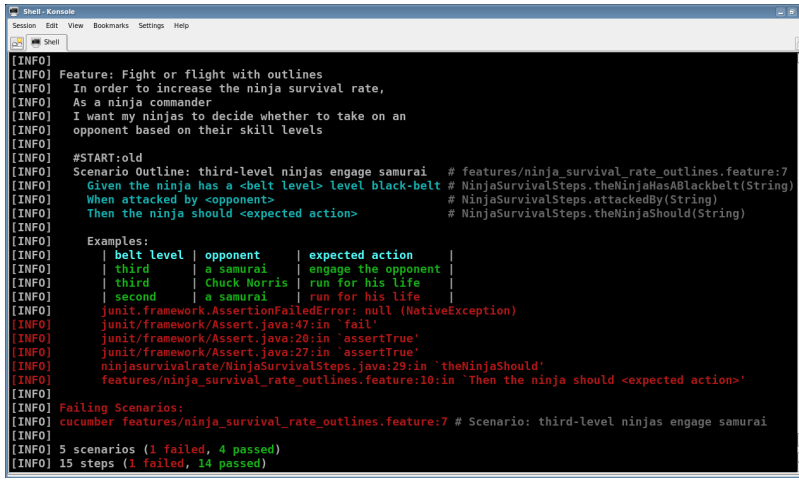
Run Cucumber again and the test should fail. If you are running Ruby, you should get an error similar to the one in Figure 7.1. If you are running Java, you should get an error similar to the one in Figure 7.2.

Figure 7.1. Ruby ninjas engage samurai even when they should not



For a homework assignment, change the Ninja class to make this test pass and re-run Cucumber to confirm that you have fixed it.

Figure 7.2. Java ninjas engage samurai even when they should not



Scenario outlines are one of the most important tools for Cucumber ninjas, as they allow us to describe complex scenario groups easily and make the files shorter and easier to understand.

Working with sets of attributes

Given-When-Then plain language scenarios can quickly become too verbose when they have many important attributes or work with lists of objects. If the list of Given or Then steps starts growing too much, the scenarios become hard to understand and maintain. For example:

```
Given a ninja has a third-level black belt
and the ninja is a level two katana user
and the ninja is a level three sake drinker
and the ninja has never fought Chuck Norris before
and the ninja has fought samurai before
and the ninja has level five magic powers
When ...
```

Writing scenarios like this complicates both the feature file and the step definition code unnecessarily and makes it hard to understand the behaviour the scenario is illustrating. Cucumber allows us to specify related attributes as a table. For example:

```
Scenario: Fully armed
  Given a ninja with the following experience
    | belt_level  | katana | sake     | fought  | magic |
    | third       | two    | three    | samurai | five  |
  When attacked by a samurai
  Then the ninja should engage the opponent
```

Reading tables in Ruby

In Ruby, tables are converted automatically into `Cucumber::Ast::Table` objects. A step definition will receive the whole table as an argument. The simplest way to use a table is to invoke the `hashes` method which converts the entire table into an array of hashes, mapping the header values to cell values. Each element of the array represents a single data row. We can implement the step to handle creating a ninja from several properties by grabbing the first (and only) hash and passing that to the constructor:

```
Given /^a ninja with the following experience$/ do |table|
   @ninja=Ninja.new table.hashes.first
end
```

See *Cucumber AST Table API docs*[2] on RDoc for more information on other table methods.

Reading tables in Java

In Java, tables are converted automatically into `cuke4duke.Table` objects, which mirror the API of the `Cucumber::Ast::Table` ruby class. A step definition will receive the entire table as an argument. The easiest way to process a table is to invoke the `hashes` method to get a list of maps that represents the table. Each element of the list represents one data row as a `Map<String,String>`, mapping column headers into cell values. For example, this is how we can create a ninja using the `belt_level` property of the first map in the list:

[2] **http://rdoc.info/github/aslakhellesoy/cucumber/master/Cucumber/Ast/Table**

```
@Given ("^a ninja with the following experience$")
public void ninjaWithExperience(cuke4duke.Table table) {
 List<Map<String,String>> hashes=table.hashes();
 Map<String,String> ninjaProperties=hashes.get(0);
 ninja=new Ninja(ninjaProperties.get("belt_level"));
}
```

See *Cuke4Duke API docs*[3] on Cukes.info for more information on the Java table API.

Reading tables in .NET

In .NET, tables are converted automatically into `Cuke4Nuke.Framework.Table` objects, which mirror parts of the API of the `Cucumber::Ast::Table` Ruby class. The Table API in .NET is not as nice and clean as it is in Ruby and Java, but it does the job. A step definition will receive the entire table as an argument. The easiest way to process a table is to invoke the `Hashes` method to get a list of dictionaries that represents the table. Each element of the list represents one data row as a `Dictionary<string,string>`, mapping column headers into cell values. For example, this is how we can create a ninja using the `belt_level` property of the first map in the list:

```
[Given("^a ninja with the following experience$")]
public void NinjaWithExperience(Table table)
{
  List<Dictionary<string, string>> hashes = table.Hashes();
  Dictionary<String, String> ninjaProperties = hashes[0];
  ninja = new Ninja(ninjaProperties["belt_level"]);
}
```

See *Cuke4Nuke Table source*[4] on github for more information on the .NET table API.

Working with lists of objects

We can use tables to efficiently describe specifications that work with lists of objects. Instead of a very lengthy description that explains individual objects with Given-When-Then, we can just specify batches of objects in a table. For example, it is a well known fact that only Chuck Norris can perform

[3] **http://cukes.info/cuke4duke/cuke4duke/apidocs/cuke4duke/Table.html**
[4] **http://github.com/richardlawrence/Cuke4Nuke/blob/master/Cuke4Nuke/Framework/Table.cs**

the roundhouse-kick without wire-fu and that he is always extremely dangerous. We can specify that with the following scenario:

```
Feature: Skill availability
  As a ninja trainer,
  I want ninjas to understand the dangers of various opponents
  so that they can engage them in combat more effectively

  Scenario: Samurai are dangeous with katanas, no advanced kicks
    Given the following skills are allowed
      | katana          |
      | karate-kick     |
      | roundhouse-kick |
    When a ninja faces a samurai
    Then he should expect the following attack techniques
      | technique     | danger |
      | katana        | high   |
      | karate-kick   | low    |

  Scenario: Chuch Norris can do anything and is always dangerous
    Given the following skills are allowed
      | katana          |
      | karate-kick     |
      | roundhouse-kick |
    When a ninja faces Chuck Norris
    Then he should expect the following attack techniques
      | technique       | danger   |
      | katana          | extreme  |
      | karate-kick     | extreme  |
      | roundhouse-kick | extreme  |
```

In this case we are using tables for inputs and expected outputs. Note that the input table does not really have any headers. Unlike most other tools based on tables, Cucumber does not require the first row to be a header if that is not required. In addition, although the API to pass tables to validation step definitions is the same, Cucumber does not require us to check every value individually. It has a helper method to compare expected tabular outputs to actual values.

We'll go through the step definitions in all three platforms now. For homework, implement the `Skill` class to make the feature from this example pass. If you want to cheat, have a look at our web site and download the solution from there. Once you have it running, the result should be nice and green like in Figure 7.3.

Then break the test by changing the code in the `Skill` class and re-run the test again to see it fail. For example, the result in Figure 7.4 is the result when all the skills were available to everyone. Notice how Cucumber prints the outputs nicely to signal exactly what is wrong.

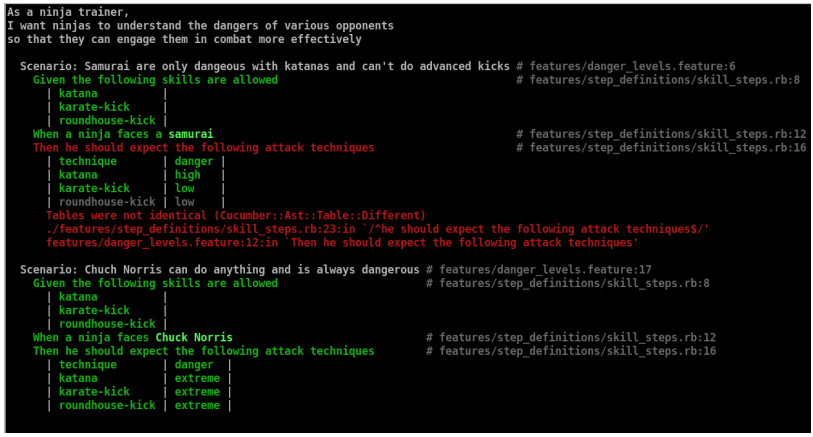
Figure 7.3. Table comparisons in Ruby

```
Feature: Skill availability
As a ninja trainer,
I want ninjas to understand the dangers of various opponents
so that they can engage them in combat more effectively

  Scenario: Samurai are only dangeous with katanas and can't do advanced kicks # features/danger_levels.feature:6
    Given the following skills are allowed                                     # features/step_definitions/skill_steps.rb:8
      | katana       |
      | karate-kick  |
      | roundhouse-kick |
    When a ninja faces a samurai                                               # features/step_definitions/skill_steps.rb:12
    Then he should expect the following attack techniques                      # features/step_definitions/skill_steps.rb:16
      | technique    | danger |
      | katana       | high   |
      | karate-kick | low    |

  Scenario: Chuch Norris can do anything and is always dangerous # features/danger_levels.feature:17
    Given the following skills are allowed                       # features/step_definitions/skill_steps.rb:8
      | katana          |
      | karate-kick     |
      | roundhouse-kick |
    When a ninja faces Chuck Norris                              # features/step_definitions/skill_steps.rb:12
    Then he should expect the following attack techniques        # features/step_definitions/skill_steps.rb:16
      | technique       | danger  |
      | katana          | extreme |
      | karate-kick     | extreme |
      | roundhouse-kick | extreme |
```

What is the order of execution?

It's not important. When teams start with specification by example they often ask for the order of execution to be able to reuse context. For example, both scenarios in this case have the same `Given` step, so setting it up only in one place and using it for both scenarios makes sense. Making one scenario dependent on another one or relying on the order of execution is bad practice. This makes it impossible to check individual scenarios in isolation or get quick feedback from a subset of tests. Instead of making the scenarios interdependent, we can clean this up with a common setup. We explain this in the section Using backgrounds to group related pre-conditions on page 95

Figure 7.4. Cucumber clearly prints what's wrong when table comparisons fail



Advanced table operations in Ruby

To process a table without using the first row as a header, we can call the `raw` method and get the entire table as an array of arrays of strings. The first array will contain rows and the second-level arrays will contain cell values inside each row. In this case, the table has only one column, so we'll get the following array `[["katana"],["karate-kick"],["roundhouse-kick"]]`. We can now process this easily and for example initialise an array of `Skill` objects with it.

```
Given /^the following skills are allowed$/ do |table|
  @skill_list=table.raw.flatten.map {|skill| Skill.new(skill) }
end
```

To validate the results in the third step, we should go through the list of skill objects and create an array of hashes corresponding to the expected outcome table. Then we can use the `diff!` method of the Cucumber table to compare the entire table with the expected list of hashes.

```
Then /^he should expect the following attack techniques$/ do |table|
  actual_skill_list=[]
  @skill_list.each do |skill|
    if skill.available_to @opponent then
      actual_skill_list <<
```

```
        { "technique" => skill.name,
          "danger" => skill.danger(@opponent) }
    end
  end
  table.diff! actual_skill_list
end
```

The `diff!` method can compare a table against another table, an array of hashes, or an array of arrays of strings for cases when a table does not have a header row.

Advanced table operations in Java

To process a table without using the first row as a header, we can call the `raw` method and get the entire table as a `List<List<String>>`. The first list will contain rows and the second-level lists will contain cell values inside each row. In this case, the table has only one column, so we'll get the following List `[["katana"],["karate-kick"],["roundhouse-kick"]]`. We can now process this easily and for example initialise an array of `Skill` objects with it.

```
@Given ("^the following skills are allowed$")
public void setAllowedSkills(cuke4duke.Table table) {
  skills=new ArrayList<Skill>();
  for (List<String> rows:table.raw()){
    skills.add(new Skill(rows.get(0)));
  }
}
```

To validate the results in the third step, we should go through the list of skill objects and create a `List<Map<String,String>>` corresponding to the expected outcome table in the same format as the result of the `hashes` method we used earlier. Then we can use the `diff!` method of the Cucumber table to compare the entire table with the expected list of hashes.

```
@Then ("^he should expect the following attack techniques$")
public void checkExpectedTechniques(cuke4duke.Table table) {
  List<Map<String,String>> actualTechniques=
      new ArrayList<Map<String,String>>();
  for (Skill skill:skills){
    if (skill.availableTo(opponent)){
      actualTechniques.add(toHash(skill,opponent));
    }
  }
```

```
    table.diffHashes(actualTechniques);
  }
  private Map<String,String> toHash(Skill skill,String opponent){
    Map<String,String> map=new HashMap<String,String>();
    map.put("technique",skill.getName());
    map.put("danger",skill.getDanger(opponent));
    return map;
  }
```

The `diff!` method can compare a table against another table, a list of hash-maps, or a list of lists of strings for cases when a table does not have a header row.

Advanced table operations in .NET

To process a table without using the first row as a header, we can access the raw data using the `Data` property. This is the replacement for the Ruby `hashes` method, it gives us access to the entire table as a `List<List<string>>`. The first list will contain rows. The second-level lists will contain cell values inside each row. In this case, the table has only one column, so we'll get the following List `[["katana"],["karate-kick"],["roundhouse-kick"]]`. We can now process this easily and for example initialise an array of `Skill` objects with it.

```
[Given (@"^the following skills are allowed$")]
public void setAllowedSkills(Table table) {
  skills=new List<Skill>();
  foreach (List<String> rows in table.Data){
    skills.Add(new Skill(rows[0]));
  }
}
```

To validate the results in the third step, we create a new table using the actual values. The API in .NET differs significantly from the Ruby and Java counterparts in this aspect. We can use the `Data` property of the new table to populate it with the actual data, in the same format as the result that we expect. Then we can use the `AssertSameAs` method of the Cucumber table to compare both tables:

```
[Then (@"^he should expect the following attack techniques$")]
public void checkExpectedTechniques(Table table) {
 //   System.Diagnostics.Debugger.Break();
    Table actualTable = new Table();
    actualTable.Data.Add(ToList ( "technique", "danger" ));
```

```
    foreach (Skill skill in skills){
      if (skill.AvailableTo(opponent)){
            actualTable.Data.Add(
             ToList(skill.Name(), skill.Danger(opponent)));
      }
    }
      Console.WriteLine(actualTable.ToString());
      table.AssertSameAs(actualTable);
  }
```

Unlike Java and Ruby that print a nice report when the comparison fails and show us exactly where the problem is, the .NET version just fails. You might need to debug it, as explained in the section *Debugging Cuke4Nuke steps* on page 24, if you get stuck.

Using backgrounds to group related pre-conditions

Both scenarios we use to demonstrate advanced table operations start with the same list of skills. Related scenarios often start with the same context, or a very similar context. In such cases, it is often useful to centralise the common setups. This will make it easier to change the setup globally in the future. It also makes scenarios shorter and easier to understand. Cucumber supports this with backgrounds. A Background is a special scenario executed before each scenario in the feature file. It starts with the `Background` keyword instead of the usual `Scenario`. Here is how the previous example looks with a background:

```
Feature: Skill availability
  As a ninja trainer,
  I want ninjas to understand the dangers of various opponents
  so that they can engage them in combat more effectively

  Background: Allowed skills
    Given the following skills are allowed
    | katana          |
    | karate-kick     |
    | roundhouse-kick |

  Scenario: Samurai are dangeous with katanas, no advanced kicks
    When a ninja faces a samurai
    Then he should expect the following attack techniques
```

```
    | technique   | danger |
    | katana      | high   |
    | karate-kick | low    |

  Scenario: Chuch Norris can do anything and is always dangerous
    When a ninja faces Chuck Norris
    Then he should expect the following attack techniques
    | technique       | danger   |
    | katana          | extreme  |
    | karate-kick     | extreme  |
    | roundhouse-kick | extreme  |
```

Avoid complex backgrounds

If a feature file background starts becoming long or complex, it is often a sign that you are describing how the context needs to be set up rather than what the context is. Separate the specification of the context from the procedure to set it up and automate the procedure using step definitions. Another reason for complex backgrounds is often trying to define technical initialisation, for example allocating resources etc. Technical context setup should be moved to a technical hook (explained in the section Using hooks to manage cross-cutting concerns on page 96).

Using hooks to manage cross-cutting concerns

Cucumber feature files should ideally describe expected functionality without going into a lot of technical detail. This makes the feature files simpler, easier to read and understand, and less bound to a particular technical implementation. It helps us implement the principle of symmetric change (see tip *The principle of symmetric change* on page 111). Technical initialisation or teardown activities are sometimes useful to avoid duplication in step definitions, so Cucumber provides for that with Scenario Hooks.

Scenario Hooks are technical actions that will be executed before or after a scenario is validated. They can be set up globally (for all scenarios) or filtered by tags (mentioned in the section *Tagging for fun and profit* on page 59).

To demonstrate how hooks work, we'll use a simple feature demonstrating how Chuck Norris kills ninjas. We'll call this one a killer feature (`features/killer.feature`).

```
Feature: Killer
  Every time you run a scenario, Chuck Norris will kill one ninja

  Scenario: First blood
    When this scenario is executed
    Then Chuck Norris should expect 3 ninjas
    And Chuck Norris should kill one ninja

  Scenario: Second blood
    When this scenario is executed
    Then Chuck Norris should expect 2 ninjas
    And he should kill 2 ninjas
```

The number of expected ninjas decreases between scenarios and the number of killed ninjas rises. We'll implement this using hooks and without doing anything in the step definition of the action. This is quite a misuse of hooks and actions but considering that Chuck ends up being really cool, I'm sure he won't mind. And we'll use it to demonstrate the syntax of hooks in different languages.

Scenario hooks in Ruby

You can define scenario hooks in Ruby using the following syntax:

```
Before do
 # something that happens before a scenario is executed
end
After do |scenario|
 # something that happens after a scenario is executed
end
```

We can define the steps of the killer feature using hooks to keep global state of ninjas killed so far. Save the following content into a file, for example `features/step_definitions/killer_steps.rb`.

```
require 'rspec/expectations'
require 'cucumber/formatter/unicode'

@@killed=0
@@remaining=3
```

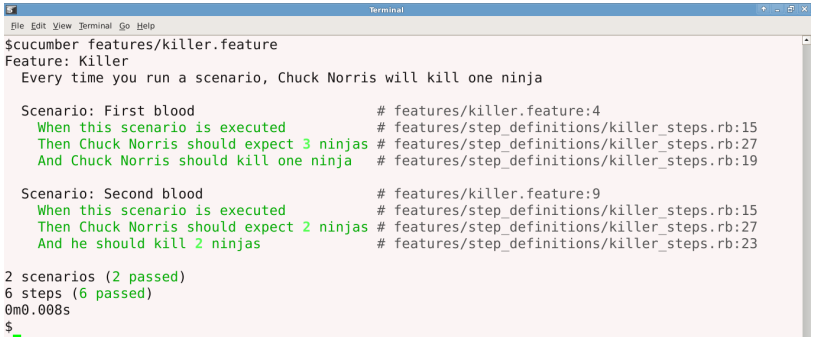
```
Before do
  @@killed=@@killed+1
end

After do |scenario|
  @@remaining=@@remaining-1
end

When /^this scenario is executed$/ do
  # do absolutely nothing
end

Then /^[A-z ]* should kill one ninja$/ do
  @@killed.should == 1
end

Then /^[A-z ]* should kill (\d+) ninjas$/ do |expected|
  expected.to_i.should == @@killed
end

Then /^[A-z ]* should expect (\d+) ninjas$/ do |expected|
  expected.to_i.should == @@remaining
end
```

Run only the killer feature now using the following command:

```
cucumber features/killer.feature
```

The result should appear as in Figure 7.5. The global class variables (remaining ninjas and killed ninjas) change every time a scenario is executed, so that you can see the effects.

Figure 7.5. Chuck kills ninjas using hooks in Ruby



Scenario hooks are global

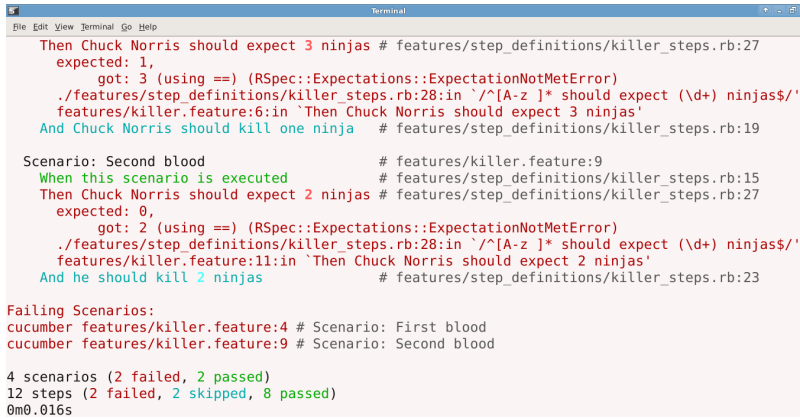
Regardless of whether you define a hook in a step definition or in a global support file, all scenario hooks are global.

You can set up hooks in a global file, ideally in a `features/support` directory. You can also set them up in a feature file as in the previous example. All hooks execute before/after each scenario, regardless of where they are defined. To see this effect, execute the feature again but run one more feature before it. For example, use the following command:

```
cucumber features/ninja_survival_rate.feature features/killer.feature
```

The result is now quite different, with failing steps in our killer feature (Figure 7.6). Make sure to remember this if you use hooks, or you'll end up on the one Chuck Norris uses to hang his punch bag.

Figure 7.6. Too many ninjas fall at the hand of Chuck because of global hooks



You can avoid this problem by assigning tags to scenarios (the section *Tagging for fun and profit* on page 59) and specifying the relevant tags as arguments of `Before` and `After`). For example, the following will execute only before scenarios marked with `@killer`:

```
Before('@killer') do
  # only before scenarios tagged with @killer
end
```

For more information on scenario hooks in Ruby, see *the hooks page in the Cukes wiki*[5].

Scenario hooks in Java

You can define scenario hooks in Java using the `@Before` and `@After` annotations. For example:

```
@Before
public void thisGetsCalledBeforeEachScenario(){
  // ...
```

[5] http://github.com/aslakhellesoy/cucumber/wiki/Hooks

```
}
@After
public void thisGetsCalledAfterEachScenario(){
  // ...
}
```

@Before and @After annotations are in the cuke4duke.annotation package, not the cuke4duke.annotation.I18n.EN as all the other annotations we used so far.

We can define the steps to implement our killer feature using the tags:

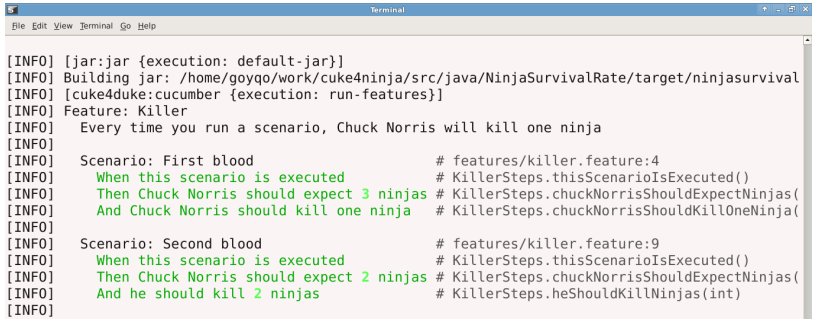
```
private static int killed=0;
private static int remaining=3;

@Before
public void killNinjaBeforeEachScenario(){
  killed++;
}
@After
public void expectLessNinjasAfterScenario(){
  remaining--;
}
@When ("^this scenario is executed$")
public void thisScenarioIsExecuted() {
 // do absolutely nothing
}
@Then ("^Chuck Norris should expect ([0-9]+)* ninjas$")
public void chuckNorrisShouldExpectNinjas(int expected) {
  assertEquals(expected, remaining);
}
@Then ("^Chuck Norris should kill one ninja$")
public void chuckNorrisShouldKillOneNinja() {
  heShouldKillNinjas(1);
}
@Then ("^he should kill ([0-9]+)* ninjas$")
public void heShouldKillNinjas(int expected) {
  assertEquals(expected, killed);
}
```

If you change the POM file to execute just the killer feature now (or delete other features), you should see the feature passing (Figure 7.7). Remember that the hooks are global, although they are defined in a class file with the other steps from this feature, so they will be executed even for other features.

This means that the same test will fail if you execute all the other features from the `features` folder. To avoid this problem, mark your scenarios with tags and list the relevant tags in the annotation. For example:

```
@Before("@killer")
public void thisGetsCalledBeforeKillerScenarios(){
  // ...
}
```

Figure 7.7. Ninjas fall at Chuck's hand in Java



For more information on scenario hooks in other JVM languages, see *the hooks page in the Cuke4Duke wiki*[6].

Scenario hooks in .NET

You can define scenario hooks in Java using the `[Before]` and `[After]` attributes. For example:

```
[Before]
public void ThisGetsCalledBeforeEachScenario(){
   // ...
}
[After]
public void ThisGetsCalledAfterEachScenario(){
   // ...
}
```

[6] **http://github.com/aslakhellesoy/cuke4duke/wiki/Hooks**

We can define the steps to implement our killer feature using the tags:

```
private static int killed = 0;
private static int remaining = 3;

[Before]
public void KillNinjaBeforeEachScenario()
{
    killed++;
}

[After]
public void ExpectLessNinjasAfterScenario()
{
    remaining--;
}

[When(@"^this scenario is executed$")]
public void ThisScenarioIsExecuted()
{
    // do absolutely nothing
}

[Then("^Chuck Norris should expect ([0-9]+)* ninjas$")]
public void ChuckNorrisShouldExpectNinjas(int expected)
{
    Assert.AreEqual(expected, remaining);
}

[Then("^Chuck Norris should kill one ninja$")]
public void ChuckNorrisShouldKillOneNinja()
{
    HeShouldKillNinjas(1);
}

[Then("^he should kill ([0-9]+)* ninjas$")]
public void HeShouldKillNinjas(int expected)
{
    Assert.AreEqual(expected, killed);
}
```

If you execute just the killer feature now, you should see the feature passing. Remember that the hooks are global, although they are defined in a class file with the other steps from this feature, so they will be executed even for other features. This means that the same test will fail if you execute all the other

features from the `features` folder. To avoid this problem, mark your scenarios with tags and list the relevant tags in the annotation. For example:

```
[Before("@killer")]
 public void ThisGetsCalledBeforeKillerScenarios(){
   // ...
 }
```

For more information on scenario hooks in Cuke4Nuke, see *hooks examples in the Cuke4Nuke source code*[7].

Advanced hooks

The Ruby version of Cucumber (but not Cuke4Duke and Cuke4Nuke) supports many other hooks, such as step hooks (executed before or after each step) or global hooks (executed once before any test runs or after all tests finish). They are defined similarly to the scenario hooks mentioned in this section. For more information, see the hooks page in the Cukes wiki[8].

[7] **http://github.com/richardlawrence/Cuke4Nuke/blob/master/features/hooks.feature**
[8] **https://github.com/aslakhellesoy/cucumber/wiki/Hooks**

Part III. Web automation

In this part, we show how Cucumber integrates with the most popular frameworks for web browser automation.

Chapter 8

The Three Layers of UI Automation

Many ninja children are told the story of Cucumber and the three layers of UI Automation, in which the the abstract concept of layers of responsibility are represented by bears who are too weak to eat hot porridge and the little ninja comes in and eats the porridge and totally flips out, absolutely killing everyone in the story and other nearby stories. Ahh the carnage is truly a sight to behold...

Sorry, ignore that last bit.

Ninjas learn how to use many deadly weapons during their years of training — Nunchucks, Sai and many others. None are more treacherous than the ones for UI test automation. Some of these tools use the dark magic of the Geisha to seduce young ninjas into believing that they can build up a suite of tests by recording actions performed on a web site to produce automation scripts. Geishas promise that these scripts solve the problem of ongoing UI automation.

At first, all seems to go well. Young ninjas record scripts quickly and the tests all pass. Ninjas can easily add more tests. After a few months, new features bring some small styling changes to a number of core pages of Ninja web sites. Suddenly all the tests start failing! The ninjas are starting to realise that they have been tricked by the Geisha magic. Their anger turns to despair, however, once they look at the scripts produced in the first stage of their journey. The scripts are written in Selenese (Figure 8.1), a strange language from a far off land. Selenese uses twelve words where a real ninja would usually use one.

The ninjas realize that their UI record and replay testing was a trap. They were describing tests at the technical level of user interface interactions. The resulting tests are very brittle and many of them break with even the smallest change in the UI. Since the scripting language is quite verbose, it is hard to understand the reasons why a test fails. The ninjas realise that the test scripts are completely unmaintainable and must be thrown away. Chuck Norris, of course, refactors Selenese using roundhouse kicks but ninjas do not have

that level of skill. Ninjas then fall under the Mount Fuji death spell (also known as the sine of death[1]), illustrated in Figure 8.2.

Figure 8.1. The Secret language of Selenese

Auto-generated at Mon Jan 2 16:11:54 2006

Auto-generated from zen/web/selenium/tests/1-zen.wiki at Mon Jan 2 16:11:54 2006

Zen Koans Web Sanity Tests		
open	/zen/	
verifyLocation	/zen/	
verifyTitle	Zen Koans	
verifyTextPresent	Zen Koans	
verifyText	koan_source	Ashidakim Zen Koans
verifyAttribute	koan_source@href	exact:http://www.ashidakim.com/zenkoans/zenindex.html
verifyText	koan	These koans, or parables, were translated *
assertElementPresent	next_link	
verifyText	next_link	Start
click	next_link	
pause	1200	
verifyText	next_link	Next
verifyText	koan_title	A Cup of Tea
verifyText	koan_body	Nan-in, *
click	next_link	
pause	1200	
verifyText	next_link	Next
verifyText	koan_title	Finding a Diamond on a Muddy Road
verifyText	koan_body	Gudo was *
assertAttribute	koan_titles@style	*visibility: hidden*
assertElementPresent	titles_btn	
verifyText	titles_btn	Show all Koans

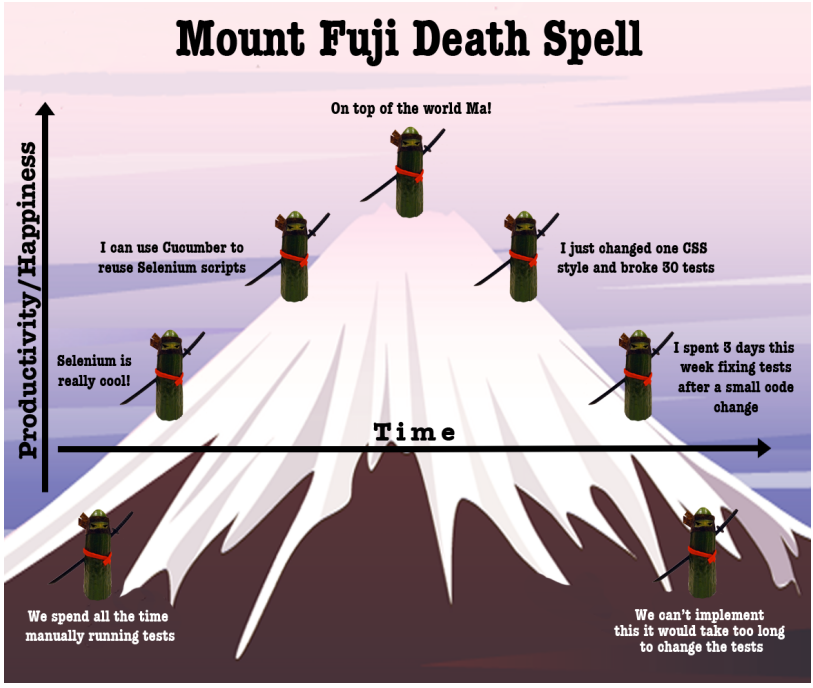
Some ninjas start to describe tests with user interface workflows, not low-level technical actions. This becomes a bit more stable. They reuse actions in workflows, making test scripts shorter, easier to understand and maintain. These tests are not bound to a particular layout, but they are bound to the implementation of a user interface, still not really aligned with the underlying business model. When the page workflow changes, or when the underlying technology changes, the tests still break although the business functionality stays the same.

This is when ninjas reach zen illumination: business rules don't change as much as technical implementations. Technology moves much faster than business. The closer tests are to business rules, the more stable they are. Ninjas realise that a test script describes two things: the specification of a

[1] http://gojko.net/2010/07/29/the-sine-of-death-by-ui-test-automation/

business rule and the way to validate it. The specification does not depend on the user interface, just the validation process.

Figure 8.2. The Mount Fuji of Death

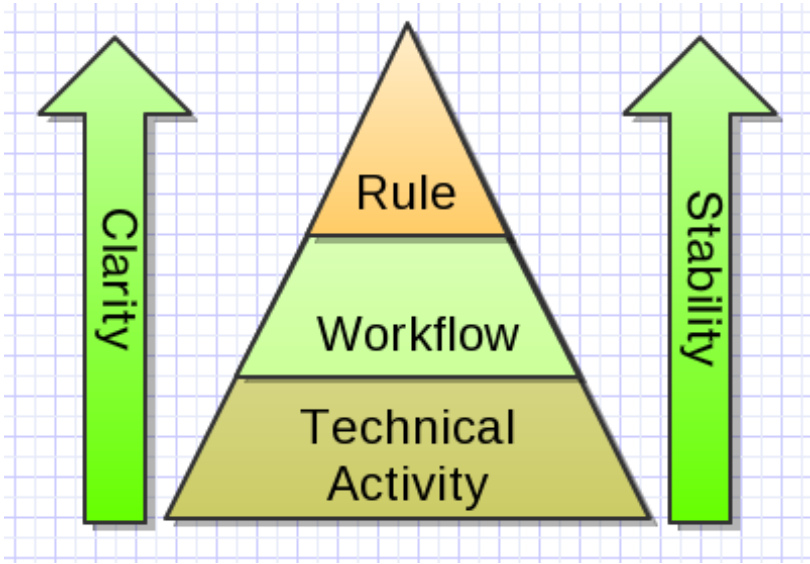


Ninjas then start to divide the two: the specification of a business rule can be written in a language readable even to junior ninjas who don't know the secrets of ancient calligraphy. These specifications are stable and do not change very often. They capture the validation process and automate it with different levels of granularity, applying what they learned about reusable workflows. Ninjas start thinking about the three levels of UI test automation, shown in Figure 8.3).

1. Business rule or functionality level: what is this test demonstrating or exercising. Ideally illustrated with realistic key examples. For example: Free delivery is offered to customers who order two or more books, illustrated with an example of a customer who orders one book and doesn't get free delivery and an example of a customer who orders two books and gets free delivery.

2. User interface workflow level: what does a user have to do to exercise the functionality through the UI, on a higher activity level. For example, put the specified number of books in a shopping cart, enter address details, verify that delivery options include or not include free delivery as expected.
3. Technical activity level: what are the technical steps required to exercise the functionality. For example, open the shop homepage, log in with `testuser` and `testpassword` go to the `/book` page, click on the first image with the `book` CSS class, wait for page to load, click on the 'Buy now' link and so on.

Figure 8.3. The three levels of web automation



And there was much rejoicing. The idea of thinking about these different levels allows ninjas to write UI-level tests that were easy to understand, efficient to write and relatively inexpensive to maintain. This is because there is a natural hierarchy of concepts on these three levels. Checking that delivery is available for two books involves putting a book in a shopping cart. Putting a book in a shopping cart involves a sequence of technical steps. Entering address details does as well. Breaking things down like that and combining lower level concepts into higher level concepts reduces the cognitive load and promotes reuse.

The principle of symmetric change

The best software is the one where the technical software model is aligned with the relevant business domain model. This ensures that one small change in the business domain (new requirements or changes to existing features) results in one small change in software. The same is true for the other artefacts produced for software — tests and documentation. When the tests are aligned with the business domain model, one small change in business will result in one small change in tests, making the tests easy to maintain.

Tests described at a low level technical detail, in the language of technical UI interactions, are everything but aligned with a business model. If anything, they are aligned with the current design and layout of user interfaces. A small change in business requirements can have a huge impact on such tests, requiring hours of updates to tests after just a few minutes of changes to the code.

Easy to understand

From the bottom up, the clarity of the test increases. At the technical activity level, tests are very opaque and full of clutter — it's hard to see the ninja in the forest.[2] At the user interface workflow level, tests describe how something is done, which is easier to understand but still has too much detail to efficiently describe several possibilities. At the business rule level, the intention of the test is described in a relatively terse form. We can use that level to effectively communicate all different possibilities in key examples. It is much more efficient to give another example as 'Free delivery is not offered to customers who have one book' than to talk about logging in, putting only a single book in a cart, checking out etc. Because health and safety regulations we had to take out the text specifying that other example in the language of clicking check boxes and links, because readers noses would bleed after reading it as if they were punched in the face by Chuck Norris himself.

[2]Unless you are Chuck Norris

Efficient to write

From the bottom up, the technical level of tests decreases. At the technical activity level, you need people who understand the design of a system, HTTP calls, DOM and such to write the test. To write tests at the user interface workflow level, you only need to understand the web site workflow. At the business rule level, you need to understand what the business rule is. Given a set of third-level components (eg login, adding a book), testers who are not automation specialists and business users can happily write the definition of second level steps. This allows them to engage more efficiently during development and reduce the automation load on developers.

More importantly, the business rule and the workflow level can be written before the UI is actually there. Tests at these levels can be written before the development starts, and be used as guidelines for development and as acceptance criteria to verify the output.

Relatively inexpensive to maintain

The business rule level is not tied to any particular web site design or activity flow, so it remains stable and unchanged during most web user interface changes, be it layout or workflow improvements. The user interface workflow level is tied to the activity workflow, so when the flow for a particular action changes we need to rewrite only that action. The technical level is tied to the layout of the pages, so when the layout changes we need to rewrite or re-record only the implementation of particular second-level steps affected by that (without changing the description of the test at the business or the workflow level). To continue with the free delivery example from above, if the login form was suddenly changed not to have a button but an image, a ninja would only need to re-write the `login` action at the technical level.

The benefits of three levels

You don't need to be Chuck Norris to know that the changes happen most frequently the technical level — layout, not the activity workflow and definitely not the business rule level. So by breaking up the implementation into this hierarchy, ninjas can create several layers of insulation and limit

the propagation of changes. This reduces the cost of maintenance significantly.

At the same time, introducing the business level is a good start of aligning the test specifications with the underlying business domain model, which then allows ninjas to benefit from the principle of symmetric change. It also enables them to use tests as a living documentation system.

Chapter 9

Getting Ruby ready to exercise the UI

In this chapter we will be covering the basic requirements for testing Web applications in Ruby. We will introduce Rails quickly, but go into more detail about how to set up Cucumber with Sinatra.

Rails

If you are at all serious about using Cucumber to test Rails sites, then you should get hold of the RSpec Book [8], which includes a section discussing BDD outside-in rails development using cucumber in great detail. We will, however, give a very brief overview of the basics for Rails here.

Rails 2

To use Cucumber with Rails 2, you will need the following gems installed on your system:

- **rails**
- **cucumber**
- **cucumber-rails**
- **rspec-rails**
- **webrat**
- **selenium-client**

Install each of these using ruby gems.

Once you have these installed, get things running using the following commands:

```
rails ninjas4Hire
cd ninjas4Hire
script/generate rspec
```

```
script/generate cucumber --webrat --rspec
```

You should now be able to run cucumber tests using the command:

```
rake cucumber
```

... and get a response like this:

```
0 scenarios
0 steps
```

You should now be able to add cucumber features into the `features` directory, and step definitions into the `features/step_definitions` directory.

Rails 3

For a Rails 3 application, first create the Rails site using the following command in a terminal window.

```
rails new ninjas4hire
```

This will create a directory called `ninjas4hire`. Navigate to that directory in your terminal and create a file called `gemfile`. Edit this file so it looks like the example below.

```
source 'http://rubygems.org'
gem 'capybara'
gem 'database_cleaner'
gem 'cucumber-rails'
gem 'cucumber'
gem 'rspec-rails'
gem 'spork'
gem 'launchy'
```

You can now install the gem dependencies using `bundler`:

```
bundle install
```

Once this is complete you should be able to setup the cucumber configuration using the command:

```
ruby script/rails generate cucumber:install --rspec --capybara
```

Sinatra

Due to the fact that Ninjas like to travel light (their deep appreciation of 20th century easy listening crooners is also a factor) we will be using *Sinatra*[1] to illustrate how to use Cucumber to support web application development.

What is Sinatra, Sensei?

Sinatra is a lightweight framework for creating web applications using Ruby. As an example, the following is given for a helloworld example on the Sinatra site.

Create a ruby file called myapp.rb:

```
# myapp.rb
require 'sinatra'

get '/' do
 'Hello world!'
end
```

Install the sinatra gem

```
gem install sinatra
```

Then, run in the directory where you created the file

```
ruby -rubygems myapp.rb
```

View the output by pointing your browser at `http://localhost:4567`. Pretty hep, no?

[1] **http://www.sinatrarb.com/intro**

Gordon is alive?!

To get started, we are going to set up a simple sinatra app, and exercise it with cucumber. First, create a directory called `ninjas4hire`, and get a terminal window pointing at the directory.

Now add a subdirectory called `src` and create a file in there called `ninjas4hire_app.rb`. Edit the file, so it looks like the example below.

```
require 'rubygems'
require 'sinatra/base'
class Ninjas4HireApp < Sinatra::Base
  get '/cukeTest' do
    "up"
  end
end
```

If you haven't done it already, install the sinatra gem.

```
gem install sinatra
```

Now we're going to test that what we have done so far runs. To do this we can use irb.[2] In the terminal window, type

```
$ irb
irb(main):001:0> require "src/ninjas4hire_app.rb"
=> true
irb(main):002:0> Ninjas4HireApp.run!
```

> What did I just do?
>
> **These commands do three things**
>
> 1. **Start the interactive ruby environment**
> 2. **Load the `Ninjas4HireApp` class we created in the `ninjas4hire_app.rb` file**
> 3. **Run the Sinatra app**

[2] **Interactive Ruby (see http://www.ruby-lang.org/en/documentation/quickstart/ for more info)**

You should see some output in the terminal window like this:

```
== Sinatra/1.1.0 has taken the stage on 4567 for development with
backup from WEBrick
[2010-11-23 08:38:49] INFO  WEBrick 1.3.1
[2010-11-23 08:38:49] INFO  ruby 1.8.7 (2010-01-10) [i486-linux]
[2010-11-23 08:38:55] INFO  WEBrick::HTTPServer#start: pid=2588
port=4567
```

If you open a browser and navigate to `http://localhost:4567/cukeTest` **then you should see the word "up" displayed. You can shut down the irb session by typing** `Ctrl+C` **to shut down the app and then entering** `quit`.

Since we don't want to go through such a long winded process each time we run the app, the next step is to create a rake file. In the `ninjas4hire` **directory, create a file called** `rakefile` **and edit it so that it looks like the example below**

```
require 'rake'

desc "run the server"
task :run do |t|
  require "src/ninjas4hire_app.rb"
  Ninjas4HireApp.run!
end
```

If you don't have rake installed, install it now using rubygems:

```
gem install rake
```

Now test that you can run the app using the rake command:

```
rake run
```

Feeding Sinatra with cucumbers

Now we have Sinatra up and running, the next step is to get Cucumber involved. To help with this we will use the `cucumber-sinatra` **gem.[3] This will create some directories and files in our sinatra project, so it is configured for testing with cucumber, RSpec and Capybara.[4]**

[3] **https://github.com/aslakhellesoy/cucumber/wiki/Sinatra**
[4] **https://github.com/jnicklas/capybara**

You will need to have the cucumber gem

If you haven't installed cucumber, gherkin or RSpec yet, there are instructions in the section Installing Cucumber on page 9

First we will need to install the gems for capybara and cucumber-sinatra. The example below also installs a gem called launchy[5] which is a helper library for launching cross-platform applications.

```
gem install cucumber-sinatra capybara launchy
```

Then we will use cucumber-sinatra to set up the configuration for our project

```
cucumber-sinatra init Ninjas4HireApp src/ninjas4hire_app.rb
```

You should see an output like the one in the figure below

```
Generating with init generator:
      [ADDED]  features/support/env.rb
      [ADDED]  features/support/paths.rb
      [ADDED]  features/step_definitions/web_steps.rb
```

The three files it created are:

1. `features/support/env.rb` **sets up capybara and rspec.**
2. `features/support/paths.rb` **enables the overriding of paths with readable alternatives.**
3. `features/step_definitions/web_steps.rb` **created some reusable steps.**

Now we are going to create a simple feature file to check everything is working. First, add the following lines to `paths.rb`**. These will map the text 'the cucumber test page' to the path** `/cukeTest`

```
    when /the home\s?page/
      '/'
  #Add these lines
  when /^the cucumber test page$/
    '/cukeTest'
  #end of added lines
    else
```

[5] http://rubygems.org/gems/launchy/versions/0.3.7

```
      raise "Can't find mapping from \"#{page_name}\" to a path.\n" +
        "Now, go and add a mapping in #{__FILE__}"
    end
```

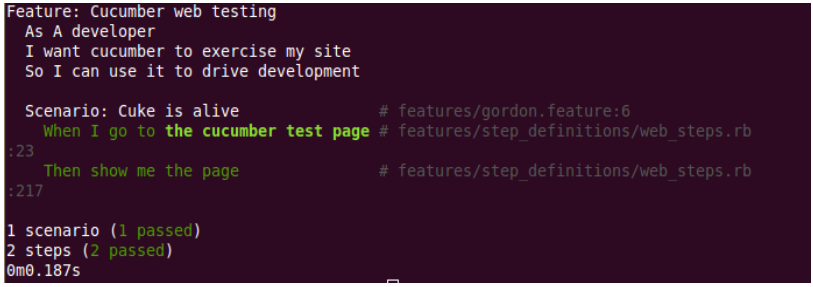
The next step is to add a cucumber feature file to the features directory. You can call it whatever you want, so long as it has the correct extension (`.feature`). [6] Edit the feature file so it looks like the example below.

```
Feature: Cucumber web testing
  As a developer
  I want cucumber to exercise my site
  So I can use it to drive development

  Scenario: Cuke is alive
    When I go to the cucumber test page
    Then show me the page
```

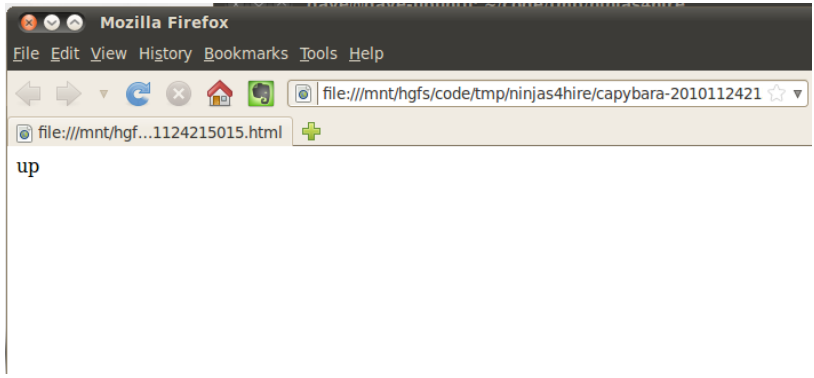
Now run the cucumber feature by opening a terminal window in the root of the project, and typing

```
cucumber features
```

You should see some terminal output similar to the figure below, and a browser window should open, displaying the page.



[6] I called mine gordon.feature because I thought of Brian Blessed shouting "Gordon is ALIVE?!" in the Flash Gordon film



So, we wrote a Cucumber feature and it ran and passed without having to write any step definitions!

This is possible because the `cucumber-sinatra` gem sets up some default steps out of the box. If you take a look at `features/step_definitions/web_steps.rb` you should see a function definition like the one illustrated below.

```
When /^(?:|I )go to (.+)$/ do |page_name|
  visit path_to(page_name)
end
```

This is the step corresponding to the feature file line:

```
When I go to the cucumber test page
```

The body of the method uses two pieces of functionality. Firstly there is a call to `path_to(page_name)`. This function is defined in `paths.rb`. We added a url mapping to this earlier in the chapter. Secondly, the result of this function is passed to visit, which is picked up by capybara to automate the loading of the page at the `/cukeTest` url.

Take a few minutes to look at the step definitions provided in `web_steps.rb`. In the next chapter we will be putting into practice some of the theory outlined in the previous chapter, so we will be using these predefined steps as a template for the technical layer we will be creating.

Raking your cucumber patch

As a final step, we are going to edit our rake file so that we can run our cucumber features using the `rake` command. Happily, Cucumber ships with a rake task so it is simple to do.

Rake adds some overhead, so it is not recommended to use this directly. The usefulness of the rake task comes when running cucumber as part of an automated process because it allows a greater level of control over how and when the cucumber feature are run.

First, edit the file called `rakefile` in the root directory of your ninjas4hire project so it looks like the example below

```
require 'rake'
require 'rubygems'
require 'cucumber'
require 'cucumber/rake/task'

desc "run the server"
task :run do |t|
  require "src/ninjas4hire_app.rb"
  Ninjas4HireApp.run!
end

desc "run cucumber features"
Cucumber::Rake::Task.new(:features) do |t|
  t.cucumber_opts = "features --format pretty"
end
```

Now create a file called `gemfile` if you don't have one already, and edit it so it looks like the example below

```
source 'http://rubygems.org'
gem 'sinatra'
gem 'rspec'
gem 'gherkin' , "2.2.9"
gem 'cucumber'
gem 'cucumber-sinatra'
gem 'capybara'
gem 'launchy'
```

Once you have done this, you should be able run the task using the command:
`rake features`

Chapter 10

.Net and WatiN

In this chapter we will put into practice some of the principles from Chapter 8. To recap, our aim is to automate the testing of a feature in a way that is flexible enough to allow for future changes. We will divide our test code into three layers in order to achieve this. These layers will have the following aims and responsibilities:

- The Business Rules will be expressed in a Cucumber feature file. The feature file will be supported by a step definition class written in C# that will use the API defined in the Workflow layer.
- The Site Workflow will be encapsulated in a fluent API composed of C# classes. This API will attempt to provide a DSL which describes how users navigate through the site to achieve goals that are of interest for the feature(s) described in the Business Rules. It will depend on the Technical Implementation API to provide the means of communicating with the application being tested.
- The Technical Implementation API will provide a finely grained set of classes which will hide the details of UI Automation. It will aim to promote reuse and to insulate the Workflow layer from UI changes

Getting Ready

Before we start, we need to set up the project and get the WatiN libraries.

WatiN

WatiN (pronounced What-in) is a library for Web Application Testing in .Net (the initials form the name). It was inspired by WatiR (pronounced water, where the R stands for Ruby). There is more information about WatiN on SourceForge.[1]

Download the files from SourceForge[2]. For the example in this chapter, I used the 2.x / 2.0 Release candidate.[3]

[1] http://watin.sourceforge.net/
[2] http://sourceforge.net/projects/watin/files/
[3] http://sourceforge.net/projects/watin/files/WatiN%202.x/2.0%20Release%20candidates/

The files should come in zip format, so you will need to unzip them to a suitable directory. Inside the root directory you should find a directory called `Bin`. Inside this directory, there should be a file called `WatiN.Core.dll`. This will need to be referenced in the project we create.

Creating a Project

Create a new project, using the same methods described in Chapter 3 (excepting the creation of feature files and step definition classes). Once this is done, there are a couple of extra steps required so we can use WatiN.

1. Add a reference to `WatiN.Core.dll`
2. Because our example uses Internet Explorer (which is not thread safe![4]), you will need to run the tests as Single Threaded Apartments. The details of how to configure this are in the sidebar *Configuring apartment threading* on page 126.

Configuring apartment threading

Add an `App.config` file to the root of the project. Edit it, adding the lines

```
  <configSections>
    <sectionGroup name="NUnit">
      <section name="TestRunner"
    type="System.Configuration.NameValueSectionHandler"/>
    </sectionGroup>
  </configSections>
  <NUnit>
    <TestRunner>
      <!-- Valid values are STA,MTA. Others ignored. -->
      <add key="ApartmentState" value="STA" />
    </TestRunner>
  </NUnit>
</configuration>
```

[4] http://watin.sourceforge.net/apartmentstateinfo.html

Making sure WatiN is working

Before adding any feature files, it would be useful to check that WatiN is configured correctly. To achieve this, let's add a simple NUnit test. [5]

First, create a folder in your project called `test`. Then add a class file in the new folder called `SmokeTest.cs`. Edit the class so it looks like the example below. Once you have done that, run the test using NUnit.

```
using NUnit.Framework;
using WatiN.Core;

namespace WatiNinja.test
{
    [TestFixture]
    public class SmokeTest
    {
        [Test]
        public void RunSmokeTest()
        {
            using (var browser = new IE("http://www.google.com"))
            {
                const string search = "WatiN";
                browser.TextField(Find.ByName("q")).TypeText(search);
                browser.Button(Find.ByName("btnG")).Click();
                Assert.IsTrue(browser.ContainsText(search));
            }
        }
    }
}
```

CodeTrack

In our example project, we will be using an open source bug tracking tool called CodeTrack[6] as a system to test.

Codetrack requires Apache and PHP, so you will need to install both of these if they are not on your system

[5] I've used the one in the WatiN getting started guide http://watin.sourceforge.net/gettingstarted.html for this example

[6] http://kennwhite.sourceforge.net/codetrack/

Installing Apache

To install apache on windows, download the msi installer from the apache site.[7]

Installing PHP

To install PHP on windows, download the msi installed from the PHP site.[8] Install PHP as an apache module. You may need to alter the apache configuration file so it correctly points at the php-apache library. To do this edit the `httpd.conf` file in the `conf` directory where you installed Apache (on my 64-bit windows system this was `C:\Program Files (x86)\Apache Software Foundation\Apache2.2\conf`). There should be some lines at the bottom of this file

```
#BEGIN PHP INSTALLER EDITS - REMOVE ONLY ON UNINSTALL
PHPIniDir "C:/Program Files (x86)/PHP"
LoadModule php5_module "C:/Program Files (x86)/PHP/
php5apache2_2.dll"
#END PHP INSTALLER EDITS - REMOVE ONLY ON UNINSTALL
```

When I installed they were blank and needed to be edited so that the correct path to `php5apache2_2.dll`was set.

Once these prerequisites are in place, download the zip or tarball from the codetrack site, and extract into the httpd docs directory.[9] After restarting Apache, pointing your browser at `http://localhost/codetrack` should bring up the codetrack login screen.

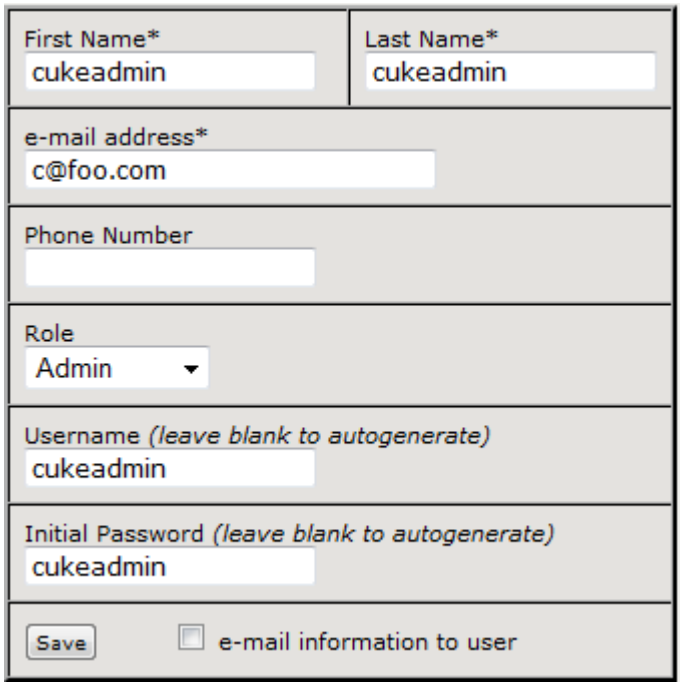
The final step is to create some logins for the test to use.

1. **Login as admin (password codetrack) and enter a new password**

[7] **http://httpd.apache.org/download.cgi**
[8] **http://windows.php.net/download/**
[9] **For example:** `C:\Program Files (x86)\Apache Software Foundation\Apache2.2\htdocs`

2. Select the Admin button at the top of the screen, select the 'Add a User' link, and create an administrative user for the tests to use as illustrated in Figure 10.1
3. Repeat the procedure for users 'Ninja 1' and 'Ninja 2' as illustrated in Figure 10.2 and Figure 10.3

Figure 10.1. Creating cukeadmin user in CodeTrack



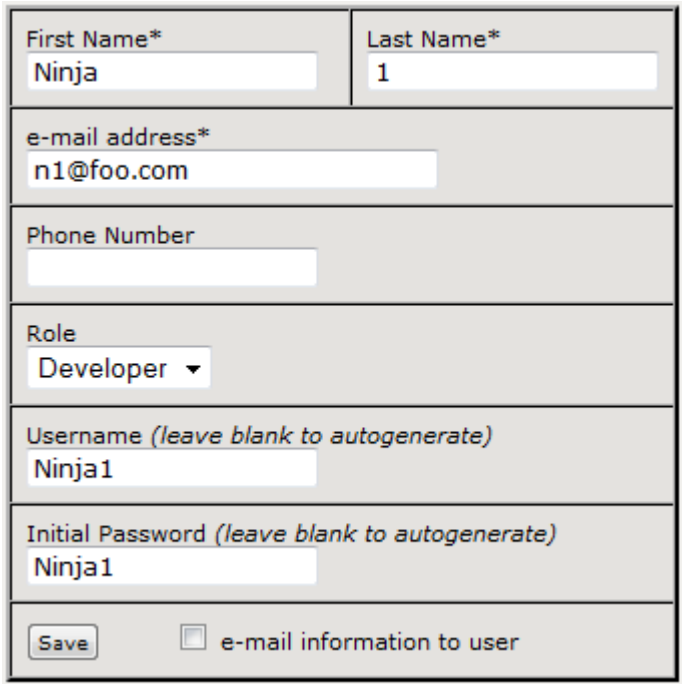
Putting it to the test

Creating the Feature file

Now we have a system to test, we can get on with the main order of business. We will begin by creating a cucumber feature file, and then work down through the step definitions, application workflow and finally the technical layer. In this chapter we will cover the key classes in the project. There is a

completed version of the project that you can download from the cuke4ninja site.[10]

Figure 10.2. Creating ninja1 user in CodeTrack



If you haven't created a `features` folder in the project yet, do this now.[11] Add a new file to the `features` folder called `reportassigned.feature`. Edit it so it looks like the following example.

```
Feature: Report on assigned problems
  In order to see the bugs that are assigned to me
  As a Ninja Developer
  I want to run a run a report to see the problems assigned to me

Scenario: Assigned problem
Given there are open issues with the properties
|Title                  |Severity |
```

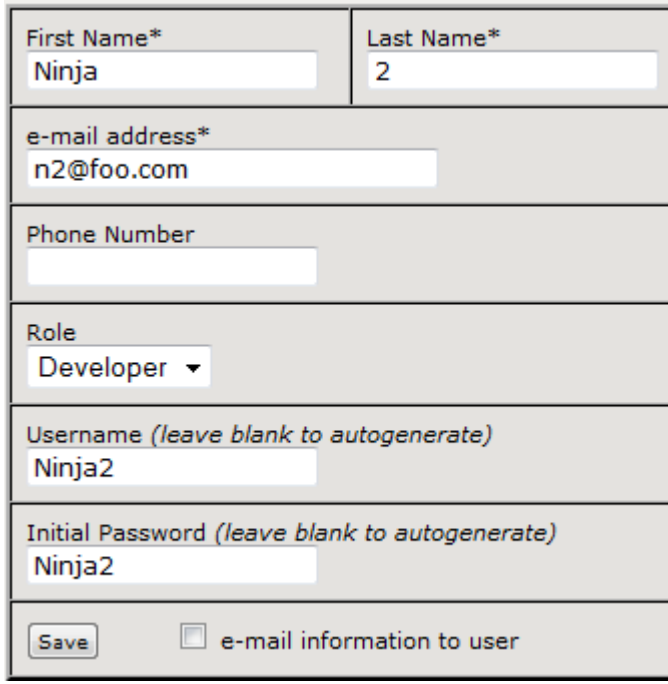
[10] http://cuke4ninja.com/download.html
[11] Don't forget the step_definitions folder and cucumber.wire file as well!

```
|Chuck Norris beat me   |Fatal    |
When the issue "Chuck Norris beat me" is assigned to Ninja2
Then Ninja2 sees the following issues in his report
|Title                  |Severity |
|Chuck Norris beat me   |Fatal    |
And Ninja1 sees no issues in his report
```

So, we want a user to be able to log in and see the issues assigned to them. Though this seems simple enough on first inspection, there are a number of complicating factors. For instance consider the the first part of the scenario

```
Given there are open issues with the properties
|Title                  |Severity |
|Chuck Norris beat me   |Fatal    |
```

Figure 10.3. Creating ninja2 user in CodeTrack



There are a couple of issues that need to be considered when implementing this step, some of which will be non-obvious on first inspection.

- We need to ensure that the issues defined in the step are created. In some cases this could involve direct calls to the business layer API to create the issues, which would involve transforming the table in the feature definition to domain objects. In other scenarios this may be achieved by direct access to the database. In this example we do not have access to either of these options, so we will be creating records using the UI. We will still transform the table into domain objects, however, since doing so will decouple the workflow and technical layers from the test harness.
- The test needs to be repeatable, so issues from previous runs need to be removed, or not visible in the current run. Though post test cleanup may seem advantageous, it is not always possible. Also cleaning up after the test run implies cleaning up after a *failed* test, which can make finding the causes of test failures difficult. In this example we will, instead, be isolating our test by creating a new project for each run.

Step Definitions

Lets start creating our step definitions. In the project we set up earlier, create a folder called `watininja`, and a folder under that called `business`. Create a class called `ReportAssignedSteps.cs` in this folder. At the top, above the `namespace WatiNinja.watininja.business` definition, make sure there are the following imports.

```
using System.Collections.Generic;
using Cuke4Nuke.Framework;
using NUnit.Framework;
using WatiN.Core;
using Table=Cuke4Nuke.Framework.Table;
```

Next, lets set up some fields

```
public class ReportAssignedSteps {
  private const string Admin = "cukeadmin";
  private UserWorkflow _userWorkFlow;
  private string _project;
  private Browser _browser;
```

`Browser` is a WatiN class that allows us to control an instance of a browser. In this example, we will be using IE to test. We will need to reuse the browser field between different step definitions, but close the browser at the end of each scenario. To achieve this behaviour, let's create a lazy loading property to expose the Browser field

```
Browser Browser {
    get {
        if (_browser == null) _browser = new IE();
        return _browser;
    }
}
```

UserWorkflow is one of the workflow layer classes, so we will need to create this class. Create a folder called workflow under the watininja folder we created earlier, and in this folder, add a class called UserWorkflow.cs. We can leave it as an empty class for now. We will need to reuse the workflow between different step definitions, but clean it up at the end of each scenario. As with Browser, let's create a lazy loading property.

```
UserWorkflow UserWorkflow {
    get {
        if (_userWorkFlow == null)
            _userWorkFlow = new UserWorkflow(
        new CodeTrack("http://localhost/codetrack/codetrack.php",
        Browser,
        new UserRepository()));
        return _userWorkFlow;
    }
}
```

We will also add an After method for clean up when the feature is done

```
[After]
public void Cleanup() {
    Browser.Close();
    _browser = null;
  _userWorkFlow = null;
}
```

We are injecting a couple of dependencies into the UserWorkflow constructor. CodeTrack is the entry point for the technical layer and UserRepository is a simulation of a domain API. The CodeTrack class is itself constructed with the Browser and a root URL. In this example the url is hard coded. In a real life example it would be configured. For now though, let's create the stub classes for CodeTrack and UserRepository. Create a class called UserRepository.cs in the watininja folder, create a folder under watininja called technical and create a file called CodeTrack in this new folder. Edit the CodeTrack class so it has the following lines at the top.

```
private readonly string _baseUrl;
private readonly Browser _browser;
private readonly UserRepository _userRepository;

public CodeTrack(string baseUrl,
          Browser browser,
          UserRepository userRepository)
{
    _baseUrl = baseUrl;
    _browser = browser;
    _userRepository = userRepository;
}
```

Now, we will create the step definition for the `given` step. First though, a quick recap of our aims.

- **Set up a new project so test runs are isolated. We need to be refer to this new project later, so some kind of identifier is needed**
- **Create the issues defined in the table, while ensuring that the workflow layer API is not dependent on Cucumber**
- **Create a workflow API which understandable, reusable and in line with the domain language**

Edit the `ReportAssignedSteps` class, adding the following lines

```
[Given("^there are open issues with the properties$")]
public void ThereAreOpenIssuesWithThePropertiesWithTable(Table issues) {
    _project = UserWorkflow.LogonAs(Admin)
                           .CreateNewProject()
                           .AddIssues(issues.ToIssues())
                           .CurrentProject;
}
```

User Workflow

Next, we will need to implement the methods used in the step definition in the workflow classes. For the logon functionality, we need to check we are not logged in (and log out if necessary), then log in with the required user. In the `UserWorkflow` class, add the following code.

```
private void Logout() {
    if (_codeTrack.IsLoggedIn)
        _codeTrack.Logout();
}
```

```
public UserWorkflow LogonAs(string user) {
    Logout();
    LogonForm logonForm = _codeTrack.GotoLogonPage();
    logonForm.Name = user;
    logonForm.Password = user;
    logonForm.Submit();
    return this;
}
```

Notice how the method returns the containing class. This means we can chain method calls which change state in a fluent manner. This method is implemented using the (as yet unwritten) technical layer API, which we will implement in the next section.

The next part of the workflow API to implement is the creation of new projects. We will need to navigate to the part of the application where new projects are created, enter the details for a new project (including a unique project name) and save the new project (returning the identifier for the project for later use). Add the following code into the `UserWorkflow` class.

```
public ProjectWorkflow CreateNewProject() {
    ProjectForm projectForm = _codeTrack.GotoAdminPage().GotoProjectForm();
    string projectName = NextProjectName();
    projectForm.Name = projectName;
    projectForm.Description = "Test Project";
    projectForm.Submit();
    _codeTrack.GotoHomePage();
    return UsingProject(projectName);
}

public ProjectWorkflow UsingProject(string name) {
    var homePage = _codeTrack.GotoHomePage();
    homePage.ProjectName = name;
    _projectWorkflow.CurrentProject = name;
    return _projectWorkflow;
}
private static String NextProjectName() {
    return "CP" + DateTime.Now.ToString("yyyyMMddhhmmss");
}
```

Notice how from these new methods we are returning a `ProjectWorkFlow` rather than a `UserWorkflow`. We will now create a new class called `Project-WorkFlow.cs` in the `workflow` folder, so we can implement the methods for creating new issues there.

Project Workflow

In the `ProjectWorkFlow` we need to implement a constructor, so we can inject the technical layer entry point.

```
private readonly CodeTrack _codeTrack;
public ProjectWorkflow(CodeTrack codeTrack)
{
    _codeTrack = codeTrack;
}
```

We will also need a property, to define the project we are working on

```
public string CurrentProject { set; get; }
```

... and a method that will add issues into the application

```
public ProjectWorkflow AddIssues(IList<Issue> issues) {
    foreach (var issue in issues) {
        IssueForm issueForm = _codeTrack.GotoNewIssueForm();
        issueForm.ProjectName = CurrentProject;
        issueForm.Title = issue.Title;
        issueForm.Description = issue.Title;
        issueForm.Severity = issue.Severity;
        issueForm.Submit();
    }
    return this;
}
```

Issue and Table

We will also need a way of converting between the Cucumber Table and a collection of `Issue` objects. First, let's create a class called `Issue.cs` in the `watininja` folder. Edit so it looks like the example below.

```
public Issue(string title, string severity) {
    Title = title;
    Severity = severity;
}
public string Title { get; private set; }
public string Severity { get; private set; }
public bool Equals(Issue other) {
    if (ReferenceEquals(null, other)) return false;
    if (ReferenceEquals(this, other)) return true;
```

```
    return Equals(other.Title, Title) && Equals(other.Severity, Severity);
}
public override bool Equals(object obj) {
    if (ReferenceEquals(null, obj)) return false;
    if (ReferenceEquals(this, obj)) return true;
    if (obj.GetType() != typeof(Issue)) return false;
    return Equals((Issue)obj);
}
public override int GetHashCode() {
    unchecked {
        return ((Title != null ? Title.GetHashCode() : 0) * 397) ^
        (Severity != null ? Severity.GetHashCode() : 0);
    }
}
```

Then add a class called `TableConverter.cs` into the `business` folder, and edit it to be like the following example.

```
    public static class TableConverter
    {
        public static IList<Issue> ToIssues(this Table propertiesList)
        {
            var list = new List<Issue>();
            foreach (var properties in propertiesList.Hashes())
            {
                var issue = new Issue(
                    properties["Title"].Trim(),
                    properties["Severity"].Trim());
                list.Add(issue);
            }
            return list;
        }
    }
```

This will allow the `ToIssues` method to be available as an extension method.

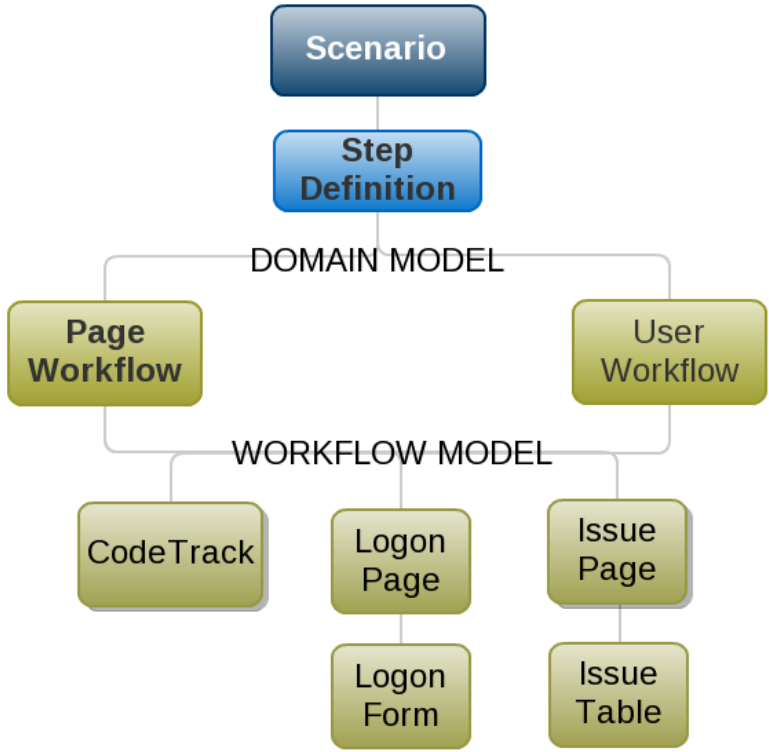
The Technical Layer

So far we have created a step definition, and some workflow classes. We have also created a class to represent the entry point to the technical layer API. This is illustrated in Figure 10.4.We can now progress with the classes that make up the technical layer API.

With the exception of the `CodeTrack` class, all the classes in the technical layer can be described as being *Pages* or *Forms*, with the difference being that

Forms are submitted. Both will need access to the `Browser` in order to manipulate the application UI. Though complex object hierarchies should in general be avoided, in this case putting some shared functionality into base classes seems reasonable.

Figure 10.4. Overview of the Business, Workflow and Technical Layers



Lets start by creating a class called `Page.cs` in the `technical` folder. Edit it so it looks like the example below.

```
public class Page {
  protected Browser Browser { private set; get; }
  public Page(Browser browser) {
      Browser = browser;
```

```
  }
  public void SelectOption(AttributeConstraint by, String text) {
      var select = Browser.SelectList(by);
      select.Select(text);
  }
}
```

Now create a class called `Form.cs`, again in the technical folder. Edit it so it looks like the example below.

```
public class Form : Page {
    public Form(Browser browser) : base(browser){}
    public virtual void Submit() {
        Browser.Button(Find.By("type", "submit")).Click();
    }
}
```

Now we are ready to create all the Page and Form classes which are used in the Workflow methods we defined earlier.

- **LogonForm**
- **AdminPage**
- **ProjectForm**
- **HomePage**
- **IssueForm**

We will also need to implement methods on the CodeTrack class.

- **GotoLogonPage**
- **GotoHomePage**
- **Logout**
- **GotoNewIssueForm**
- **IsLoggedIn**

Most of these classes are fairly similar, and the source for these classes is can be downloaded from the cuke4ninja site, so I'm not going to list every class here. We will look at a some of the implementation details though.

First, create the class `LogonForm.cs` in the `technical` folder, and edit it so it looks like the example below.

```
public class LogonForm : Form
{
```

```
    public LogonForm(Browser browser) : base(browser) {}
    public string Name {
        set {
            Browser.TextField(Find.ByName("userLogin[username]"))
        .TypeText(value);
        }
    }

    public string Password {
        set {
            Browser.TextField(Find.ByName("userLogin[password]"))
        .TypeText(value);
        }
    }

}
```

As you can see, the Name property setter is implemented using WatiN functionality. The WatiN method `Browser.TextField` will find a DOM element on the page matching the supplied predicate, in this case `Find.ByName(...)`. We can then set the value of this text element by typing text into it, using the `.TypeText(value)` method. Most of the WatiN functionality can be used in a similar manner, selecting an element, or group of elements using a predicate expression, then reading from the element or performing operations on it.

Another example of WatiN interacting with elements can be seen in the `IssuePage` class. Here we are using the `Click()` method os simulate the user clicking on a link

```
public class IssuePage : Page {
    private readonly UserRepository _repository;

    public IssuePage(Browser browser, UserRepository repository) : base(browser)
        _repository = repository;
    }

    public IssueForm StartEdit() {
        Browser.Element(Find.By("type","submit")).Click();
        return new IssueForm(Browser, _repository);
    }
}
```

The methods we need to add to the `CodeTrack` class are, in the most part, navigational. The implementation for `GotoLogonPage` is given below.

```
private string Url(string relativePath) {
    return string.Format("{0}{1}", _baseUrl, relativePath);
}

public LogonForm GotoLogonPage() {
    _browser.GoTo(Url("?page=login"));
    return new LogonForm(_browser);
}
```

Here, we are using the WatiN `Browser` method `Goto(url)` to navigate to a particular page on the site. The private helper method `Url(string relativePath)` is adding the relative path to the base url we constructed the `CodeTrack` object with.

See if you can implement the remaining methods on the technical layer. Download the source from the cuke4ninja site and refer to that if you get stuck.

The When and Then Step Definitons

Let's go back to the step definitions and implement the `When` and `Then` steps. Add the following code for the `when` step

```
[When("^the issue \"(.*)\" is assigned to (.*)$")]
public void TheIssueWithTitleIsAssignedToUser(
  string issueTitle, string user) {
    UserWorkflow.LogonAs(Admin)
                .UsingProject(_project)
                .AssignIssueToUser(issueTitle, user);
}
```

Already, we can see some reuse beginning to pay back the effort we put in earlier. We only have to implement the `AssignIssueToUser` method to get this step done. This method should be added to the `ProjectWorkFlow`

```
public ProjectWorkflow AssignIssueToUser(string issueTitle, String user) {
    HomePage homePage = _codeTrack.GotoHomePage();
    homePage.ProjectName = CurrentProject;
    IssuePage issuePage = homePage.ShowIssueWithTitle(issueTitle);
    IssueForm issueForm = issuePage.StartEdit();
    issueForm.AssignTo(user);
    issueForm.Submit();
    return this;
}
```

There are two `Then` steps to implement, for users having issues assigned to them and for those without issues

```
[Then("^(.*) sees the following issues in his report$")]
public void UserSeesTheFollowingIssuesInHisReportWithTable(
  string user, Table issues) {
    IList<Issue> reportedIssues = UserWorkflow.LogonAs(user)
                                              .UsingProject(_project)
                                              .ViewAssignedIssuesReport()
                                              .Issues;

    Assert.AreEqual(issues.ToIssues(), reportedIssues);
}

[Then("^(.*) sees no issues in his report$")]
public void UserSeesNoIssuesInHisReport(string user) {
    Assert.AreEqual(0, UserWorkflow.LogonAs(user)
                                   .UsingProject(_project)
                                   .ViewAssignedIssuesReport()
                                   .NumberOfIssues);
}
```

These require just the implementation of one method in `ProjectWorkflow`

```
public AssignedIssuesReport ViewAssignedIssuesReport()
{
    ReportsPage reportsPage = _codeTrack.GotoReportsPage();
    IssueTable issueTable = reportsPage.ShowIssuesAssignedToLoggedInUser();
    return new AssignedIssuesReport(issueTable.Issues);
}
```

We will need to create a few new classes in the technical layer to support this implementation

- **ReportsPage**
- **IssueTable**
- **AssignedIssuesReport**

See if you can implement the remaining methods on the technical layer. Download the source from the cuke4ninja site and refer to that if you get stuck.

Appendix A.

Resources

Books

[1] Gojko Adzic. Copyright © 2009. Neuri. *Bridging the Communication Gap*. Specification by Example and Agile Acceptance Testing. 0955683610.

[2] Lisa Crispin and Janet Gregory. Copyright © 2009. Addison-Wesley Professional. *Agile Testing*. A Practical Guide for Testers and Agile Teams. 0201835959.

[3] Craig Larman and Bas Vodde. Copyright © 2010. Pearson Education. *Practices for Scaling Lean and Agile Development*. Large, Multisite, and Offshore Product Development with Large-Scale Scrum. 9780321636409.

[4] Eric Evans. Copyright © 2003. Addison-Wesley Professional. *Domain-Driven Design*. Tackling Complexity in the Heart of Software. 0321125215.

[5] Gerald M. Weinberg and Donald C. Gause. Copyright © 1989. Dorset House Publishing Company. *Exploring Requirements: Quality Before Design*. 0932633137.

[6] Gerald Weinberg. Copyright © 1992. Dorset House Publishing. *Quality Software Management : Vol. 1*. Systems Thinking. 978-0932633224.

[7] Mike Cohn. Copyright © 2004. Addison-Wesley Professional. *User Stories Applied: For Agile Software Development*. 978-0321205681.

[8] David Chelimsky. Dave Astels. Zach Dennis. Aslak Hellesøy. Bryan Helmkamp. Dan North. Copyright © 2010. Pragmatic Bookshelf. *The RSpec Book*. Behaviour Driven Development with RSpec, Cucumber and Friends. 978-1934356371.

Tools

- **Cucumber main web site http://cukes.info**
- **Cucumber wiki http://wiki.github.com/aslakhellesoy/cucumber/**
- **Specflow http://specflow.org**
- **Cuke4Nuke http://github.com/richardlawrence/Cuke4Nuke**
- **Cuke4Duke http://wiki.github.com/aslakhellesoy/cuke4duke/**
- **Cuke4Nuke Visual Studio Plugin http://github.com/henritersteeg/cuke4vs**

Articles

- **Dan North: Introducing BDD http://blog.dannorth.net/introducing-bdd/**
- **Dan North: Whose domain is it anyway? http://dannorth.net/2011/01/31/whose-domain-is-it-anyway/**
- **Liz Keogh: Pulling Power http://www.infoq.com/articles/pulling-power**

Videos

- **Dan North: How to Sell BDD to the business http://skillsmatter.com/podcast/java-jee/how-to-sell-bdd-to-the-business**
- **Gojko Adzic: Behaviour-Driven Development with Cucumber http://gojko.net/2010/02/08/behaviour-driven-development-with-cucumber-video-slides-and-links/**
- **Eric Evans: Folding Together DDD and Agile http://skillsmatter.com/podcast/design-architecture/folding-together-ddd-agile**
- **Gojko Adzic: TDD, DDD and BDD http://skillsmatter.com/podcast/design-architecture/ddd-tdd-bdd**