The Thucydides Reference Manual

John Ferguson Smart

The Thucydides Reference Manual

John Ferguson Smart

Table of Contents

List of Figures

Copyright

Chapter 1. Introducing Thucydides

Thucydides (Thoo-SID-a-dees) is a tool designed to make writing automated acceptance and regression tests easier (Refer to Pt. 3). It provides features that make it easier to organize and structure your acceptance tests, associating them with the user stories or features that they test. As the tests are executed, Thucydides generates illustrated documentation describing how the application is used based on the stories described by the tests.

Thucydides provides strong support for automated web tests based on Selenium 2 (http://docs.seleniumhq.org/projects/webdriver/), though it can also be used effectively for non-web tests (?).

Thucydides was a Greek historian (http://en.wikipedia.org/wiki/Thucydides) known for his astute analysis skills who rigorously recorded events that he witnessed and participated in himself. In the same way, the Thucydides framework observes and analyzes your acceptance tests, and records a detailed account of their execution.

Chapter 2. Basic concepts of Acceptance and Regression Testing

To get the most out of Thucydides, it is useful to understand some of the basic principles behind Acceptance Test Driven Development. Thucydides is commonly used for both Automated Acceptance Tests and Regression tests, and the principles discussed here apply, with minor variations, to both.

Acceptance Test Driven Development, or ATDD, is an advanced form of Test Driven Development (TDD) in which automated acceptance criteria — defined in collaboration with users — drive and focus the development process. This helps ensure that everyone understands what features are under development.

One of the important things about ATDD is the idea of "Specification by Example". Specification by Example refers to the use of relatively concrete examples to illustrate how a system should work, as opposed to more formally written specifications.

Let’s look at an example. In many projects, requirements are expressed as simple stories along the following lines:

```
In order to earn money to buy a new car
As a car owner
I want to sell my old car online
```

If we were implementing an online car sales web site that helps people achieve this goal, we would typically define a set of acceptance criteria to flesh out this story. For example, we might have the following criteria in our list of acceptance criteria:

- Car owner can place a standard car ad online
- Car owner can place a premium car ad online
- Car ad should display the brand, model and year of the car

and so on.

A tester planning the tests for these acceptance criteria might draw up a test plan outlining of the way she expects to test these criteria. For the first criteria, she might start off with a high-level plan like the following:

- Go to car ads section and choose to post a standard car ad
- Enter car details
- Choose publication options
- Preview ad
- Enter payment details
- See ad confirmation

Each of these steps might need to be broken down into smaller steps.

- *Enter car details*
  - Enter car make, model and year
  - Select options
  - Add photos
  - Enter description

These steps are often fleshed out with more concrete details:

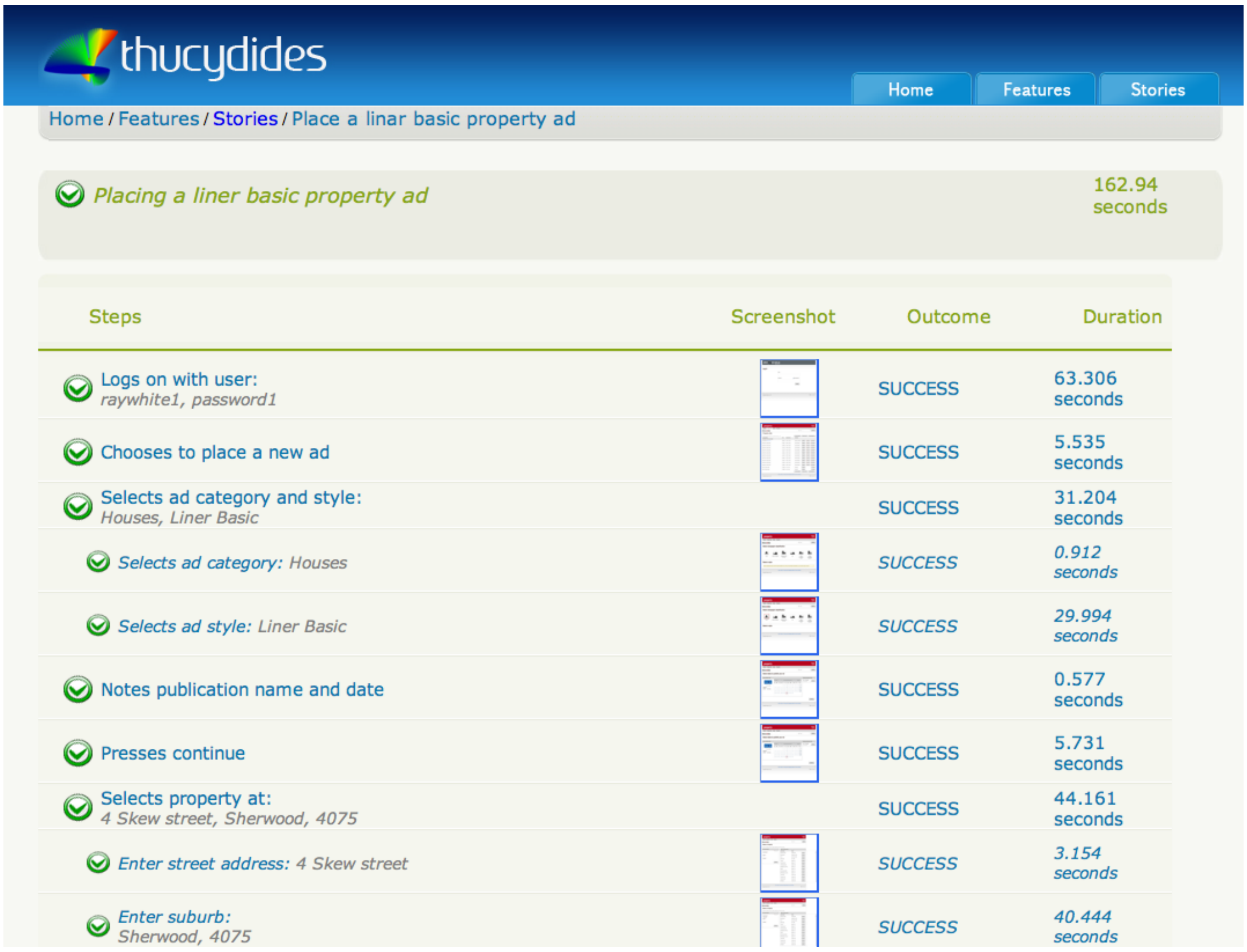
- *Enter car details*
  - Create an ad for a 2006 Mitsubishi Pajero
  - Add Air Conditioning and CD Player
  - Add three photos
  - Enter a description

For our purposes, Regression Tests can be defined as end-to-end tests that ensure that an application behaves as expected, and that it continues to behave as expected in future releases. Whereas ATDD Acceptance Tests are defined very early on in the piece, before development starts, Regression Tests involve an existing system. Other than that, the steps involved in defining and automating the tests are very similar.

Now different project stakeholders will be interested in different levels of detail. Some, such as project managers and management in general, will be interested only in which application features work, and which need to be done. Others, such as business analysts and QA, will be interested in the details of how each acceptance scenario is implemented, possible down to the screen level.

Thucydides helps you structure your automated acceptance tests into steps and sub-steps like the ones illustrated above. This tends to make the tests clearer, more flexible and easier to maintain. In addition, when the tests are executed, Thucydides produces illustrated, narrative-style reports like the one in Figure 2.1, “A test report generated by Thucydides”.

Figure 2.1. A test report generated by Thucydides



Home / Features / Stories / Place a linar basic property ad

Placing a liner basic property ad — 162.94 seconds

Steps	Screenshot	Outcome	Duration
✓ Logs on with user: *raywhite1, password1*		SUCCESS	63.306 seconds
✓ Chooses to place a new ad		SUCCESS	5.535 seconds
✓ Selects ad category and style: *Houses, Liner Basic*		SUCCESS	31.204 seconds
✓ *Selects ad category: Houses*		*SUCCESS*	*0.912 seconds*
✓ *Selects ad style: Liner Basic*		*SUCCESS*	*29.994 seconds*
✓ Notes publication name and date		SUCCESS	0.577 seconds
✓ Presses continue		SUCCESS	5.731 seconds
✓ Selects property at: *4 Skew street, Sherwood, 4075*		SUCCESS	44.161 seconds
✓ *Enter street address: 4 Skew street*		*SUCCESS*	*3.154 seconds*
✓ *Enter suburb: Sherwood, 4075*		*SUCCESS*	*40.444 seconds*

When it comes to implementing the tests themselves, Thucydides also provides many features that make it easier, faster and cleaner to write clear, maintainable tests. This is particularly true for automated web tests using Selenium 2, but Thucydides also caters for non-web tests as well. Thucydides currently works well with JUnit and easyb - integration with other BDD frameworks is in progress.

Chapter 3. Getting started with Thucydides

3.1. Creating a new Thucydides project

The easiest way to start a new Thucydides project is to use the Maven archetype. Three archetypes are currently available: one for using Thucydides with JUnit, another if you also want to write your acceptance tests (or a part of them) using easyb [http://www.easyb.org/], and finally one more to help you write acceptance tests in jBehave. In this section, we will create a new Thucydides project using the Thucydides archetype, and go through the essential features of this project.

From the command line, you can run **mvn archetype:generate** and then select the **net.thucydides.thucydides-easyb-archetype** archetype from the proposed list of archetypes. Or you can use your favorite IDE to generate a new Maven project using an archetype.

```
$ mvn archetype:generate
...
Define value for property 'groupId': : com.mycompany
Define value for property 'artifactId': : webtests
Define value for property 'version':  1.0-SNAPSHOT: :
Define value for property 'package':  com.mycompany: :
Confirm properties configuration:
groupId: com.mycompany
artifactId: webtests
version: 1.0-SNAPSHOT
package: com.mycompany
 Y: :
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2:33.290s
[INFO] Finished at: Fri Oct 28 07:20:41 NZDT 2011
[INFO] Final Memory: 7M/81M
[INFO] ------------------------------------------------------------------------
```

This will create a simple Thucydides project, complete with a Page Object, a Step library and two test cases, one using JUnit, and one using easyb. The actual tests run against the online dictionary at Wiktionary.org. Before going any further, take the project for a spin. First, however, you will need to add the `net.thucydides.maven.plugins` to your plugin groups in your `settings.xml` file:

```
<settings>
   <pluginGroups>
       <pluginGroup>net.thucydides.maven.plugins</pluginGroup>
       ...
      </pluginGroups>
  ...
</settings>
```

This will let you invoke the Maven **thucydides** plugin from the command line in the short-hand form shown here. Now go into the generated project directory, run the tests and generate the reports:

```
$ mvn test thucydides:aggregate
```

This should run some web tests and generate a report in `target/site/thucydides` directory (open the `index.html` file).

If you drill down into the individual test reports, you will see an illustrated narrative for each test similar to the one shown in Figure 2.1, “A test report generated by Thucydides”

Now for the details. the project directory structure is shown here:

```
+ src
   + main
      + java
         + com.mycompany.pages
            - HomePage.java

   + test
      + java
         + com.mycompany.pages
            + requirements
               - Application.java
            + steps
               - EndUserSteps.java
            - LookupADefinitionStoryTest.java

      + stories
         + com.mycompany
            - LookupADefinition.story
```

This project is designed to provide a starting point for your Thucydides acceptance tests, and to illustrate some of the basic features. The tests come in two flavors: *easyb* and *JUnit*. easyb is a Groovy-based BDD (Behaviour Driven Development) library which works well for this kind of test. The sample easyb story can be found in the `LookupADefinition.story` file, and looks something like this:

```
using "thucydides"

thucydides.uses_default_base_url "http://en.wiktionary.org/wiki/Wiktionary:Main_Page"
thucydides.uses_steps_from EndUserSteps
thucydides.tests_story SearchByKeyword

scenario "Looking up the definition of 'apple'", {
    given "the user is on the Wikionary home page", {
        end_user.is_the_home_page()
    }
    when "the end user looks up the definition of the word 'apple'", {
        end_user.looks_for "apple"
    }
    then "they should see the definition of 'apple", {
       end_user.should_see_definition_containing_words "A common, round fruit"
    }
}
```

A cursory glance at this story will show that it relates a user looking up the definition of the word *apple*. However only the “what” is expressed at this level – the details are hidden inside the test steps and, further down, inside page objects.

If you prefer pure Java tests, the JUnit equivalent can be found in the `LookupADefinitionStoryTest.java` file:

```
@Story(Application.Search.SearchByKeyword.class)
@RunWith(ThucydidesRunner.class)
public class LookupADefinitionStoryTest {

    @Managed(uniqueSession = true)
    public WebDriver webdriver;

    @ManagedPages(defaultUrl = "http://en.wiktionary.org/wiki/Wiktionary:Main_Page")
    public Pages pages;

    @Steps
    public EndUserSteps endUser;

    @Issue("#WIKI-1")
    @Test
    public void looking_up_the_definition_of_apple_should_display_the_corresponding_arti
        endUser.is_the_home_page();
                endUser.looks_for("apple");
        endUser.should_see_definition_containing_words("A common, round fruit");

    }
}
```

As you can see, this is a little more technical but still very high level.

The step libraries contain the implementation of each of the steps used in the high-level tests. For complex tests, these steps can in turn call other steps. The step library used in this example can be found in `EndUserSteps.java`:

```
public class EndUserSteps extends ScenarioSteps {

        public EndUserSteps(Pages pages) {
                super(pages);
        }

    @Step
    public void searches_by_keyword(String keyword) {
        enters(keyword);
        performs_search();
    }

        @Step
        public void enters(String keyword) {
        onHomePage().enter_keywords(keyword);
        }

    @Step
    public void performs_search() {
        onHomePage().starts_search();
    }

    private HomePage onHomePage() {
        return getPages().currentPageAt(HomePage.class);
    }

    @Step
        public void should_see_article_with_title(String title) {
        assertThat(onHomePage().getTitle(), is(title));
        }
    @Step
    public void is_on_the_wikipedia_home_page() {
        onHomePage().open();
    }
}
```

Page Objects are a way of encapsulating the implementation details about a particular page. Selenium 2 has particularly good support for page objects, and Thucydides leverages this. The sample page object can be found in the HomePage.java class:

```
@DefaultUrl("http://en.wiktionary.org/wiki/Wiktionary:Main_Page")
public class SearchPage extends PageObject {

    @FindBy(name="search")
        private WebElement searchInput;

        @FindBy(name="go")
        private WebElement searchButton;

        public SearchPage(WebDriver driver) {
                super(driver);
        }

        public void enter_keywords(String keyword) {
                searchInput.sendKeys(keyword);
        }

    public void starts_search() {
        searchButton.click();
    }

    public List<String> getDefinitions() {
        WebElement definitionList = getDriver().findElement(By.tagName("ol"));
        List<WebElement> results = definitionList.findElements(By.tagName("li"));
        return convert(results, new ExtractDefinition());
    }

    class ExtractDefinition implements Converter<WebElement, String> {
        public String convert(WebElement from) {
            return from.getText();
        }
    }
}
```

The final piece in the puzzle is the `Application.java` class, which is a way of representing the structure of your requirements in Java form, so that your easyb and JUnit tests can be mapped back to the requirements they are testing:

```
        public class Application {
            @Feature
            public class Search {
                public class SearchByKeyword {}
                public class SearchByAnimalRelatedKeyword {}
                public class SearchByFoodRelatedKeyword {}
                public class SearchByMultipleKeywords {}
                public class SearchForQuote{}
            }
            @Feature
            public class Backend {
                public class ProcessSales {}
                public class ProcessSubscriptions {}
            }

            @Feature
            public class Contribute {
                public class AddNewArticle {}
                public class EditExistingArticle {}
            }
        }
```

This is what enables Thucydides to generate the aggregate reports about features and stories.

In the following sections, we will look at different aspects of writing automated tests with Thucydides in more detail.

3.2. Setting custom web driver capabilities

You can set custom web driver capabilities by passing a semi-colon separated list of capabilities in the property `thucydides.driver.capabilities`. For example,

```
"build:build-1234; max-duration:300; single-window:true; tags:[tag1,tag2,tag3]"
```

Chapter 4. Writing Acceptance Tests with Thucydides

In this section, we look at the things you need to know to write your acceptance or regression tests using Thucydides in more detail. We will also outline a general approach to writing your web-based acceptance tests that has worked well for us in the past.

1. Define and organize the requirements or user stories you need to test
2. Write high level pending tests for the acceptance criteria
3. Choose a test to implement, and break it into a small (typically between 3 and 7) high-level steps
4. Implement these steps, either by breaking them down into other steps, or by accessing Page Objects.
5. Implement any new Page Object methods that you have discovered.

Note

These steps should not been seen as a linear or waterfall-style approach. Indeed, the process is usually quite incremental, with requirements being added to the `Application` class as they are required, and pending tests being used to defined tests before they are fleshed out.

4.1. Organizing your requirements

To get the most out of automated tests in Thucydides, you need to tell Thucydides which features of your application you are testing in each test. While this step is optional, it is highly recommended.

The current version of Thucydides uses a simple, three-level organization to structure acceptance tests into more manageable chunks. At the highest level, an application is broken into *features*, which is a high-level functionality or group of related functions. A feature contains a number of *stories* (corresponding to user stories, use cases, and so on). Each story is validated by a number of examples, or acceptance criteria, which are automated in the form of web tests (sometimes called scenarios). Each test, in turn, is implemented using a number of steps.

Of course this structure and these terms are merely a convenience to allow a higher-level vision of your acceptance tests. However, this sort of three-level abstraction seems to be fairly common.

In the current version of Thucydides, you define this structure within the test code, as (very light-weight) Java classes [1]. This makes it easier to refactor and rename user stories and features within the tests, and gives a central point of reference in the test suite illustrating what features are being tested. A simple example is shown here. The Application class is simply a convenient way of placing the features and user stories in the one file. Features are marked with the @Feature annotation. User stories are declared as inner classes nested inside a @Feature class.

```
public class Application {
```

[1]Future versions of Thucydides will support other ways of defining your user requirements.

```
    @Feature
    public class ManageCompanies {
        public class AddNewCompany {}
        public class DeleteCompany {}
        public class ListCompanies {}
    }

    @Feature
    public class ManageCategories {
        public class AddNewCategory {}
        public class ListCategories {}
        public class DeleteCategory {}
    }

    @Feature
    public class ManageTags {
        public class DisplayTagCloud {}
    }

    @Feature
    public class ManageJobs {}

    @Feature
    public class BrowseJobs {
        public class UserLookForJobs {}
        public class UserBrowsesJobTabs {}
    }
}
```

Chapter 5. Defining high-level tests

There are two approaches to automated acceptance criteria or regression tests with Thucydides. Both involve implementing the tests as a sequence of very high-level steps, and then fleshing out those steps by drilling down into the details, until you get to the Page Objects. The difference involves the language used to implement the high-level tests. Tools like easyb are more focused on communication with non-developers, and allow high level tests to be expressed more easily in business terms. On the other hand, developers often find it more comfortable to work directly with JUnit, so if communication with non-technical stakeholders is not a high priority, this might be a preferred option.

In the current version of Thucydides, you can write your tests using easyb (for a more BDD-style approach) or in JUnit using Java or another JVM language (Groovy is a popular choice). Other BDD tools will be supported in future versions. We will discuss both here, but you can use whatever you and your team are more comfortable with.

5.1. Defining high-level tests in easyb

Easyb (http://easyb.org) is a Groovy-based BDD tool. It makes it easy to write light-weight stories and scenarios using the classic BDD-style "given-when-then" structure, and then to implement them in Groovy. The Thucydides easyb plugin is designed to make it easy to write Thucydides tests using easyb.

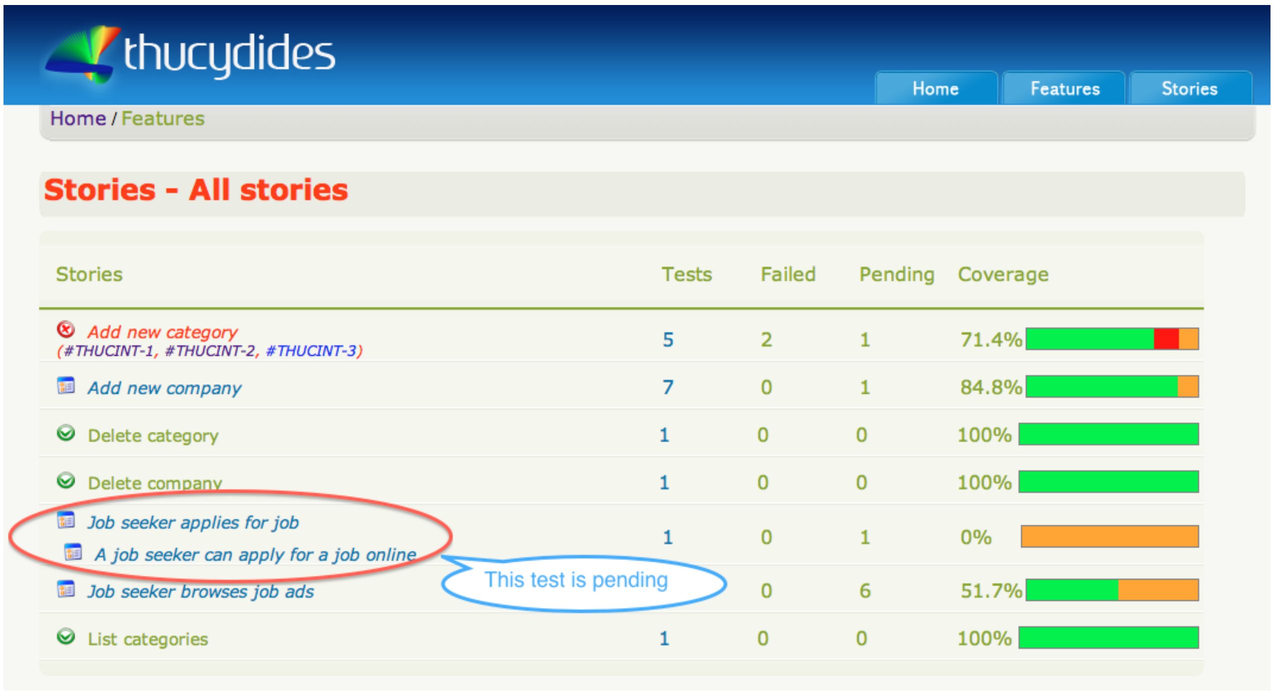
5.1.1. Writing a pending easyb story

In easyb, you write tests (referred to as "scenarios") that, when using Thucydides, correspond to automated acceptance criteria. Tests are grouped into "stories" - each story has it’s own file.

Scenarios are first written as "pending". These are just high-level outlines, describing a set of acceptance criteria for a particular story in a "given-when-then" structure.

When the tests are executed, pending scenarios are skipped. However they appear in the reports, so that you know what features still need to be implemented. An example of how pending scenarios appear in a Thucydides report can be seen in Figure 5.1, “Pending tests are shown with the *calendar* icon”.

Figure 5.1. Pending tests are shown with the *calendar* icon



Stories - All stories

Stories	Tests	Failed	Pending	Coverage
Add new category (#THUCINT-1, #THUCINT-2, #THUCINT-3)	5	2	1	71.4%
Add new company	7	0	1	84.8%
Delete category	1	0	0	100%
Delete company	1	0	0	100%
Job seeker applies for job / A job seeker can apply for a job online	1	0	1	0%
Job seeker browses job ads		0	6	51.7%
List categories	1	0	0	100%

Here is an example of a pending easyb story using Thucydides:

```
using "thucydides"

import net.thucydides.demos.jobboard.requirements.Application.ManageCategories.AddNewCat

thucydides.tests_story AddNewCategory

scenario "The administrator adds a new category to the system",
{
        given "a new category needs to be added to the system"
        when "the administrator adds a new category"
        then "the system should confirm that the category has been created"
        and "the new category should be visible to job seekers"
}

scenario "The administrator adds a category with an existing code to the system",
{
        given "the administrator is on the categories list page"
        when "the user adds a new category with an existing code"
        then "an error message should be displayed"
}
```

Let’s examine this story piece-by-piece. First, you need to declare that you are using Thucydides. You do this by using the easyb using keyword:

```
using "thucydides"
```

This will, among other things, inject the `thucydides` object into your story context so that you can configure Thucydides to run your story correctly.

Next, you need to tell Thucydides what story you are testing. You do this by referencing one of the story classes you defined earlier. That’s what we are doing here:

```
import net.thucydides.demos.jobboard.requirements.Application.ManageCategories.AddNewCat

thucydides.tests_story AddNewCategory
```

The rest of the easyb story is just a set of regular easyb pending scenarios. For the moment, there is no implementation, so they will appear as "pending" in the reports:

```
scenario "The administrator adds a new category to the system",
{
        given "a new category needs to be added to the system"
        when "the administrator adds a new category"
        then "the system should confirm that the category has been created"
        and "the new category should be visible to job seekers"
}

scenario "The administrator adds a category with an existing code to the system",
{
        given "the administrator is on the categories list page"
        when "the user adds a new category with an existing code"
        then "an error message should be displayed"
}
```

You typically declare many pending stories, preferably in collaboration with the product owner or BAs, at the start of an iteration. This lets you get a good picture of what stories need to be implemented in a given iteration, and also an idea of the relative complexity of each story.

5.1.2. Implementing the easyb stories

The next step is to implement your stories. Let’s look at an implemented version of the first of these scenarios:

```
using "thucydides"

import net.thucydides.demos.jobboard.requirements.Application.ManageCategories.AddNewCat
import net.thucydides.demos.jobboard.steps.AdministratorSteps
import net.thucydides.demos.jobboard.steps.JobSeekerSteps

thucydides.uses_default_base_url "http://localhost:9000"
thucydides.uses_steps_from AdministratorSteps
thucydides.uses_steps_from JobSeekerSteps
thucydides.tests_story AddNewCategory

def cleanup_database() {
    administrator.deletes_category("Scala Developers");
}

scenario "The administrator adds a new category to the system",
{
    given "a new category needs to be added to the system",
    {
      administrator.logs_in_to_admin_page_if_first_time()
      administrator.opens_categories_list()
    }
    when "the administrator adds a new category",
    {
       administrator.selects_add_category()
       administrator.adds_new_category("Scala Developers","SCALA")
    }
    then "the system should confirm that the category has been created",
    {
        administrator.should_see_confirmation_message "The Category has been created"
    }
    and "the new category should be visible to job seekers",
    {
        job_seeker.opens_jobs_page()
        job_seeker.should_see_job_category "Scala Developers"
    }
}
```

Again, let’s break this down. In the first section, we import the classes we need to use:

```
using "thucydides"

import net.thucydides.demos.jobboard.requirements.Application.ManageCategories.AddNewCat
import net.thucydides.demos.jobboard.steps.AdministratorSteps
import net.thucydides.demos.jobboard.steps.JobSeekerSteps
```

Next, we declare the default base URL to be used for the tests. Like the equivalent annotation in the JUnit tests, this is used for tests executed from within the IDE, or if no base URL is defined on the command line using the `webdriver.base.url` parameter.

```
thucydides.uses_default_base_url "http://localhost:9000"
```

We also need to declare the test step libraries we will be using. We do this using thucydides.uses_steps_from. This will inject an instance variable into the easyb context for each declared step library. If the step library class name ends in *Steps* (e.g. JobSeekerSteps), the name of the variable will be the class name less the *Steps* suffix, converted to lower case and underscores (e.g. "job_seeker"). We will learn more about implementing test step libraries further on.

```
thucydides.uses_steps_from AdministratorSteps
thucydides.uses_steps_from JobSeekerSteps
thucydides.tests_story AddNewCategory
```

Finally we implement the scenario. Notice, that since this is Groovy, we can declare fixture methods to help set up and tear down the test environment as required:

```
def cleanup_database() {
    administrator.deletes_category("Scala Developers");
}
```

The implementation usually just invokes step methods, as illustrated here:

```
scenario "The administrator adds a new category to the system",
{
    given "a new category needs to be added to the system",
    {
      administrator.logs_in_to_admin_page_if_first_time()
      administrator.opens_categories_list()
    }
    when "the administrator adds a new category",
    {
       administrator.selects_add_category()
       administrator.adds_new_category("Scala Developers","SCALA")
    }
    then "the system should confirm that the category has been created",
    {
        administrator.should_see_confirmation_message "The Category has been created"
    }
    and "the new category should be visible to job seekers",
    {
        job_seeker.opens_jobs_page()
        job_seeker.should_see_job_category "Scala Developers"
        cleanup_database()
    }
}
```

5.2. Defining high-level tests in JUnit

Thucydides integrates smoothly with ordinary JUnit 4 tests, using the ThucydidesRunner test runner and a few other specialized annotations. This is one of the easiest ways to start out with Thucydides, and is very well suited for regression testing, where communication and clarification with the various stakeholders is less of a requirement.

Here is an example of a Thucydides JUnit web test:

```
@RunWith(ThucydidesRunner.class)
@Story(UserLookForJobs.class)
public class LookForJobsStory {

    @Managed
    public WebDriver webdriver;

    @ManagedPages(defaultUrl = "http://localhost:9000")
    public Pages pages;

    @Steps
    public JobSeekerSteps job_seeker;

    @Test
    public void user_looks_for_jobs_by_key_word() {
        job_seeker.opens_jobs_page();
        job_seeker.searches_for_jobs_using("Java");
        job_seeker.should_see_message("No jobs found.");
    }

    @Test
    public void when_no_matching_job_found_should_display_error_message() {
        job_seeker.opens_jobs_page();
        job_seeker.searches_for_jobs_using("unknownJobCriteria");
        job_seeker.should_see_message("No jobs found.");
    }

    @Pending @Test
    public void tags_should_be_displayed_to_help_the_user_find_jobs() {}

    @Pending @Test
    public void the_user_can_list_all_of_the_jobs_for_a_given_tag() {}

    @Pending @Test
    public void the_user_can_see_the_total_number_of_jobs_on_offer() {}
}
```

Let's examine this section-by-section. The class starts with the @RunWith annotation, to indicate that this is a Thucydides test. We also use the @Story annotation to indicate which user story (defined as nested classes of the the @Feature classes above) is being tested. This is used to generate the aggregate reports.

```
@RunWith(ThucydidesRunner.class)
@Story(UserLookForJobs.class)
public class LookForJobsStory {
    ...
```

Next, come two essential annotations for any web tests. First of all, your test case needs a public `Webdriver` field, annotated with the `@Managed` annotation. This enables Thucydides to take care of opening and closing a WebDriver driver for you, and lets Thucydides use this driver in the pages and test steps when the tests are executed:

```
    @Managed
    public WebDriver webdriver;
```

The second essential field is an instance of the `Pages` class, annotated with the `@ManagedPages` annotation. This is essentially a page factory, that Thucydides uses to provide you with instantiated page objects. The `defaultUrl` attribute lets you define a URL to use when your pages open, if no other base URL has been defined. This is useful for IDE testing:

```
    @ManagedPages(defaultUrl = "http://localhost:9000")
    public Pages pages;
```

Note that these two annotations are only required for web tests. If your Thucydides test does not use web tests, you can safely leave them out.

For high-level acceptance or regression tests, it is a good habit to define the high-level test as a sequence of high-level steps. It will make your tests more readable and easier to maintain if you delegate the implementation details of your test (the "how") to reusable "step" methods. We will discuss how to define these step methods later. However, the minimum you need to do is to define the class where the steps will be defined, using the `@Steps` annotation. This annotation tells Thucydides to listen to method calls on this object, and (for web tests) to inject the WebDriver instance and the page factory into the Steps class so that they can be used in the step methods.

```
    @Steps
    public JobSeekerSteps job_seeker;
```

5.2.1. Pending tests

Tests that contain no steps are considered to be pending. Alternatively, you can force a step to be skipped (and marked as pending) by using the `@Pending` annotation or the `@Ignore` annotation. Note that the semantics are slightly different: `@Ignore` indicates that you are temporarily suspending execution of a test, whereas `@Pending` means that the test has been specified but not yet implemented. So both these tests will be pending:

```
@Test
public void administrator_adds_an_existing_company_to_the_system() {}
```

```
@Pending @Test
public void administrator_adds_a_company_with_an_existing_code_to_the_system() {
    steps.login_to_admin_page();
    steps.open_companies_list();
    steps.select_add_company();
    // More to come
}
```

A test is also considered pending if any of the steps used in that test are pending. For a step to be pending, it needs to be annotated with the `@Pending` annotation.

Junit assumptions

You can use junit assumptions [http://junit.sourceforge.net/javadoc/org/junit/Assume.html] in your tests or step methods to . Steps where the conditions under assumptions fail are marked as PENDING instead of ERROR. Subsequent steps are also marked as PENDING.

```
@Test
public void administrator_adds_an_existing_company_to_the_system() {}

@Test
public void administrator_adds_a_company_with_an_existing_code_to_the_system() {
    steps.login_to_admin_page();
    steps.open_companies_list();
    steps.select_add_company();
    // More to come
}
```

```
...
@Step
public void open_companies_list() {
  Assume.assumeThat(user.role, is("admin"));
  CompaniesListPage page = pages().get(CompanyListPage.class);
  String companieslist = page.getCompaniesList();

}
...
```

In the above example, if the assumption in step `open_companies_list` fails, it and all susbequent steps will be marked PENDING.

5.2.2. Running tests in a single browser session

Normally, Thucydides opens a new browser session for each test. This helps ensure that each test is isolated and independent. However, sometimes it is useful to be able to run tests in a single browser session, in particular for performance reasons on read-only screens. You can do this by using the *uniqueSession* attribute in the @Managed annotation, as shown below. In this case, the browser will open at the start of the test case, and not close until all of the tests have been executed.

```
@RunWith(ThucydidesRunner.class)
public class OpenStaticDemoPageSample {

    @Managed(uniqueSession=true)
    public WebDriver webdriver;
    @ManagedPages(defaultUrl = "classpath:static-site/index.html")
    public Pages pages;

    @Steps
    public DemoSiteSteps steps;

    @Test
    @Title("The user opens the index page")
    public void the_user_opens_the_page() {
        steps.should_display("A visible title");
    }

    @Test
    @Title("The user selects a value")
    public void the_user_selects_a_value() {
        steps.enter_values("Label 2", true);
        steps.should_have_selected_value("2");
    }

    @Test
    @Title("The user enters different values.")
    public void the_user_opens_another_page() {
        steps.enter_values("Label 3", true);
        steps.do_something();
        steps.should_have_selected_value("3");
    }
}
```

If you do not need WebDriver support in your test, you can skip the `@Managed` and `@Pages` annotations, e.g.

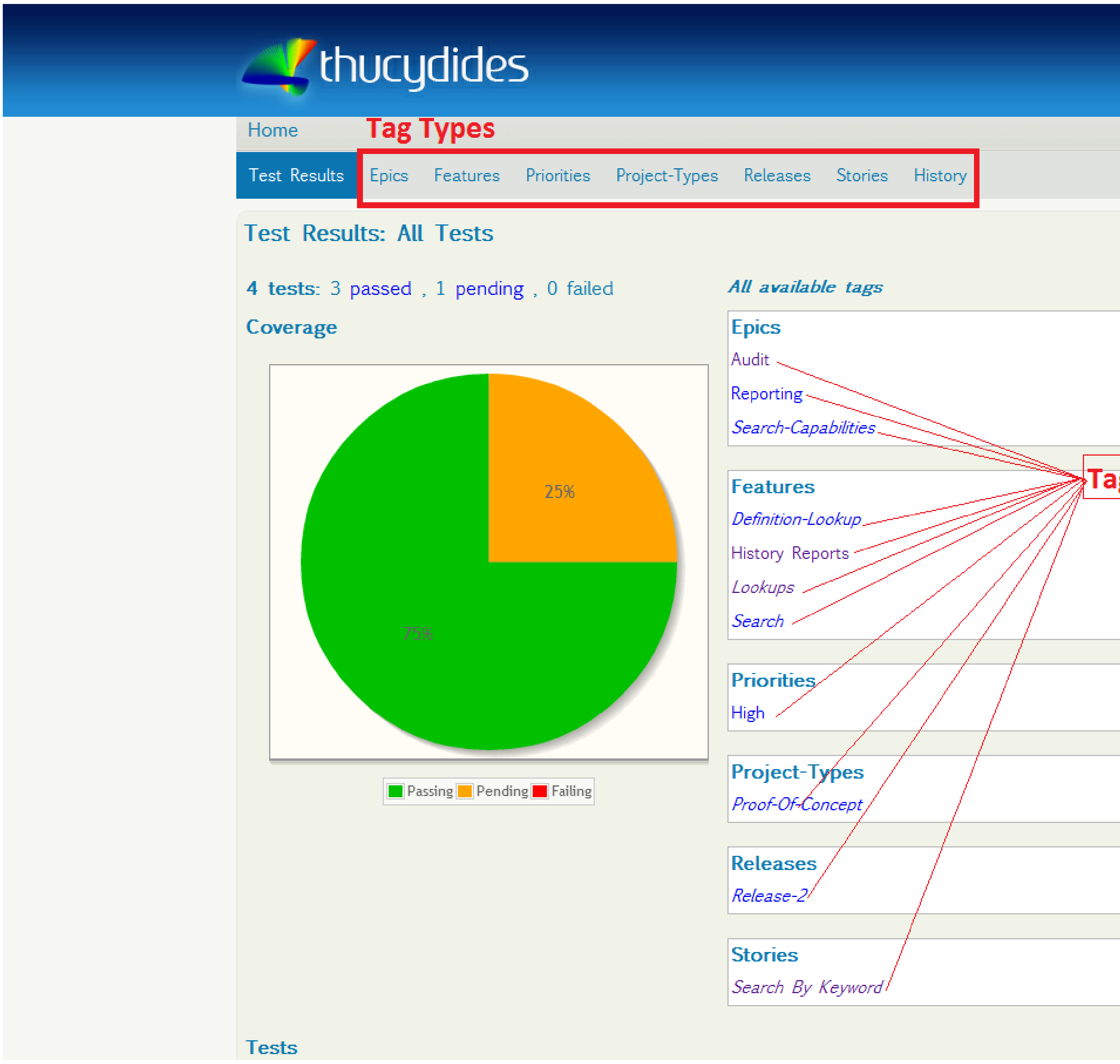
```
@RunWith(ThucydidesRunner.class)
@Story(Application.Backend.ProcessSales.class)
public class WorkWithBackendTest {

    @Steps
    public BackendSteps backend;

    @Test
    public void when_processing_a_sale_transation() {
        backend.accepts_a_sale_transaction();
        backend.should_the_update_mainframe();
    }
}
```

5.3. Adding tags to test cases

You can add arbitrary tags to your tests both in junit and easyb. Tags provide context to tests. A tag has two parts - a `type` and a `name`. Thucydides reports categorize tests based on the specified tag types.

Tag types are arbitrary and you can add as many types as you wish. By default, a `story` tag type is automatically added to each test. An example of tags on Thucydides reports is given in ???

Figure 5.2. Tag types appear on top. Each tag type displays the tag names.



5.3.1. Adding tags to junit tests

Tags are added to junit tests using `@WithTag` annotation. The following will add a tag of type `epic` with name "Audit".

```
@WithTag(type="epic", name="Audit")
```

If no type is defined, the default tag type is assumed to be `feature`. In other words, the following two tags are equivalent.

```
@WithTag(type="feature", name="Definition-lookup")
```

```
@WithTag(name="Definition-lookup")
```

`@WithTag` has an alternative, more concise syntax using a colon (:) to separate the tag type and name. For example,

```
@WithTag("epic:Audit")
```

or,

```
@WithTag("feature:Definition-lookup")
```

Multiple tags can be added using `@WithTags` annotation or it’s shorter cousin - `@WithTagValuesOf`. For example,

```
@WithTags (
        {
                @WithTag(name="lookups", type="feature"),
                @WithTag(name="release-2", type="release")

        }
)
```

Using `@WithTagValuesOf`, the above can be written more succinctly as:

```
@WithTagValuesOf({"lookups", "release:release-2"})
```

5.3.2. Adding tags to easyb tests

Tags can be easily added to easyb stories in the form of `thucydides.tests.<type>` to the stories. For example,

```
thucydides.tests.feature "history reports"
thucydides.tests.epic "reporting"
thucydides.tests.epic "audit"
thucydides.tests.priority "high"
```

5.3.3. Filter tests by tags in jUnit

You can filter tests by tag while running Thucydides. This can be achieved by providing a single tag or a comma separated list of tags from command line. If provided, only classes and/or methods with tags in this list will be executed.

Example:

```
mvn verify -Dtags="iteration:I1"
```

or

```
mvn verify -Dtags="color:red,flavor:strawberry"
```

5.4. Running Thucydides in different browsers

Thucydides supports all browser-based WebDriver drivers, i.e. Firefox, Internet Explorer and Chrome, as well as HTMLUnit. By default, it will use Firefox. However, you can override this option using the `webdriver.driver` system property. To set this from the command line, you could do the following:

```
$ mvn test -Dwebdriver.driver=iexplorer
```

If you are not using Firefox by default, it is also useful to define this variable as a property in your Maven pom.xml file, e.g.

```
<properties>
    <webdriver.driver>iexplorer</webdriver.driver>
</properties>
```

For this to work with JUnit, however, you need to pass the webdriver.driver property to JUnit. JUnit runs in a separate JVM, and will not see the system properties defined in the Maven build. To get around this, you need to pass them into JUnit explicitly using the systemPropertyVariables configuration option, e.g.

```
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>2.7.1</version>
            <configuration>
                <systemPropertyVariables>
                    <webdriver.driver>${webdriver.driver}</webdriver.driver>
                </systemPropertyVariables>
            </configuration>
        </plugin>
```

5.4.1. Chrome switches

Thucydides supports `chrome.switches` system property to define options for the Chrome driver. This lets you set useful chrome options such as `"--homepage=about:blank" or "--no-first-run"`. You can provide any number of options, separated by commas, e.g.:

```
$mvn verify -Dchrome.switches="homepage=about:blank,--no-first-run"
```

5.5. Forcing the use of a particular driver in a test case or test

The `@Managed` annotation also lets you specify what driver you want to use for a particular test case, via the `driver` attribute. Current supported values are “firefox”, “iexplorer”, “chrome” and “htmlunit”. The `driver` attribute lets you override the system-level default driver for specific requirements. For example, the following test case will run in Chrome, regardless of the `webdriver.driver` system property value used:

```
@RunWith(ThucydidesRunner.class)
@Story(Application.Search.SearchByKeyword.class)
public class SearchByFoodKeywordStoryTest {

    @Managed(uniqueSession = true, driver="chrome")
    public WebDriver webdriver;

    @ManagedPages(defaultUrl = "http://www.google.co.nz")
    public Pages pages;

    @Steps
    public EndUserSteps endUser;

    @Test
    public void searching_by_keyword_pears_should_display_the_corresponding_article() {
        endUser.is_the_google_home_page();
        endUser.enters("pears");
        endUser.starts_search();
        endUser.should_see_article_with_title_containing("Pear");
    }

    @Test
    @WithDriver("firefox")
    public void searching_by_keyword_pineapples_should_display_the_corresponding_article
        endUser.is_the_google_home_page();
        endUser.enters("pineapples");
        endUser.starts_search();
        endUser.should_see_article_with_title_containing("Pineapple");
    }
}
```

In **easyb**, you can use the `uses_driver` directive, as shown here:

```
using "thucydides"
...
thucydides.uses_default_base_url "http://localhost:9000"
thucydides.uses_driver chrome

...

scenario "The administrator adds a new category to the system",
{
    given "a new category needs to be added to the system",
    {
      administrator.logs_in_to_admin_page_if_first_time()
      administrator.opens_categories_list()
    }
    when "the administrator adds a new category",
    {
       administrator.selects_add_category()
       administrator.adds_new_category("Scala Developers","SCALA")
    }
    then "the system should confirm that the category has been created",
    {
        administrator.should_see_confirmation_message "The Category has been created"
    }
    and "the new category should be visible to job seekers",
    {
        job_seeker.opens_jobs_page()
        job_seeker.should_see_job_category "Scala Developers"
    }
}
```

In JUnit, you can also use the `@WithDriver` annotation to specify a driver for an individual test. This will override both the system-level driver and the `@Managed` annotation's driver attribute, if provided. For example, the following test will always run in Firefox:

```
    @Test
    @WithDriver("firefox")
    public void searching_by_keyword_pineapples_should_display_the_corresponding_article
```

```
        endUser.is_the_google_home_page();
        endUser.enters("pineapples");
        endUser.starts_search();
        endUser.should_see_article_with_title_containing("Pineapple");
    }
```

Chapter 6. Writing Acceptance Tests with JBehave

Thucydides is an open source library designed to make it easier to define, implement and report on automated acceptance criteria. Until now, Thucydides tests have been implemented using JUnit or easyb. However the most recent version of Thucydides, version 0.9.x, now lets you write your acceptance criteria using the popular JBehave framework.

6.1. JBehave and Thucydides

JBehave is an open source BDD framework originally written by Dan North, the inventor of BDD. It is strongly integrated into the JVM world, and widely used by Java development teams wanting to implement BDD practices in their projects.

In JBehave, you write automate your acceptance criteria by writing test stories and scenarios using the familiar BDD "given-when-then" notation, as shown in the following example:

```
Scenario: Searching by keyword and category

Given Sally wants to buy some antique stamps for her son
When she looks for ads in the 'Antiques' category containing 'stamps'
Then she should obtain a list of ads related to 'stamps' from the 'Antiques' category
```

Scenarios like this go in `.story` files: a story file is designed to contain all the scenarios (acceptence criteria) of a given user story. A story file can also have a narrative section at the top, which gives some background and context about the story being tested:

```
In order to find the items I am interested in faster
As a buyer
I want to be able to list all the ads with a particular keyword in the description or ti

Scenario: Searching by keyword and category

Given Sally wants to buy some antique stamps for her son
When she looks for ads in the 'Antiques' category containing 'stamps'
Then she should obtain a list of ads related to 'stamps' from the 'Antiques' category

Scenario: Searching by keyword and location

Given Sally wants to buy a puppy for her son
When she looks for ads in the Pets & Animals category containing puppy in New South Wale
Then she should obtain a list of Pets & Animals ads containing the word puppy
  from advertisers in New South Wales
```

You usually implement a JBehave story using classes and methods written in Java, Groovy or Scala. You implement the story steps using annotated methods to represent the steps in the text scenarios, as shown in the following example:

```
public class SearchSteps {
    @Given("Sally wants to buy a $gift for her son")
    public void sally_wants_to_buy_a_gift(String gift) {
        // test code
    }

    @When("When she looks for ads in the $category category containing $keyword in $regi
    public void looking_for_an_ad(String category, String keyword, String region){
        // more test code
    }
}
```

6.2. Working with JBehave and Thucydides

Thucydides and JBehave work well together. Thucydides uses simple conventions to make it easier to get started writing and implementing JBehave stories, and reports on both JBehave and Thucydides steps, which can be seamlessly combined in the same class, or placed in separate classes, depending on your preferences.

To get started, you will need to add the Thucydides JBehave plugin to your project. In Maven, just add the following dependencies to your pom.xml file:

```
<dependency>
    <groupId>net.thucydides</groupId>
    <artifactId>thucydides-core</artifactId>
    <version>0.9.2</version>
</dependency>
<dependency>
    <groupId>net.thucydides</groupId>
    <artifactId>thucydides-jbehave-plugin</artifactId>
    <version>0.9.0</version>
</dependency>
```

New versions come out regularly, so be sure to check the Maven Central repository (http://search.maven.org) to know the latest version numbers for each dependency.

6.3. Setting up your project and organizing your directory structure

JBehave is a highly flexible tool. The downside of this is that, out of the box, JBehave requires quite a bit of bootstrap code to get started. Thucydides tries to simplify this process by using a convention-over-configuration approach, which significantly reduces the amount of work needed to get started with your acceptance tests. In fact, you can get away with as little as an empty JUnit test case and a sensibly-organized directory structure for your JBehave stories.

6.3.1. The JUnit test runner

The JBehave tests are run via a JUnit runner. This makes it easier to run the tests both from within an IDE or as part of the build process. All you need to do is to extend the ThucydidesJUnitStories, as shown here:

```
package net.thucydides.showcase.jbehave;

import net.thucydides.jbehave.ThucydidesJUnitStories;

public class JBehaveTestCase extends ThucydidesJUnitStories {
    public JBehaveTestCase() {}
}
```

When you run this test, Thucydides will run any JBehave stories that it finds in the default directory location. By convention, it will look for a ‘stories` folder on your classpath, so `src/test/resources/stories’ is a good place to put your story files.

6.3.2. Organizing your requirements

Placing all of your JBehave stories in one directory does not scale well; it is generally better to organize them in a directory structure that groups them in some logical way. In addition, if you structure your requirements well, Thucydides will be able to provide much more meaningful reporting on the test results.

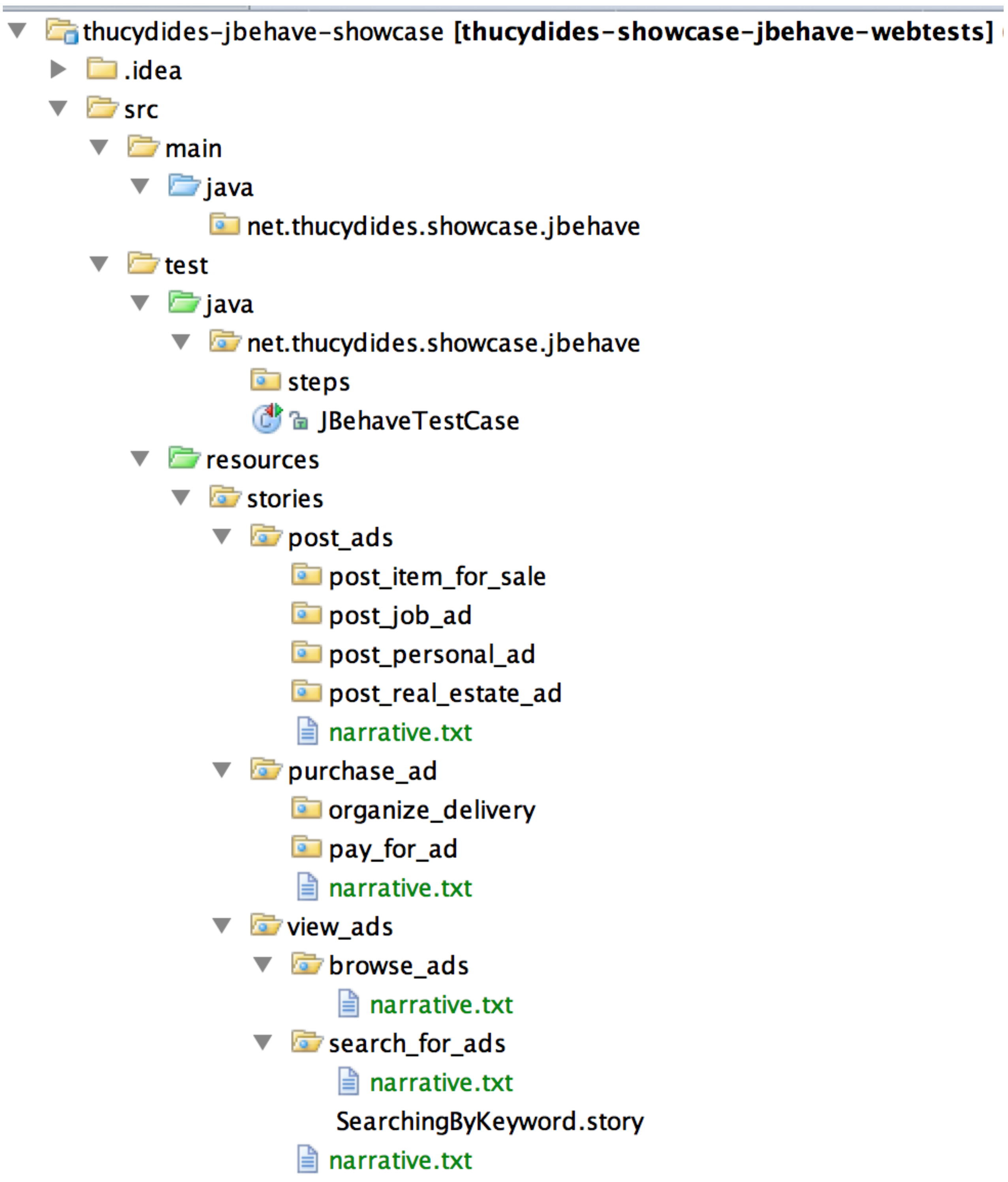
By default, Thucydides supports a simple directory-based convention for organizing your requirements. The standard structure uses three levels: capabilities, features and stories. A story is represented by a JBehave .story file so two directory levels underneath the `stories` directory will do the trick. An example of this structure is shown below:

```
+ src
  + test
    + resources
      + stories
        + grow_potatoes                     [a capability]
          + grow_organic_potatoes           [a feature]
            - plant_organic_potatoes.story  [a story]
            - dig_up_organic_potatoes.story [another story]
          + grow_sweet_potatoes             [another feature]
          ...
```

If you prefer another hierarchy, you can use the `thucydides.capability.types` system property to override the default convention. For example. if you prefer to organize your requirements in a hierachy consisting of epics, theme and stories, you could set the `thucydides.capability.types` property to *epic,theme* (the story level is represented by the .story file).

When you start a project, you will typically have a good idea of the capabilities you intent to implement, and probably some of the main features. If you simply store your .story files in the right directory structure, the Thucydides reports will reflect these requirements, even if no tests have yet been specified for them. This is an excellent way to keep track of project progress. At the start of an iteration, the reports will show all of the requirements to be implemented, even those with no tests defined or implemented yet. As the iteration progresses, more and more acceptance criteria will be implemented, until acceptance criteria have been defined and implemented for all of the requirements that need to be developed.

Figure 6.1. A Thucyides project using JBehave can organize the stories in an appropriate directory structure



```
▼ thucydides-jbehave-showcase [thucydides-showcase-jbehave-webtests]
  ▶ .idea
  ▼ src
    ▼ main
      ▼ java
          net.thucydides.showcase.jbehave
    ▼ test
      ▼ java
        ▼ net.thucydides.showcase.jbehave
            steps
            JBehaveTestCase
      ▼ resources
        ▼ stories
          ▼ post_ads
              post_item_for_sale
              post_job_ad
              post_personal_ad
              post_real_estate_ad
              narrative.txt
          ▼ purchase_ad
              organize_delivery
              pay_for_ad
              narrative.txt
          ▼ view_ads
            ▼ browse_ads
                narrative.txt
            ▼ search_for_ads
                narrative.txt
                SearchingByKeyword.story
              narrative.txt
```

An optional but useful feature of the JBehave story format is the narrative section that can be placed at the start of a story to help provide some more context about that story and the scenarios it contains. This narrative will appear in the Thucydides reports, to help give product owners, testers and other team members more information about the background and motivations behind each story. For example, if you are working on an online classifieds website, you might want users to be able to search ads using keywords. You could describe this functionality with a textual description like this one:

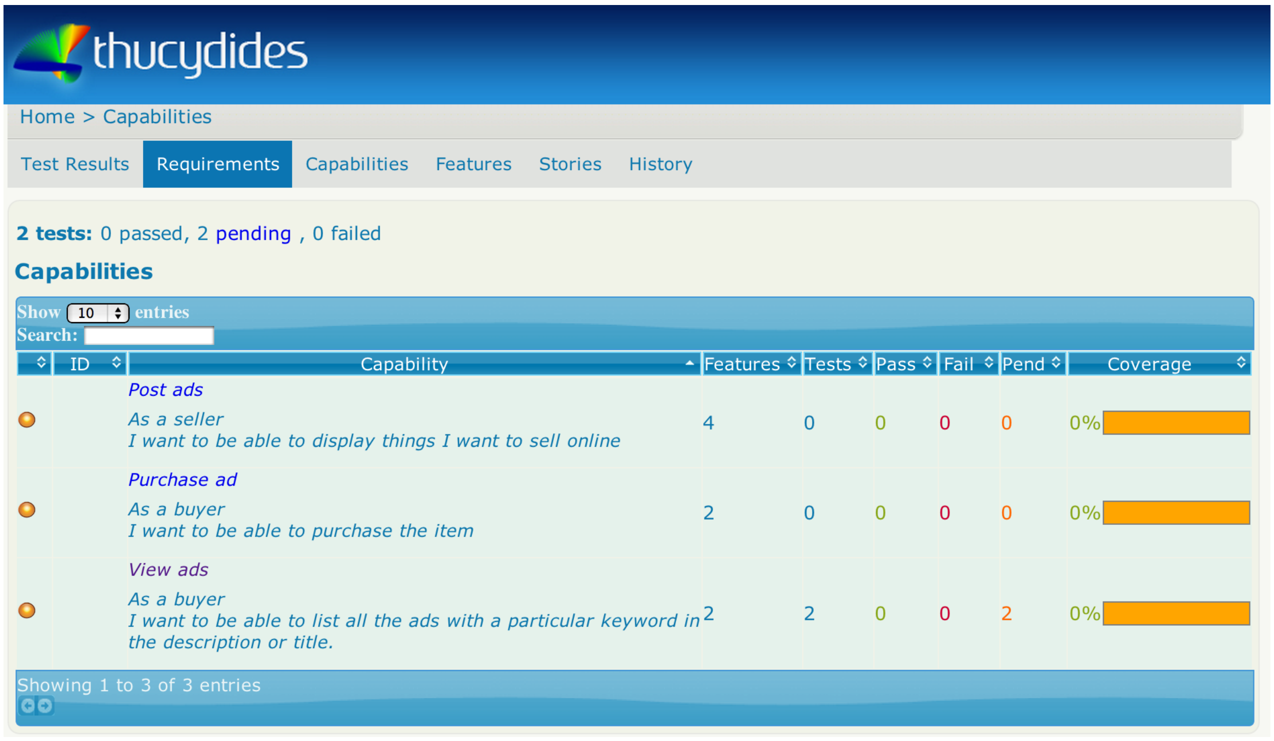
```
Story: Search for ads by keyword
In order to find the items I am interested in faster
As a buyer
I want to be able to list all the ads with a particular keyword
in the description or title.
```

However to make the reports more useful still, it is a good idea to document not only the stories, but to also do the same for your higher level requirements. In Thucydides, you can do this by placing a text file called `narrative.txt` in each of the requirements directories you want to document (see below). These files follow the JBehave/Cucumber convention for writing narratives, with an optional title on the first line, followed by a narrative section started by the keyword `Narrative:`. For example, for a search feature for an online classifieds web site, you might have a description along the following lines:

```
Search for online ads

Narrative:
In order to increase sales of advertised articles
As a seller
I want potential buyers to be able to display only the ads for
articles that they might be interested in purchasing.
```

When you run these stories (without having implemented any actual tests), you will get a report containing lots of pending tests, but more interestingly, a list of the requirements that need to be implemented, even if there are no tests or stories associated with them yet. This makes it easier to plan an iteration: you will initially have a set of requirements with only a few tests, but as the iteration moves forward, you will typically see the requirements fill out with pending and passing acceptance criteria as work progresses.

Figure 6.2. You can see the requirements that you need to implement n the requirements report



2 tests: 0 passed, 2 pending , 0 failed

Capabilities

	ID	Capability	Features	Tests	Pass	Fail	Pend	Coverage
●		*Post ads* *As a seller* *I want to be able to display things I want to sell online*	4	0	0	0	0	0%
●		*Purchase ad* *As a buyer* *I want to be able to purchase the item*	2	0	0	0	0	0%
●		*View ads* *As a buyer* *I want to be able to list all the ads with a particular keyword in the description or title.*	2	2	0	0	2	0%

Showing 1 to 3 of 3 entries

Narrative in asciidoc format

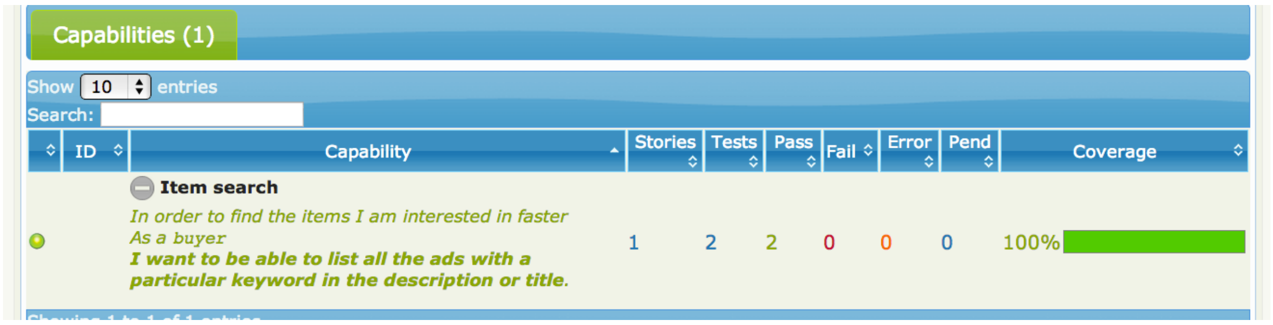
Narratives can be written in Asciidoc [http://www.methods.co.nz/asciidoc/] for richer formatting. Set the `narrative.format` property to `asciidoc` to allow Thucydides to parse the narrative in asciidoc format.

For example, the following narrative,

```
Item search

Narrative:
In order to find the items I am interested in faster
As a +buyer+
*I want to be able to list all the ads with a particular keyword in the description or t
```

will be rendered on the report as shown below.

Figure 6.3. Narrative with asciidoc formatting



Capabilities (1)

Show 10 entries
Search:

	ID	Capability	Stories	Tests	Pass	Fail	Error	Pend	Coverage
●		**Item search** *In order to find the items I am interested in faster* *As a buyer* ***I want to be able to list all the ads with a particular keyword in the description or title.***	1	2	2	0	0	0	100%

6.3.3. Customizing the requirements module

You can also easily extend the Thucydides requirements support so that it fits in to your own system. This is a two-step process. First, you need to write an implementation of the `RequirementsTagProvider` interface.

```
package com.acme.tests

public class MyRequirementsTagProvider implements RequirementsTagProvider {
    @Override
    public List<Requirement> getRequirements() {
        // Return the full list of available requirements from your system
    }

    @Override
    public Optional<Requirement> getParentRequirementOf(TestOutcome testOutcome) {
        // Return the requirement, if any, associated with a particular test result
    }

    @Override
    public Set<TestTag> getTagsFor(TestOutcome testOutcome) {
        // Return all the requirements, and other tags, associated with a particular tes
    }
}
```

Next, create a text file in your `src/main/resources/META-INF/serices` directory called `net.thucydides.core.statistics.service.TagProvider`, and put the fullly qualified name of your RequirementsTagProvider implementation.

6.3.4. Story meta-data

You can use the JBehave Meta tag to provide additional information to Thucydides about the test. The @driver annotation lets you specify what WebDriver driver to use, eg.

```
Meta:
@driver htmlunit

Scenario: A scenario that uses selenium

Given I am on the test page
When I enter the first name <firstname>
And I enter the last name <lastname>
Then I should see <firstname> and <lastname> in the names fields
And I should be using HtmlUnit

Examples:
|firstname|lastname|
|Joe      | Blow|
|John     | Doe   |
```

You can also use the @issue annotation to link scenarios with issues, as illustrated here:

```
Meta:
@issue MYPROJ-1, MYPROJ-2

Scenario: A scenario that works
Meta:
@issues MYPROJ-3,MYPROJ-4
@issue MYPROJ-5

Given I have an implemented JBehave scenario
And the scenario works
When I run the scenario
Then I should get a successful result
```

You can also attribute tags to the story as a whole, or to individual scenarios:

```
Meta:
@tag capability:a capability

Scenario: A scenario that works
Meta:
@tags domain:a domain, iteration: iteration 1

Given I have an implemented JBehave scenario
And the scenario works
When I run the scenario
Then I should get a successful result
```

6.3.5. Implementing the tests

If you want your tests to actually do anything, you will also need classes in which you place your JBehave step implementations. If you place these in any package at or below the package of your main JUnit test, JBehave will find them with no extra configuration.

Thucydides makes no distinction between the JBehave-style @Given, @When and @Then annotations, and the Thucydides-style @Step annotations: both will appear in the test reports. However you need to start with the @Given, @When and @Then-annotated methods so that JBehave can find the correct methods to call for your stories. A method annotated with @Given, @When or @Then can call Thucydides @Step methods, or call page objects directly (though the extra level of abstraction provided by the @Step methods tends to make the tests more reusable and maintainable on larger projects).

A typical example is shown below. In this implementation of one of the scenarios we saw above, the high-level steps are defined using methods annotated with the JBehave @Given, @When and @Then annotations. These methods, in turn, use steps that are implemented in the BuyerSteps class, which contains a set of Thucydides @Step methods. The advantage of using this two-leveled approach is that it helps maintain a degree of separation between the definition of what is being done in a test, and how it is being implemented. This tends to make the tests easier to understand and easier to maintain.

```
public class SearchScenarioSteps {
    @Steps
    BuyerSteps buyer;

    @Given("Sally wants to buy a $present for her son")
    public void buyingAPresent(String present) {
        buyer.opens_home_page();
    }

    @When("she looks for ads in the $category category containing $keyword in $region")
    public void adSearchByCategoryAndKeywordInARegion(String category,String keyword,Str
        buyer.chooses_region(region);
        buyer.chooses_category_and_keywords(category, keyword);
        buyer.performs_search();
    }

    @Then("she should obtain a list of $category ads containing the word $keyword from a
    public void resultsForACategoryAndKeywordInARegion(String category,String keyword,St
        buyer.should_only_see_results_with_titles_containing(keyword);
        buyer.should_only_see_results_from_region(region);
        buyer.should_only_see_results_in_category(category);
    }
}
```

The Thucydides steps can be found in the BuyserSteps class. This class in turn uses Page Objects to interact with the actual web application, as illustrated here:

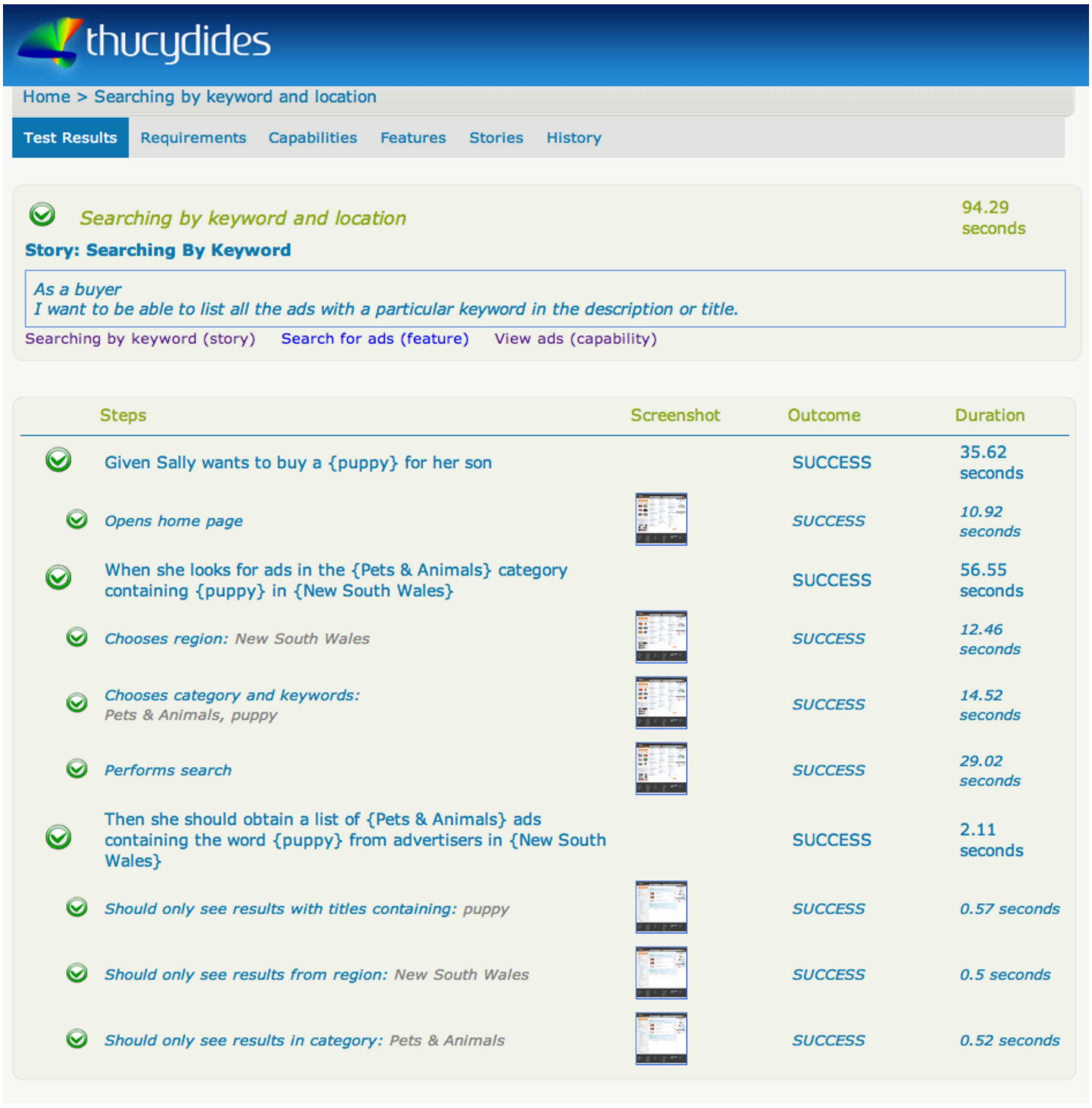
```
public class BuyerSteps extends ScenarioSteps {

    HomePage homePage;
    SearchResultsPage searchResultsPage;

    public BuyerSteps(Pages pages) {
        super(pages);
        homePage = getPages().get(HomePage.class);
        searchResultsPage = getPages().get(SearchResultsPage.class);
    }

    @Step
    public void opens_home_page() {
        homePage.open();
    }

    @Step
    public void chooses_region(String region) {
        homePage.chooseRegion(region);
    }

    @Step
    public void chooses_category_and_keywords(String category, String keywords) {
        homePage.chooseCategoryFromDropdown(category);
        homePage.enterKeywords(keywords);
    }

    @Step
    public void performs_search() {
        homePage.performSearch();
    }

    @Step
    public void should_only_see_results_with_titles_containing(String title) {
        searchResultsPage.allTitlesShouldContain(title);
    }
    ...
}
```

The Page Objects are similar to those you would find in any Thucydides project, as well as most WebDriver projects. An example is listed below:

```
@DefaultUrl("http://www.newsclassifieds.com.au")
public class HomePage extends PageObject {

    @CacheLookup
    @FindBy(name="adFilter.searchTerm")
    WebElement searchTerm;

    @CacheLookup
    @FindBy(css=".keywords button")
    WebElement search;

    public HomePage(WebDriver driver) {
        super(driver);
    }

    public void chooseRegion(String region) {
        findBy("#location-select .arrow").then().click();
        waitFor(500).milliseconds();
        findBy("//ul[@class='dropdown-menu']//a[.='" + region + "']").then().click();
    }

    public void chooseCategoryFromDropdown(String category) {
        getDriver().navigate().refresh();
        findBy("#category-select").then(".arrow").then().click();
        findBy("//span[@id='category-select']//a[contains(.,'" + category + "')]").then(
    }

    public void enterKeywords(String keywords) {
        element(searchTerm).type(keywords);
    }

    public void performSearch() {
        element(search).click();
    }
}
```

When these tests are executed, the JBehave steps combine with the Thucydides steps to create a narrative report of the test results:

Figure 6.4. You can see the requirements that you need to implement in the requirements report



Searching by keyword and location 94.29 seconds

Story: Searching By Keyword

As a buyer
I want to be able to list all the ads with a particular keyword in the description or title.

Searching by keyword (story) Search for ads (feature) View ads (capability)

Steps	Screenshot	Outcome	Duration
Given Sally wants to buy a {puppy} for her son		SUCCESS	35.62 seconds
Opens home page		*SUCCESS*	*10.92 seconds*
When she looks for ads in the {Pets & Animals} category containing {puppy} in {New South Wales}		SUCCESS	56.55 seconds
Chooses region: New South Wales		*SUCCESS*	*12.46 seconds*
Chooses category and keywords: Pets & Animals, puppy		*SUCCESS*	*14.52 seconds*
Performs search		*SUCCESS*	*29.02 seconds*
Then she should obtain a list of {Pets & Animals} ads containing the word {puppy} from advertisers in {New South Wales}		SUCCESS	2.11 seconds
Should only see results with titles containing: puppy		*SUCCESS*	*0.57 seconds*
Should only see results from region: New South Wales		*SUCCESS*	*0.5 seconds*
Should only see results in category: Pets & Animals		*SUCCESS*	*0.52 seconds*

6.4. JBehave Maven Archetype

A jBehave archetype is availble to help you jumpstart a new project. As usual, you can run mvn archetype:generate from the command line and then select the net.thucydides.thucydides-jbehave-archetype archetype from the proposed list of archetypes. Or you can use your favorite IDE to generate a new Maven project using an archetype.

This archetype creates a project directory structure similar to the one shown here:

```
+ main
    + java
       + SampleJBehave
           + pages
               - DictionaryPage.java
           + steps
               - EndUserSteps.java
+ test
    + java
       + SampleJBehave
           + jbehave
               - AcceptanceTestSuite.java
               - DefinitionSteps.java
    + resources
        + SampleJBehave
            + stories
                + consult_dictionary
                    - LookupADefinition.story
```

6.5. Running all tests in a single browser window

All web tests can be run in a single browser window using either by setting the thucydides.use.unique.browser system property or programmatically using runThucydides().inASingleSession() inside the junit runner.

```
package net.thucydides.showcase.jbehave;

import net.thucydides.jbehave.ThucydidesJUnitStories;

public class JBehaveTestCase extends ThucydidesJUnitStories {
    public JBehaveTestCase() {
      runThucydides().inASingleSession();
    }
}
```

Chapter 7. Implementing Step Libraries

Once you have defined the steps you need to describe your high level tests, you need to implement these steps. In an automated web test, test steps represent the level of abstraction between your Page Objects (which are designed in terms of actions that you perform on a given page) and higher-level stories (sequences of more business-focused actions that illustrate how a given user story has been implemented). Steps can contain other steps, and are included in the Thucydides reports. Whenever a step is executed, a screenshot is stored and displayed in the report.

7.1. Creating Step Libraries

Test steps are regular java methods, annotated with the `@Step` annotation. You organize steps and step groups in step libraries. A step library is just a normal Java class. If you are running web tests, your step library should either have a `Pages` member variable, or (more simply) extend the `ScenarioSteps` class, e.g.

```
public class JobSeekerSteps extends ScenarioSteps {
    public JobSeekerSteps(Pages pages) {
        super(pages);
    }


    @Step
    public void opens_jobs_page() {
        FindAJobPage page = getPages().get(FindAJobPage.class);
        page.open();
    }

    @Step
    public void searches_for_jobs_using(String keywords) {
        FindAJobPage page = getPages().get(FindAJobPage.class);
        page.look_for_jobs_with_keywords(keywords);

    }
}
```

Note that step methods can take parameters. The parameters that are passed into a step method will be recorded and reported in the Thucydides reports, making this an excellent technique to make your tests more maintainable and more modular.

Steps can also call other steps, which is very useful for more complicated test scenarios. The result is the sort of nested structure you can see in Figure 2.1, “A test report generated by Thucydides”.

Chapter 8. Defining Page Objects

If you are working with WebDriver web tests, you will be familiar with the concept of Page Objects. Page Objects are a way of isolating the implementation details of a web page inside a class, exposing only business-focused methods related to that page. They are an excellent way of making your web tests more maintainable.

In Thucydides, page objects can be just ordinary WebDriver page objects, on the condition that they have a constructor that accepts a WebDriver parameter. However, the Thucydides `PageObject` class provides a number of utility methods that make page objects more convenient to work with, so a Thucydides Page Object generally extends this class.

Here is a simple example:

```
@DefaultUrl("http://localhost:9000/somepage")
public class FindAJobPage extends PageObject {

    WebElement keywords;
    WebElement searchButton;

    public FindAJobPage(WebDriver driver) {
        super(driver);
    }

    public void look_for_jobs_with_keywords(String values) {
        typeInto(keywords, values);
        searchButton.click();
    }

    public List<String> getJobTabs() {
        List<WebElement> tabs = getDriver().findElements(By.xpath("//div[@id='tabs']//a"
        return extract(tabs, on(WebElement.class).getText());
    }
}
```

The `typeInto` method is a shorthand that simply clears a field and enters the specified text. If you prefer a more fluent-API style, you can also do something like this:

```
@DefaultUrl("http://localhost:9000/somepage")
public class FindAJobPage extends PageObject {
        WebElement keywordsField;
        WebElement searchButton;

        public FindAJobPage(WebDriver driver) {
            super(driver);
        }

        public void look_for_jobs_with_keywords(String values) {
            **enter(values).into(keywordsField);**
            searchButton.click();
        }

        public List<String> getJobTabs() {
            List<WebElement> tabs = getDriver().findElements(By.xpath("//div[@id='tabs']
            return extract(tabs, on(WebElement.class).getText());
        }
}
```

You can use an even more fluent style of expressing the implementation steps by using methods like `find`, `findBy` and `then`.

For example, you can use webdriver "By" finders with element name, id, css selector or xpath selector as follows:

```
page.find(By.name("demo")).then(By.name("specialField")).getValue();

page.find(By.cssSelector(".foo")).getValue();

page.find(By.xpath("//th")).getValue();
```

You can also use `findBy` method and pass the css or xpath selector directly. For example,

```
page.findBy("#demo").then("#specialField").getValue(); //css selectors

page.findBy("//div[@id='dataTable']").getValue(); //xpath selector
```

8.1. Using pages in a step library

When you need to use a page object in one of your steps, you just ask for one from the Page factory, providing the class of the page object you need, e.g.

```
FindAJobPage page = getPages().get(FindAJobPage.class);
```

If you want to make sure you are on the right page, you can use the `currentPageAt()` method. This will check the page class for any `@At` annotations present in the Page Object class and, if present, check that the current URL corresponds to the URL pattern specified in the annotation. For example, when you invoke it using `currentPageAt()`, the following Page Object will check that the current URL is precisely http://www.apache.org.

```
@At("http://www.apache.org")
public class ApacheHomePage extends PageObject {
    ...
}
```

The `@At` annotation also supports wildcards and regular expressions. The following page object will match any Apache sub-domain:

```
@At("http://.*.apache.org")
public class AnyApachePage extends PageObject {
    ...
}
```

More generally, however, you are more interested in what comes after the host name. You can use the special `#HOST` token to match any server name. So the following Page Object will match both http://localhost:8080/app/action/login.form an http://staging.acme.com/app/action/login.form. It will also ignore parameters, so http://staging.acme.com/app/action/login.form?username=toto&password=oz will work fine too.

```
@At(urls={"#HOST/app/action/login.form"})
public class LoginPage extends PageObject {
   ...
}
```

8.2. Opening the page

A page object is usually designed to work with a particular web page. When the `open()` method is invoked, the browser will be opened to the default URL for the page.

The `@DefaultUrl` annotation indicates the URL that this test should use when run in isolation (e.g. from within your IDE). Generally, however, the host part of the default URL will be overridden by the `webdriver.base.url` property, as this allows you to set the base URL across the board for all of your tests, and so makes it easier to run your tests on different environments by simply changing this property value. For example, in the test class above, setting the `webdriver.base.url` to *https://staging.mycompany.com* would result in the page being opened at the URL of *https://staging.mycompany.com/somepage*.

You can also define named URLs that can be used to open the web page, optionally with parameters. For example, in the following code, we define a URL called *open.issue*, that accepts a single parameter:

```
@DefaultUrl("http://jira.mycompany.org")
@NamedUrls(
  {
    @NamedUrl(name = "open.issue", url = "http://jira.mycompany.org/issues/{1}")
  }
)
public class JiraIssuePage extends PageObject {
    ...
}
```

You could then open this page to the http://jira.mycompany.org/issues/ISSUE-1 URL as shown here:

```
page.open("open.issue", withParameters("ISSUE-1"));
```

You could also dispense entirely with the base URL in the named URL definition, and rely on the default values:

```
@DefaultUrl("http://jira.mycompany.org")
@NamedUrls(
  {
    @NamedUrl(name = "open.issue", url = "/issues/{1}")
  }
)
public class JiraIssuePage extends PageObject {
    ...
}
```

And naturally you can define more than one definition:

```
@NamedUrls(
  {
          @NamedUrl(name = "open.issue", url = "/issues/{1}"),
          @NamedUrl(name = "close.issue", url = "/issues/close/{1}")
  }
)
```

You should never try to implement the `open()` method yourself. In fact, it is final. If you need your page to do something upon loading, such as waiting for a dynamic element to appear, you can use the

@WhenPageOpens annotation. Methods in the PageObject with this annotation will be invoked (in an unspecified order) after the URL has been opened. In this example, the `open()` method will not return until the dataSection web element is visible:

```
@DefaultUrl("http://localhost:8080/client/list")
    public class ClientList extends PageObject {

     @FindBy(id="data-section");
     WebElement dataSection;
     ...

     @WhenPageOpens
     public void waitUntilTitleAppears() {
         element(dataSection).waitUntilVisible();
     }
}
```

8.3. Working with web elements

Important

The element() method described below is no longer needed. See Web Element Facade section for details.

8.3.1. Checking whether elements are visible

The element method of the PageObject class provides a convenient fluent API for dealing with web elements, providing some commonly-used extra features that are not provided out-of-the-box by the WebDriver API. For example, you can check that an element is visible as shown here:

```
public class FindAJobPage extends PageObject {

    WebElement searchButton;

    public boolean searchButtonIsVisible() {
        return element(searchButton).isVisible();
    }
    ...
}
```

If the button is not present on the screen, the test will wait for a short period in case it appears due to some Ajax magic. If you don’t want the test to do this, you can use the faster version:

```
public boolean searchButtonIsVisibleNow() {
    return element(searchButton).isCurrentlyVisible();
}
```

You can turn this into an assert by using the `shouldBeVisible()` method instead:

```
public void checkThatSearchButtonIsVisible() {
    element(searchButton).shouldBeVisible();
}
```

This method will through an assertion error if the search button is not visible to the end user.

If you are happy to expose the fact that your page has a search button to your step methods, you can make things even simpler by adding an accessor method that returns a WebElementFacade, as shown here:

```
public WebElementFacade searchButton() {
    return element(searchButton);
}
```

Then your steps will contain code like the following:

```
        searchPage.searchButton().shouldBeVisible();
```

8.3.2. Checking whether elements are enabled

You can also check whether an element is enabled or not:

```
element(searchButton).isEnabled() element(searchButton).shouldBeEnabled()
```

There are also equivalent negative methods:

```
element(searchButton).shouldNotBeVisible();
element(searchButton).shouldNotBeCurrentlyVisible();
element(searchButton).shouldNotBeEnabled()
```

You can also check for elements that are present on the page but not visible, e.g:

```
element(searchButton).isPresent();
element(searchButton).isNotPresent();
element(searchButton).shouldBePresent();
element(searchButton).shouldNotBePresent();
```

8.3.3. Manipulating select lists

There are also helper methods available for drop-down lists. Suppose you have the following dropdown on your page:

```
<select id="color">
    <option value="red">Red</option>
    <option value="blue">Blue</option>
    <option value="green">Green</option>
</select>
```

You could write a page object to manipulate this dropdown as shown here:

```
public class FindAJobPage extends PageObject {

        @FindBy(id="color")
        WebElement colorDropdown;

        public selectDropdownValues() {
            element(colorDropdown).selectByVisibleText("Blue");
            assertThat(element(colorDropdown).getSelectedVisibleTextValue(), is("Blue"))

            element(colorDropdown).selectByValue("blue");
            assertThat(element(colorDropdown).getSelectedValue(), is("blue"));

            page.element(colorDropdown).selectByIndex(2);
            assertThat(element(colorDropdown).getSelectedValue(), is("green"));
        }
        ...
}
```

8.3.4. Determining focus

You can determine whether a given field has the focus as follows:

```
element(firstName).hasFocus()
```

You can also wait for elements to appear, disappear, or become enabled or disabled:

```
element(button).waitUntilEnabled()
element(button).waitUntilDisabled()
```

or

```
element(field).waitUntilVisible()
element(button).waitUntilNotVisible()
```

8.3.5. Using WebElementFacade variables directly

Instead of declaring WebElement variables in Page Objects and then calling element() or $() to wrap them in WebElementFacades, you can now declare WebElementFacade variables directly inside the Page Objects. This will make the Page Object code simpler more readable.

So, instead of writing,

```
public class FindAJobPage extends PageObject {

    WebElement searchButton;

    public boolean searchButtonIsVisible() {
        return element(searchButton).isVisible();
    }
    ...
}
```

you can write,

```
public class FindAJobPage extends PageObject {

    WebElementFacade searchButton;

    public boolean searchButtonIsVisible() {
        return searchButton.isVisible();
    }
    ...
}
```

8.3.6. Using direct XPath and CSS selectors

Another way to access a web element is to use an XPath or CSS expression. You can use the `element` method with an XPath expression to do this more simply. For example, imagine your web application needs to click on a list item containing a given post code. One way would be as shown here:

```
WebElement selectedSuburb = getDriver().findElement(By.xpath("//li/a[contains(.,'" + pos
selectedSuburb.click();
```

However, a simpler option would be to do this:

```
element(By.xpath("//li/a[contains(.,'" + postcode + "')]")).click();
```

8.4. Working with Asynchronous Pages

Asynchronous pages are those whose fields or data is not all displayed when the page is loaded. Sometimes, you need to wait for certain elements to appear, or to disappear, before being able to proceed with your tests. Thucydides provides some handy methods in the PageObject base class to help with these scenarios. They are primarily designed to be used as part of your business methods in your page objects, though in the examples we will show them used as external calls on a PageObject instance for clarity.

8.4.1. Checking whether an element is visible

In WebDriver terms, there is a distinction between when an element is present on the screen (i.e. in the HTML source code), and when it is rendered (i.e. visible to the user). You may also need to check whether an element is visible on the screen. You can do this in two ways. Your first option is to use the isElementVisible method, which returns a boolean value based on whether the element is rendered (visible to the user) or not:

```
assertThat(indexPage.isElementVisible(By.xpath("//h2[.='A visible title']")), is(true));
```

or

```
assertThat(indexPage.isElementVisible(By.xpath("//h2[.='An invisible title']")), is(fals
```

Your second option is to actively assert that the element should be visible:

```
indexPage.shouldBeVisible(By.xpath("//h2[.='An invisible title']");
```

If the element does not appear immediately, you can wait for it to appear:

```
indexPage.waitForRenderedElements(By.xpath("//h2[.='A title that is not immediately visi
```

An alternative to the above syntax is to use the more fluid `waitFor` method which takes a css or xpath selector as argument:

```
indexPage.waitFor("#popup"); //css selector

indexPage.waitFor("//h2[.='A title that is not immediately visible']"); //xpath selector
```

If you just want to check if the element is present though not necessarily visible, you can use `waitForRenderedElementsToBePresent` :

```
indexPage.waitForRenderedElementsToBePresent(By.xpath("//h2[.='A title that is not immed
```

or its more expressive flavour, `waitForPresenceOf` which takes a css or xpath selector as argument.

```
indexPage.waitForPresenceOf("#popup"); //css

indexPage.waitForPresenceOf("//h2[.='A title that is not immediately visible']"); //xpat
```

You can also wait for an element to disappear by using `waitForRenderedElementsToDisappear` or `waitForAbsenceOf` :

```
indexPage.waitForRenderedElementsToDisappear(By.xpath("//h2[.='A title that will soon di

indexPage.waitForAbsenceOf("#popup");

indexPage.waitForAbsenceOf("//h2[.='A title that will soon disappear']");
```

For simplicity, you can also use the `waitForTextToAppear` and `waitForTextToDisappear` methods:

```
indexPage.waitForTextToDisappear("A visible bit of text");
```

If several possible texts may appear, you can use `waitForAnyTextToAppear` or `waitForAllTextToAppear`:

```
indexPage.waitForAnyTextToAppear("this might appear","or this", "or even this");
```

If you need to wait for one of several possible elements to appear, you can also use the `waitForAnyRenderedElementOf` method:

```
indexPage.waitForAnyRenderedElementOf(By.id("color"), By.id("taste"), By.id("sound"));
```

8.5. Executing Javascript

There are times when you may find it useful to execute a little Javascript directly within the browser to get the job done. You can use the `evaluateJavascript()` method of the `PageObject` class to do this. For example, you might need to evaluate an expression and use the result in your tests. The following command will evaluate the document title and return it to the calling Java code:

```
String result = (String) evaluateJavascript("return document.title");
```

Alternatively, you may just want to execute a Javascript command locally in the browser. In the following code, for example, we set the focus to the *firstname* input field:

```
        evaluateJavascript("document.getElementById('firstname').focus()");
```

And, if you are familiar with JQuery, you can also invoke JQuery expressions:

```
        evaluateJavascript("$('#firstname').focus()");
```

This is often a useful strategy if you need to trigger events such as mouse-overs that are not currently supported by the WebDriver API.

8.6. Uploading files

Uploading files is easy. Files to be uploaded can be either placed in a hard-coded location (bad) or stored on the classpath (better). Here is a simple example:

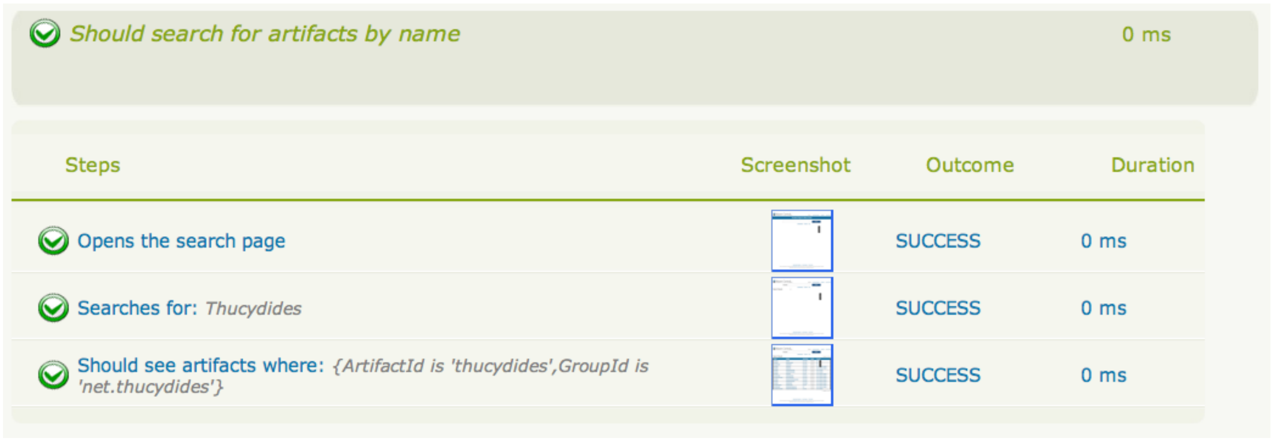
```
public class NewCompanyPage extends PageObject {
    ...
    @FindBy(id="object_logo")
    WebElement logoField;

    public NewCompanyPage(WebDriver driver) {
        super(driver);
    }

    public void loadLogoFrom(String filename) {
        upload(filename).to(logoField);
    }
}
```

8.7. Using Fluent Matcher expressions

When writing acceptance tests, you often find yourself expressing expectations about individual domain objects or collections of domain objects. For example, if you are testing a multi-criteria search feature, you will want to know that the application finds the records you expected. You might be able to do this in a very precise manner (for example, knowing exactly what field values you expect), or you might want to make your tests more flexible by expressing the ranges of values that would be acceptable. Thucydides provides a few features that make it easier to write acceptance tests for this sort of case.

In the rest of this section, we will study some examples based on tests for the Maven Central search site (see Figure 8.1, "The results page for the Maven Central search page"). This site lets you search the Maven repository for Maven artifacts, and view the details of a particular artifact.

Figure 8.1. The results page for the Maven Central search page



Should search for artifacts by name 0 ms

Steps	Screenshot	Outcome	Duration
✓ Opens the search page		SUCCESS	0 ms
✓ Searches for: *Thucydides*		SUCCESS	0 ms
✓ Should see artifacts where: *{ArtifactId is 'thucydides',GroupId is 'net.thucydides'}*		SUCCESS	0 ms

We will use some imaginary regression tests for this site to illustrate how the Thucydides matchers can be used to write more expressive tests. The first scenario we will consider is simply searching for an artifact by name, and making sure that only artifacts matching this name appear in the results list. We might express this acceptance criteria informally in the following way:

- Give that the developer is on the search page,
- And the developer searches for artifacts called *Thucydides*
- Then the developer should see at least 16 Thucydides artifacts, each with a unique artifact Id

In JUnit, a Thucydides test for this scenario might look like the one:

```
...
import static net.thucydides.core.matchers.BeanMatchers.the_count;
import static net.thucydides.core.matchers.BeanMatchers.each;
import static net.thucydides.core.matchers.BeanMatchers.the;
import static org.hamcrest.Matchers.greaterThanOrEqualTo;
import static org.hamcrest.Matchers.is;
import static org.hamcrest.Matchers.startsWith;

@RunWith(ThucydidesRunner.class)
public class WhenSearchingForArtifacts {

    @Managed
    WebDriver driver;

    @ManagedPages(defaultUrl = "http://search.maven.org")
    public Pages pages;

    @Steps
    public DeveloperSteps developer;

    @Test
    public void should_find_the_right_number_of_artifacts() {
        developer.opens_the_search_page();
        developer.searches_for("Thucydides");
        developer.should_see_artifacts_where(the("GroupId", startsWith("net.thucydides"))
                                             each("ArtifactId").isDifferent(),
                                             the_count(is(greaterThanOrEqualTo(16))));

    }
}
```

Let’s see how the test in this class is implemented. The `should_find_the_right_number_of_artifacts()` test could be expressed as follows:

1. When we open the search page

2. And we search for artifacts containing the word *Thucydides*

3. Then we should see a list of artifacts where each Group ID starts with "net.thucydides", each Artifact ID is unique, and that there are at least 16 such entries displayed.

The implementation of these steps is illustrated here:

```
...
import static net.thucydides.core.matchers.BeanMatcherAsserts.shouldMatch;

public class DeveloperSteps extends ScenarioSteps {

    public DeveloperSteps(Pages pages) {
        super(pages);
    }

    @Step
    public void opens_the_search_page() {
        onSearchPage().open();
    }

    @Step
    public void searches_for(String search_terms) {
        onSearchPage().enter_search_terms(search_terms);
        onSearchPage().starts_search();
    }

    @Step
    public void should_see_artifacts_where(BeanMatcher... matchers) {
        shouldMatch(onSearchResultsPage().getSearchResults(), matchers);
    }

    private SearchPage onSearchPage() {
        return getPages().get(SearchPage.class);
    }

    private SearchResultsPage onSearchResultsPage() {
        return getPages().get(SearchResultsPage.class);
    }
}
```

The first two steps are implemented by relatively simple methods. However the third step is more interesting. Let’s look at it more closely:

```
    @Step
    public void should_see_artifacts_where(BeanMatcher... matchers) {
        shouldMatch(onSearchResultsPage().getSearchResults(), matchers);
    }
```

Here, we are passing an arbitrary number of expressions into the method. These expressions actually *matchers*, instances of the BeanMatcher class. Not that you usually have to worry about that level of detail - you create these matcher expressions using a set of static methods provided in the BeanMatchers class. So you typically would pass fairly readable expressions like `the("GroupId", startsWith("net.thucydides"))` or `each("ArtifactId").isDifferent()`.

The `shouldMatch()` method from the BeanMatcherAsserts class takes either a single Java object, or a collection of Java objects, and checks that at least some of the objects match the constraints specified by the matchers. In the context of web testing, these objects are typically POJOs provided by the Page Object to represent the domain object or objects displayed on a screen.

There are a number of different matcher expressions to choose from. The most commonly used matcher just checks the value of a field in an object. For example, suppose you are using the domain object shown here:

```
     public class Person {
        private final String firstName;
        private final String lastName;

        Person(String firstName, String lastName) {
            this.firstName = firstName;
            this.lastName = lastName;
        }

        public String getFirstName() {...}

        public String getLastName() {...}
    }
```

You could write a test to ensure that a list of Persons contained at least one person named "Bill" by using the "the" static method, as shown here:

```
    List<Person> persons = Arrays.asList(new Person("Bill", "Oddie"), new Person("Tim",
    shouldMatch(persons, the("firstName", is("Bill"))
```

The second parameter in the the() method is a Hamcrest matcher, which gives you a great deal of flexibility with your expressions. For example, you could also write the following:

```
    List<Person> persons = Arrays.asList(new Person("Bill", "Oddie"), new Person("Tim",

    shouldMatch(persons, the("firstName", is(not("Tim"))));
    shouldMatch(persons, the("firstName", startsWith("B")));
```

You can also pass in multiple conditions:

```
    List<Person> persons = Arrays.asList(new Person("Bill", "Oddie"), new Person("Tim",

    shouldMatch(persons, the("firstName", is("Bill"), the("lastName", is("Oddie"));
```

Thucydides also provides the DateMatchers class, which lets you apply Hamcrest matches to standard java Dates and `JodaTime` DateTimes. The following code samples illustrate how these might be used:

```
    DateTime january1st2010 = new DateTime(2010,01,01,12,0).toDate();
    DateTime may31st2010 = new DateTime(2010,05,31,12,0).toDate();

    the("purchaseDate", isBefore(january1st2010))
    the("purchaseDate", isAfter(january1st2010))
    the("purchaseDate", isSameAs(january1st2010))
    the("purchaseDate", isBetween(january1st2010, may31st2010))
```

You sometimes also need to check constraints that apply to all of the elements under consideration. The simplest of these is to check that all of the field values for a particular field are unique. You can do this using the `each()` method:

```
    shouldMatch(persons, each("lastName").isDifferent())
```

You can also check that the number of matching elements corresponds to what you are expecting. For example, to check that there is only one person who’s first name is Bill, you could do this:

```
     shouldMatch(persons, the("firstName", is("Bill"), the_count(is(1)));
```

You can also check the minimum and maximum values using the max() and min() methods. For example, if the Person class had a `getAge()` method, we could ensure that every person is over 21 and under 65 by doing the following:

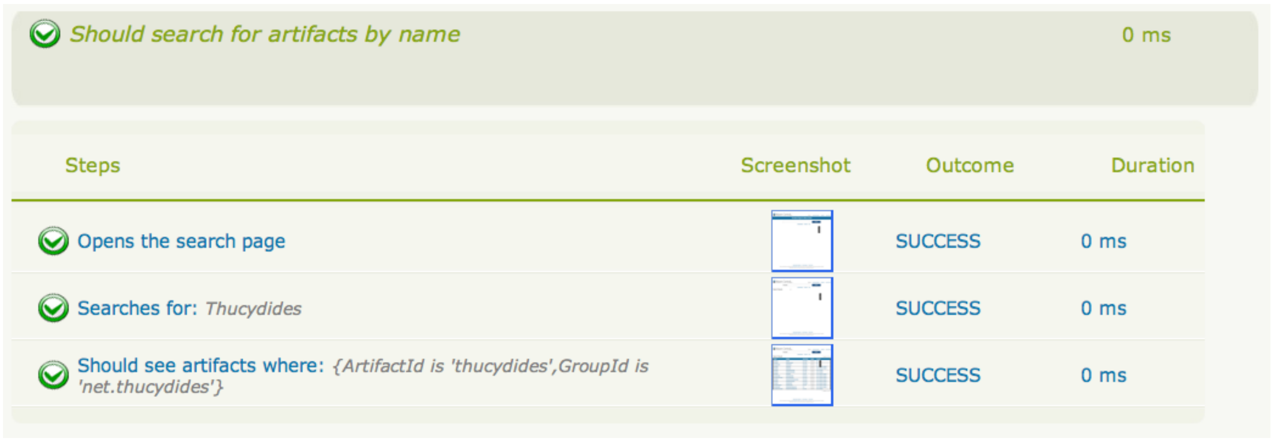
```
     shouldMatch(persons, min("age", greaterThanOrEqualTo(21)),
                          max("age", lessThanOrEqualTo(65)));
```

These methods work with normal Java objects, but also with Maps. So the following code will also work:

```
    Map<String, String> person = new HashMap<String, String>();
    person.put("firstName", "Bill");
    person.put("lastName", "Oddie");

    List<Map<String,String>> persons = Arrays.asList(person);
    shouldMatch(persons, the("firstName", is("Bill"))
```

The other nice thing about this approach is that the matchers play nicely with the Thucydides reports. So when you use the BeanMatcher class as a parameter in your test steps, the conditions expressed in the step will be displayed in the test report, as shown in Figure 8.2, “Conditional expressions are displayed in the test reports”.

Figure 8.2. Conditional expressions are displayed in the test reports



Should search for artifacts by name 0 ms

Steps	Screenshot	Outcome	Duration
Opens the search page		SUCCESS	0 ms
Searches for: *Thucydides*		SUCCESS	0 ms
Should see artifacts where: *{ArtifactId is 'thucydides',GroupId is 'net.thucydides'}*		SUCCESS	0 ms

There are two common usage patterns when building Page Objects and steps that use this sort of matcher. The first is to write a Page Object method that returns the list of domain objects (for example, Persons) displayed on the table. For example, the getSearchResults() method used in the should_see_artifacts_where() step could be implemented as follows:

```
    public List<Artifact> getSearchResults() {
        List<WebElement> rows = resultTable.findElements(By.xpath(".//tr[td]"));
        List<Artifact> artifacts = new ArrayList<Artifact>();
        for (WebElement row : rows) {
            List<WebElement> cells = row.findElements(By.tagName("td"));
            artifacts.add(new Artifact(cells.get(0).getText(),
                                       cells.get(1).getText(),
                                       cells.get(2).getText()));

        }
        return artifacts;
    }
```

The second is to access the HTML table contents directly, without explicitly modelling the data contained in the table. This approach is faster and more effective if you don’t expect to reuse the domain object in other pages. We will see how to do this next.

8.7.1. Working with HTML Tables

Since HTML tables are still widely used to represent sets of data on web applications, Thucydides comes the HtmlTable class, which provides a number of useful methods that make it easier to write Page Objects that contain tables. For example, the rowsFrom method returns the contents of an HTML table as a list of Maps, where each map contains the cell values for a row indexed by the corresponding heading, as shown here:

```
...
import static net.thucydides.core.pages.components.HtmlTable.rowsFrom;

public class SearchResultsPage extends PageObject {

    WebElement resultTable;
    public SearchResultsPage(WebDriver driver) {
        super(driver);
    }

    public List<Map<String, String>> getSearchResults() {
        return rowsFrom(resultTable);
    }

}
```

This saves a lot of typing - our `getSearchResults()` method now looks like this:

```
    public List<Map<String, String>> getSearchResults() {
        return rowsFrom(resultTable);
    }
```

And since the Thucydides matchers work with both Java objects and Maps, the matcher expressions will be very similar. The only difference is that the Maps returned are indexed by the text values contained in the table headings, rather than by java-friendly property names.

You can also read tables without headers (i.e., <th> elements) by specifying your own headings using the `withColumns` method. For example:

```
    List<Map<Object, String>> tableRows =
                    HtmlTable.withColumns("First Name","Last Name", "Favorite Colour")
                             .readRowsFrom(page.table_with_no_headings);
```

You can also use the HtmlTable class to select particular rows within a table to work with. For example, another test scenario for the Maven Search page involves clicking on an artifact and displaying the details for that artifact. The test for this might look something like this:

```
    @Test
    public void clicking_on_artifact_should_display_details_page() {
        developer.opens_the_search_page();
        developer.searches_for("Thucydides");
        developer.open_artifact_where(the("ArtifactId", is("thucydides")),
                                      the("GroupId", is("net.thucydides")));

        developer.should_see_artifact_details_where(the("artifactId", is("thucydides")),
                                                    the("groupId", is("net.thucydides"))
    }
```

Now the open_artifact_where() method needs to click on a particular row in the table. This step looks like this:

```
    @Step
    public void open_artifact_where(BeanMatcher... matchers) {
        onSearchResultsPage().clickOnFirstRowMatching(matchers);
    }
```

So we are effectively delegating to the Page Object, who does the real work. The corresponding Page Object method looks like this:

```
import static net.thucydides.core.pages.components.HtmlTable.filterRows;
...
    public void clickOnFirstRowMatching(BeanMatcher... matchers) {
        List<WebElement> matchingRows = filterRows(resultTable, matchers);
        WebElement targetRow = matchingRows.get(0);
        WebElement detailsLink = targetRow.findElement(By.xpath(".//a[contains(@href,'ar
        detailsLink.click();
    }
```

The interesting part here is the first line of the method, where we use the filterRows() method. This method will return a list of WebElements that match the matchers you have passed in. This method makes it fairly easy to select the rows you are interested in for special treatment.

8.8. Running several steps using the same page object

Sometimes, querying the browser can be expensive. For example, if you are testing tables with large numbers of web elements (e.g. a web element for each cell), performance can be slow, and memory usage high. Normally, Thucydides will requery the page (and create a new Page Object) each time you call `Pages.get()` or `Pages.currentPageAt()`. If you are certain that the page will not change (i.e., that you are only performing read-only operations on the page), you can use the onSamePage() method of the ScenarioSteps class to ensure that subsequent calls to `Pages.get()` or `Pages.currentPageAt()` will return the same page object:

```
@RunWith(ThucydidesRunner.class)
public class WhenDisplayingTableContents {

    @Managed
    public WebDriver webdriver;

    @ManagedPages(defaultUrl = "http://my.web.site/index.html")
    public Pages pages;

    @Steps
    public DemoSiteSteps steps;

    @Test
    public void the_user_opens_another_page() {
        steps.navigate_to_page_with_a_large_table();
        steps.onSamePage(DemoSiteSteps.class).check_row(1);
        steps.onSamePage(DemoSiteSteps.class).check_row(2);
        steps.onSamePage(DemoSiteSteps.class).check_row(3);
    }
}
```

8.9. Switching to another page

A method, switchToPage() is provided in PageObject class to make it convenient to return a new PageObject after navigation from within a method of a PageObject class. For example,

```
@DefaultUrl("http://mail.acme.com/login.html")
public class EmailLoginPage extends PageObject {


    ...
    public void forgotPassword() {
        ...
        forgotPassword.click();
        ForgotPasswordPage forgotPasswordPage = this.switchToPage(ForgotPasswordPage.cla
        forgotPasswordPage.open();
        ...
    }
    ...
}
```

Chapter 9. Advanced JIRA Integration

Thucydides provides tight integration with JIRA, though an extensible plugin architecture. You can store requirements in JIRA, and associate the automated tests with this requirements, so that the requirements defined in JIRA appear in the Thuydides reports. If you are using the JIRA Zephyr plugin to manage manual tests, you can also read manual test results from Zephyr and include them in your Thucydides reports.

9.1. JIRA Integration plugins

A number of Thucydides plugins for JIRA are available for different JIRA configurations. A Requirements plugin implements the `RequirementsTagProvider` interface, and helps Thucydides retrieve a list of project requirements and associate these requirements with test results.

There are several plugins available, that are used for different purposes. All of the JIRA plugins use the **@Issue** annotation (or equivalent) to associate executed tests with a requirement defined in JIRA. In addition to using them out-of-the-box, the source code for the JIRA Thucydides plugins can be used to write your own custom integration plugins for JIRA or other systems.

Thucydides JIRA plugins

- thucydides-jira-plugin: A client library for the JIRA RESTful interface. This library is used by the other plugins, and is not usually used directly unless you want to write your own JIRA integration plugin.
- thucydides-jira-requirements-provider: Reads the requirements structure from the Epics and Stories in JIRA. It reads Epic cards as the top-level requirements and Story cards underneath the Epics. It also reads the versions defined in the JIRA project, using the **Fix Version** field to associated test results with particular versions. Many organizations customize their JIRA card structure, so this plugin is a good place to start if you have a more specific card organization.
- thucydides-jira-customfield-requirements-provider: You use this plugin to define requirements and versions in custom JIRA fields..
- thucydides-structure-plugin-requirements-provider This plugin provides integration with the JIRA structure plugin.

You should not include a dependency on more than one of the JIRA requirements provider plugins.

A few configuration options are used for all of the plugins:

```
jira.url=http://my.jira.server
jira.project=DEMO
jira.username=scott
jira.password=tiger
```

9.2. Reporting on versions

Thucydides lets you report on test results from several different points of view, including requirements (epics, stories, features, capabilites etc.) and versions. You can deactivate this reporting using the `thucydides.report.show.releases` property in your `thucydides.properties` file, e.g.

```
thucydides.report.show.releases = false
```

9.3. Using JIRA versions

By default, Thucydides will read version details from the Releases in JIRA. Test outcomes will be associated with a particular version using the "Fixed versions" field.

JIRA uses a flat version structure - you can’t have for example releases that are made up of a number of sprints. Thucydides lets you organize these in a hierarchical structure based on a simple naming convention. By default, Thucydides uses "release" as the highest level release, and either "iteration" or "sprint" as the second level. For example, suppose you have the the following list of versions in JIRA

- Release 1
- Iteration 1.1
- Iteration 1.2
- Release 2
- Release 3

This will produce Release reports for Release 1, Release 2, and Release 3, with Iteration 1.2 and Iteration 1.2 appearing underneath Release 1. The reports will contain the list of requirements and test outcomes associated with each release.

You can customize the names of the types of release usinge the `thucydides.release.types` property, e.g.

```
thucydides.release.types=milestone, release, version
```

9.4. Retrieving manual test results from Zephyr

Zephyr is a JIRA plugin that lets you store and manage manual test cases in JIRA. To get a complete picture of how well an application has been tested, we need to take into account both automated and manual test results. To let you do this with Zephyr, Thucydides provides an Adaptor to import data from Zephyr. This Adaptor reads test cases from Zephyr and converts them into manual Thucydides test outcomes, stored in the Thucydides working directory (usually `target/site/thucydides`). Thucydides can then include them in the aggregate reports when you run `mvn thucydides:aggregate`.

You can integrate with Zephyr simply by adding a dependency on the `thucydides-jira-zephyr-adaptor` in your `pom.xml` file.

```
<dependency>
        <groupId>net.thucydides.plugins.jira</groupId>
        <artifactId>thucydides-jira-zephyr-adaptor</artifactId>
        <version>0.9.220</version>
</dependency>
```

You also need to declare the adaptor in your `thucydides.properties` file:

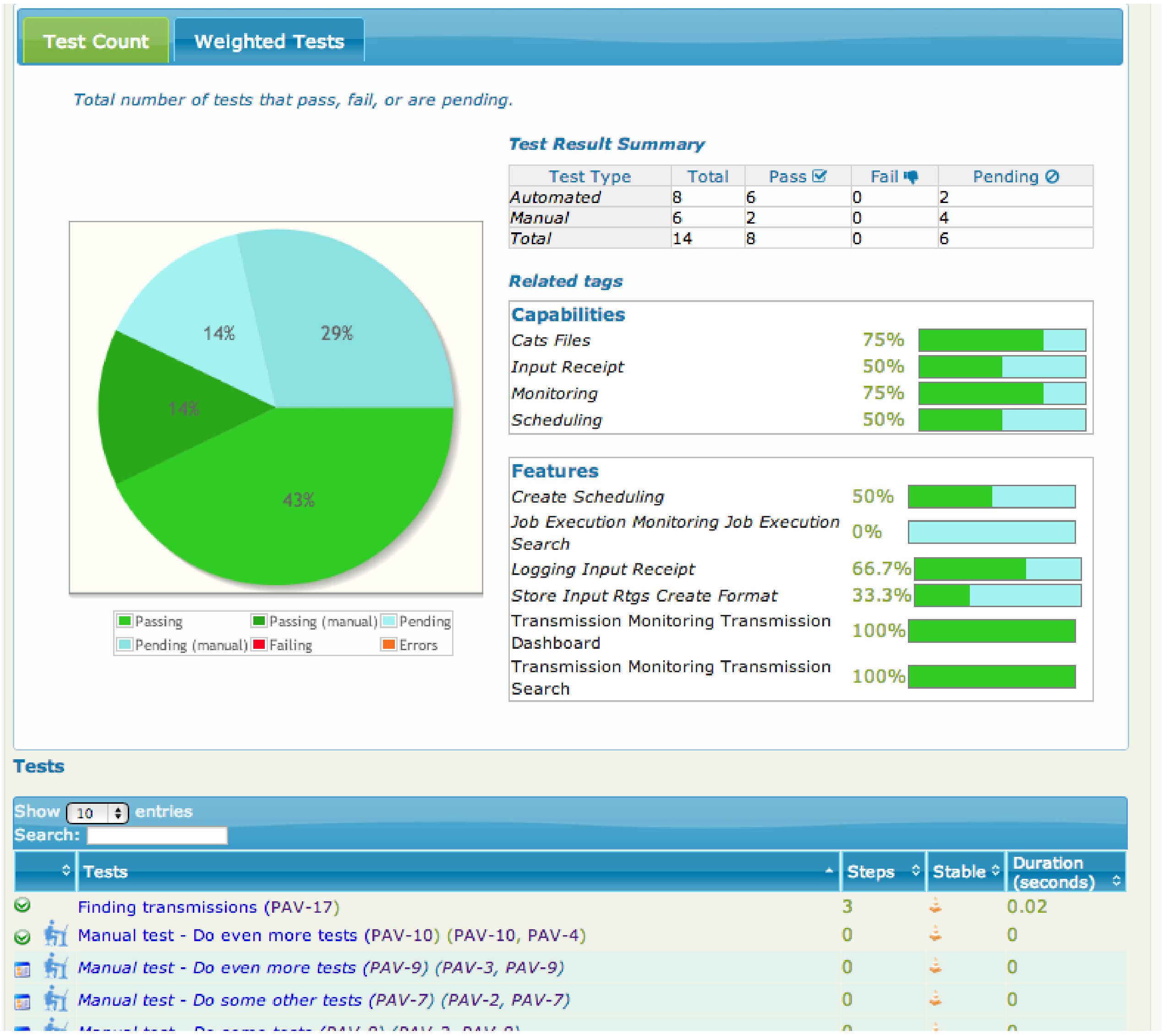
```
thucydides.adaptors.zephyr=net.thucydides.plugins.jira.adaptors.ZephyrAdaptor
```

Thucydides will now let you use it to import the test results from Zephyr, using the `thucydides:import` command. You need to provide the `import.format` parameter to tell Thucydides what adaptor you want to use:

```
$ thucydides:import -Dimport.format=zephyr
```

The manual test results will then appear in the reports, as shown here:

Figure 9.1. Manual test results imported from Zephyr



Total number of tests that pass, fail, or are pending.

Test Result Summary

Test Type	Total	Pass	Fail	Pending
Automated	8	6	0	2
Manual	6	2	0	4
Total	14	8	0	6

Related tags

Capabilities	
Cats Files	75%
Input Receipt	50%
Monitoring	75%
Scheduling	50%

Features	
Create Scheduling	50%
Job Execution Monitoring Job Execution Search	0%
Logging Input Receipt	66.7%
Store Input Rtgs Create Format	33.3%
Transmission Monitoring Transmission Dashboard	100%
Transmission Monitoring Transmission Search	100%

Tests

	Tests	Steps	Stable	Duration (seconds)
✓	Finding transmissions (PAV-17)	3		0.02
✓	Manual test - Do even more tests (PAV-10) (PAV-10, PAV-4)	0		0
	Manual test - Do even more tests (PAV-9) (PAV-3, PAV-9)	0		0
	Manual test - Do some other tests (PAV-7) (PAV-2, PAV-7)	0		0

Chapter 10. Spring Integration

If you are running your acceptance tests against an embedded web server (for example, using Jetty), it can occasionally be useful to access the service layers directly for fixture or infrastructure-related code. For example, you may have a scenario where a user action must, as a side effect, record an audit log in a table in the database. To keep your test focused and simple, you may want to call the service layer directly to check the audit logs, rather than logging on as an administrator and navigating to the audit logs screen.

Spring provides excellent support for integration tests, via the SpringJUnit4ClassRunner test runner. Unfortunately, if you are using Thucydides, this is not an option, as a test cannot have two runners at the same time. Fortunately, however, there is a solution! To inject dependencies using a Spring configuration file, you just need to include the Thucydides SpringIntegration rule in your test class. You instantiate this variable as shown here:

```
@Rule
public SpringIntegration springIntegration = new SpringIntegration();
```

Then you use the `@ContextConfiguration` annotation to define the configuration file or files to use. The you can inject dependencies as you would with an ordinary Spring integration test, using the usual Spring annotations such as @Autowired or `@Resource`. For example, suppose we are using the following Spring configuration file, called ‘config.xml’:

```
<beans>
    <bean id="widgetService" class="net.thucydides.junit.spring.WidgetService">
        <property name="name"><value>Widgets</value></property>
        <property name="quota"><value>1</value></property>
    </bean>
    <bean id="gizmoService" class="net.thucydides.junit.spring.GizmoService">
        <property name="name"><value>Gizmos</value></property>
        <property name="widgetService"><ref bean="widgetService" /></property>
    </bean>
</beans>
```

We can use this configuration file to inject dependencies as shown here:

```
@RunWith(ThucydidesRunner.class)
@ContextConfiguration(locations = "/config.xml")
public class WhenInjectingSpringDependencies {

    @Managed
    WebDriver driver;

    @ManagedPages(defaultUrl = "http://www.google.com")
    public Pages pages;

    @Rule
    public SpringIntegration springIntegration = new SpringIntegration();

    @Autowired
    public GizmoService gizmoService;

    @Test
    public void shouldInstanciateGizmoService() {
        assertThat(gizmoService, is(not(nullValue())));
    }

    @Test
    public void shouldInstanciateNestedServices() {
        assertThat(gizmoService.getWidgetService(), is(not(nullValue())));
    }
}
```

Other context-related annotations such as `@DirtiesContext` will also work as they would in a traditional Spring Integration test. Spring will create a new ApplicationContext for each test, but it will use a single ApplicationContext for all of the methods in your test. If one of your tests modifies an object in the ApplicationContext, you may want to tell Spring so that it can reset the context for the next test. You do this using the `@DirtiesContext` annotation. In the following test case, for example, the tests will fail without the `@DirtiesContext` annotation:

```
@RunWith(ThucydidesRunner.class)
@ContextConfiguration(locations = "/spring/config.xml")
public class WhenWorkingWithDirtyContexts {

    @Managed
    WebDriver driver;

    @ManagedPages(defaultUrl = "http://www.google.com")
    public Pages pages;

    @Rule
    public SpringIntegration springIntegration = new SpringIntegration();

    @Autowired
    public GizmoService gizmoService;

    @Test
    @DirtiesContext
    public void shouldNotBeAffectedByTheOtherTest() {
        assertThat(gizmoService.getName(), is("Gizmos"));
        gizmoService.setName("New Gizmos");
    }

    @Test
    @DirtiesContext
    public void shouldNotBeAffectedByTheOtherTestEither() {
        assertThat(gizmoService.getName(), is("Gizmos"));
        gizmoService.setName("New Gizmos");
    }

}
```

Chapter 11. Thucydides Report Configuration

To generate the full Thucydides reports, run mvn thucydides:aggregate. For this to work, you need to add the right plugins group to your settings.xml file, as shown here:

```
<settings>
 <pluginGroups>
   <pluginGroup>net.thucydides.maven.plugins</pluginGroup>
   ...
 </pluginGroups>
 ...
</settings>
```

You can run this in the same command as your tests by setting the maven.test.failure.ignore property to true: if you don't do this, Maven will stop if any errors occur and not proceed to the report generation:

```
$ mvn clean verify thucydides:aggregate -Dmaven.test.failure.ignore=true
```

You can also integrate the Thucydides reports into the standard Maven reports. If you are using Maven 2, just add the Thucydides Maven plugin to the reporting section:

```
<reporting>
    <plugins>
        ...
        <plugin>
            <groupId>net.thucydides.maven.plugins</groupId>
            <artifactId>maven-thucydides-plugin</artifactId>
            <version>${thucydides.version}</version>
        </plugin>
    </plugins>
</reporting>
```

If you are using Maven 3, you need to add the Maven Thucydides report to the maven-site-plugin configuration as shown here:

```
<build>
    <plugins>
        ...
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-site-plugin</artifactId>
            <version>3.0-beta-3</version>
            <configuration>
                <reportPlugins>
                    ...
                    <plugin>
                        <groupId>net.thucydides.maven.plugins</groupId>
                        <artifactId>maven-thucydides-plugin</artifactId>
                        <version>${thucydides.version}</version>
                    </plugin>
                </reportPlugins>
            </configuration>
        </plugin>
    </plugins>
```

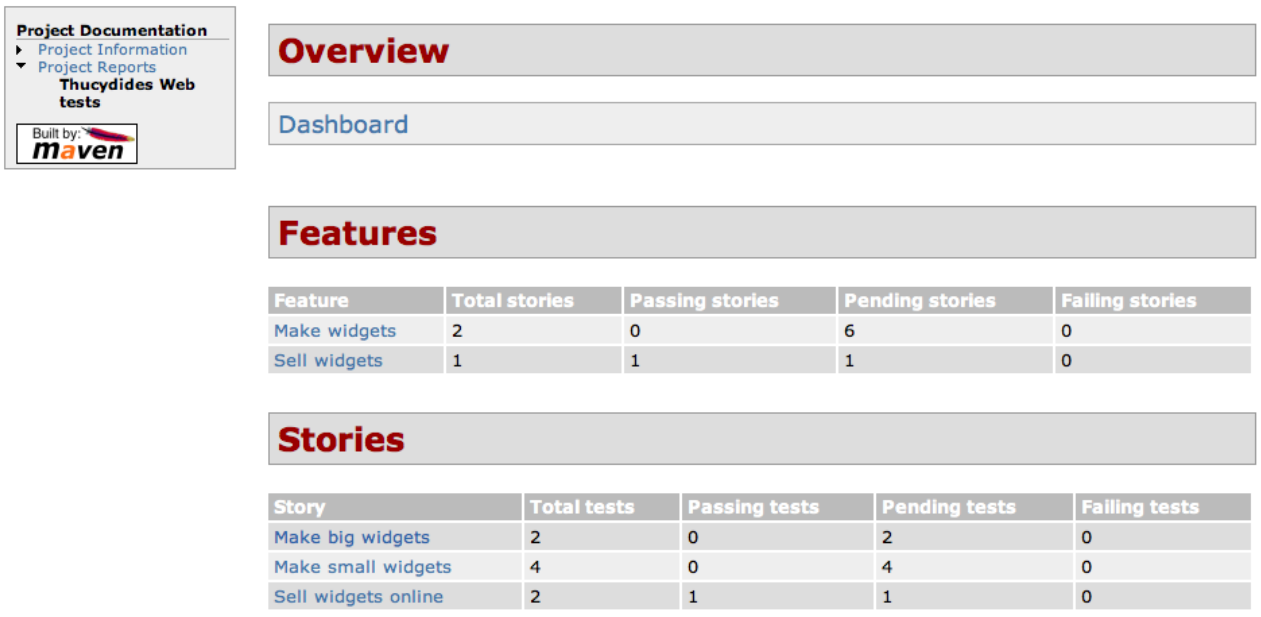
```
</build>
```

To generate this report, run the mvn site command after running mvn verify, e.g.

```
$ mvn clean verify site
```

This will produce a summary report in the generated Maven site documentation, with links to the more detailed Thucydides reports:

Figure 11.1. Thucydides test reports in the Maven site



Overview

Dashboard

Features

Feature	Total stories	Passing stories	Pending stories	Failing stories
Make widgets	2	0	6	0
Sell widgets	1	1	1	0

Stories

Story	Total tests	Passing tests	Pending tests	Failing tests
Make big widgets	2	0	2	0
Make small widgets	4	0	4	0
Sell widgets online	2	1	1	0

Chapter 12. Converting existing xUnit, specFlow and Lettuce test cases into Thucydides report

With the Thucydides maven plugin added as described in the previous section, you can also import existing xUnit, specflow [http://www.specflow.org/] and Lettuce [http://lettuce.it/index.html] test cases into a Thucydides report. For this, add the `source` directory of test cases and `format` (xunit, specflow or lettuce) parameters to the plugin configuration.

```
<reporting>
    <plugins>
        ...
        <plugin>
            <groupId>net.thucydides.maven.plugins</groupId>
            <artifactId>maven-thucydides-plugin</artifactId>
            <version>${thucydides.version}</version>
            <configuration>
              <source>src</source>
              <format>xunit</format>
            </configuration>
        </plugin>
    </plugins>
</reporting>
```

There are also a number of ways you can fine-tune the reports. You can fine-tune the tag categories that appear in the *Related Tags* section by using the `dashboard.tag.list` or `dashboard.excluded.tag.list`. The `thucydides.report.show.manual.tests` property lets you show or hide manual test results. And you can initially hide the pie chart by setting `show.pie.charts` to false (users can still display it themselves).

Chapter 13. Running Thucydides tests from the command line

You typically run Thucydides as part of the build process (either locally or on a CI server). In addition to the `webdriver.driver` option discussed about, you can also pass a number of parameters in as system properties to customize the way the tests are run. The full list is shown here:

- **properties**: Absolute path of the property file where Thucydides system property defaults are defined. Defaults to `~/thucydides.properties`
- **webdriver.driver**: What browser do you want your tests to run in: firefox, chrome or iexplorer. Basic support for iPhone and Android drivers is also available.
- **webdriver.base.url**: The default starting URL for the application, and base URL for relative paths.
- **webdriver.remote.url**: The URL to be used for remote drivers (including a selenium grid hub)
- **phantomjs.webdriver.port** What port to run PhantomJS on (used in conjunction with webdriver.remote.url to register with a Selenium hub, e.g. -Dphantomjs.webdriver=5555 -Dwebdriver.remote.url=http://localhost:4444
- **webdriver.remote.driver**: The driver to be used for remote drivers
- **webdriver.timeouts.implicitlywait**: How long webdriver waits for elements to appear by default, in milliseconds.
- **thucydides.home**: The home directory for Thucydides output and data files - by default, $USER_HOME/.thucydides
- **thucydides.outputDirectory**: Where should reports be generated.
- **thucydides.only.save.failing.screenshots** : Should Thucydides only store screenshots for failing steps? This can save disk space and speed up the tests a little. It is very useful for data-driven testing. This property is now deprecated. Use `thucydides.take.screenshots` instead.
- **thucydides.verbose.screenshots** : Should Thucydides take screenshots for every clicked button and every selected link? By default, a screenshot will be stored at the start and end of each step. If this option is set to true, Thucydides will record screenshots for any action performed on a WebElementFacade, i.e. any time you use an expression like element(…).click(), findBy(…).click() and so on. This will be overridden if the ONLY_SAVE_FAILING_SCREENSHOTS option is set to true. @Deprecated This property is still supported, but thucydides.take.screenshots provides more fine-grained control.
- **thucydides.take.screenshots** : Set this property to have more finer control on how screenshots are taken. This property can take the folowing values:
  - **FOR_EACH_ACTION** : Similar to `thucydides.verbose.screenshots`
  - **BEFORE_AND_AFTER_EACH_STEP**,
  - **AFTER_EACH_STEP**, and

  - **FOR_FAILURES** : Similar to `thucydides.only.save.failing.screenshots`

- **thucydides.verbose.steps** : Set this property to provide more detailed logging of WebElementFacade steps when tests are run.

- **thucydides.report.show.manual.tests**: Show statistics for manual tests in the test reports.

- **thucydides.report.show.releases**: Report on releases.

- **thucydides.restart.browser.frequency**: During data-driven tests, some browsers (Firefox in particular) may slow down over time due to memory leaks. To get around this, you can get Thucydides to start a new browser session at regular intervals when it executes data-driven tests.

- **thucycides.step.delay**: Pause (in ms) between each test step.

- **untrusted.certificates**: Useful if you are running Firefox tests against an HTTPS test server without a valid certificate. This will make Thucydides use a profile with the AssumeUntrustedCertificateIssuer property set.

- **thucydides.timeout**: How long should the driver wait for elements not immediately visible.

- **thucydides.browser.width** and **thucydides.browser.height**: Resize the browser to the specified dimensions, in order to take larger screenshots. This should work with Internet Explorer and Firefox, but not with Chrome.

- **thucydides.resized.image.width** : Value in pixels. If set, screenshots are resized to this size. Useful to save space.

- **thucydides.keep.unscaled.screenshots** : Set to `true` if you wish to save the original unscaled screenshots. This is set to `false` by default.

- **thucydides.store.html.source** : Set this property to `true` to save the HTML source code of the screenshot web pages. This is set to `false` by default.

- **thucydides.issue.tracker.url**: The URL used to generate links to the issue tracking system.

- **thucydides.activate.firebugs** : * Activate the Firebugs and FireFinder plugins for Firefox when running the WebDriver tests. This is useful for debugging, but is not recommended when running the tests on a build server.

- **thucydides.batch.strategy** : Defines batch strategy. Allowed values - DIVIDE_EQUALLY (default) and DIVIDE_BY_TEST_COUNT. DIVIDE_EQUALLY will simply divide the tests equally across all batches. This could be inefficient if the number of tests vary a lot between test classes. A DIVIDE_BY_TEST_COUNT strategy could be more useful in such cases as this will create batches based on number of tests.

- **thucydides.batch.count**: If batch testing is being used, this is the size of the batches being executed.

- **thucydides.batch.number** :If batch testing is being used, this is the number of the batch being run on this machine.

- **thcydides.reports.show.step.details** : Displays detailed step results in the test result tables. This property is set to false by default.

- **thucydides.use.unique.browser** : Set this to run all web tests in a single browser.

- **thucydides.locator.factory** : Set this property to override the default locator factory with another locator factory (for ex., AjaxElementLocatorFactory or DefaultElementLocatorFactory). By default, Thucydides uses a custom locator factory called DisplayedElementLocatorFactory.

- **thucydides.driver.capabilities** : A set of user-defined capabilities to be used to configure the WebDriver driver. Capabilities should be passed in as a semi-colon-separated list of key:value pairs, e.g. "build:build-1234; max-duration:300; single-window:true; tags:[tag1,tag2,tag3]"

- **thucydides.native.events** : Activate and deactivate native events for Firefox by setting this property to `true` or `false`.

- **security.enable_java** : Set this to true to enable Java support in Firefox. By default, this is set to false as it slows down the web driver.

- **thucydides.test.requirements.basedir** : The base folder of the sub-module where the jBehave stories are kept. It is assumed that this directory contains sub folders src/test/resources. If this property is set, the requirements are read from src/test/resources under this folder instead of the classpath or working directory. This property is used to support situations where your working directory is different from the requirements base dir (for example when building a multi-module project from parent pom with requirements stored inside a sub-module)

- **thucydides.proxy.http**: HTTP Proxy URL configuration for Firefox and PhantomJS

- **thucydides.proxy.http_port**: HTTP Proxy port configuration for Firefox and PhantomJS

- **thucydides.proxy.type**: HTTP Proxy type configuration for Firefox and PhantomJS

- **thucydides.proxy.user**: HTTP Proxy username configuration for Firefox and PhantomJS

- **thucydides.proxy.password**: HTTP Proxy password configuration for Firefox and PhantomJS

- **chrome.switches**: Arguments to be passed to the Chrome driver, separated by commas.

An example of using these parameters is shown here:

```
$ mvn test -Dwebdriver.driver=iexplorer -Dwebdriver.base.url=http://myapp.staging.acme.co
```

This will run the tests against the staging server using Internet Explorer.

- **webdriver.firefox.profile**: The path to the directory of the profile to use when starting firefox. This defaults to webdriver creating an anonymous profile. This is useful if you want to run the web tests using your own Firefox profile. If you are not sure about how to find the path to your profile, look here: http://support.mozilla.com/en-US/kb/Profiles. For example, to run the default profile on a Mac OS X system, you would do something like this:

```
$ mvn test -Dwebdriver.firefox.profile=/Users/johnsmart/Library/Application\ Support/Fir
```

On Windows, it would be something like:

```
C:\Projects\myproject>mvn test -Dwebdriver.firefox.profile=C:\Users\John Smart\AppData\Ro
```

- **firefox.preferences**: A semicolon separated list of Firefox configuration settings. For ex.,

```
-Dfirefox.preferences="browser.download.folderList=2;browser.download.manager.showWhenSta
```

Integer and boolean values will be converted to the corresponding types in the Firefox preferences; all other values will be treated as Strings. You can set a boolean value to true by simply specifying the property name, e.g. `-Dfirefox.preferences=app.update.silent`.

A complete reference to Firefox's configuration settings is given here [http://kb.mozillazine.org/Firefox_:_FAQs_:_About:config_Entries].

- **thucydides.history**: The directory in which build history summary data is stored for each project. Each project has it's own sub-folder within this directory. Defaults to ~./thucydides.

If you want to set default values for some of these properties for your own development environment (e.g. to always activate the Firebugs plugin on your development machine), create a file called `thucydides.properties` in your home directory (or any property file as defined by setting the system property `properties`), and set any default values here. These values will still be overridden by any values defined in the environment variables. An example is shown here:

```
thucydides.activate.firebugs = true
thucydides.browser.width = 1200
```

13.1. Providing your own Firefox profile

If you need to configure your own customized Firefox profile, you can do this by using the Thucydidies.useFirefoxProfile() method before you start your tests. For example:

```
@Before
public void setupProfile() {
    FirefoxProfile myProfile = new FirefoxProfile();
    myProfile.setPreference("network.proxy.socks_port",9999);
    myProfile.setAlwaysLoadNoFocusLib(true);
    myProfile.setEnableNativeEvents(true);
    Thucydides.useFirefoxProfile(myProfile);
}

@Test
public void aTestUsingMyCustomProfile() {...}
```

- **tags**: Comma separated list of tags. If provided, only jUnit classes and/or methods with tags in this list will be executed. For example,

```
mvn verify -Dtags="iteration:I1"

mvn verify -Dtags="color:red,flavor:strawberry"
```

- **narrative.format**: Set this property to *asciidoc* to activate using Asciidoc [http://www.methods.co.nz/asciidoc/] format in narrative text.

Chapter 14. Integrating with issue tracking systems

14.1. Basic issue tracking integration

```
http://my.jira.server/browse/MYPROJECT-{0}
```

To do this in Maven, you need to pass this system property to JUnit using the maven-surefire-plugin as shown here:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.7.1</version>
    <configuration>
        <systemPropertyVariables>
            <thucydides.issue.tracker.url>http://my.jira.server/browse/MYPROJECT-{0}</thu
        </systemPropertyVariables>
    </configuration>
</plugin>
```

Thucydides also provides special support for the Atlassian JIRA issue tracking tool. If you provide the `jira.url` system property instead of the `thucydides.issue.tracker.url`, you only need to provide the base URL for your JIRA instance, rather than the full path:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.7.1</version>
    <configuration>
        <systemPropertyVariables>
            <jira.url>http://my.jira.server</jira.url>
        </systemPropertyVariables>
    </configuration>
</plugin>
```

You need to provide the issue number. You can place this in the test title, prefixed by the # character. For easyb tests, this just means mentioning the issue number (always starting with a # character) somewhere in the scenario name. For JUnit tests, you use the @Title annotation as shown here:

```
@RunWith(ThucydidesRunner.class)
public class FixingAnIssueScenario {

    @Managed
    public WebDriver webdriver;

    @ManagedPages(defaultUrl = "http://www.mysite.com")
    public Pages pages;

    @Steps
    public SampleScenarioSteps steps;

    @Title("Shopping cart should let users add multiple articles - fixes issues #123")
    @Test
    public void shopping_cart_should_let_users_add_multiple_articles() {
        steps.add_item_to_cart("nuts");
        steps.add_item_to_cart("bolts");
        steps.cart_should_contain("nuts","bolts");
    }
}
```

Another way to specify issues in JUnit is to use the @Issue or @Issues annotations. You can use the @Issue annotation to associate an individual test with a specific issue

```
@Issue("#123")
@Test
public void shopping_cart_should_let_users_add_multiple_articles() {
    steps.add_item_to_cart("nuts");
    steps.add_item_to_cart("bolts");
    steps.cart_should_contain("nuts","bolts");
}
```

You can also place the @Issue annotation at the class level, in which case the issue will be associated with every test in the class:

```
@RunWith(ThucydidesRunner.class)
@Issue("#123")
public class FixingAnIssueScenario {

        @Managed
        public WebDriver webdriver;

        @ManagedPages(defaultUrl = "http://www.mysite.com")
        public Pages pages;

        @Steps
        public SampleScenarioSteps steps;

        @Test
        public void shopping_cart_should_let_users_add_multiple_articles() {
            steps.add_item_to_cart("nuts");
            steps.add_item_to_cart("bolts");
            steps.cart_should_contain("nuts","bolts");
        }

        @Test
        public void some_other_test() {
            ...
        }
}
```

If a test needs to be associated with several issues, you can use the @Issues annotation instead:

```
@Issues({"#123", "#456"})
@Test public void shopping_cart_should_let_users_add_multiple_articles() {
    steps.add_item_to_cart("nuts"); steps.add_item_to_cart("bolts");
        steps.cart_should_contain("nuts","bolts");
}
```

When you do this, issues will appear in the Thucydides reports with a hyperlink to the corresponding issue in your issue tracking system.

Overriding the default reports directory

By default, Thucydides generates it’s reports in the `target/site/thucydides` directory. There are a couple of ways to override this, if need be. First of all, if you are overriding the default Maven output directory, you can override the `<directory>` element in the `<build>` section of your `pom.xml` file. However, since the Maven Surefire plugin runs the tests in a forked JVM by default, you will also need to pass in the `project.build.directory` property to the unit tests by using the `<systemPropertyVariables>` configuration element, as shown here:

```
     ...
    <build>
        <directory>${basedir}/build</directory>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>2.11</version>
                <configuration>
                    <includes>
                        <include>**/*TestScenario.java</include>
                    </includes>
                    <systemPropertyVariables>
                        <project.build.directory>${project.build.directory}</project.bui
                    </systemPropertyVariables>
                </configuration>
            </plugin>
            <plugin>
                <groupId>net.thucydides.maven.plugins</groupId>
                <artifactId>maven-thucydides-plugin</artifactId>
                <version>${thucydides.version}</version>
            </plugin>
        </plugins>
    </build>
```

This will result in the Thucydides reports being generated in ‘build/site/thucydides` instead of `target/site/thucydides’.

If you only want to override the Thucydides output directory, you can use the `thucydides.outputDirectory` and `thucydides.sourceDirectory` properties, either in the `pom.xml` file, or from the commnd line. For example, the following properties will generate the Thucydides reports in the `build/thucydides-reports` directory:

```
<properties>
    <thucydides.outputDirectory>${basedir}/build/thucydides-reports</thucydides.outputDi
    <thucydides.sourceDirectory>${basedir}/build/thucydides-reports</thucydides.sourceDi
</properties>
```

Chapter 15. Using Thucydides tags

Viewing test results is certainly useful, but, from a release and deployment point of view, it is just scratching the surface. Even more interesting is the ability to view test results in terms of stories, features, behaviors, scenarios, or whatever other categorizations you find useful.

Thucydides Tags provide a very flexible mechanism for categorizing and reporting on your test results, which serve as an alternative to the Story/Feature structure described earlier. You can decide on an appropriate set of tag types (such as "feature", "behavior", "epic", "scenario", "non-functional requirement", etc.), and then assign tags of the different types to your tests. To declare a tag type, you simply use a tag of the specified type - it will then automatically appear in the Thucydides reports.

You can assign a tag manually to a test using the `@WithTag` annotation:

```
    @WithTag(name="important functionality", type = "functionality")
    class SomeTestScenarioWithTags {
        @Test
        public void a_simple_test_case() {
        }

        @WithTag(name="simple story",type = "story")
        @Test
        public void should_do_this() {
        }

        @Test
        public void should_do_that() {
        }
    }
```

Note that tags can be assigned at the test or the class level. Tag names and types are free text -

15.1. Writing a Thucydides tags plugin

Thucydides tags are easy to integrate with other applications, such as issue tracking or agile project management systems. In this section, we look at how to write a plugin that will let Thucydides automatically assign tags to your tests, based on your specific requirements.

In Thucydides, tags are arbitrary tuples of String values (name and type), represented by the `TagType` class. You can create a tag using the `TagTest.withName()` method, as shown here:

```
  TestTag specialFeatureTag = TestTag.withName("special feature").andType("feature");
```

Any feature types you provide will be displayed as separate tabs at the top of the reports screen, and will provide all of the usual aggregation and filtering features that come with the standard reports.

To define your own tags, you need to write your own tag provider, by implementing the TagProvider interface, shown below:

```
  public interface TagProvider {
      Set<TestTag> getTagsFor(final TestOutcome testOutcome);
  }
```

The unique method of this interface, `getTagsFor()`, takes a `TestOutcome` object, and returns the set of tags associated with this test outcome. The `TestOutcome` class provides a large number of fields describing the test and it's results. For example, to obtain the list of the issues specified for this test using the `getIssues()` method. The following code is an example of a tag provider that provides a list of tags based on the test's associated issues (specified by the `@Issue` and `@Issues` annotations).

```
import ch.lambdaj.function.convert.Converter;
import net.thucydides.core.model.TestOutcome;
import net.thucydides.core.model.TestTag;
import java.util.Set;
import static ch.lambdaj.Lambda.convert;

public class IssueBasedTagProvider implements TagProvider {

    public IssueBasedTagProvider() {
    }

    public Set<TestTag> getTagsFor(final TestOutcome testOutcome) {

        Set<String> issues = testOutcome.getIssues();
        return Sets.newHashSet(convert(issues, toTestTags()));
    }

    private Converter<String, String> toTestTags() {
        return new Converter<Object, TestTag>() {

            @Override
            public TestTag convert(String issue) {
                String tagName = getNameForTag(issue);
                String tagType = getTypeForTag(issue);
                return TestTag.withName(tagName).andType(tagType);
            }
        };
    }

    String getNameForTag(String issue) {...}
    String getTypeForTag(String issue) {...}
}
```

You also need to provide a service definition in the `/META-INF/services` folder on the classpath, so that Thucydides can register and use your plugin. A simple way to do this is to create a Maven project with a file called `net.thucydides.core.statistics.service.TagProvider` in the `src/resources/META-INF/sevices` folder. This file is a text file containing the fully-qualified name of your tag provider, e.g.

```
com.mycompany.thucydides.MyThucydidesTagProvider
```

Now just include the generated JAR file in your dependencies, and Thucydides will use it automatically to include your custom tags in the reports.

15.2. Bi-directional JIRA integration

A common strategy for organizations using JIRA is to represent story cards, and/or the associated acceptance criteria, as JIRA issues. It is useful to know what automated tests have been executed for a given JIRA story card, and what story is being tested for a given test.

You can add both of these features to your Thucydides project by using the `thucydides-jira-plugin`. First, you need to add the `thucydides-jira-plugin` to your Maven dependencies. The dependencies you will need (including the normal Thucydides ones) are listed here:

```
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit-dep</artifactId>
        <version>4.10</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.hamcrest</groupId>
        <artifactId>hamcrest-all</artifactId>
        <version>1.1</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>net.thucydides</groupId>
        <artifactId>thucydides-junit</artifactId>
        <version>0.6.1</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>net.thucydides.easyb</groupId>
        <artifactId>thucydides-easyb-plugin</artifactId>
        <version>0.6.1</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>net.thucydides.plugins.jira</groupId>
        <artifactId>thucydides-jira-plugin</artifactId>
        <version>0.6.1</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.codehaus.groovy</groupId>
        <artifactId>groovy-all</artifactId>
        <version>1.8.5</version>
    </dependency>
    ...
```

Note that the JIRA workflow integration needs Groovy 1.8.5 or higher to work properly.

You will also need an `slf4j` implementation, e.g. ‘slf4j-log4j12# (if you are using Log4j) or ‘logback-classic’ (if you are using LogBack) (see http://www.slf4j.org/codes.html#StaticLoggerBinder for more details). If you’re stuck, just add slf4j-simple:

```
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-simple</artifactId>
        <version>1.6.1</version>
    </dependency>
```

In Thucydides, you can refer to a JIRA issue by placing a reference to the corresponding JIRA issue number either in the name of the test (using the @Title annotation, for example), or, more simply, by using the @Issue or @Issues annotation as shown here:

```
    @RunWith(ThucydidesRunner.class)
    public class SearchByKeywordStoryTest {

        @Managed(uniqueSession = true)
        public WebDriver webdriver;

        @ManagedPages(defaultUrl = "http://www.wikipedia.com")
        public Pages pages;

        @Steps
        public EndUserSteps endUser;

        @Issue("#WIKI-1")
        @Test
        public void searching_by_unambiguious_keyword_should_display_the_corresponding_a
            endUser.is_on_the_wikipedia_home_page();
            endUser.looks_up_cats();
            endUser.should_see_article_with_title("Cat - Wikipedia, the free encyclopedia

        }
    }
```

In this example, the test will be associated with issue WIKI-1.

Alternatively, you may want to associate an issue (such as a story card) with all of the stories in a test case by placing the @Issue (or @Issues) annotation at the class level:

```
        @RunWith(ThucydidesRunner.class)
        @Issue("#WIKI-1")
        public class SearchByKeywordStoryTest {

            @Managed(uniqueSession = true)
            public WebDriver webdriver;

            @ManagedPages(defaultUrl = "http://www.wikipedia.com")
            public Pages pages;

            @Steps
            public EndUserSteps endUser;

            @Test
            public void searching_by_unambiguious_keyword_should_display_the_correspondin
                endUser.is_on_the_wikipedia_home_page();
                endUser.looks_up_cats();
                endUser.should_see_article_with_title("Cat - Wikipedia, the free encyclop

            }
        }
```

Thucydides can use these annotations to integrate with the issues in JIRA. The most simple JIRA integration involves adding links to the corresponding JIRA issues in the Thucydides reports. To activate this, you simply need to provide the **jira.url** command line option. You do however need to pass this option to JUnit using the maven-surefire-plugin, as shown here:

```
  <build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>2.10</version>
            <configuration>
                <argLine>-Xmx1024m</argLine>
                <systemPropertyVariables>
                    <jira.url>http://jira.acme.com</jira.url>
                </systemPropertyVariables>
            </configuration>
        </plugin>
        ...
```

For tighter, round-trip integration you can also use thucydides-jira-plugin. This will not only include links to JIRA in the Thucydides reports, but it will also update the corresponding JIRA issues with links to the corresponding Story page in the Thucydides reports. To set this up, add the thucydides-jira-plugin dependency to your project dependencies:

```
    <dependency>
        <groupId>net.thucydides.plugins.jira</groupId>
        <artifactId>thucydides-jira-plugin</artifactId>
        <version>0.6.1</version>
        <scope>test</scope>
    </dependency>
```

You also need to provide a username and password to connect to JIRA, and the URL where your Thucydides reports will be published (for example, on your CI server). You do using by passing in the **jira.username**, **jira.password**, **build.id** and **hucydides.public.url** system parameters. The **build.id** parameter identifies the current test run, and helps Thucydides know whether a given JIRA issue has already been updated by another test in the current test run. Thucydides lists all of the tests for a given JIRA card, along with their results, in the JIRA comment, and optionally updates the state of the JIRA issue accordingly (see below).

```
  <build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>2.10</version>
            <configuration>
                <argLine>-Xmx1024m</argLine>
                <systemPropertyVariables>
                    <jira.url>http://jira.acme.com</jira.url>
                    <jira.username>${jira.demo.user}</jira.username>
                    <jira.password>${jira.demo.password}</jira.password>
                    <build.id>${env.BUILD_ID}</build.id>
                    <thucydides.public.url>http://localhost:9000</thucydides.public.url>
                </systemPropertyVariables>
            </configuration>
        </plugin>
        ...
```

Thucydides also generates aggregate reports grouping results for stories and features. To include the JIRA links in these reports as well, you need to set the `jiraUrl` configuration option in the `maven-thucydides-plugin`, as illustrated here:

```
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-site-plugin</artifactId>
        <version>3.0-beta-3</version>
        <configuration>
            <reportPlugins>
                <plugin>
                    <groupId>net.thucydides.maven.plugins</groupId>
                    <artifactId>maven-thucydides-plugin</artifactId>
                    <version>@project.version@</version>
                    <configuration>
                         <jiraUrl>http://jira.acme.com</jiraUrl>
                     </configuration>
                 </plugin>
            </reportPlugins>
        </configuration>
    </plugin>
```

If you do not want Thucydides to update the JIRA issues for a particular run (e.g. for testing or debugging purposes), you can also set `thucydides.skip.jira.updates` to true, e.g.

```
$ mvn verify -Dthucydides.skip.jira.updates=true
```

You can also configure the plugin to update the status of JIRA issues. This is deactivated by default: to use this option, you need to set the `thucydides.jira.workflow.active` option to ‘true’, e.g.

```
$ mvn verify -Dthucydides.jira.workflow.active=true
```

The default configuration will work with the default JIRA workflow: open or in progress issues associated with successful tests will be resolved, and closed or resolved issues associated with failing tests will be reopened. If you are using a customized workflow, or want to modify the way the transitions work, you can write your own workflow configuration. Workflow configuration uses a simple Groovy DSL. The following is an example of the configuration file used for the default workflow:

```
    when 'Open', {
        'success' should: 'Resolve Issue'
    }

    when 'Reopened', {
        'success' should: 'Resolve Issue'
    }

    when 'Resolved', {
        'failure' should: 'Reopen Issue'
    }

    when 'In Progress', {
        'success' should: ['Stop Progress','Resolve Issue']
    }

    when 'Closed', {
        'failure' should: 'Reopen Issue'
    }
```

You can write your own configuration file and place it on the classpath of your test project (e.g. in the src/test/resources directory). Then you can override the default configuration by using the `thucydides.jira.workflow` property in the Maven `pom.xml` file or directly on the command line e.g.

```
$ mvn verify -Dthucydides.jira.workflow=my-workflow.groovy
```

Alternatively, you can simply create a file called `jira-workflow.groovy` and place it somewhere on your classpath. Thucydides will then use this workflow. In both these cases, you don't need to explicitly set the `thucydides.jira.workflow.active` property.

You can also integrate JIRA issues into your easyb Thucydides stories. When using the Thucydides easyb integration, you associate one or more issues with the easyb story as a whole, but not with the individual scenarios. You do this using the thucydides.tests_issue notation:

```
    using "thucydides"

    thucydides.uses_default_base_url "http://www.wikipedia.com"
    thucydides.uses_steps_from EndUserSteps
    thucydides.tests_story SearchByKeyword

    thucydides.tests_issue "#WIKI-2"

    scenario "Searching for cats", {
        given "the user is on the home page", {
            end_user.is_on_the_wikipedia_home_page()
        }
        when "the end user searches for 'cats'", {
            end_user.looks_up_cats()
        }
        then "they should see the corresponding article", {
           end_user.should_see_article_with_title("Cat - Wikipedia, the free encyclopedi
        }
    }
```

You can also associate several issues using `thucydides.tests_issues`:

```
    thucydides.tests_issue "#WIKI-2", "#WIKI-3"
```

To use easyb with Thucydides, you need to add the latest version of `thucydides-easyb-plugin` to your dependencies if it is not already there:

```
    <dependency>
        <groupId>net.thucydides.easyb</groupId>
        <artifactId>thucydides-easyb-plugin</artifactId>
        <version>0.6.1</version>
        <scope>test</scope>
    </dependency>
```

As with JUnit, you will need to pass in the proper parameters to easyb for this to work. You will also need to be using the maven-easyb-plugin version 1.4 or higher, configured to pass in the JIRA parameters as shown here:

```
    <plugin>
        <groupId>org.easyb</groupId>
        <artifactId>maven-easyb-plugin</artifactId>
        <version>1.4</version>
        <executions>
            <execution>
                <goals>
                    <goal>test</goal>
                </goals>
            </execution>
        </executions>
        <configuration>
```

```
            <storyType>html</storyType>
            <storyReport>target/easyb/easyb.html</storyReport>
            <easybTestDirectory>src/test/stories</easybTestDirectory>
            <parallel>true</parallel>
            <jvmArguments>
                <jira.url>http://jira.acme.com</jira.url>
                <jira.username>${jira.demo.user}</jira.username>
                <jira.password>${jira.demo.password}</jira.password>
                <thucydides.public.url>http://localhost:9000</thucydides.public.url>
            </systemPropertyVariables>
            </jvmArguments>
        </configuration>
    </plugin>
```

Once this is done, Thucydides will update the relevant JIRA issues automatically whenever the tests are executed.

Chapter 16. Managing screenshots

By default, Thucydides saves a screenshot for every step executed during the tests. Thucydides can be configured to control when screenshots are stored.

16.1. Configuring when screenshots are taken

The property `thucydides.take.screenshots` can be set to configure how often the sreenshots are taken. This property can take the folowing values:

- `FOR_EACH_ACTION` : Saves a screenshot at every web element action (like click(), typeAndEnter(), type(), typeAndTab() etc.).
- `BEFORE_AND_AFTER_EACH_STEP` : Saves a screeshot before and asfter every step.
- `AFTER_EACH_STEP` : Saves a screenshot after every step
- `FOR_FAILURES` : Saves screenshots only for failing steps. This can save disk space and speed up the tests a little. It is very useful for data-driven testing.

16.2. Using annotations to control screenshots

An even more granular level of control is possible using annotations. You can annotate any test or step method (or any method used by a step or test) with the `@Screenshots` annotation to override the number of screenshots taken within this step (or sub-step). Some sample uses are shown here:

```
@Step
@Screenshots(onlyOnFailures=true)
public void screenshots_will_only_be_taken_for_failures_from_here_on() {…}

@Test
@Screenshots(forEachStep=true)
public void should_take_screenshots_for_each_step_in_this_test() {…}

@Test
@Screenshots(forEachAction=true)
public void should_take_screenshots_for_each_action_in_this_test() {…}
```

16.3. Taking screenshots at any arbitrary point during a step

It is possible to have even finer control on capturing screenshots in the tests. Using the `takeScreenshot` method, you can instruct Thucydides to take a screenshot at any arbitrary point in the step irrespective of the screenshot level set using configuration or annotations.

Simply call `Thucydides.takeScreenshot()` in the step methods whenever you want a screenshot to be captured.

16.4. Increasing the size of screenshots

Sometimes the default window size is too small to display all of the application screen in the screenshots. You can increase the size of the window Thucydides opens by providing the `thucydides.browser.width` and `thucydides.browser.height` system properties. For example, to use a browser window with dimensions of 1200x1024, you could do the following:

```
$ mvn clean verify -Dthucydides.browser.width=1200 -Dthucydides.browser.height=1024
```

Typically, the width parameter is the only one you will need to specify, as the height will be determined by the contents of the browser page.

If you are running Thucydides with JUnit, you can also specify this parameter (and any of the others, for that matter) directly in your pom.xml file, in the maven-surefire-plugin configuration, e.g:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>2.7.1</version>
            <configuration>
                <argLine>-Xmx1024m</argLine>
                <systemPropertyVariables>
                    <thucydides.browser.width>1200</thucydides.browser.width>
                </systemPropertyVariables>
            </configuration>
        </plugin>
        ...
```

When the browser width is larger than 1000px, the slideshow view in the reports will expand to show the full screenshots.

Note there are some caveats with this feature. In particular, it will not work at all with Chrome, as Chrome, by design, does not support window resizing. In addition, since WebDriver uses a real browser, so the maximum size will be limited by the physical size of the browser. This limitation applies to the browser width, as the full vertical length of the screen will still be recorded in the screenshot even if it scrolls beyond a single page.

16.4.1. Screenshots and OutOfMemoryError issues

Selenium needs memory to take screenshots, particularly if the screens are large. If Selenium runs out of memory when taking screenshots, it will log an error in the test output. In this case, configure the maven-surefire-plugin to use more memory, as illustrated here:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.7.1</version>
    <configuration>
        <argLine>-Xmx1024m</argLine>
    </configuration>
</plugin>
```

16.5. Saving raw screenshots

Thucydides saves only rescaled screenshots by default. This is done to help reduce the disk space taken by reports. If you require to save the original unscaled screenshots, this default can be easily overridden by setting the property, `thucydides.keep.unscaled.screenshots` to `true`.

16.6. Saving HTML source files for screenshots

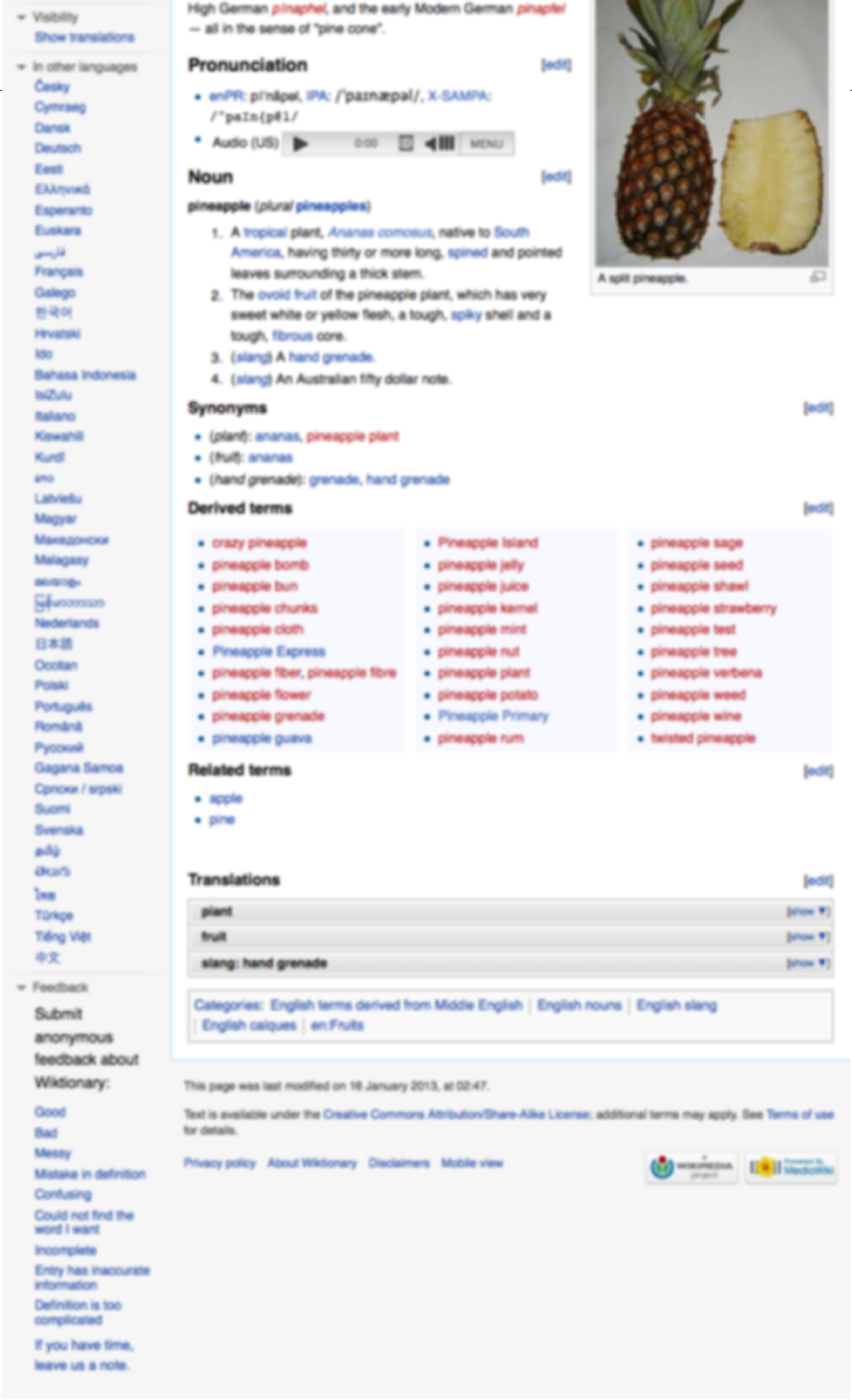
It is possible to save html source files for the screenshots by setting the property, `thucydides.store.html.source` to `true`. Html source files are not saved by default to conserve disk space.

16.7. Blurring sensitive screenshots

For security/privacy reasons, it may be required to blur sensitive screenshots in Thucydides reports. This can be done by annotating the test methods or steps with the annotation `@BlurScreenshots`. When defined on a test, all screenshots for that test will be blurred. When defined on a step, only the screenshot for that step will be blurred. @BlurredScreenshot takes a string parameter with values `LIGHT, MEDIUM` or `HEAVY` to indicate the amount of blurring. For example,

```
@Test
@BlurScreenshots("HEAVY")
public void looking_up_the_definition_of_pineapple_should_display_the_corresponding_artic
    endUser.is_the_home_page();
    endUser.looks_for("pineapple");
    endUser.should_see_definition_containing_words("A thorny fruit");
}
```

A screen at various blur levels is shown below.



High German *pinaphel*, and the early Modern German *pinapfel* — all in the sense of "pine cone".

Pronunciation [edit]

- enPR: pī'năpəl, IPA: /ˈpaɪnæpəl/, X-SAMPA: /"paIn{p@l/
- Audio (US) ▶ 0:00 MENU

Noun [edit]

pineapple (*plural* **pineapples**)

1. A tropical plant, *Ananas comosus*, native to South America, having thirty or more long, spined and pointed leaves surrounding a thick stem.
2. The ovoid fruit of the pineapple plant, which has very sweet white or yellow flesh, a tough, spiky shell and a tough, fibrous core.
3. (slang) A hand grenade.
4. (slang) An Australian fifty dollar note.

A split pineapple.

Synonyms [edit]

- (*plant*): ananas, pineapple plant
- (*fruit*): ananas
- (*hand grenade*): grenade, hand grenade

Derived terms [edit]

- crazy pineapple
- pineapple bomb
- pineapple bun
- pineapple chunks
- pineapple cloth
- Pineapple Express
- pineapple fiber, pineapple fibre
- pineapple flower
- pineapple grenade
- pineapple guava
- Pineapple Island
- pineapple jelly
- pineapple juice
- pineapple kernel
- pineapple mint
- pineapple nut
- pineapple plant
- pineapple potato
- Pineapple Primary
- pineapple rum
- pineapple sage
- pineapple seed
- pineapple shawl
- pineapple strawberry
- pineapple test
- pineapple tree
- pineapple verbena
- pineapple weed
- pineapple wine
- twisted pineapple

Related terms [edit]

- apple
- pine

Translations [edit]

plant [show ▼]

fruit [show ▼]

slang: hand grenade [show ▼]

Categories: English terms derived from Middle English | English nouns | English slang | English calques | en:Fruits

▾ Visibility

Show translations

▾ In other languages

Česky
Cymraeg
Dansk
Deutsch
Eesti
Ελληνικά
Esperanto
Euskara
فارسی
Français
Galego
한국어
Hrvatski
Ido
Bahasa Indonesia
IsiZulu
Italiano
Kiswahili
Kurdî
[illegible]
Latviešu
Magyar
Македонски
Malagasy
[illegible]
[illegible]
Nederlands
日本語
Occitan
Polski
Português
Română
Русский
Gagana Samoa
Српски / srpski
Suomi
Svenska
தமிழ்
[illegible]
ไทย
Türkçe
Tiếng Việt
中文

▾ Feedback

Submit anonymous feedback about Wiktionary:

Good
Bad
Messy
Mistake in definition
Confusing
Could not find the word I want
Incomplete
Entry has inaccurate information
Definition is too complicated
If you have time, leave us a note.

This page was last modified on 16 January 2013, at 02:47.

Text is available under the Creative Commons Attribution/Share-Alike License; additional terms may apply. See Terms of use for details.

Privacy policy About Wiktionary Disclaimers Mobile view

[illegible]

Chapter 17. Managing state between steps

Sometimes it is useful to be able to pass information between steps. For example, you might need to check that client detailed entered on a registration appears correctly on a confirmation page later on.

You can do this by passing values from one step to another, however this tends to clutter up the steps. Another approach is to use the Thucydides test session, which is essentially a hash map where you can store variables for the duration of a single test. You can obtain this session map using the `Thucydides.getCurrentSession()` static method.

As illustrated here, you can p

```
@Step
public void notes_publication_name_and_date() {
    PublicationDatesPage page = pages().get(PublicationDatesPage.class);
    String publicationName = page.getPublicationName();
    DateTime publicationDate = page.getPublicationDate();

    Thucydides.getCurrentSession().put("publicationName", publicationName);
    Thucydides.getCurrentSession().put("publicationDate", publicationDate);
}
```

Then, in a step invoked later on in the test, you can check the values stored in the session:

```
public void checks_publication_details_on_confirmation_page() {

    ConfirmationPage page = pages().get(ConfirmationPage.class);

    String selectedPublicationName = (String) Thucydides.getCurrentSession().get("publica
    DateTime selectedPublicationDate = (DateTime) Thucydides.getCurrentSession().get("pul

    assertThat(page.getPublicationDate(), is(selectedPublicationName));
    assertThat(page.getPublicationName(), is(selectedPublicationDate));

}
```

If no variable is found with the requested name, the test will fail. The test session is cleared at the start of each test.

Chapter 18. Data-Driven Testing

18.1. Data-Driven Tests in JUnit

In JUnit 4, you can use the Parameterized test runner to perform data-driven tests. In Thucydides, you use the `ThucydidesParameterizedRunner`. This runner is very similar to the JUnit Parameterized test runner, except that you use the TestData annotation to provide test data, and you can use all of the other Thucydides annotations (`@Managed`, `@ManagedPages`, `@Steps` and so on). This test runner will also generate proper Thucydides HTML and XML reports for the executed tests.

An example of a data-driven Thucydides test is shown below. In this test, we are checking that valid ages and favorite colors are accepted by the sign-on page of an (imaginary) application. To test this, we use several combinations of ages and favorite colors, specified by the testData() method. These values are represented as instance variables in the test class, and instantiated via the constructor.

```
@RunWith(ThucydidesParameterizedRunner.class)
public class WhenEnteringPersonalDetails {

    @TestData
    public static Collection<Object[]> testData() {
        return Arrays.asList(new Object[][]{
                {25, "Red"},
                {40, "Blue"},
                {36, "Green"},
        });
    }

    @Managed
    public WebDriver webdriver;

    @ManagedPages(defaultUrl = "http://www.myapp.com")
    public Pages pages;

    @Steps
    public SignupSteps signup;

    private Integer age;
    private String favoriteColor;

    public WhenEnteringPersonalDetails(Integer age, String favoriteColor) {
        this.age = age;
        this.favoriteColor = favoriteColor;
    }

    @Test
    public void valid_personal_details_should_be_accepted() {
        signup.navigateToPersonalDetailsPage();
        signup.enterPersonalDetails(age, favoriteColor);
    }
}
```

18.2. Reporting on data-driven web tests

When you generate reporting on data-driven web tests, the reports display full test outcomes and screenshots for each set of data. An overall story report is displayed for the data-driven test, which a test case for each row of test data. The test data used for each test is displayed in the report.

18.3. Running data-driven tests in parallel

Data-driven web tests can be long, especially if you need to navigate to a particular page before testing a different field value each time. In most cases, however, this is necessary, as it is unsafe to make assumptions about the state of the web page after a previous data-driven test. One effective way to speed them up, however, is to run them in parallel. You can configure `ThucydidesParameterizedRunner` tests to run in parallel by using the Concurrent annotation.

```
@RunWith(ThucydidesParameterizedRunner.class)
@Concurrent
public class WhenEnteringPersonalDetails {...
```

By default, this will run your tests concurrently, by default using two threads per CPU core. If you want to fine-tune the number of threads to be used, you can specify the *threads* annotation property.

```
@RunWith(ThucydidesParameterizedRunner.class)
@Concurrent(threads="4")
public class WhenEnteringPersonalDetails {...
```

You can also express this as a value relative to the number of available processors. For example, to run 4 threads per CPU, you could specify the following:

```
@RunWith(ThucydidesParameterizedRunner.class)
@Concurrent(threads="4x")
public class WhenEnteringPersonalDetails {...
```

18.4. Data-driven testing using CSV files

Thucydides lets you perform data-driven testing using test data in a CSV file. You store your test data in a CSV file (by default with columns separated by commas), with the first column acting as a header:

```
        NAME,AGE,PLACE OF BIRTH
        Jack Smith, 30, Smithville
        Joe Brown, 40, Brownville
        Mary Williams, 20, Williamsville
```

Next, create a test class containing properties that match the columns in the test data. Each property should be a property by the JavaBeans definition, with a matching getter and setter. The test class will typically contain one or more tests that use these properties as parameters to the test step or Page Object methods.

The class will also contain the **@UseTestDataFrom** annotation to indicate where to find the CSV file (this can either be a file on the classpath or a relative or absolute file path - putting the data set on the class path (e.g. in `src/test/resources`) makes the tests more portable).

You also use the **@RunWith** annotation as well as the other usual Thucydides annotations (`@Managed`, `@ManagedPages` and `@Steps`).

An example of such a class is shown here:

```
        @RunWith(ThucydidesParameterizedRunner.class)
        @UseTestDataFrom("test-data/simple-data.csv")
        public class SampleCSVDataDrivenScenario {

            private String name;
            private String age;
            private String placeOfBirth;

            public SampleCSVDataDrivenScenario() {
            }

            @Qualifier
            public String getQualifier() {
                return name;
            }

            @Managed
            public WebDriver webdriver;

            @ManagedPages(defaultUrl = "http://www.google.com")
            public Pages pages;

            @Steps
            public SampleScenarioSteps steps;

            @Test
            public void data_driven_test() {
                System.out.println(getName() + "/" + getAge() + "/" + getCity());
            }

            public String getName() {
                return name;
            }

            public void setName(String name) {
                this.name = name;
            }

            public String getAge() {
                return age;
            }

            public void setAge(String age) {
                this.age = age;
            }

            public String getPlaceOfBirth() {
                return placeOfBirth;
            }

            public void setPlaceOfBirth(String placeOfBirth) {
                this.placeOfBirth = placeOfBirth;
            }
        }
```

You can also specify multiple file paths separated by path separators – colon, semi-colon or comma. For example:

```
        @UseTestDataFrom("test-data/simple-data.csv,test-data-subfolder/simple-data.csv"
```

You can also configure an arbitrary directory using system property `thucydides.data.dir` and then refer to it as `$DATADIR` variable in the annotation.

```
        @UseTestDataFrom("$DATADIR/simple-data.csv")
```

Each row of test data needs to be distinguished in the generated reports. By default, Thucydides will call the `toString()` method. If you provide a public method returning a String that is annotated by the `@Qualifier` annotation, then this method will be used to distinguish data sets. It should return a value that is unique to each data set.

The test runner will create a new instance of this class for each row of data in the CSV file, assigning the properties with corresponding values in the test data. SoWhen we run this test, we will get an output like this:

```
        Jack Smith/30/Smithville
        Joe Brown/40/Brownville
        Mary Williams/20/Williamsville
```

There are a few points to note. The columns in the CSV files are converted to camel-case property names (so "NAME" becomes `name` and "PLACE OF BIRTH" becomes `placeOfBirth`). Since we are testing web applications, all of the fields should be strings.

If some of the field values contain commas, you will need to use a different separator. You can use the **separator** attribute of the **@UseTestDataFrom** annotation to specify an alternative separator. For example, the following data uses a semi-colon separator:

```
    NAME;AGE;ADDRESS
    Joe Smith; 30; 10 Main Street, Smithville
    Jack Black; 40; 1 Main Street, Smithville
    Mary Williams, 20, 2 Main Street, Williamsville
```

To run our tests against this data, we would use a test class like the following:

```
        @RunWith(ThucydidesParameterizedRunner.class)
        @UseTestDataFrom(value="test-data/simple-semicolon-data.csv", separator=';')
        public class SampleCSVDataDrivenScenario {

            private String name;
            private String age;
            private String address;

            public SampleCSVDataDrivenScenario() {
            }

            @Qualifier
            public String getQualifier() {
                return name;
            }

            @Managed
            public WebDriver webdriver;

            @ManagedPages(defaultUrl = "http://www.google.com")
            public Pages pages;

            @Steps
            public SampleScenarioSteps steps;

            @Test
            public void data_driven_test() {
                System.out.println(getName() + "/" + getAge() + "/" + getAddress());
            }

            public String getName() {
                return name;
            }

            public void setName(String name) {
                this.name = name;
            }

            public String getAge() {
                return age;
            }

            public void setAge(String age) {
                this.age = age;
            }

            public String getAddress() {
                return address;
            }

            public void setAddress(String address) {
                this.address = address;
            }
        }
```

This will generate an output like this:

```
        Joe Smith/30/10 Main Street, Smithville
        Jack Black/40/1 Main Street, Smithville
        Mary Williams/20/2 Main Street, Williamsville
```

Excel support will be added in a future version. However if you store your test data in CSV form, it becomes easier to keep track of changes to test data in your version control system.

18.5. Using data-driven testing for individual steps

Sometimes you want to use data-driven testing at the step level, rather than at the test level. For example, you might want to navigate to a particular screen in the application, and then try many combinations of data, or loop over a sequence of steps with data from a CSV file. This avoids having to reopen the browser for each row of data.

You can do this by adding property values to your Step files. Consider the following steps file:

```
        public class SampleDataDrivenSteps extends ScenarioSteps {

            public SampleDataDrivenSteps(Pages pages) {
                super(pages);
            }

            private String name;
            private String age;
            private String address;

            public void setName(String name) {
                this.name = name;
            }

            public void setAge(String age) {
                this.age = age;
            }

            public void setAddress(String address) {
                this.address = address;
            }

            @StepGroup
            public void enter_new_user_details() {
                enter_name_and_age(name, age);
                enter_address(address);
            }

            @Step
            public void enter_address(String address) {
                ...
            }

            @Step
            public void enter_name_and_age(String name, String age) {
                ...
            }

                @Step
                public void navigate_to_user_accounts_page() {
                        ...
                }
        }
```

The `enter_personal_details` step group uses the step fields to run the `enter_name_and_age` and `enter_address steps`. We want to fetch this data from a CSV file, and loop through the **enter_personal_details** step for each row of data.

We do this using the `withTestDataFrom()` method of the **StepData** class:

```
        import net.thucydides.core.annotations.ManagedPages;
        import net.thucydides.core.annotations.Steps;
        import net.thucydides.core.pages.Pages;
        import net.thucydides.junit.annotations.Managed;
        import net.thucydides.junit.runners.ThucydidesRunner;
        import org.junit.Test;
        import org.junit.runner.RunWith;
        import org.openqa.selenium.WebDriver;

        import static net.thucydides.core.steps.StepData.withTestDataFrom;
        @RunWith(ThucydidesRunner.class)
        public class SamplePassingScenarioWithTestSpecificData {

            @Managed
            public WebDriver webdriver;

            @ManagedPages(defaultUrl = "http://www.google.com")
            public Pages pages;

            @Steps
            public SampleDataDrivenSteps steps;


            @Test
            public void happy_day_scenario() throws Throwable {
                        steps.navigate_to_user_accounts_page();
                withTestDataFrom("test-data/simple-data.csv").run(steps).enter_new_user_
            }
        }
```

This will call the `data_driven_test_step()` multiple times, each time injecting data from the `test-data/simple-data.csv` file into the step.

You also can use as many data files as you want, even in the same test. You can also use the same data file for more than one test step. Remember only the properties that match columns in the CSV file will be instantiated - the others will be ignored:

```
        @RunWith(ThucydidesRunner.class)
        public class SamplePassingScenarioWithTestSpecificData {

            @Managed
            public WebDriver webdriver;

            @ManagedPages(defaultUrl = "http://www.google.com")
            public Pages pages;

            @Steps
            public SampleDataDrivenSteps steps;

            @Steps
            public DifferentDataDrivenSteps different_steps;


            @Test
            public void happy_day_scenario() throws Throwable {
                        steps.navigate_to_user_accounts_page();

                withTestDataFrom("test-data/simple-data.csv").run(steps).enter_new_user_

                withTestDataFrom("test-data/some_other-data.csv").run(different_steps).e
            }
        }
```

By the way we need to use `ThucydidesRunner` for the test cases instead of `ThucydidesParameterizedRunner`.

Note that, as a shortcut, you can dispense with the setter methods and just declare the relevant fields public. So the Steps class shown above could be rewritten like this:

```
        public class SampleDataDrivenSteps extends ScenarioSteps {

            public SampleDataDrivenSteps(Pages pages) {
                super(pages);
            }

            public String name;
            public String age;
            public String address;

            @StepGroup
            public void enter_new_user_details() {
                enter_name_and_age(name, age);
                enter_address(address);
            }

            @Step
            public void enter_address(String address) {
                ...
            }

            @Step
            public void enter_name_and_age(String name, String age) {
                ...
            }

                @Step
                public void navigate_to_user_accounts_page() {
                        ...
                }
        }
```

Chapter 19. Running Thucydides tests in parallel batches

Web tests make good candidates for concurrent testing, in theory at least, but the implementation can be tricky. For example, although it is easy enough to configure both JUnit and easyb to run tests in parallel, running several webdriver instances of Firefox in parallel on the same display, for example, tends to become unreliable.

The natural solution in this case is to split the web tests into smaller batches, and to run each batch on a different machine and/or on a different virtual display. When each batch has finished, the results can be retrieved and aggregated into the final test reports.

However splitting tests into batches by hand tends to be tedious and unreliable – it is easy to forget to add a new test to a batch, for example, or have unevenly-distributed batches.

The latest version of Thucydides lets you do this automatically, by splitting your test cases evenly into batches of a given size. In practice, you run a build job for each batch. You need to specify two parameters when you run each build: the total number of batches being run (`thucydides.batch.count`), and the number of the batch being run in this build (`thucydides.batch.number`).

For example, the following will divide the test cases into 3 batches (`thucydides.batch.count`), and only run the first test in each batch (`thucydides.batch.number`):

```
mvn verify -Dthucydides.batch.count=3 -Dthucydides.batch.number=1
```

This will only work with the JUnit integration. However this feature is also supported in easyb (as of easyb version 1.5), though using different parameters. When using the Thucydides easyb integration, you also need to provide the equivalent options for easyb:

```
mvn verify -Deasyb.batch.count=3 -Deasyb.batch.number=1
```

If you have both easyb and JUnit Thucydides tests, you will need to specify both options.

19.1. Test count based batch strategy

By default, test cases are divided equally among batches. This could be inefficient if some test cases have more tests than others. In such situations, a different batch strategy, `DIVIDE_BY_TEST_COUNT` can be defined using the system property `thucydides.batch.strategy`. This strategy will evenly distribute test cases across batches based on number of test methods in each test case.

```
mvn verify -Dthucydides,batch.strategy=DIVIDE_BY_TEST_COUNT -Dthucydides.batch.count=3 -
```

Chapter 20. Experimental features

20.1. Integration with FluentLineum

You can use FluentLenium‘s [https://github.com/FluentLenium/FluentLenium] fluent API with Thucydides. The best way to use FluentLenium within Thucydides is to use ThucydidesFluentAdapter which is available in PageObject. Here’s an example of the same PageObject written in the traditional style and with FluentLenium.

```
import ch.lambdaj.function.convert.Converter;
import net.thucydides.core.annotations.DefaultUrl;
import org.openqa.selenium.By;
import org.openqa.selenium.Keys;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;

import net.thucydides.core.pages.PageObject;

import java.util.List;

import static ch.lambdaj.Lambda.convert;

@DefaultUrl("http://en.wiktionary.org/wiki/Wiktionary:Main_Page")
public class DictionaryPage extends PageObject {

    @FindBy(name="search")
    private WebElement searchTerms;

    @FindBy(name="go")
    private WebElement lookupButton;

    public DictionaryPage(WebDriver driver) {
        super(driver);
    }

    public void enter_keywords(String keyword) {
        element(searchTerms).type(keyword);
    }

    public void lookup_terms() {
        element(lookupButton).click();
    }

    public List getDefinitions() {
        WebElement definitionList = getDriver().findElement(By.tagName("ol"));
        List results = definitionList.findElements(By.tagName("li"));
        return convert(results, toStrings());
    }

    private Converter<WebElement, String> toStrings() {
        return new Converter<WebElement, String>() {
            public String convert(WebElement from) {
                return from.getText();
            }
        };
    }
}
```

and with FluentLineum

```
import ch.lambdaj.function.convert.Converter;
import net.thucydides.core.annotations.DefaultUrl;
import net.thucydides.core.pages.PageObject;
import org.fluentlenium.core.domain.FluentList;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;

import java.util.List;

import static ch.lambdaj.Lambda.convert;
import static org.fluentlenium.core.filter.FilterConstructor.withName;

@DefaultUrl("http://en.wiktionary.org/wiki/Wiktionary:Main_Page")
public class FluentDictionaryPage extends PageObject {

    public FluentDictionaryPage(WebDriver driver) {
        super(driver);
    }

    public void enter_keywords(String keyword) {
        fluent().fill("input", withName("search")).with(keyword);
    }

    public void lookup_terms() {
        fluent().click("input", withName("go"));
    }

    public List getDefinitions() {
        FluentList results = fluent().findFirst("ol").find("li");
        return results.getTexts();
    }
}
```

20.2. Shortcut for the element() method

Another new experimental feature introduces the ability to replace the commonly-used element() method with ‘$’, as illustrated in the following examples:

```
    ...
    @FindBy(name="search")
    private WebElement searchTerms;

    @FindBy(name="go")
    private WebElement lookupButton;

    public DictionaryPage(WebDriver driver) {
        super(driver);
    }

    public void enter_keywords(String keyword) {
        $(searchTerms).type(keyword);
    }

    public void lookup_terms() {
        $(lookupButton).click();
    }

    public void click_on_article(int articleNumber) {
        $("//section[@id='searchResults']/article[" + articleNumber + "]//a").click();
    }

    public String getHeading() {
        return $("section>h1").getText()
    }
}
```

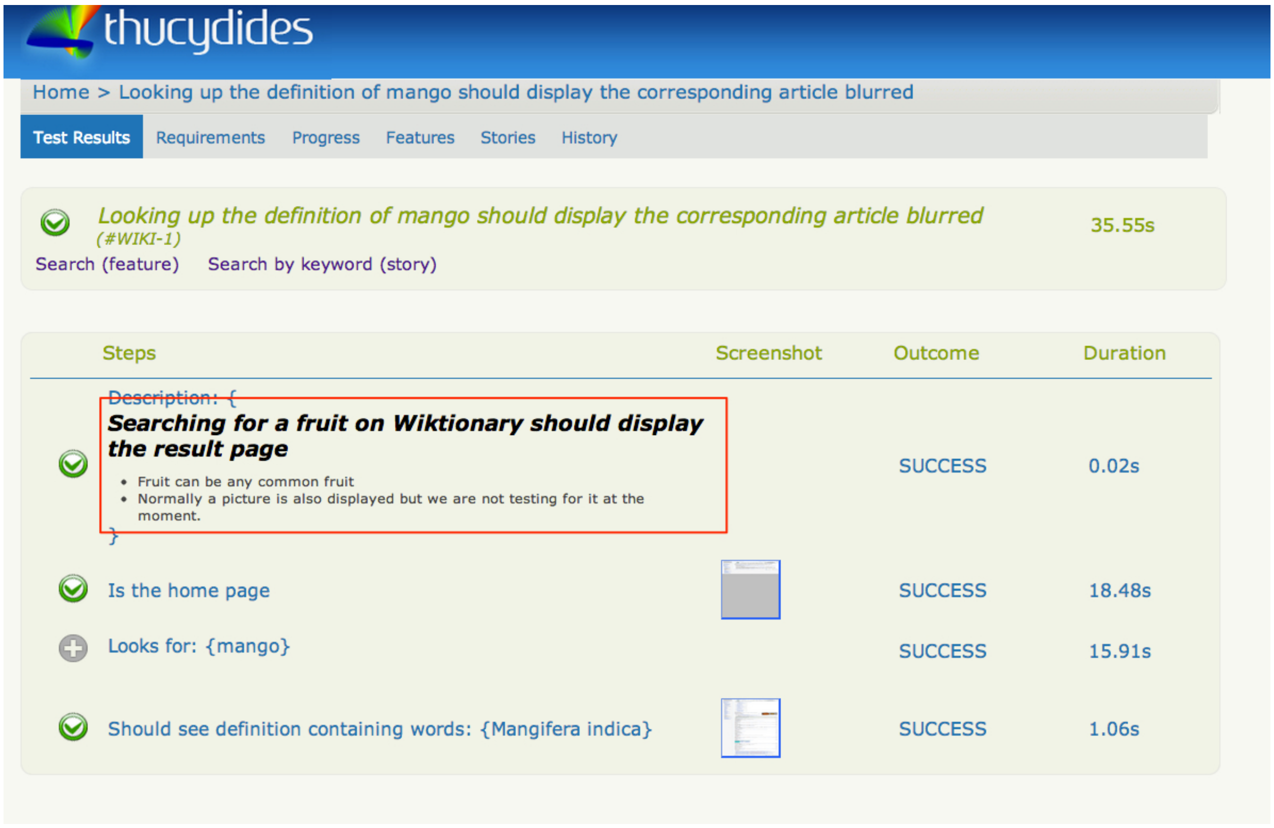
20.3. Retrying failed tests

Sometimes it is required to retry a failed test. This can be achieved by setting the system property `max.retries` to the number of times you want failed tests to be retried.

20.4. Using Step methods to document test cases

Methods in the Step library can be used to provide additional documentation for the test scenarios in Thucydides reports. You can pass valid HTML text as parameter to @Step methods in the step library. This will show up as formatted text in the reports on the step details page. The following screenshot demonstrates this.

Figure 20.1. HTML formatted text, if passed to a step method will be displayed as shown. This can be useful for annotating or documenting the tests with helpful information.


thucydides
Home > Looking up the definition of mango should display the corresponding article blurred
Test Results
Requirements
Progress
Features
Stories
History

Looking up the definition of mango should display the corresponding article blurred 35.55s
(#WIKI-1)
Search (feature) Search by keyword (story)

Steps	Screenshot	Outcome	Duration
~~Description: {~~ ***Searching for a fruit on Wiktionary should display the result page*** • Fruit can be any common fruit • Normally a picture is also displayed but we are not testing for it at the moment. }		SUCCESS	0.02s
Is the home page		SUCCESS	18.48s
Looks for: {mango}		SUCCESS	15.91s
Should see definition containing words: {Mangifera indica}		SUCCESS	1.06s

This is achieved by creating a dummy @Step method called description that takes a String parameter. At runtime, the tests supply this method with formatted html text as parameter.

```
...
@Step
public void description(String html) {
    //do nothing
}

public void about(String description, String...remarks) {
    String html =
    "<h2 style=\"font-style:italic;color:black\">" + description + "</h2>" +
    "<div><p>Remarks:</p>" +
    "<ul style=\"margin-left:5%; font-weight:200; color:#434343; font-size:10px;\">";

    for (String li : remarks) html += "<li>" + li + "</li>";

    html += "<ul></div>";

    description(html);
}
...
```

Chapter 21. Further Reading

Articles

- Dr. Dobb’s Journal. Project of the Month: Thucydides [http://drdobbs.com/open-source/232300277/], December. 2011.
- JavaWorld. Acceptance test driven development for web applications [http://www.javaworld.com/javaworld/jw-08-2011/110823-atdd-for-web-apps.html], August. 2011.
- JavaWorld. Selenium 2 and Thucydides for ATDD [http://www.javaworld.com/javaworld/jw-10-2011/111018-thucydides-for-atdd.html], August. 2011.

Books

- Bdd in Action, BDD In Action - Manning - Summer, 2014 [http://manning.com/smart]