Cucumber and Gherkin for Java

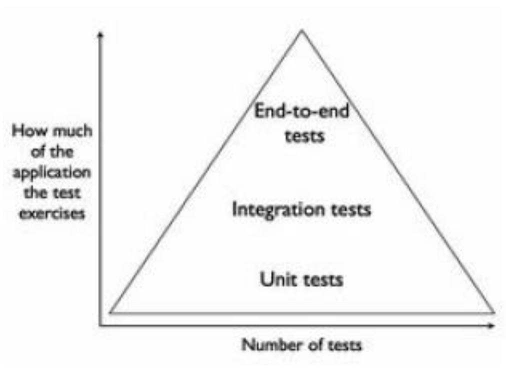
Behaviour-Driven Development (BDD)

Last updated: 10 July 2017

Contents

Introduction

Behaviour-Driven Development (BDD) extends traditional unit testing with sensible automated acceptance testing. BDD facilitates clearer and unambiguous communication and continuous feedback between the business stakeholders and development and testing. Traditional unit testing is still the foundation of automated testing and is required for fast code checks. The focus of traditional unit testing is on the program code internals (such as statement, branch, and path coverage), while BDD ensures software quality from a business-oriented perspective. Traditional unit testing will still have the highest number of test cases and coverage.



BDD formalises and builds on the principles of Test-Driven Development (TDD). TDD reverses the usual sequence of coding first and testing afterwards. In TDD, tests are written in advance of the code needed to satisfy them.

Instead of business stakeholders passing requirements to the development team without much opportunity for feedback, BDD allows cooperation between business stakeholders and development. BDD ensures that the software design meets the need of the actual code and leaves behind a suite of tests and up-to-date documentation to help preserve the integrity of the code.

Cucumber tests interact directly with the developers' code, but they're written in a medium and language that business stakeholders can understand. By working together to write these tests not only do the team members decide what behaviour they need to implement next, but they learn how to describe that behaviour in a common language that everyone understands.

What makes Cucumber stand out from the crowd of other tools is that it has been designed specifically to ensure that automated acceptance tests can easily be read - and written - by anyone on the team. This reveals the true value of acceptance tests as a communication and collaboration tool. The easy readability of Cucumber acceptance tests draws business stakeholders into the process, helping developers explore and understand their requirements.

Cucumber test description files (also called "feature files") share the benefit of traditional specification documents in that they can be written and read by business stakeholders, but they have a distinct advantage in that a computer can understand them, too. In practice, this means that the documentation, rather than being something that's written once and then gradually goes out of date, becomes a living thing that reflects the true state of the project.

The decision about whether or not a behaviour should be specified in the feature files (or covered by traditional unit tests) should be based on whether the business stakeholders are interested in it.

So that Cucumber can understand these feature files, they must follow some basic syntax rules. The name for this set of rules is Gherkin. Although Gherkin is sometimes called a programming language, its primary design goal is human readability, meaning you can write automated tests that read like documentation.

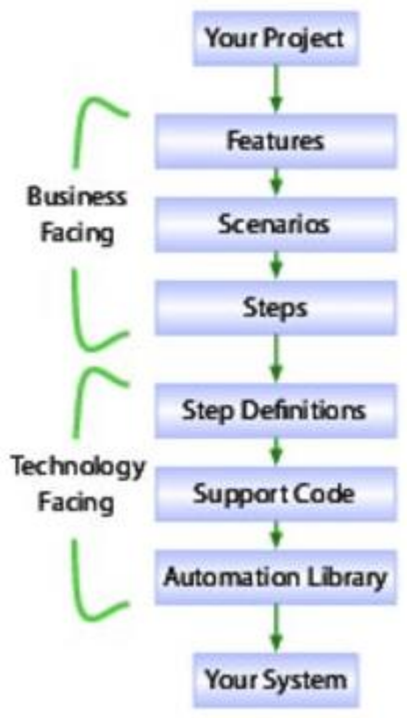
Cucumber Workflow

Each test case in Cucumber is called a scenario, and scenarios are grouped into features. Each scenario contains several steps.

The business-facing parts of a Cucumber test suite, stored in feature files, must be written according to syntax rules - known as Gherkin - so that Cucumber can read them.

Under the hood, step definitions translate from the business-facing language of steps into code.

This is a high-level overview of a typical Cucumber test suite:



Cucumber reads in the specifications from plain language text files called features, examines them for scenarios to test, and runs the scenarios against the system. Each scenario is a list of steps for Cucumber to work through. So that Cucumber can understand these feature files, they must follow some basic syntax rules. The name for this set of rules is Gherkin.

Along with the features, you give Cucumber a set of step definitions, which map the business readable language of each step into code (written in Java) to carry out whatever action is being described by the step. In a mature test suite, the step definition itself will probably just be one or two lines of code that delegate to a library of support code, specific to the domain of your application, that knows how to carry out common tasks on the system. Sometimes that may involve using an automation library, like the browser automation library Selenium, to interact with the system itself.

If the code in the step definition executes without error, Cucumber proceeds to the next step in the scenario. If it gets to the end of the scenario without any of the steps raising an error, it marks the scenario as having passed. If any of the steps in the scenario fail, however, Cucumber marks the scenario as having failed and moves on to the next one. As the scenarios run, Cucumber prints out the results showing you exactly what is working and what isn't.

Gherkin

Keywords

Keyword	Purpose
`Feature`	Each Gherkin file begins with the "`Feature`" keyword. This keyword doesn't really affect the behaviour of your Cucumber tests at all; it just gives you a convenient place to put some summary documentation about the group of tests that follow. The description can span multiple lines. The text immediately following on the same line as the Feature keyword is the "name" of the feature, and the remaining lines are its "description". In valid Gherkin, a "`Feature`" must be followed by one of the following: • `Scenario` • `Background` • `Scenario Outline`
`Background`	Allows you to add some context to the scenarios in a single feature. A "`Background`" is much like a scenario containing a number of steps. The difference is when it is run. The background is run before each of your scenarios but after any of your "`Before`" Hooks. You can have a single "`Background`" element per feature file, and it must appear before any of the "`Scenario`" or "`Scenario Outline`" elements. Backgrounds are useful for taking "`Given`" (and sometimes "`When`") steps that are repeated in each scenario and moving them to a single place. This helps keep your scenarios clear and concise.
`Scenario`	To actually express the behaviour we want, each feature contains several scenarios. Each scenario is a single concrete example of how the system should behave in a particular situation. Like "`Feature`", a "`scenario`" description can also use multiple lines.
`Scenario Outline`	Allows a scenario to use its own data table for parameters. We indicate placeholders within the scenario outline using angle brackets (<.. >) where we want real values to be substituted. The scenario outline itself is useless without an "`Examples`" table, which lists rows of values to be substituted for each placeholder. In a scenario outline, each row of an "`Examples`" table represents a whole scenario to be executed by Cucumber. In fact, you might want to use the keyword "`Scenarios`" (note the extra s) in place of "`Examples`" if you find that more readable. We can use a placeholder to replace any of the text we like in a step. It doesn't

	matter what order the placeholders appear in the table; what counts is that the column header matches the text in the placeholder in the scenario outline. A single "`Scenario Outline`" can have one or multiple "`Example`" tables attached to it. This allows (logical) grouping and an almost unlimited amount of rows of values.
`Given`	Set up the context where the scenario happens.
`When`	Interact with the system (Actions).
`Then`	Check that the outcome of the "`When`" interaction was what we expected.
`And` (or alternatively "`But`")	Combines multiple "`Given`", "`When`" or "`Then`" statements together.
`*`	Can be used as (less verbose) alternative for the "`Given`", "`When`", "`Then`", "`And`" and "`But`" keywords.
`#`	Comments start with a "`#`" character and have to be the first and only thing on a line.
`"""` … `"""`	Doc strings (the text between triple quotes) allow you to specify a larger piece of text than you could fit on a single line. Just like a data table, the entire string between the `"""` triple quotes is attached to the step above it. Doc strings open up all kinds of possibilities for specifying data in your steps.
`@`*tagname*	Tag names can be any text. One or multiple tags can be used before "`Scenario`", or "`Feature`", or "`Scenario Outline`" elements and the individual "`Examples`" tables under them. If a tag is used before "`Feature`", it will apply to all scenarios within the feature. There are three main reasons for tagging: 1. Documentation: You want to use a tag to attach a label to certain scenarios, for example to label them with an ID from a project management tool. 2. Filtering: Cucumber allows you to use tags as a filter to pick out specific scenarios to run or report on. You can even have Cucumber fail your test run if a certain tag appears too many times. 3. Hooks: Run a block of code whenever a scenario with a particular tag is about to start or has just finished.

Data tables

Data tables are a great feature of Gherkin. They're really versatile (allowing multiple formats of data, for example with or without headers), and they help you express data concisely, as you'd want to in a normal specification document.

The documentation for "`cucumber.api.DataTable`" can be found at:
http://cucumber.github.io/api/cucumber/jvm/javadoc/cucumber/api/DataTable.html .

Data tables are not to be confused with the tables used under the "`Scenario Outline`" keyword. Data tables just describe a lump of data to attach to a single step of a single scenario. "`Scenario Outline`" on the other hand generates multiple scenarios during execution, where every scenario uses a different set of data.

Hooks

Cucumber supports hooks, which are methods that run before or after each scenario. You can define them anywhere in your support or step definition layers, using the annotations "`@Before`" and "`@After`".

Hooks are global by default, meaning they run for every scenario in all of your features. If you want them to run for just certain scenarios, you need to tag those scenarios and then use a tagged hook.

Tagged Hooks

Both "`@Before`" and "`@After`" accept a tag expression, which you can use to selectively add the hook to only certain scenarios.

If a tagged hook is attached to a feature, then it will run before or after every scenario in this particular feature.

If a tagged hook is attached to a scenario, it will only run before or after that particular scenario.

Hook Examples:

The following hook runs before every scenario:

```
@Before
public void beforeCallingScenario() {
      System.out.println("*********** About to start the scenario.");
}
```

The following tagged hook runs just before a scenario (or feature) that is tagged with "ExampleTaggedScenario":

```
@Before("@ExampleTaggedScenario")
public void beforeCallingTaggedScenario() {
      System.out.println("*********** About to start the tagged scenario
'ExampleTaggedScenario'.");
}
```

Hook Order

Sometimes it's important to be able to specify the exact order that your hooks run in. The "`@Before`" and "`@After`" annotations have an order parameter that you can set. The default value is 10000 for any hook that doesn't have a specific order set. The following example gives a "`@Before`" hook an order of 20:

```
@Before( order = 20 )
public void exampleHook() {…}
```

Cucumber runs “`@Before`” hooks from low to high. A “`@Before`” hook with an order of 10 will run before one with an order of 20. “`@After`” hooks run in the opposite direction— from high to low— so a “`@After`” hook with an order of 20 will run before one with an order of 10.

If you need to use order on a tagged hook, you have to use the “`value`” parameter for the tag name:

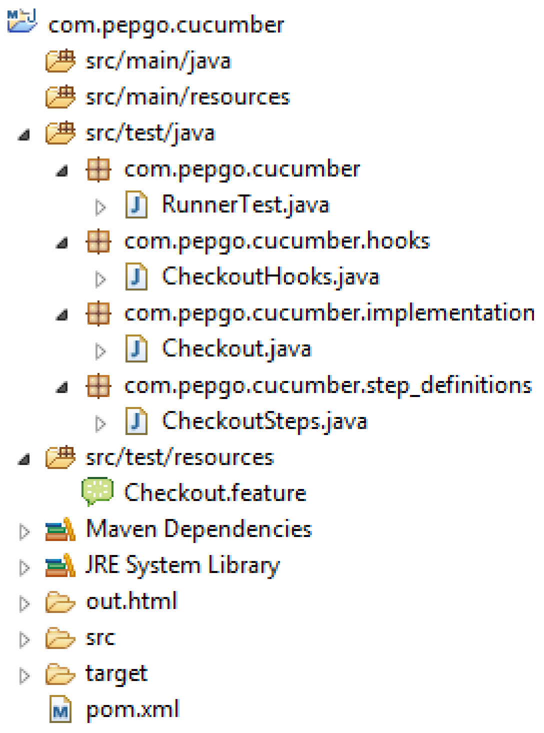
```
@Before( value ="myTag", order = 20)
public void exampleHook() {…}
```

A practical Cucumber and Gherkin example

This example tests a supermarket checkout, where customers can buy apples and bananas.

The example project has six custom files:

1. “`RunnerTest.java`” contains the runner class that drives the execution.
2. “`CheckoutHooks.java`” contains hooks.
3. “`Checkout.java`” contains support code (additional program code not bound to a particular step).
4. “`CheckoutSteps.java`” contains the step definitions.
5. “`Checkout.feature`” contains the tests in Gherkin syntax.
6. “`pom.xml`” contains the Maven setup.



```
com.pepgo.cucumber
    src/main/java
    src/main/resources
  ◢ src/test/java
    ◢ com.pepgo.cucumber
      ▷ RunnerTest.java
    ◢ com.pepgo.cucumber.hooks
      ▷ CheckoutHooks.java
    ◢ com.pepgo.cucumber.implementation
      ▷ Checkout.java
    ◢ com.pepgo.cucumber.step_definitions
      ▷ CheckoutSteps.java
  ◢ src/test/resources
      Checkout.feature
  ▷ Maven Dependencies
  ▷ JRE System Library
  ▷ out.html
  ▷ src
  ▷ target
    pom.xml
```

The feature file “Checkout.feature”

This file contains the features and scenarios, written in Gherkin. In this example, one of the scenarios contains a table with 2 different sets of data. Cucumber will therefore execute this scenario twice (one execution for each set of data).

This example feature also contains multiple hooks that are used by the “`CheckoutHooks.java`” class.

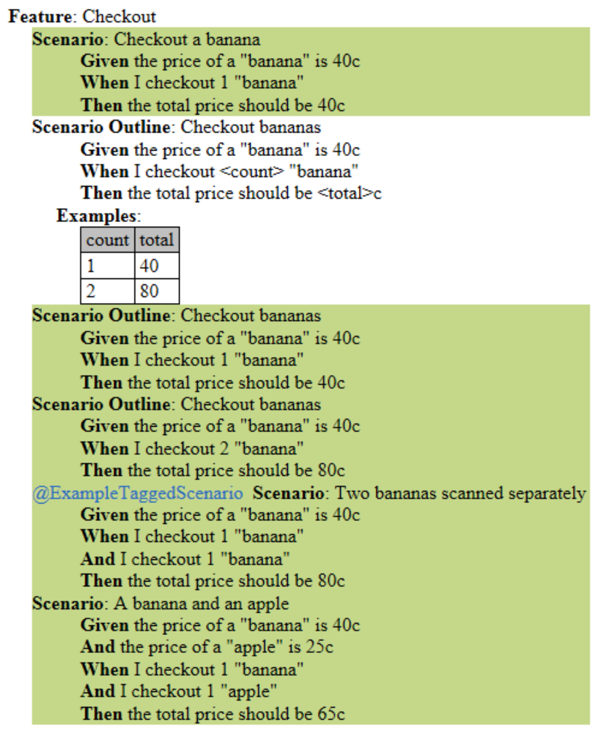
```
Feature: Checkout

Scenario: Checkout a banana
Given the price of a "banana" is 40c
When I checkout 1 "banana"
Then the total price should be 40c

Scenario Outline: Checkout bananas
Given the price of a "banana" is 40c
When I checkout <count> "banana"
Then the total price should be <total>c
Examples:
| count | total |
| 1     | 40    |
| 2     | 80    |

@ExampleTaggedScenario
Scenario: Two bananas scanned separately
Given the price of a "banana" is 40c
When I checkout 1 "banana"
And I checkout 1 "banana"
Then the total price should be 80c

Scenario: A banana and an apple
Given the price of a "banana" is 40c
And the price of a "apple" is 25c
When I checkout 1 "banana"
And I checkout 1 "apple"
Then the total price should be 65c
```

The runner class "RunnerTest.java"

This class drives the execution. The runner class itself must be empty and have the word "Test" in its name (in this example the class is called "RunnerTest"), if test should be executed through Maven from command line, for example with the command "`mvn clean test`".

```
package com.pepgo.cucumber;

import org.junit.runner.RunWith;

import cucumber.api.CucumberOptions;
import cucumber.api.junit.Cucumber;

@RunWith(Cucumber.class)
@CucumberOptions(
        features="src/test/resources",
        plugin="html:out.html")
public class RunnerTest {

}
```

The "plugin" setting of this example produces a HTML report that looks like this (in this example in a folder called "out.html"):



Feature: Checkout

Scenario: Checkout a banana

Given the price of a "banana" is 40c

When I checkout 1 "banana"

Then the total price should be 40c

Scenario Outline: Checkout bananas

Given the price of a "banana" is 40c

When I checkout <count> "banana"

Then the total price should be <total>c

Examples:

count	total
1	40
2	80

Scenario Outline: Checkout bananas

Given the price of a "banana" is 40c

When I checkout 1 "banana"

Then the total price should be 40c

Scenario Outline: Checkout bananas

Given the price of a "banana" is 40c

When I checkout 2 "banana"

Then the total price should be 80c

@ExampleTaggedScenario **Scenario**: Two bananas scanned separately

Given the price of a "banana" is 40c

When I checkout 1 "banana"

And I checkout 1 "banana"

Then the total price should be 80c

Scenario: A banana and an apple

Given the price of a "banana" is 40c

And the price of a "apple" is 25c

When I checkout 1 "banana"

And I checkout 1 "apple"

Then the total price should be 65c

The step definitions class “CheckoutSteps.java”

This class contains the methods that map the scenario steps of the feature file

```
package com.pepgo.cucumber.step_definitions;

import com.pepgo.cucumber.implementation.Checkout;

import cucumber.api.java.en.*;
import static org.junit.Assert.*;

public class CheckoutSteps {

        int bananaPrice = 0, applePrice = 0;
        Checkout checkout = new Checkout();

        @Given("^the price of a \"(.*?)\" is (\\d+)c$")
        public void thePriceOfAIsC(String name, int price) throws Throwable {
                        if (name.equals("banana")) {
                                bananaPrice = price;
                        }
                        else {
                                applePrice = price;
                        }
        }

        @When("^I checkout (\\d+) \"(.*?)\"$")
        public void iCheckout(int itemCount, String itemName) throws Throwable {
                if (itemName.equals("banana")) {
                        checkout.add(itemCount, bananaPrice);
                }
                else {
                        checkout.add(itemCount, applePrice);
                }
        }
```

```
	@Then("^the total price should be (\\d+)c$")
	public void theTotalPriceShouldBeC(int total) throws Throwable {
		assertEquals(total, checkout.total());
	}
}
```

The hooks class “CheckoutHooks.java”

This class contains sample hooks to demonstrate the use and lifecycle of hooks.

```
package com.pepgo.cucumber.hooks;

import cucumber.api.java.After;
import cucumber.api.java.Before;
import cucumber.api.Scenario;

public class CheckoutHooks {

	// Runs before every scenario
	@Before
	public void beforeCallingScenario() {
		System.out.println("*********** About to start the scenario.");
	}

	// Runs after every scenario
	@After
	public void afterRunningScenario(Scenario scenario) {
		System.out.println("*********** Just finished running scenario: "
			+ scenario.getStatus());
	}

	// Runs just before the scenario that is tagged with "ExampleTaggedScenario"
	@Before("@ExampleTaggedScenario")
	public void beforeCallingTaggedScenario() {
		System.out.println("*********** About to start the tagged scenario 
'ExampleTaggedScenario'.");
	}

	// Runs just after the scenario that is tagged with "ExampleTaggedScenario"
	@After("@ExampleTaggedScenario")
	public void afterRunningTaggedScenario(Scenario scenario) {
		System.out.println("*********** Just finished running the tagged scenario 
'ExampleTaggedScenario': "
			+ scenario.getStatus());
	}
}
```

The support code class “Checkout.java”

This class just contains support code. It this example, the class provides an object that is used to exchange data between different steps.

```
package com.pepgo.cucumber.implementation;

public class Checkout {

	private int runningTotal = 0;

	public void add(int count, int price) {
		runningTotal += (count * price);
```

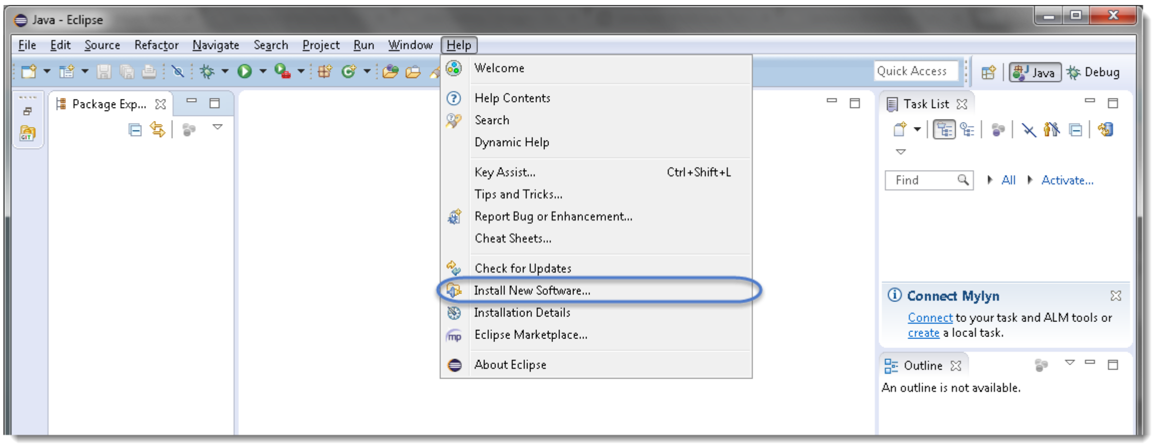
```
        }

        public int total() {
                return runningTotal;
        }
}
```

The Maven settings file "pom.xml"

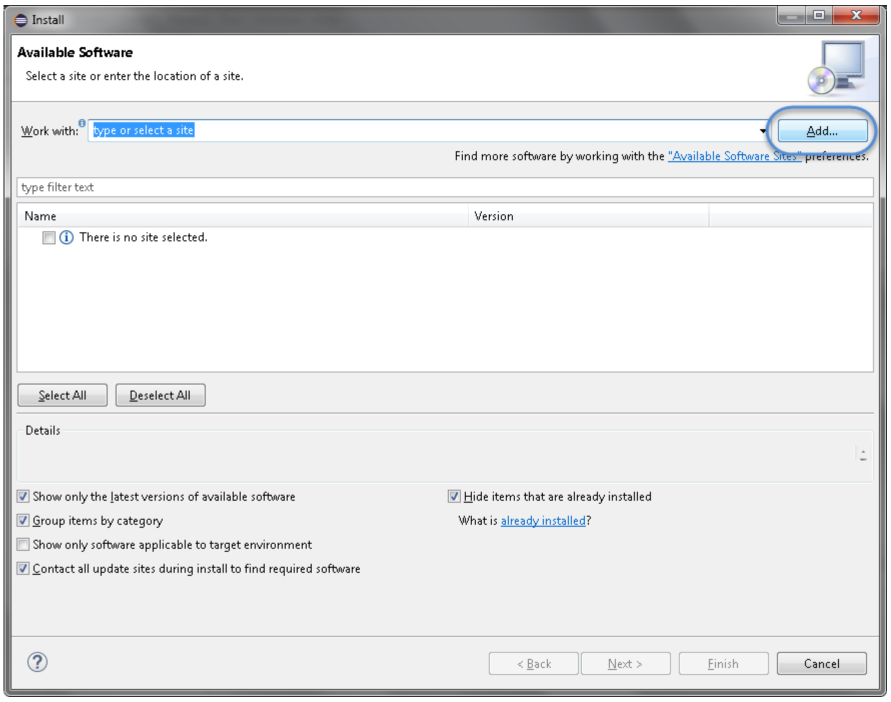
This file is described separately later in this document.

Install the Cucumber Eclipse Plugin

1) Launch the Eclipse IDE and from Help menu, click “Install New Software”.

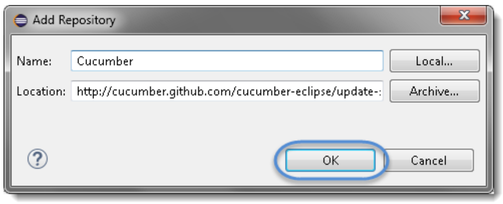


2) You will see a dialog window, click the “Add” button.


Install
Available Software
Select a site or enter the location of a site.
Work with:
type or select a site
Add...
Find more software by working with the "Available Software Sites" preferences.
type filter text
Name
Version
There is no site selected.
Select All
Deselect All
Details
Back
Next >
Finish
Cancel

[✓] Show only the latest versions of available software
[✓] Group items by category
[ ] Show only software applicable to target environment
[✓] Contact all update sites during install to find required software
[✓] Hide items that are already installed
What is already installed?

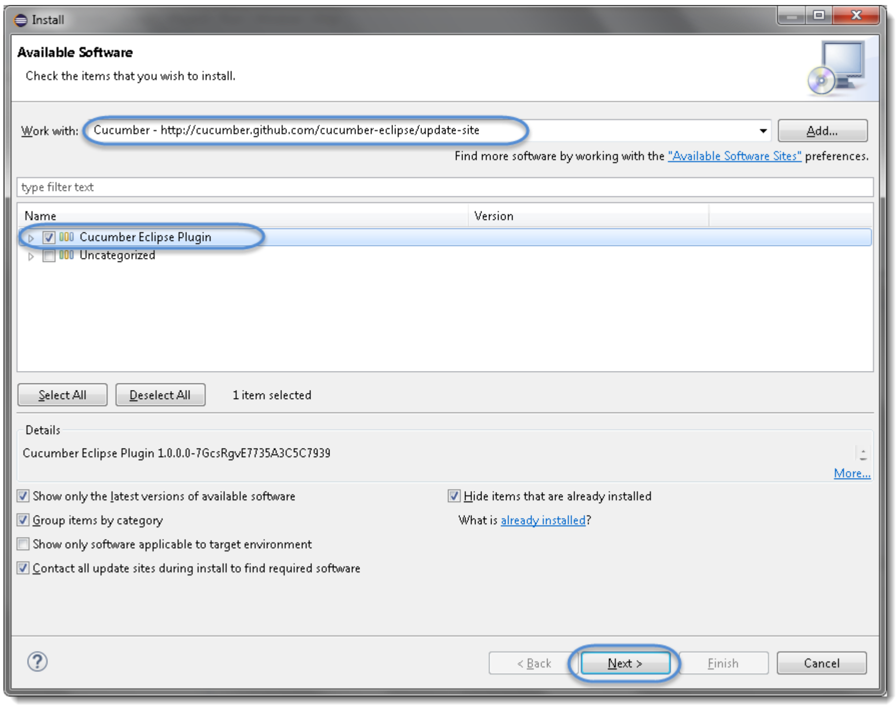
3) Type a name as you wish, let’s take “Cucumber” and type “http://cucumber.github.com/cucumber-eclipse/update-site” as location. Click “OK”.


Add Repository

Name: Cucumber

Location: http://cucumber.github.com/cucumber-eclipse/update-:

4) You come back to the previous window but this time you must see Cucumber Eclipse Plugin option in the available software list. Just Check the box and press the “Next” button.


Install

Available Software

Check the items that you wish to install.

Work with: Cucumber - http://cucumber.github.com/cucumber-eclipse/update-site

Find more software by working with the "Available Software Sites" preferences.

type filter text

Name	Version
[✓] Cucumber Eclipse Plugin	
[ ] Uncategorized	

1 item selected

Details

Cucumber Eclipse Plugin 1.0.0.0-7GcsRgvE7735A3C5C7939

[✓] Show only the latest versions of available software

[✓] Group items by category

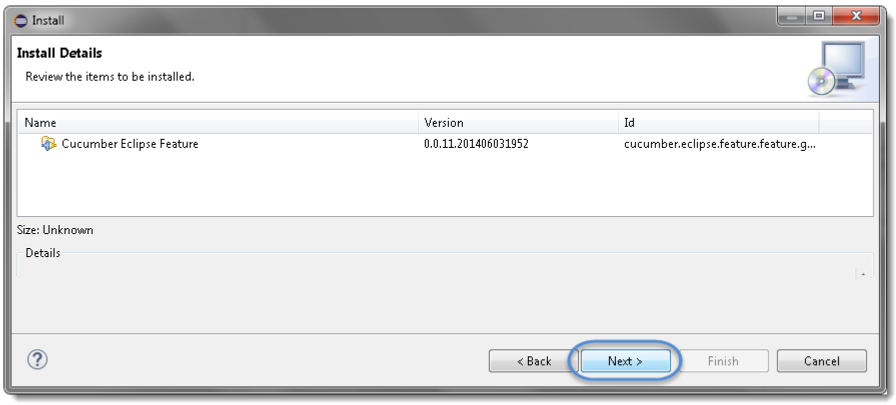
[ ] Show only software applicable to target environment

[✓] Contact all update sites during install to find required software

[✓] Hide items that are already installed

What is already installed?

5) Click on “Next”.


Install

Install Details

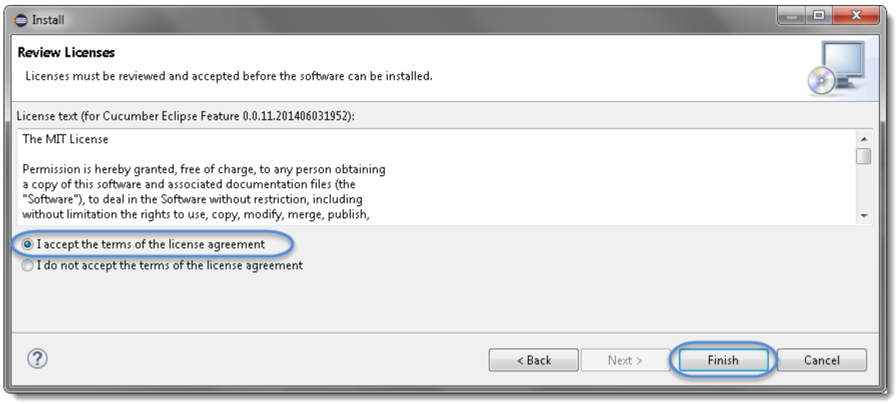
Review the items to be installed.

Name	Version	Id
Cucumber Eclipse Feature	0.0.11.201406031952	cucumber.eclipse.feature.feature.g...

Size: Unknown

Details

6) Click “I accept the terms of the license agreement” then click “Finish”.



Review Licenses

Licenses must be reviewed and accepted before the software can be installed.

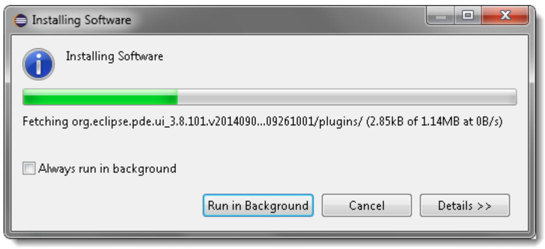
License text (for Cucumber Eclipse Feature 0.0.11.201406031952):

The MIT License

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,

[✓ I accept the terms of the license agreement]
I do not accept the terms of the license agreement

7) Let it install, it will take few seconds to complete.

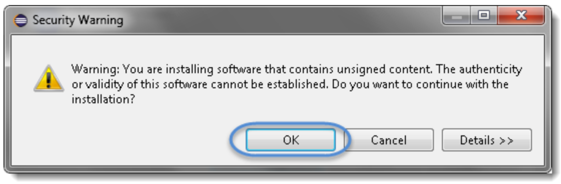


Installing Software

Fetching org.eclipse.pde.ui_3.8.101.v2014090...09261001/plugins/ (2.85kB of 1.14MB at 0B/s)

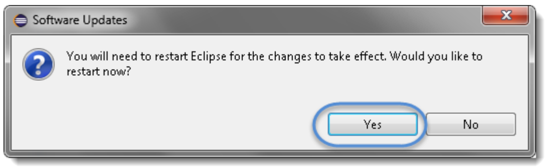
Always run in background

8) You may or may not encounter a Security warning, if in case you do just click “OK”.



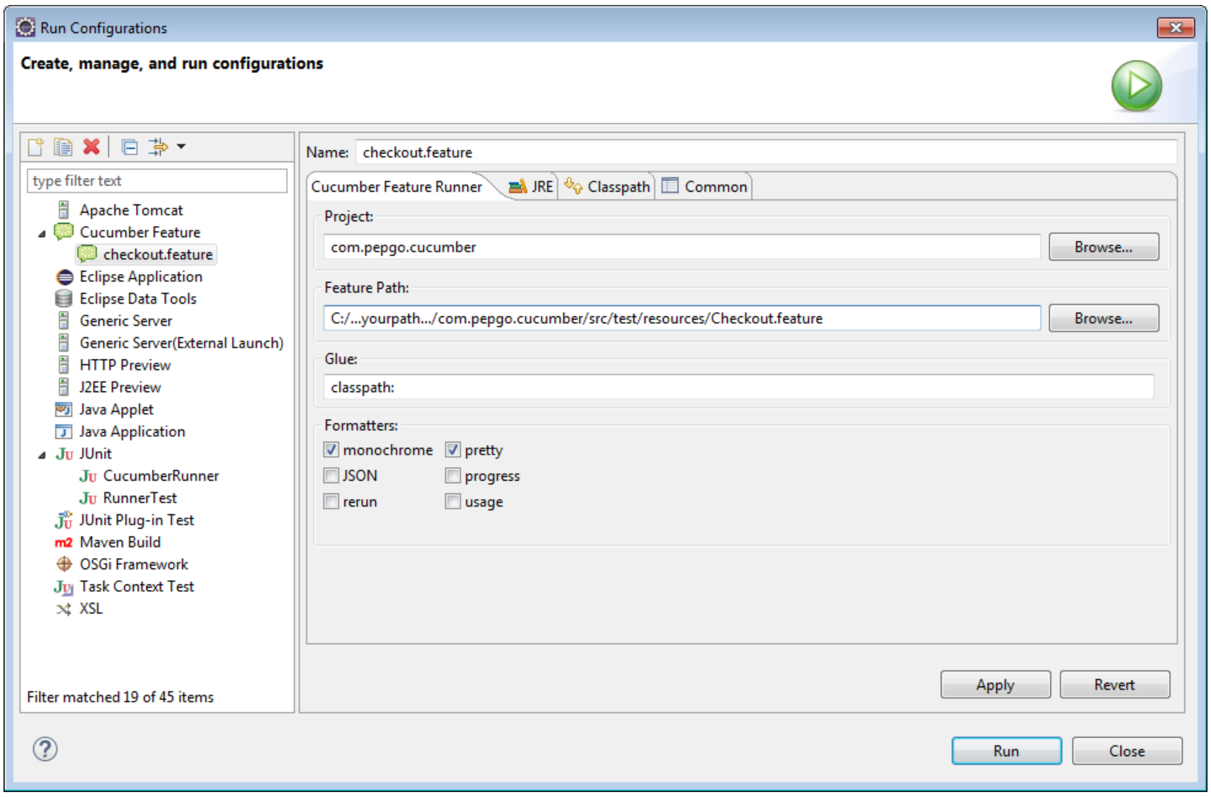
Warning: You are installing software that contains unsigned content. The authenticity or validity of this software cannot be established. Do you want to continue with the installation?

9) You are all done now, just click “Yes”.



You will need to restart Eclipse for the changes to take effect. Would you like to restart now?

After the installation, you will get the following new tab in the Eclipse “Run Configurations” (example picture):



Name: checkout.feature

Project: com.pepgo.cucumber

Feature Path: C:/...yourpath.../com.pepgo.cucumber/src/test/resources/Checkout.feature

Glue: classpath:

Formatters:

✓ monochrome	✓ pretty
JSON	progress
rerun	usage

- “Feature Path” contains the path to your feature file(s).
- “Glue” contains the path to your step definitions (if required by your file structure).
- “Formatters” contains the default formatter(s) that will be applied when no other formatter(s) are set:

Formatter	Description
`monochrome`	By default, the bundled Cucumber plugins will produce coloured output to help emphasize the outcome of each step. This is done by inserting control characters into the output. However, this isn’t supported natively in Microsoft Windows, so you have to install a tool called “`ANSICON`” to see colour output in Microsoft Windows.
`pretty`	Prints the feature as is.
`JSON`	Prints the feature as JSON.
`progress`	Prints one character per scenario. It prints a dot “`.`” for each step that passed, an “`F`” for one that failed, a “`U`” for an undefined step, and a hyphen “`–`” for a skipped step.
`rerun`	Prints failing scenarios (tests) with line numbers. This is great in large runs for rerunning just the failed scenarios, instead of the whole feature.
`usage`	Lists all the step definitions, as well as the steps that are using it. It shows unused step definitions, and it sorts the step definitions by their average execution time. This output is useful for quickly finding slow parts in your code, but also a great way to get an overview of the step definitions.

Recommended Maven settings for Cucumber with JUnit

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <groupId>com.pepgo.cucumber</groupId>
        <artifactId>com.pepgo.cucumber</artifactId>
        <version>1.0-SNAPSHOT</version>
        <packaging>jar</packaging>
        <name>Cucumber-JVM Example</name>

        <properties>
                <junit.version>4.12</junit.version>
                <cucumber.version>1.2.5</cucumber.version>
        </properties>

        <dependencies>
                <dependency>
                        <groupId>junit</groupId>
                        <artifactId>junit</artifactId>
                        <version>${junit.version}</version>
                        <scope>test</scope>
                </dependency>
                <dependency>
                        <groupId>info.cukes</groupId>
                        <artifactId>cucumber-java</artifactId>
                        <version>${cucumber.version}</version>
                        <scope>test</scope>
                </dependency>
                <dependency>
                        <groupId>info.cukes</groupId>
                        <artifactId>cucumber-junit</artifactId>
                        <version>${cucumber.version}</version>
                        <scope>test</scope>
                </dependency>
        </dependencies>

        <build>
                <plugins>
                        <plugin>
                                <groupId>org.apache.maven.plugins</groupId>
                                <artifactId>maven-surefire-plugin</artifactId>
                                <version>2.20</version>
                                <configuration>
                                        <argLine>-Duser.language=en</argLine>
                                        <argLine>-Xmx1024m</argLine>
                                        <argLine>-XX:MaxPermSize=256m</argLine>
                                        <argLine>-Dfile.encoding=UTF-8</argLine>
                                        <useFile>false</useFile>
                                </configuration>
                        </plugin>
                </plugins>
        </build>
</project>
```

If regular unit tests need to be executed separately from the (Cucumber) scenarios, then tests can be run explicitly using “`cucumber.api.cli.Main`”. To do this within Maven, you’d use the “`exec-maven-plugin`”: http://www.mojohaus.org/exec-maven-plugin/

Cucumber Regular Expressions Cheat Sheet

The following list shows just a few common regular expression samples. A good tool for creating regular expressions online is: http://rubular.com/

Pattern	Notes	Match Examples
`.`	One of anything (except a newline)	a B 3
`.*`	The star modifier means “any number of times”. In this example, it means: Any character (except a newline) 0 or more times	a AbCD Words with punctuation! 123-456 An empty string
`.+`	The plus modifier means “at least one of anything (except a newline)”	All of the above except the empty string
`.{2}`	Exactly two of any character	aa Ab !n 23
`.{1,3}`	One to three of any character	aa Ab !n2 9
`^`*pattern*	Anchors match to beginning of string	"`/^foo/`" matches "foo" and "foo bar"" but not "bar foo"
pattern`$`	Anchors match to end of string	"`/foo$/`" matches "foo" and "bar foo" but not "foo bar"
`[0-9]*` *or* `\d*`	Matches a series of digits (or nothing)	123456 9 An empty string
`[0-9]+` *or* `\d+`	Matches one or more digits	All of the above except the empty string
`"[^\"]*"`	Matches something (or nothing) in double quotes; literally, “any character except a double quote zero or more times inside double quotes”	"foo" "foo bar" "12345"
`?`	Makes the previous character or group optional	“/an?/” matches “a” and “an” but not “n”
`|`	Like a logical OR; can be used with parentheses to include an OR in a larger pattern	“`/I’m|I am/`” matches “I’m” and “I am” “`/I (eat|ate)/`” matches “I eat” and “I ate”
`(`*pattern*`)`	A group; typically used in Cucumber to capture a substring for a step definition argument	"`/(\d+)`" users/ matches "3 users" and captures the "3" for use later
`(?:`*pattern*`)`	A non-capturing group	"`/I (?:eat|ate)/`" matches "I eat" and "I ate" but does not capture eat or ate for use later. This is particularly useful for keeping step definitions flexible.