Thucydides: manuel de référence

John Ferguson Smart

Thucydides: manuel de référence

John Ferguson Smart

Table of Contents

List of Figures

Copyright

Copyright © 2011-2013 John Ferguson Smart.

Version en ligne publiée par Wakaleo Consulting.

Ce travail est sous licence Creative Commons Attribution - Pas d’utilisation Commerciale - Pas de Modification 3.0 United States Pour plus d’information sur cette licence, voir creativecommons.org/licenses/by-nc-nd/3.0/fr/ [http://creativecommons.org/licenses/by-nc-nd/3.0/fr/].

Java™ et toutes les marques et logos basés sur Java sont des marques commerciales ou marques commerciales enregistrées de Sun Microsystems, Inc., aux États-Unis et autres pays.

Éclipse™ est une marque commerciale de Eclipse Foundation, Inc., aux États-Unis et autres pays.

Apache et le logo plume Apache sont des marques commerciales de la Apache Software Foundation.

Plusieurs des appellations utilisées par les fabricants et les vendeurs pour désigner leurs produits sont revendiqués comme marques commerciales. Quand ces appellations apparaissent dans ce livre, et quand Wakaleo Consulting était au fait de cette revendication de marque commerciale, les appellations ont été affichées en majuscules ou avec des initiales en majuscule.

Bien que toutes les précautions ont été prises dans la préparation de ce livre, l'éditeur et les auteurs déclinent toute responsabilité pour les erreurs ou omissions, ou pour les dommages résultant de l’utilisation des informations fournies ci-après.

Chapter 1. Introduction à Thucydides

Thucydides (Tou-saille-didz) est un outil conçu pour faciliter l'écriture des tests de recette et de non régression automatisés. Il offre des fonctionnalités pour faciliter l’organisation et la structuration de vos tests de recette en les reliant avec vos histoires utilisateur ou les fonctionnalités qu’ils testent. Quand les tests sont exécutés, Thucydides génère une documentation illustrée qui décrit comment l’application est utilisée en s’appuyant sur les histoires décrites par les tests.

Thucydides apporte une prise en charge robuste des tests web automatisés utilisant Selenium 2, bien qu’il puisse être utilisé efficacement pour des tests non web.

Thucydides était un historien grec connu pour ses analyses astucieuses et qui consignait rigoureusement les événements dont il était témoin ou auxquels il participait lui-même. De la même manière, le framework Thucydides observe et analyse vos tests de recette et consigne de façon détaillée leur exécution.

Chapter 2. Concepts fondamentaux du test de recette et de non régression

Pour tirer le meilleur parti de Thucydides, il est utile de comprendre quelques principes de base sous-jacents au développement dirigé par les tests de recette (Acceptance Test Driven Development). Thucydides est communément utilisé à la fois pour les tests de recette automatisés et les tests de non régression et les principes abordés ici s'appliquent aux deux, à quelques variations mineures près.

Acceptance Test Driven Development, ou ATDD, est une forme avancée du développement dirigé par les tests (Test Driven Development ou TDD) dans lequel les critères de recette automatisées - définis en collaboration avec les utilisateurs - pilotent et orientent le processus de développement. Ceci aide à s'assurer que tout le monde comprend quelles sont les fonctionnalités en cours de développement.

Un des points importants au sujet de l'ATDD est l'idée de la "Spécification par l'exemple". Spécification par l'exemple fait référence à l'utilisation d'exemples relativement concrets pour illustrer comment un système doit fonctionner, par opposition à des spécifications écrites plus formellement.

Prenons un exemple. Dans de nombreux projets, les exigences sont exprimées à l'aide de simples histoires telles que ce qui suit:

```
De façon à obtenir de l'argent pour acheter une nouvelle voiture
En tant que propriétaire d'une voiture
Je veux vendre ma vieille voiture sur Internet
```

Si nous développons un site web de vente de voitures qui aide les gens à atteindre cet objectif, nous devrions typiquement définir une série de critères de recette pour donner corps à notre histoire. Par exemple, nous pourrions avoir les critères suivants dans notre liste de critères de recette:

- Le propriétaire de voiture peut publier une annonce de vente en ligne standard
- Le propriétaire de voiture peut publier une annonce de vente en ligne premium
- L'annonce voiture peut afficher la marque, le modèle et l'année de la voiture

et ainsi de suite.

Un testeur planifiant les tests pour ces critères de recette peut concevoir un plan de tests décrivant la façon dont il entend tester ces critères de recette. Pour le premier critère, il peut commencer par un plan de haut niveau tel que ce qui suit:

- Aller à la section des annonces voiture et choisir de poster une annonce voiture standard
- Saisir les informations relatives à la voiture
- Choisir les options de publication
- Pré visualiser l'annonce
- Saisir les informations de paiement

- Visualiser la confirmation de l’annonce

Chacune de ces étapes peut nécessiter d'être décomposée en étapes plus fines.

- *Saisir les informations relatives à la voiture*
  - Saisir le fabriquant, le modèle et l’année de la voiture
  - Choisir les options
  - Ajouter des photos
  - Saisir une description

Ces étapes sont souvent étoffées avec plus de détails concrets:

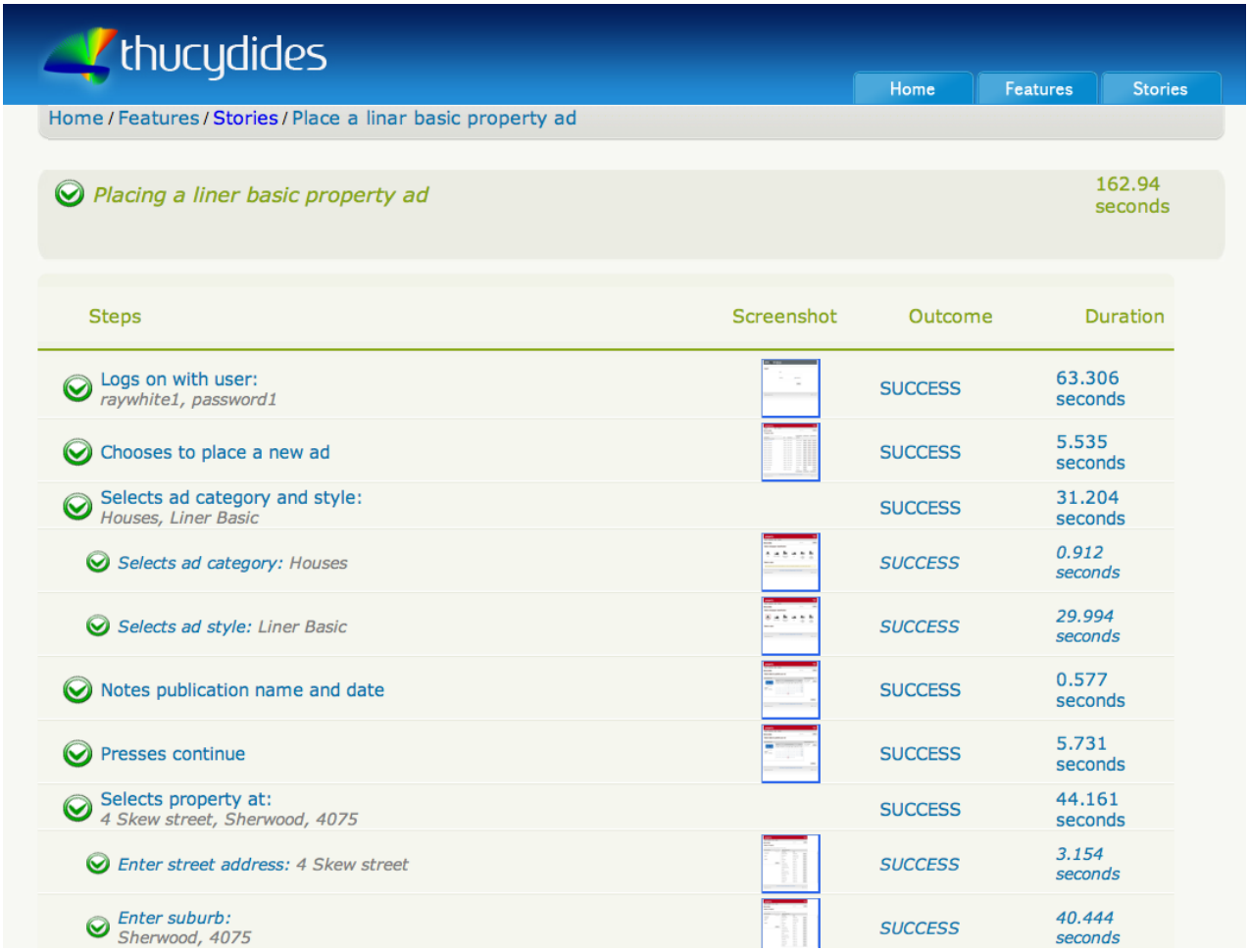
- *Saisir les informations relatives à la voiture*
  - Créer une annonce pour une Mitsubishi Pajero de 2006
  - Ajouter Air conditionné et lecteur CD
  - Ajouter trois photos
  - Saisir une description

Selon nos objectifs, les tests de non régression peuvent être définis comme des tests de bout en bout qui assurent que l’application se comporte comme attendu et qui continuera à se comporter comme attendu dans les prochaines livraisons. Autant les tests de recette sont définis très tôt dans le processus, avant le début du développement, autant les tests de non régression implique l’existence du système. A par cela, les étapes nécessaires à la définition et à l’automatisation des tests sont très semblables.

Maintenant différentes parties prenantes du projet seront impliquées à différents niveaux de détail. Certains, tels que les gestionnaires de projet et les gestionnaires en général, ne seront intéressés que par les fonctionnalités qui fonctionnent et par celles qui doivent être réalisées. D’autres, tels que les analystes métier et les qualiticiens, seront intéressés par la façon détaillée dont les scénarios de recette sont implémentés, éventuellement en allant jusqu’aux écrans.

Thucydides vous aide à structurer vos tests de recette automatisés en étapes et sous-étapes comme illustré ci-dessous. Ceci a tendance à rendre les tests plus clairs, plus flexibles et plus faciles à maintenir. De plus, quand les tests sont exécutés, Thucydides produit des rapports illustrés en style narratif comme dans Figure 2.1, “Un rapport de test généré par Thucydides”.

Figure 2.1. Un rapport de test généré par Thucydides



Quand vient le moment d'implémenter les tests eux-mêmes, Thucydides apporte également de nombreuses fonctionnalités pour rendre ceci plus facile, plus rapide et plus propre de façon à écrire des tests clairs et maintenables. Ceci est particulièrement vrai pour les tests web automatisés utilisant Selenium 2, mais Thucydides répond aussi bien aux besoins des tests non-web. Thucydides fonctionne actuellement avec JUnit et easyb - l'intégration avec d'autres framework BDD est en cours.

Chapter 3. Débuter avec Thucydides

3.1. Créer un nouveau projet Thucydides

La façon la plus simple de démarrer un nouveau projet Thucydides est d'utiliser l'archétype Maven. Trois archétypes sont actuellement disponibles: un pour utiliser Thucydides avec JUnit, un autre si vous voulez également écrire vos tests de recette (ou une partie d'entre eux) en utilisant easyb [http://www.easyb.org/] et enfin un supplémentaire pour vous aider à écrire vos tests de recette avec jBehave. Dans cette section, nous créerons un nouveau projet Thucydides en utilisant l'archétype Thucydides, puis nous aborderons les fonctionnalités essentielles de ce projet.

En ligne de commandes, vous pouvez exécuter mvn archetype:generate puis choisir l'archétype net.thucydides.thucydides-easyb-archetype dans la liste des archétypes proposés. Ou bien, vous pouvez utiliser votre IDE préféré pour générer un nouveau projet Maven en utilisant un archétype.

```
$ 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] ------------------------------------------------------------------------
```

Ceci va créer un projet Thucydides simple complété d'un objet page, d'une bibliothèque d'étapes et de deux cas de tests, l'un utilisant JUnit et l'autre easyb. Les tests effectifs sont exécutés sur le dictionnaire en ligne Wiktionary.org. Avant d'aller plus loin, faisons un tour dans notre projet. Mais avant, nous devons ajouter `net.thucydides.maven.plugins` à nos groupes de plugin dans notre fichier `settings.xml`:

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

Ceci va nous permettre d'appeler le plugin Maven Thucydides à partir de la ligne de commandes sous la forme abrégée montrée ici. Maintenant, allons dans le répertoire généré pour notre projet, lançons les tests et générons le rapport:

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

Ceci doit lancer des tests web et générer un rapport dans le répertoire `target/site/thucydides` (ouvrez le fichier `index.html`).

Si vous descendez dans les rapports de tests individuels, vous verrez un récit illustré pour chaque test comme celui montré ici Figure 2.1, “Un rapport de test généré par Thucydides”

Maintenant les détails. La structure projet est montrée ici:

```
+ 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
```

Ce projet est conçu pour fournir un point de départ à vos tests de recette Thucydides et pour illustrer certaines des fonctionnalités de base. Les tests sont fournis dans deux styles: *easyb* et *JUnit*. easyb est une bibliothèque BDD (Behaviour Driven Development) reposant sur Groovy qui fonctionne bien pour ce type de tests. L’histoire exemple easyb se trouve dans le fichier `LookupADefinition.story` et ressemble à quelque chose comme ça:

```
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"
    }
}
```

Un coup d’oeil rapide à cette histoire montre qu’elle se rapporte à un utilisateur cherchant la définition du mot *apple*. Seul le "quoi" est exprimé à ce niveau - les détails sont masqués dans les étapes du test, voire plus profondément, dans les objets page.

Si vous préférez des tests en pur Java, l'équivalent JUnit se trouve dans le fichier `LookupADefinitionStoryTest.java`:

```
@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");

    }
}
```

Comme vous pouvez le voir, c’est un petit peu plus technique mais cela reste à très haut niveau.

Les bibliothèques d'étapes contiennent l’implémentation de chacune des étapes utilisées dans les tests de haut niveau. Pour des tests complexes, ces étapes peuvent à leur tour faire appel à d’autres étapes. La bibliothèque d'étapes utilisée dans cet exemple peut être trouvée dans `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();
    }
}
```

Les objets Page sont un moyen d'encapsuler les détails d'implémentation d'une page donnée. Selenium 2 possède un particulièrement bon support pour les objets page et ThucydidesRunner en tire parti. L'objet page d'exemple se trouve dans la classe HomePage.java:

```
@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();
        }
    }
}
```

La pièce finale du puzzle est la classe `Application.java` qui est un moyen de représenter la structure de vos exigences sous forme Java de telle sorte que vos tests easyb et JUnit puissent être reliés aux exigences qu'ils testent:

```
        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 {}
            }
        }
```

C’est ce qui permet à Thucydides de générer un rapport global concernant les fonctionnalités et les histoires.

Dans les sections suivantes, nous verrons plus en détails les différents aspects de l'écriture des tests automatiques avec Thucydides.

3.2. Configurer des capacités personnalisées pour le pilote web

Vous pouvez configurer des capacités personnalisées pour le pilote web en passant une liste de capacités séparées par des points virgules dans la propriété `thucydides.driver.capabilities`. Par exemple,

```
"build:build-1234; max-duration:300; single-window:true; tags:[étiquette1,étiquette2,éti
```

Chapter 4. Écrire des tests de recette avec Thucydides

Dans cette section, nous examinons plus en détail les choses dont nous avons besoin pour écrire nos tests de recette ou de non régression en utilisant Thucydides. Nous allons également présenter une approche générale pour écrire des tests de recette web qui a bien fonctionné pour nous par le passé.

1. Définir et organiser les exigences ou les histoires utilisateur que vous avez besoin de tester
2. Écrire des tests haut niveau en attente pour les critères de recette
3. Choisir un test à implémenter et le diviser en petites étapes de haut niveau (typiquement entre 3 et 7)
4. Implémenter ces étapes, soit en les divisant en d'autres étapes, soit en accédant à des objets page.
5. Implémenter toutes les méthodes d'objet Page que vous avez découvertes.

Note

Ces étapes ne doivent pas être vues comme une approche linéaire ou en cascade. En effet, le processus est habituellement assez incrémental avec des exigences ajoutées à la classe Application quand elles sont nécessaires, et des tests en attente utilisés pour définir les tests avant qu'ils ne soient complétés.

4.1. Organiser vos exigences

Pour tirer le meilleur de vos tests automatisés dans Thucydides, vous devez dire à Thucydides quelles fonctionnalités de votre application vous testez dans chaque test. Bien que cette étape soit facultative, elle est fortement recommandée.

La version actuelle de Thucydides utilise une organisation simple à trois niveaux pour structurer les tests de recette en morceaux plus facilement gérables. Au plus haut niveau, une application est divisée en *fonctionnalités* (*feature*), qui est une fonction de haut niveau ou un groupe de fonctions en rapport. Une fonctionnalité contient un certain nombre d'histoires (*stories*, correspondant aux histoires utilisateur, cas d'utilisation, etc.). Chaque histoire est validée par un certain nombre d'exemples, ou critères d'acceptation, qui sont automatisés sous forme de tests web (parfois appelés scénarios). Chaque test, à son tour, est implémenté en utilisant un certain nombre d'étapes.

Bien sûr cette structure et ces termes sont principalement des commodités pour permettre une vision haut niveau de vos tests de recette. Cependant, ce type d'abstraction à 3 niveaux semble être assez commune.

Dans la version actuelle de Thucydides, vous définissez cette structure à l'intérieur du code de test sous forme de classes Java (très légères) [1]. Ceci facilite la refactorisation et le renommage des histoires utilisateur et des fonctionnalités à l'intérieur des tests, et donne un point de référence central dans la suite de tests illustrant quelles fonctionnalités sont en cours de test. Un exemple simple est

[1] Les futures versions de Thucydides offriront d'autres façons de définir vos exigences utilisateur.

donné ici. La classe Application est simplement un moyen pratique de regrouper dans un seul fichier les fonctionnalités et les histoires utilisateur. Les fonctionnalités sont repérées avec l'annotation @Feature. Les histoires utilisateur sont déclarées sous forme de classes internes à l'intérieur d'une classe @Feature.

```
public class Application {

    @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. Définir des tests de haut niveau

Il y a deux approches pour automatiser les tests de recette ou de non régression avec Thucydides. Les deux impliquent d'implémenter les tests sous forme d'une séquence d'étapes de très haut niveau puis d'implémenter ces étapes en descendant dans les détails jusqu'à atteindre les objets page. La différence réside dans le langage utilisé pour implémenter les tests de haut niveau. Des outils tels que easyb se concentrent davantage sur la communication avec les non-développeurs et permettent aux tests de haut niveau d'être exprimés plus facilement en termes métier. D'un autre côté, les développeurs trouvent souvent plus confortable de travailler directement avec JUnit, donc, si la communication avec des parties prenantes non techniciens n'est pas une priorité, cette façon de faire peut être envisagée favorablement.

Dans la version actuelle de Thucydides, vous pouvez écrire vos tests en utilisant easyb (pour une approche davantage dans le style BDD) ou en JUnit en utilisant Java ou un autre langage JVM (Groovy est un choix populaire). D'autres outils BDD seront gérés dans de futures versions. Nous aborderons les deux ici, mais vous pouvez utiliser celle avec laquelle vous et votre équipe êtes le plus à l'aise.

5.1. Définir des tests haut niveau en easyb

Easyb (http://easyb.org) est un outil BDD basé sur Groovy. Il facilite l'écriture d'histoires et de scénarios légers en utilisant la structuration classique du style BDD "given-when-then" ("étant donné-lorsque-alors") et en les implémentant en Groovy. Le plugin easyb de Thucydides est conçu pour faciliter l'écriture des tests Thucydides en utilisant easyb.

5.1.1. Écrire une histoire easyb en attente

Avec easyb, vous écrivez des tests (appelés "scenarios") qui, lorsqu'on utilise Thucydides, correspondent aux critères de recette automatisée. Les tests sont groupés en "histoires" ou "stories" - chaque histoire possède son propre fichier.

Les scénarios sont d'abord écrits comme "pending" (en attente). Ce sont juste des aperçus haut niveau, décrivant un ensemble de critères d'acceptation pour une histoire donnée et selon la structuration "given-when-then".

Quand les tests sont exécutés, les scénarios en attente sont ignorés. Cependant, ils apparaissent dans les rapports, de telle sorte que vous savez quelles fonctionnalités doivent encore être implémentées. Un exemple de la façon dont les scénarios en attente apparaissent dans un rapport Thucydides figure dans Figure 5.1, “Les tests en attente sont affichés avec l'icône *calendrier*”.

Figure 5.1. Les tests en attente sont affichés avec l'icône *calendrier*

Voici un exemple d'histoire easyb en attente utilisant 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"
}
```

Examinons cette histoire partie par partie. D'abord, vous devez déclarer que vous utilisez Thucydides. Vous faites cela en utilisant le mot clef easyb:

```
using "thucydides"
```

Ceci va, entre autres choses, injecter l'objet `thucydides` dans le contexte de votre histoire de telle sorte que vous pouvez configurer Thucydides pour exécuter votre histoire correctement.

Ensuite, vous devez dire à Thucydides quelle histoire vous être en train de tester. Vous faites cela en référençant l'une des classes d'histoire définies précédemment. C'est ce que nous faisons ici:

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

```
thucydides.tests_story AddNewCategory
```

Le reste de l'histoire easyb est juste une série de scénarios en attente easyb habituels. Pour le moment, il n'y a aucune implémentation de telle sorte qu'ils apparaissent en attente ("pending") dans les rapports:

```
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"
}
```

Typiquement, vous déclarez de nombreuses histoires en attente, de préférence en collaboration avec le propriétaire du produit ou les analystes métier au début d'une itération. Ceci vous donne une bonne idée de quelles histoires doivent être implémentées dans une itération donnée ainsi qu'une idée de la complexité relative de chaque histoire.

5.1.2. Implémenter des histoires easyb

L'étape suivante consiste à implémenter vos histoires. Voyons une version implémentée du premier de ces scénarios:

```
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"
    }
}
```

De nouveau, décomposons ceci. Dans la première partie, nous importons les classes que nous avons besoin d'utiliser:

```
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
```

Ensuite, nous déclarons l'URL de base par défaut à utiliser pour les tests. Comme l'annotation équivalente dans les tests JUnit, celle-ci est utilisée pour les tests exécutés depuis l'IDE ou si aucune URL de base n'est définie en ligne de commande en utilisant le paramètre `webdriver.base.url`.

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

Nous avons également besoin de déclarer les bibliothèques d'étapes de test que nous allons utiliser. Nous le faisons en utilisant thucydides.uses_steps_from. Ceci va injecter une variable d'instance dans le contexte easyb pour chaque bibliothèque d'étape déclarée. Si le nom de classe de la bibliothèque d'étape finit en *Steps* (par exemple JobSeekerSteps), le nom de la variable sera le nom de la classe moins le suffixe *Steps*, converti en minuscules et sous-ligné (ex. "job_seeker"). Nous en apprendrons davantage sur l'implémentation des bibliothèques d'étapes plus loin.

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

Enfin, nous implémentons le scénario. Notez que, puisqu'il s'agit de Groovy, nous pouvons déclarer des méthodes d'outillage pour faciliter la préparation et le nettoyage de l'environnement de test selon les besoins:

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

Habituellement, l'implémentation invoque seulement des méthodes d'étape, comme illustré ici:

```
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. Définir de tests de haut niveau avec JUnit

Thucydides s'intègre facilement avec les tests JUnit 4 ordinaires en utilisant le lanceur de tests ThucydidesRunner et quelques annotations spécialisées. C'est une des manières les plus simples pour commencer avec Thucydides, et cela est très bien adapté aux tests de non régression où la communication et les clarifications avec les différentes parties prenantes n'est pas une exigence.

Voici un exemple de test web JUnit Thucydides:

```
@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() {}

}
```

Examinons ceci section par section. La classe commence par l'annotation @RunWith pour indiquer qu'il s'agit d'un test Thucydides. Nous utilisons également l'annotation @Story pour indiquer quelle histoire utilisateur est testée (définie comme classe imbriquée des classes @Feature vues plus haut). Ceci est utilisé pour générer des rapports globaux.

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

Ensuite arrivent deux annotations essentielles pour tous les tests web. Tout d'abord, votre cas de test a besoin d'un champ public `Webdriver` annoté avec `@Managed`. Ceci permet à Thucydides de prendre soin pour vous de l'ouverture et de la fermeture du pilote WebDriver et d'utiliser ce pilote dans les pages et les étapes de test quand les tests sont exécutés:

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

Le second champ essentiel est une instance de la classe `Pages`, annotée avec `@ManagedPages`. Il s'agit essentiellement d'une fabrique de page que Thucydides utilise pour vous fournir des objets page instanciés. L'attribut `defaultUrl` vous permet de définir une URL à utiliser à l'ouverture de vos pages si aucune autre URL de base n'a été définie. Ceci est utile pour les tests dans l'IDE:

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

Notez que ces annotations ne sont nécessaires que pour les tests web. Si votre test Thucydides n'utilise pas de tests web, vous pouvez les ignorer en toute sérénité.

Pour des tests haut niveau de recette ou de non régression, c'est une bonne habitude de définir les tests de haut niveau comme une suite d'étapes de haut niveau. Cela rendra vos tests plus lisibles et plus faciles à maintenir si vous déléguez les détails d'implémentation de votre test (le "comment") à des méthodes réutilisables d'étape. Nous verrons comment définir ces méthodes d'étape plus tard. Cependant, vous devez au minimum définir la classe dans laquelle les étapes seront définies, en utilisant l'annotation @Steps. Cette annotation demande à Thucydides d'être à l'écoute des appels de méthodes de cet objet et (pour les tests web) d'injecter l'instance WebDriver et la fabrique de page dans la classe Steps de telle sorte qu'elles puissent être utilisées dans les méthodes d'étapes.

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

5.2.1. Tests en attente

Les tests qui ne contiennent aucune étape sont considérés comme étant en attente (*pending*). Vous pouvez également forcer une étape à être ignorée (et signalée comme étant en attente) en utilisant l'annotation `@Pending` ou l'annotation `@Ignore`. Noter que le sens est légèrement différent: `@Ignore` indique que vous avez temporairement suspendu l'exécution d'un test tandis que `@Pending` signifie que le test a été spécifié mais pas encore implémenté. Ainsi, les deux tests suivants seront en attente:

```
@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
}
```

Un test sera également considéré comme étant en attente si au moins l'une des étapes utilisées dans ce test est en attente. Pour qu'une étape soit en attente, elle doit être annotée avec `@Pending`.

hypothèses JUnit (assumptions)

Vous pouvez utiliser les hypothèses [http://junit.sourceforge.net/javadoc/org/junit/Assume.html] JUnit dans vos tests ou vos méthodes d'étapes. Les étapes pour lesquelles les conditions sous hypothèse échouent sont marquées EN SUSPEND au lieu de ERREUR. Les étapes suivantes sont également marquées comme étant EN SUSPEND.

```
@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();

}
...
```

Dans l'exemple suivant, si l'hypothèse dans l'étape `open_companies_list` échoue, celle-ci et toutes les étapes suivantes seront marquées comme EN SUSPEND.

5.2.2. Exécuter des tests dans une unique session du navigateur

Normalement, Thucydides ouvre une session du navigateur pour chaque test. Ceci permet de s'assurer plus facilement que chaque test est isolé et indépendant. Cependant, parfois, il est utile d'être capable d'exécuter des tests dans une même session du navigateur, en particulier pour des raisons de performance sur des écrans en lecture seule. Vous pouvez faire cela en utilisant l'attribut *uniqueSession* de l'annotation @Managed comme montré ci-dessous. Dans ce cas, le navigateur va s'ouvrir au début du cas de test et ne se fermera pas tant que tous les tests n'auront pas été exécutés.

```
@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");
    }
}
```

Si vous n'avez pas besoin de la gestion de WebDriver dans vos tests, vous pouvez omettre les annotations `@Managed` et `@Pages`, par exemple:

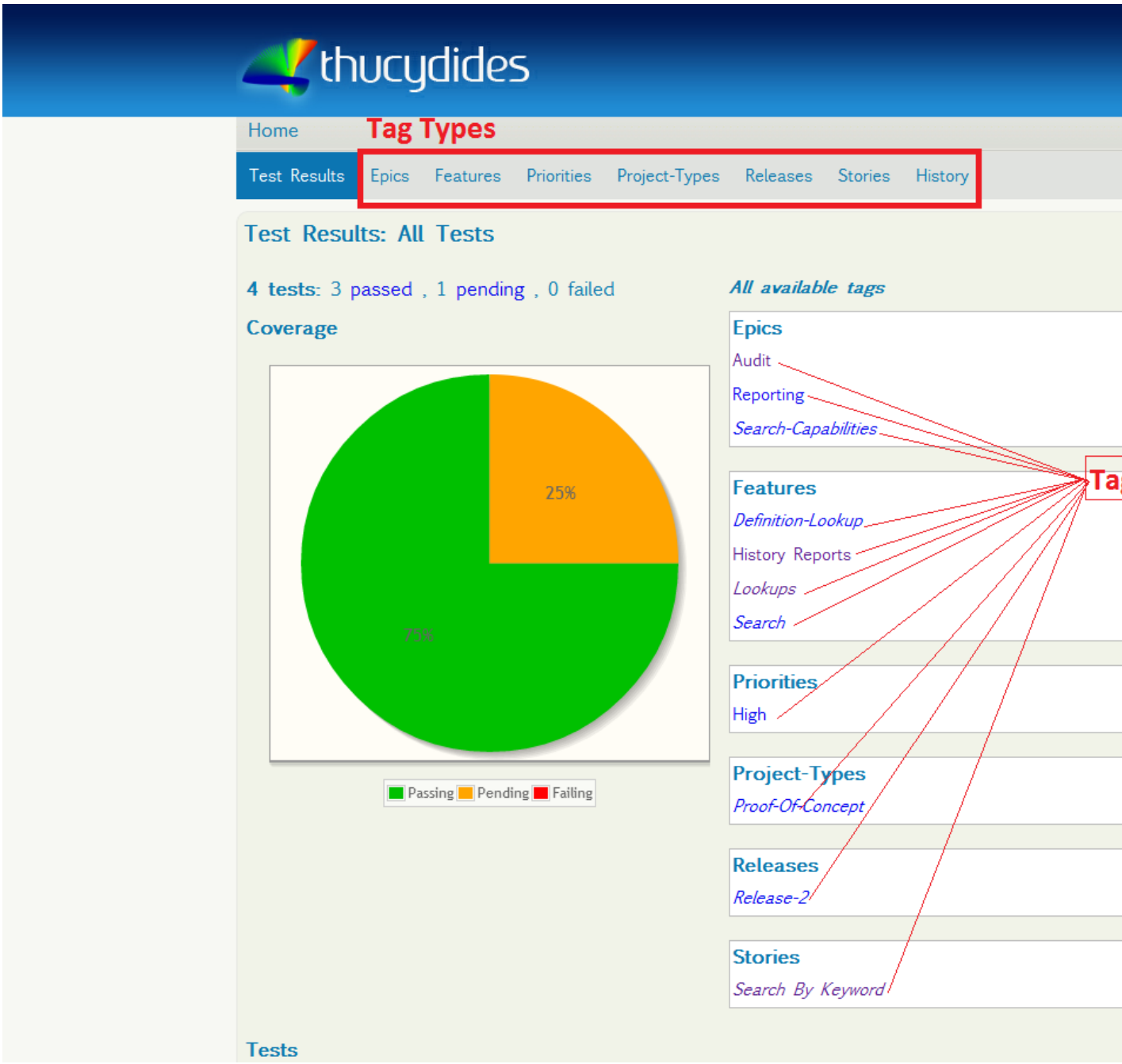
```
@RunWith(ThucydidesRunner.class)
@Story(Application.Backend.ProcessSales.class)
public class WorkWithBackendTest {

    @Steps
    public BackendSteps backend;

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

5.3. Ajouter des étiquettes aux cas de test

vous pouvez ajouter des étiquettes arbitraires à vos tests à la fois pour JUnit et easyb. Les étiquettes fournissent un contexte aux tests. Une étiquette comporte deux parties - un `type` et un `nom`. Les rapports Thucydides catégorisent les tests en se basant sur les types d'étiquettes indiqués.

Les types d'étiquettes sont arbitraires et vous pouvez ajouter autant de types que vous le souhaitez. Par défaut, un type d'étiquette `story` est automatiquement ajouté à chaque test. Un exemple d'étiquettes dans les rapports Thucydides figure dans ???

Figure 5.2. Les types d'étiquettes apparaissent en haut. Chaque type d'étiquette affiche les noms d'étiquettes.



5.3.1. Ajouter des étiquettes aux tests JUnit

Les étiquettes sont ajoutées aux tests JUnit en utilisant l'annotation `@WithTag`. Ce qui suit va ajouter une étiquette de type `epic` avec le nom "Audit".

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

Si aucun type n'est défini, on suppose que le type d'étiquette par défaut est `feature`. En d'autres modes, les deux étiquettes suivantes sont équivalentes.

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

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

`@WithTag` possède une syntaxe alternative plus concise utilisant un deux points pour séparer le type et le nom de l'étiquette. Par exemple,

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

ou,

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

Plusieurs étiquettes peuvent être ajoutées en utilisant l'annotation `@WithTags` ou sa cousine plus courte - `@WithTagValuesOf`. Par exemple,

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

        }
)
```

En utilisant `@WithTagValuesOf`, ce qui précède peut être écrit de la façon plus concise suivante :

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

5.3.2. Ajouter des étiquettes aux tests easyb

Des étiquettes peuvent facilement être ajoutées aux histoires easyb sous la forme `thucydides.tests.<type>`. Par exemple,

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

5.3.3. Filtrer les tests grâce aux étiquettes dans JUnit

Vous pouvez filtrer les tests par étiquette lors de l'exécution de Thucydides. Ceci peut être réalisé en indiquant en ligne de commande une seule étiquette ou une liste d'étiquettes séparées par des virgules. Si elle(s) est(sont) fournie(s), seules les classes et/ou les méthodes ayant une étiquette présente dans la liste seront exécutées.

Exemple:

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

ou

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

5.4. Exécuter Thucydides dans différents navigateurs

Thucydides gère tous les pilotes WebDriver de navigateurs, i.e. Firefox, Internet Explorer et Chrome ainsi que HTMLUnit. Par défaut, il utilisera Firefox. Cependant, vous pouvez passer outre cette option en utilisant la propriété système `webdriver.driver`. Pour l'utiliser en ligne de commandes, vous pouvez procéder comme suit:

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

Si vous n'utilisez pas Firefox par défaut, il vous sera utile de définir cette variable en tant que propriété dans votre fichier Maven pom.xml, par exemple:

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

Cependant, pour que ceci fonctionne avec JUnit, vous devez passer cette propriété webdriver.driver à JUnit. JUnit s'exécute dans une JVM dédiée et n'aura donc pas connaissance des propriétés système définies dans le build Maven. Pour y remédier, vous devez la passer explicitement à JUnit en utilisant l'option de configuration systemPropertyVariables, par exemple:

```
        <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. Options Chrome

Thucydides gère la propriété système `chrome.switches` pour définir des options pour le pilote Chrome. Ceci vous permet d'indiquer des options chrome très utiles telles que `"--homepage=about:blank" ou "--no-first-run"`. Vous pouvez indiquer autant d'options que vous voulez en les séparant par des virgules, par exemple :

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

5.5. Forcer l'utilisation d'un pilote spécifique dans un cas de test ou dans un test

L'annotation `@Managed` vous permet également d'indiquer le pilote que vous voulez utiliser pour un cas de test particulier, via l'attribut `driver`. Les valeurs gérées actuellement sont "firefox",

"iexplorer", "chrome" et "htmlunit". L'attribut `driver` vous permet de surcharger le pilote par défaut du système pour des exigences spécifiques. Par exemple, le cas de test suivant s'exécutera sous Chrome, quelle que soit la valeur de la propriété système `webdriver.driver` utilisée:

```
@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");
    }
}
```

Avec **easyb**, vous pouvez utiliser la directive `use_driver` comme illustré ici:

```
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"
    }
}
```

Avec JUnit, vous pouvez également utiliser l'annotation `@WithDriver` pour indiquer un pilote pour un test individuel. Ceci surchargera à la fois le pilote système et l'attribut pilote de l'annotation `@Managed` s'il y en a. Par exemple, le test suivant s'exécutera toujours avec Firefox:

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

Chapter 6. Écrire des tests de recette avec JBehave

Thucydides est une bibliothèque open source conçue pour faciliter la définition, l'implémentation et la génération de rapports pour les critères de recette automatisés. Jusqu'à maintenant, les tests Thucydides ont été implémentés en utilisant JUnit ou easyb. Cependant, la version la plus récente de Thucydides, la version 0.9.x, vous permet désormais d'écrire vos critères de recette en utilisant le framework populaire JBehave.

6.1. JBehave et Thucydides

JBehave est un framework BDD open source initialement écrit par Dan North, l'inventeur du BDD. Il est fortement intégré dans le monde JVM et largement utilisé par les équipes de développement en Java qui veulent implémenter les pratiques BDD dans leurs projets.

Avec JBehave, vous écrivez vos critères de recette en écrivant des histoires utilisateur et des scénarios en utilisant la notation BDD habituelle "étant "given-when-then" ("étant donné-quand-alors"), comme montré dans l'exemple suivant:

```
Scenario: Rechercher par mot clef et catégorie

Etant donné Sally qui veut acheter des timbres anciens pour son fils
Quand elle cherche des annonces dans la catégorie 'Anciens' contenant 'timbres'
Alors elle doit obtenir une liste d'annonces correspondant aux 'timbres'
      de la catégorie 'Anciens'
```

Les scénarii comme celui-là vont dans des fichiers *.story*: un fichier histoire est conçu pour contenir toues les scénarii (critères de recette) d'une histoire utilisateur donnée. Un fichier d'histoires peut également contenir une section descriptive au début qui donne des informations de contexte concernant les histoires testées:

```
De façon à trouver les éléments qui m'intéressent plus vite
En tant qu'acheteur
Je veux pouvoir obtenir la liste de toutes les annonces
   contenant un mot clef donné dans la description ou le titre

Scenario: Rechercher par mot clef et catégorie

Etant donné Sally qui veut acheter des timbres anciens pour son fils
Quand elle cherche des annonces dans la catégorie 'Anciens' contenant 'timbres'
Alors elle doit obtenir une liste d'annonces correspondant aux 'timbres'
   de la catégorie 'Anciens'

Scénario: Rechercher par mot clef et localisation

Etant donné Sally qui veut acheter un animal pour son fils
Quand elle cherche des annonces dans la catégorie Animaux de compagnie
   contenant animal Nouvelle-Galle du Sud
Alors elle doit obtenir une liste des annonces contenant le mot animal
   provenant d'annonceurs de Nouvelle-Galle du Sud.
```

Vous implémentez habituellement une histoire JBehave en utilisant des classes et des méthodes écrites en Java, en Groovy ou en Scala. Vous implémentez les étapes de l'histoire en utilisant des méthodes annotées pour représenter les étapes des scénarii textuels, comme montré dans l'exemple suivant:

```
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. Travailler avec JBehave et Thucydides

Thucydides et JBehave fonctionnent bien ensembles. Thucydides utilise de simples conventions pour débuter plus facilement dans l'écriture et l'implémentation d'histoires JBehave et de rapports à la fois pour les étapes JBehave et Thucydides qui peuvent être combinées de manière totalement transparente dans la même classe ou placées dans des classes séparées, selon ce que vous préférez.

Pour démarrer, vous aurez besoin d'ajouter le plugin JBehave Thucydides à votre projet. Avec Maven, ajoutez simplement les dépendances suivantes à votre fichier pom.xml:

```
<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>
```

De nouvelles versions sortent régulièrement, assurez-vous de contrôler le dépôt central Maven (http://search.maven.org) pour connaître les derniers numéros de version pour chacune des dépendances.

6.3. Configurer votre projet et organiser la structure de votre répertoire

JBehave est un outil hautement configurable. L'inconvénient de ceci est que, d'origine, JBehave nécessite du code de lancement pour démarrer. Thucydides essaie de simplifier ce processus en utilisant une approche convention plutôt que configuration qui réduit significativement le volume de travail nécessaire pour démarrer avec vos tests de recette. En fait, vous pouvez vous en débarrasser avec rien de plus qu'un cas de test JUnit vide et une structure de répertoire judicieusement organisée pour vos histoires JBehave.

6.3.1. Le lanceur de test JUnit

Les tests JBehave sont exécutés via un lanceur JUnit. Ceci facilite l'exécution des tests à la fois depuis un IDE ou comme une partie du processus de build. Tout ce dont vous avez besoin est d'hériter de ThucydidesJUnitStories, comme montré ici:

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

import net.thucydides.jbehave.ThucydidesJUnitStories;

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

Quand vous lancez ce test, Thucydides va lancer toutes les histoires JBehave qu'il trouve dans l'emplacement du répertoire par défaut. Par convention, il va chercher un répertoire *stories* dans votre classpath, aussi 'src/test/resources/stories' est un bon endroit pour placer vos fichiers histoire.

6.3.2. Organiser vos exigences

Placer toutes vos histoires JBehave dans un seul répertoire n'est pas très adapté quand le projet grandit; il est généralement mieux de les organiser dans une structure de répertoires qui les regroupe selon une logique donnée. De plus, si vous structurez bien vos exigences, Thucydides sera capable de fournir des rapports beaucoup plus significatifs concernant les résultats des tests.

Par défaut, Thucydides gère une convention simple basée sur les répertoires pour organiser vos exigences. La structure standard utilise trois niveaux: capacités, fonctionnalités et histoires (capabilities, features et stories). Une histoire est représentée par un fichier JBehave .story, aussi deux niveaux de répertoires sous le répertoire *stories* feront l'affaire. Un exemple de cette structure est montré ci-dessous:

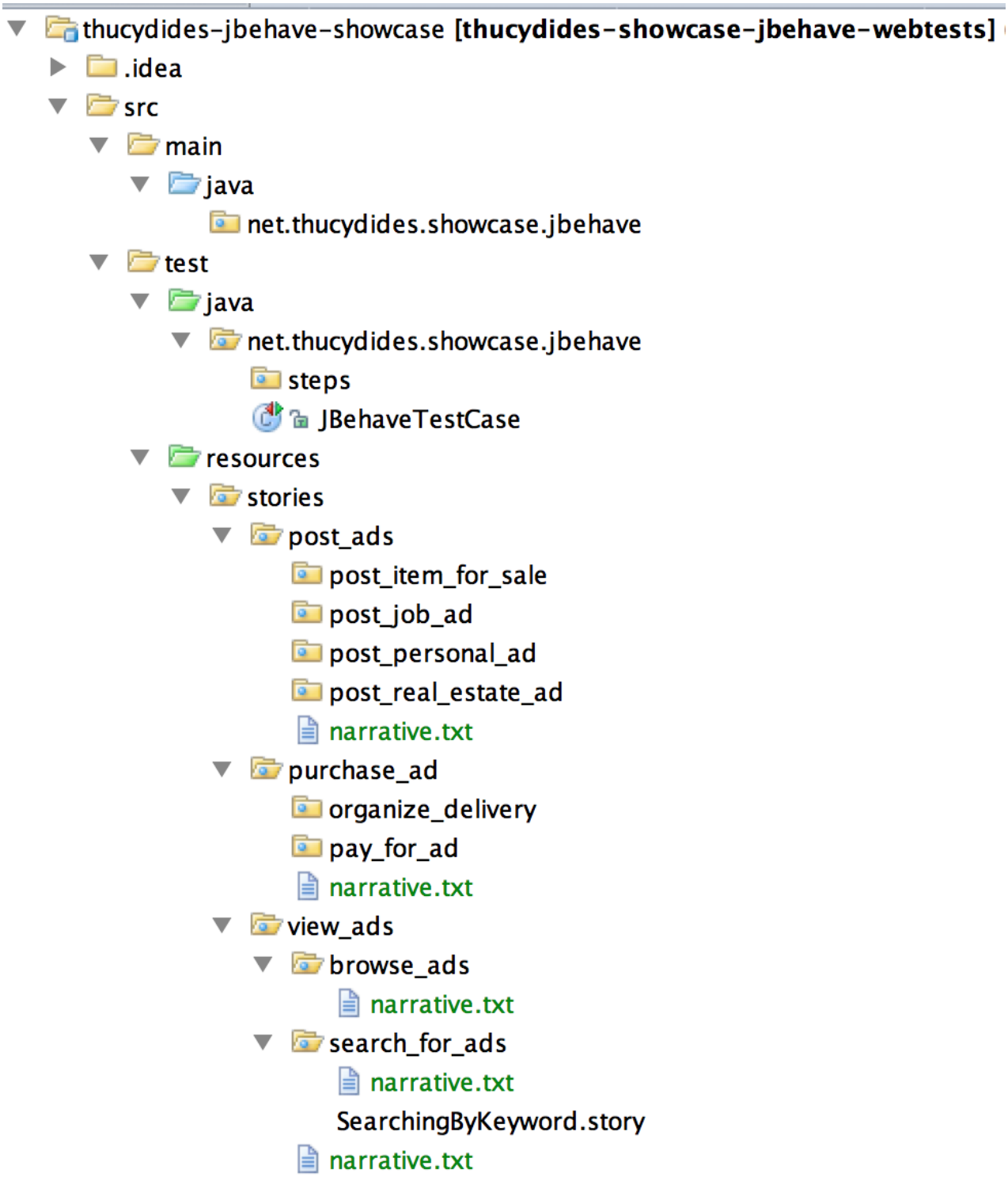
```
+ src
  + test
    + resources
      + stories
        + grow_potatoes                    [a capability]
          + grow_organic_potatoes          [a feature]
            - plant_organic_potatoes.story  [a story]
            - dig_up_organic_potatoes.story [another story]
          + grow_sweet_potatoes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

Si vous préférez une autre hiérarchie, vous pouvez utiliser la propriété système `thucydides.capability.types` pour surcharger la convention par défaut. Par exemple, si vous préférez organiser vos exigences sous forme d'une hiérarchie composée d'épopées, de thèmes et d'histoires (epics, themes et stories), vous pouvez donner à la propriété `thucydides.capability.types` la valeur *epic,theme* (le niveau story est représenté par le fichier .story).

Quand vous démarrez un projet, vous aurez typiquement une bonne idée des capacités que vous avez l'intention d'implémenter, et probablement certaines des fonctionnalités principales. Si vous enregistrez simplement vos fichiers .story dans la bonne structure de répertoires, les rapports Thucydides reflétera ces exigences, même si aucun test n'a encore été spécifié pour elles. C'est un excellent moyen pour garder une trace des progrès du projet. Au début d'une itération, les rapports

montreront toutes les exigences à implémenter, même celles dont les tests n'ont pas encore été définis ou implémentés. Quand l'itération avance, de plus en plus de critères de recette seront implémentés, jusqu'à ce que les critères de recette soient définis et implémentés pour toutes les exigences qui ont besoin d'être développées.

Figure 6.1. Un projet Thucydides utilisant JBehave peut organiser les histoires dans une structure de répertoires appropriée



Une fonctionnalité optionnelle mais très utile du format d'histoire de JBehave est la section descriptive qui peut être placée au début d'une histoire pour aider à fournir certains éléments de

contexte concernant l'histoire et les scénarii qu'elle contient. Cette description apparaîtra dans les rapports Thucydides, pour aider à donner aux propriétaires du produit (product owner), aux testeurs et aux autres membres de l'équipe plus d'information concernant le panorama et les motivations sous-jacentes à chaque histoire. Par exemple, si vous travaillez sur un site web de petites annonces, vous pourriez vouloir que les utilisateurs puissent pouvoir chercher des annonces en utilisant des mots clefs. Vous pourriez décrire cette fonctionnalité avec une description textuelle telle que celle-ci:

```
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.
```

Cependant, pour rendre les rapports encore plus utiles, c'est une bonne idée de non seulement documenter les histoires mais aussi vos exigences de plus haut niveau. Avec Thucydides, vous pouvez faire cela en plaçant un fichier texte appelé *narrative.txt* dans chacun de vos répertoires d'exigences que vous voulez documenter (voir ci-dessous). Ces fichiers se conforment à la convention JBehave/ Cucumber d'écriture des descriptions, avec un titre facultatif sur la première ligne suivi par une section descriptive commençant par le mot clef `Narrative:`. Par exemple, pour une fonctionnalité de recherche sur un site web de petites annonces, vous pourriez avoir une description contenant les lignes suivantes:

```
Recherche d'une annonce en ligne

Narrative:
De façon à augmenter les ventes des articles publicitaires
En tant que vendeur
Je veux que les acheteurs potentiels puissent n'afficher que les annonces pour
les articles qui peuvent les intéresser.
```

Quand vous lancez ces histoires (sans avoir implémenter un seul vrai test), vous obtenez un rapport contenant un tas de tests en attente, mais, plus intéressant, une liste des exigences qui ont besoin d'être implémentées, même s'il n'y a encore aucun test ou histoire associé à elles. Ceci facilite la planification d'une itération: vous aurez initialement un ensemble d'exigences avec seulement quelques tests, mais lorsque l'itération avance, vous verrez typiquement les exigences se remplir avec des critères de recette en attente ou atteints au fur et à mesure que le travail avance.

Figure 6.2. Vous pouvez voir les exigences que vous devez implémenter dans les rapports d'exigences

Les narrations au format asciidoc

Les narrations peuvent être écrites en Asciidoc [http://www.methods.co.nz/asciidoc/] pour une mise en forme plus riche. Positionnez la propriété `narrative.format` à `asciidoc` pour permettre à Thucydides d'analyser la narration au format asciidoc.

Par exemple, la narration suivante,

```
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
```

sera affichée dans le rapport comme montré ci-dessous.

Figure 6.3. Narration avec un format asciidoc

6.3.3. Personnaliser le module d’exigences

Vous pouvez également facilement étendre la gestion des exigences de Thucydides de telle façon qu’elle s’adapte à votre propre système. C’est un processus en deux étapes. D’abord, vous devez écrire une implémentation de l’interface `RequirementsTagProvider`.

```
package com.acme.tests

public class MyRequirementsTagProvider implements RequirementsTagProvider {
    @Override
    public List<Requirement> getRequirements() {
                // Retourne la liste complète des exigences disponibles
        // de votre système
    }

    @Override
    public Optional<Requirement> getParentRequirementOf(TestOutcome testOutcome) {
                // Retourne l'exigence, s'il y en a, associée avec un
        // résultat de test spécifique
    }

    @Override
    public Set<TestTag> getTagsFor(TestOutcome testOutcome) {
                // Retourne toutes les exigences, et autres étiquettes,
        // associées avec un résultat de test spéficique
    }
}
```

Ensuite, créez un fichier texte dans votre répertoire `src/main/resources/META-INF/serices` appelé `net.thucydides.core.statistics.service.TagProvider` et placez le nom pleinement qualifié de votre implémentation de RequirementsTagProvider.

6.3.4. Méta données d’histoire

Vous pouvez utiliser les étiquettes Meta de JBehave pour fournir des informations additionnelles à Thucydides concernant le test. L’annotation @driver vous permet d’indiquer quel pilote WebDriver utiliser, par exemple.

```
Meta:
@driver htmlunit

Scenario: Un scénario qui utilise Selenium

Etant donné que je suis sur la page de test
Quand je saisis le prénom <prenom>
Et que je saisis le nom <nom>
Alors je dois voir <prenom> et <nom>dans les champs noms
Et je dois utiliser HTMLUnit

Examples:
|prenom   |nom|
|Joe      | Blow|
|John     | Doe   |
```

Vous pouvez également utiliser l’annotation @issue pour relier des scénarios avec des anomalies, comme illustré ici:

```
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
```

Vous pouvez également attribuer des étiquettes à l'histoire en tant que telle, ou à des scénarios particuliers:

```
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. Implémenter les tests

Si vous voulez que vos tests fassent réellement quelque chose, vous aurez également besoin de classes dans lesquelles vous placerez vos implémentations des étapes JBehave. Si vous les placez dans un package dans ou sous le package de votre test principal JUnit, JBehave les trouvera sans qu'il soit nécessaire d'ajouter de configuration supplémentaire.

Thucydides ne fait pas de distinction entre les annotations de style JBehave @Given, @When et @Then et les annotations de style Thucydides @Step: les deux apparaîtront dans les rapports de tests. Cependant, vous devez commencer avec les méthodes annotées avec @Given, @When et @Then de façon à ce que JBehave puisse trouver les méthodes correctes à appeler pour vos histoires. Une méthode annotée avec @Given, @When ou @Then peut appeler des méthodes Thucydides @Step ou appeler des objets page directement (bien que le niveau d'abstraction supplémentaire fourni par les méthodes @Step tende à rendre les tests davantage réutilisables et maintenables pour des projets plus importants).

Un exemple typique est montré ci-dessous. Dans cette implémentation de l'un des scénarii que nous avons vus plus haut, les étapes de haut niveau sont définies en utilisant des méthodes annotées avec les annotations JBehave @Given, @When et @Then. Ces méthodes, à leur tour, utilisent des étapes qui sont implémentées dans la classe BuyerSteps qui contient une série de méthodes Thucydides @Step. L'avantage d'utiliser cette approche à deux niveaux est qu'elle aide à maintenir un degré de séparation entre la définition de ce qui est fait dans un test et comment c'est implémenté. Ceci tend à rendre les tests plus faciles à comprendre et plus facile à maintenir.

```
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);
    }
}
```

Les étapes Thucydides peuvent être trouvées dans la classe BuyerSteps. Cette classe utilise à son tour des objets page pour interagir avec l'application web réelle, comme illustré ici:

```
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);
```

```
    }
    ...
}
```

Les objets pages sont similaires à ceux que vous trouveriez dans n'importe quel projet Thucydides, ainsi que dans la plupart des projets WebDriver. Un exemple figure ci-dessous:

```
@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();
    }
}
```

Quand ces tests sont exécutés, les étapes JBehave les combinent avec les étapes Thucydides pour créer un rapport descriptif pour les résultats des tests:

Figure 6.4. Vous pouvez voir les exigences que vous devez implémenter dans le rapport d'exigences

6.4. Archétype Maven JBehave

Un archétype jBehave est disponible pour vous aider à démarrer un nouveau projet. Comme d'habitude, vous pouvez exécuter mvn archetype:generate depuis la ligne de commande puis choisir l'archétype net.thucydides.thucydides-jbehave-archetype dans la liste des archétypes proposés. Ou bien, vous pouvez utiliser votre IDE favori pour générer un nouveau projet Maven en utilisant un archétype.

Cette archétype crée une structure de répertoires du projet semblable à celle montrée ici :

```
+ 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. Exécuter tous les tests dans une seule fenêtre du navigateur

Tous les tests web peuvent être exécutés dans une seule fenêtre du navigateur en configurant la propriété système thucydides.use.unique.browser ou via programmation en utilisant runThucydides().inASingleSession() dans le lanceur JUnit.

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

import net.thucydides.jbehave.ThucydidesJUnitStories;

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

Chapter 7. Implémenter des bibliothèques d'étapes

une fois que vous avez défini les étapes dont vous avez besoin pour décrire vos tests haut niveau, vous devez les implémenter. Dans un test web automatisé, les étapes de test représentent le niveau d’abstraction situé entre vos objets pages (qui sont conçues en terme d’actions que vous exécutez sur une page donnée) et les histoires de plus haut niveau (suites d’actions davantage orientées métier qui illustrent comment une histoire utilisateur donnée a été implémentée). Les étapes peuvent contenir d’autres étapes et sont incluses dans les rapports Thucydides. A chaque fois qu’une étape est exécutée, une capture d'écran est enregistrée et affichée dans le rapport.

7.1. Créer des bibliothèques d'étapes

Les étapes de test sont des méthodes Java classiques annotées avec `@Step`. Vous rangez les étapes et les groupes d'étapes dans des bibliothèques d'étapes. Une bibliothèque d'étapes n’est qu’une classe Java classique. Si vous exécutez des tests web, votre bibliothèque d'étapes devra soit posséder une variable membre `Pages` soit, plus simplement, hériter de la classe `ScenarioSteps`, par exemple:

```
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);

    }
}
```

Notez que les méthodes d'étapes peuvent attendre des paramètres. Ces paramètres passés à la méthode d'étape seront enregistrés et apparaîtront dans les rapports Thucydides, ce qui en fait une excellente technique pour rendre vos tests davantage maintenables et modulaires.

Les étapes peuvent également appeler d’autres étapes, ce qui est très utile pour des scénarios de tests plus compliqués. Le résultat sera la sorte de structure imbriquée que vous pouvez voir dans Figure 2.1, “Un rapport de test généré par Thucydides”.

Chapter 8. Définir des objets Page

Si vous travaillez avec des tests web WebDriver, vous connaissez le concept de Page Objects ou Objets Page. Les objets Page sont une manière de masquer les détails d'implémentation d'une page web à l'intérieur d'une classe en n'exposant que les méthodes orientées métier en rapport avec cette page. C'est une excellente façon de rendre les tests web davantage maintenables.

Avec Thucydides, les objets page sont des objets page WebDriver habituels à la condition qu'ils possèdent un constructeur qui prenne un paramètre WebDriver. Toutefois, le `PageObject` de Thucydides fournit un certain nombre de méthodes utilitaires qui rendent plus commode l'utilisation des objets page, c'est pourquoi un objet page Thucydides hérite généralement de cette classe.

Voici un exemple simple:

```
@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());
    }
}
```

La méthode `typeInfo` est un raccourci qui se contente d'effacer un champ et d'y entrer le texte indiqué. Si vous préférez un style davantage orienté API, vous pouvez également faire quelque chose comme ça:

```
@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());
        }
}
```

Vous pouvez même utiliser un style encore plus fluide pour exprimer les étapes d'implémentation en utilisant des méthodes telles que `find`, `findBy` et `then`.

Par exemple, vous pouvez utiliser les finders "By" webdriver avec les sélecteurs d'élément name, id ou css ou les sélecteurs xpath comme ceci:

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

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

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

Vous pouvez même utiliser la méthode `findBy` et passer directement le sélecteur css ou xpath. Par exemple,

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

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

8.1. Utiliser des pages dans une bibliothèque d'étapes

Lorsque vous avez besoin d'utiliser un objet page dans l'une de vos étapes, vous vous contentez d'en demander une à la fabrique Page (Page factory) qui vous le fournira, par exemple:

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

Si vous voulez être sûr que vous êtes sur la bonne page, vous pouvez utiliser la méthode `currentPageAt()`. Celle-ci va chercher dans la classe Page Object toutes les annotations `@At` présentes et, s'il y en a, vérifiera que l'URL actuelle correspond au motif d'URL indiqué dans l'annotation. Par exemple, lorsque vous l'appelez en utilisant `currentPageAt()`, l'objet page suivant va vérifier que l'URL actuelle est précisément http://www.apache.org.

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

L'annotation `@At` gère également les jokers et les expressions rationnelles. L'objet page suivant correspondra à tout sous-domaine Apache:

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

Toutefois, dans le cas général, vous serez davantage intéressé par ce qui vient après le nom d'hôte. Vous pouvez utiliser le mot clef spécial `#HOST` pour que tout nom de serveur convienne. C'est

pour cette raison que l'objet page suivant correspondra à la fois à http://localhost:8080/app/action/login.form et à http://staging.acme.com/app/action/login.form. Les paramètres seront également ignorés, donc http://staging.acme.com/app/action/login.form?username=toto&password=oz correspondra également.

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

8.2. Ouvrir une page

Un objet page est habituellement conçu pour fonctionner avec une page web donnée. Quand la méthode `open()` est invoquée, le navigateur va s'ouvrir sur l'URL par défaut de la page.

L'annotation `@DefaultUrl` indique l'URL que ce test doit utiliser quand il est exécuté de manière isolée (par exemple depuis votre IDE). Généralement, cependant, la partie hôte de l'URL par défaut sera remplacée par la propriété `webdriver.base.url` ce qui vous permet de définir l'URL de base pour tous vos tests et de faciliter l'exécution de vos tests sur différents environnements, simplement en changeant cette valeur de propriété. Par exemple, dans la classe de test ci-dessus, définir `webdriver.base.url` à *https://staging.mycompany.com* fera que la page s'ouvrira à l'URL *https://staging.mycompany.com/somepage*.

Vous pouvez également définir des URL nommées qui peuvent être utilisées pour ouvrir la page web, assorties facultativement de paramètres. Par exemple, dans le code suivant, nous définissons une URL nommée *open.issue* qui accepte un unique paramètre:

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

Vous pouvez alors ouvrir cette page sur l'URL http://jira.mycompany.org/issues/ISSUE-1 comme illustré ici:

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

Vous pouvez également omettre totalement l'URL de base dans la définition de l'URL nommée en vous reposant sur les valeurs par défaut:

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

```
}
```

Et naturellement, vous pouvez faire des définitions multiples:

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

Vous ne devriez jamais essayer d'implémenter la méthode `open()` vous-même. En fait, elle est final. Si vous avez besoin que votre page fasse quelque chose au chargement, comme attendre l'apparition d'un élément dynamique, vous pouvez utiliser l'annotation @WhenPageOpens. Les méthodes de PageObject dotées de cette annotation seront appelées (dans un ordre quelconque) après que l'URL aura été ouverte. Dans cette exemple, la méthode `open()` ne rendra pas la main tant que l'élément web dataSection ne sera pas 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. Travailler avec des éléments web

Important

La méthode element() décrite ci-après n'est désormais plus nécessaire. Voir la section Façade d'élément web pour les détails.

8.3.1. Vérifier si des éléments sont visibles

La méthode element de la classe PageObject offre une API souple et commode pour interagir avec les éléments web en fournissant certaines fonctionnalités additionnelles fréquemment utilisées qui ne sont pas fournies nativement par l'API WebDriver. Par exemple, vous pouvez vérifier qu'un élément est visible comme illustré ici:

```
public class FindAJobPage extends PageObject {

    WebElement searchButton;

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

Si le bouton n’est pas présent à l'écran, le test va attendre un petit moment au cas où il apparaîtrait du fait de quelque magie Ajax. Si vous ne souhaitez pas que le test se comporte de cette façon, vous pouvez utiliser la version plus rapide:

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

Vous pouvez transformer ceci en affirmation en utilisant à la place la méthode `shouldBeVisible()`:

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

La méthode lancera une erreur d’affirmation si le bouton de recherche n’est pas visible pour l’utilisateur final.

Si vous n'êtes pas satisfait d’exposer le fait que votre page possède un bouton de recherche dans vos méthodes d'étape, vous pouvez rendre les choses encore plus simples en ajoutant une méthode accesseur qui renvoie une WebElementFacade, comme illustré ici:

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

Vos étapes contiendront alors du code tel que ce qui suit:

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

8.3.2. Vérifier si des éléments sont activés

Vous pouvez également vérifier si un élément est activé ou non:

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

Il existe également les méthodes négatives correspondantes:

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

Vous pouvez également contrôler des éléments qui sont présents sur la page mais qui ne sont pas visibles, par exemple:

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

8.3.3. Manipuler des listes déroulantes

Il existe également des méthodes d’aide pour les listes déroulantes. Supposons que vous avez la liste déroulante suivante sur votre page:

```
<select id="color">
```

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

Vous pouvez écrire un objet page pour manipuler cette liste déroulante comme suit:

```
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. Vérifier le focus

Vous pouvez savoir si un champ donné a le focus comme suit:

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

Vous pouvez également attendre que des éléments apparaissent, disparaissent, deviennent actifs ou inactifs:

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

ou

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

8.3.5. Utiliser directement des variables WebElementFacade

Au lieu de déclarer des variables WebElement dans les objets pages puis d’appeler element() ou $() pour les encapsuler dans des WebElementFacades, vous pouvez désormais déclarer directement des variables WebElementFacade dans les objets pages. Ceci rendra le code de l’objet page plus lisible.

Ainsi, au lieu d'écrire,

```
public class FindAJobPage extends PageObject {
```

```
    WebElement searchButton;

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

vous pouvez écrire,

```
public class FindAJobPage extends PageObject {

    WebElementFacade searchButton;

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

8.3.6. Utiliser des sélecteurs XPath et CSS

Un autre moyen d'accéder à des éléments web consiste à utiliser une expression XPath ou CSS. Vous pouvez utiliser la méthode `element` avec une expression XPath pour faire cela plus facilement. Par exemple, imaginez que votre application web nécessite de cliquer sur un élément de liste contenant un code postal donné. Une façon de faire serait la suivante:

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

Cependant, il est plus simple de faire comme ceci:

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

8.4. Travailler avec des pages asynchrones

Les pages asynchrones sont celles dont les champs ou les données ne sont pas tous affichés quand la page est chargée. Parfois, vous devez attendre que certains éléments apparaissent ou disparaissent pour lancer vos tests. Thucydides fournit certaines méthodes pratiques dans la classe PageObject pour faciliter la gestion de tels scénarios. Elles sont principalement conçues pour être utilisées dans les méthodes métier de vos objets page, mais dans les exemples, nous les utiliserons par appel externe sur un PageObject pour la clarté de la démonstration.

8.4.1. Vérifier si un élément est visible

Pour WebDriver, il existe une distinction entre le fait qu'un élément soit présent à l'écran (i.e. dans le code source HTML) et qu'il soit rendu (i.e. visible pour l'utilisateur). Vous pouvez également avoir besoin de savoir si un élément est visible à l'écran . Vous pouvez le faire de deux façons. La première possibilité est d'utiliser la méthode isElementVisible qui renvoie une valeur booléenne indiquant si l'élément est rendu (visible pour l'utilisateur) ou pas:

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

ou

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

La seconde possibilité est de décider activement que l'élément doit être visible:

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

Si l'élément n'apparaît pas immédiatement, vous pouvez attendre qu'il apparaisse:

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

Une alternative à la syntaxe précédente consiste à utiliser la méthode plus souple `waitFor` qui prend un sélecteur CSS ou XPath comme paramètre:

```
indexPage.waitFor("#popup"); //sélecteur CSS

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

Si vous voulez simplement vérifier si un élément est présent bien que pas forcément visible, vous pouvez utiliser `waitForRenderedElementsToBePresent` :

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

ou sa variante plus expressive, `waitForPresenceOf` qui prend un sélecteur CSS ou XPath comme paramètre:

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

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

Vous pouvez également attendre qu'un élément disparaisse en utilisant `waitForRenderedElementsToDisappear` ou `waitForAbsenceOf` :

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

indexPage.waitForAbsenceOf("#popup");

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

Pour simplifier, vous pouvez également utiliser les méthodes `waitForTextToAppear` et `waitForTextToDisappear`:

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

S'il y a plusieurs textes différents qui peuvent apparaître, vous pouvez utiliser `waitForAnyTextToAppear` ou `waitForAllTextToAppear`:

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

Si vous devez attendre qu'un élément parmi plusieurs possibles apparaisse, vous pouvez également utiliser la méthode `waitForAnyRenderedElementOf`:

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

8.5. Exécuter du Javascript

Il y a des situations dans lesquelles vous pourriez trouver utile d'exécuter un peu de Javascript directement dans le navigateur pour que le travail soit fait. Vous pouvez utiliser la méthode `evaluateJavascript()` de la classe `PageObject` pour faire cela. Par exemple, vous pourriez avoir besoin de calculer une expression et d'utiliser le résultat dans vos tests. La commande suivante va évaluer le titre du document et le renvoyer au code Java appelant:

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

Alternativement, vous pourriez simplement vouloir exécuter une commande Javascript localement dans le navigateur. Dans le code suivant, par exemple, nous positionnons le focus sur le champ de saisie *firstname*:

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

Et, si vous êtes familier avec JQuery, vous pouvez également faire appel aux expressions JQuery:

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

Ceci est souvent une stratégie utile si vous avez besoin de déclencher des événements tels que des survols de souris qui ne sont actuellement pas gérés par l'API WebDriver.

8.6. Envoi de fichiers

Envoyer des fichiers est facile. Les fichiers à envoyer peuvent soit être placés dans un emplacement codé en dur (pas bien) ou enregistrés dans le chemin d'accès (mieux). Voici un exemple simple:

```
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);
    }
}
```

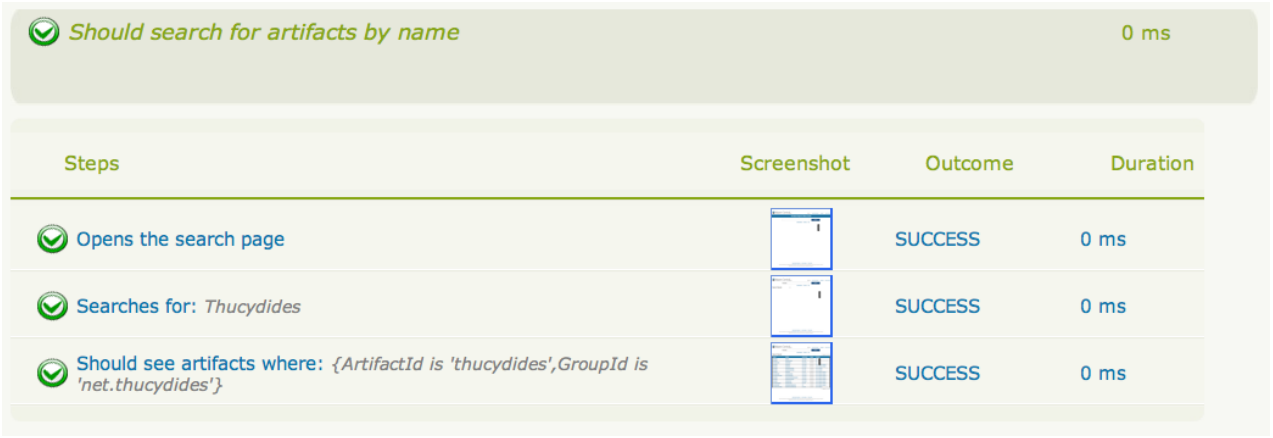
8.7. Utiliser des expressions de correspondance souples

Quand on écrit des tests de recette, on est souvent amené à exprimer des attentes relatives à des objets du domaine ou à des ensembles d'objets du domaine. Par exemple, si vous testez une fonctionnalité de recherche multi-critères, vous voulez savoir si l'application trouve les enregistrements que vous attendez. Vous pourriez être capable de faire cela d'une manière très précise (par exemple en sachant exactement les valeurs des champs que vous attendez) ou bien vous pourriez vouloir rendre vos tests

davantage flexibles en exprimant les plages de valeurs qui seraient acceptables. Thucydides fournit quelques fonctionnalités qui facilitent l'écriture des tests de recette dans ce type de cas.

Dans le reste de cette section, nous étudierons quelques exemples basés sur des tests du site de recherche Maven Central (voir Figure 8.1, “La page des résultats de la page de recherche de Maven Central”). Ce site vous permet de rechercher des artefacts Maven dans le dépôt Maven et de consulter les détails d’un artefact donné.

Figure 8.1. La page des résultats de la page de recherche de Maven Central



Nous allons utiliser quelques tests imaginaires de non régression pour ce site afin d’illustrer comment les comparateurs (matchers) de Thucydides peuvent être utilisés pour écrire des tests plus expressifs. Le premier scénario que nous allons envisager consiste à simplement chercher un artefact par son nom et à s’assurer que seuls les artefacts correspondant à ce nom apparaissent dans la liste des résultats. Nous pourrions énoncer informellement ce critère de validation de la manière suivante:

- Étant donné que le développeur est sur la page de recherche
- Et que le développeur cherche l’artefact appelé *Thucydides*
- Alors le développeur doit voir au moins 16 artefacts Thucydides, chacun doté d’un unique Id d’artefact

Avec JUnit, un test Thucydides correspondant à ce scénario pourrait ressembler à celui-ci:

```
...
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))));

    }
}
```

Voyons comment le test est implémenté dans cette classe. Le test `should_find_the_right_number_of_artifacts()` peut être explicité comme suit:

1. Quand nous ouvrons la page de recherche
2. Et que nous cherchons l'artefact contenant le mot *Thucydides*
3. Alors nous devrions voir une liste d'artefacts pour lesquels chaque Group ID commence par "net.thucydides", chaque Artifact ID est unique et qu'il y a au moins 16 entrées de ce type d'affichées.

L'implémentation de ces étapes est illustrée ici:

```
...
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);
    }
}
```

Les deux premières étapes sont implémentées par des méthodes relativement simples. Cependant, la troisième étape est plus intéressante. Regardons-la de plus près:

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

Ici, nous passons un nombre arbitraire d'expression à la méthode. Ces expressions sont en fait des *matchers*, des instances de la classe BeanMatcher. Vous n'avez normalement pas à vous soucier de ce niveau de détail - vous créez ces expressions de correspondance en utilisant un ensemble de méthodes statiques fournies par la classe BeanMatcher. Aussi vous ne devriez typiquement passer que des expressions relativement lisibles telles que `the("GroupId", startsWith("net.thucydides"))` ou `each("ArtifactId").isDifferent()`.

La méthode `shouldMatch()` de la classe BeanMatcherAsserts attend soit un unique objet Java, soit un ensemble d'objets Java et vérifie qu'au moins certains de ces objets correspondent aux contraintes indiquées par les matchers. Dans le cas du test web, ces objets sont typiquement des POJOs fournis par l'objet page pour représenter des objets du domaine ou des objets affichés à l'écran.

Il existe un certain nombre d'expressions différentes de matchers parmi lesquelles choisir. Le matcher le plus communément utilisé vérifie simplement la valeur d'un champ dans un objet. Par exemple, supposons que vous utilisez l'objet domaine montré ici:

```
     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() {...}
    }
```

Vous pourriez écrire un test pour vous assurer que la liste des personnes contienne au moins une personne appelée "Bill" en utilisant la méthode statique "the", comme montré ici:

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

    shouldMatch(persons, the("firstName", is("Bill"))
```

Le second paramètre de la méthode the() est un matcher Hamcrest qui vous donne une grande marge de flexibilité dans vos expressions. Par exemple, vous pourriez également écrire ce qui suit:

```
    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")));
```

Vous pouvez également passer des conditions multiples:

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

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

Thucydides fournit également la classe DateMatchers qui vous permet d'appliquer les matchers Hamcrest aux objets Java standard Dates et aux Datetimes `JodaTime`. Les exemples de code suivant illustrent la façon dont cela peut être utilisé:

```
    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))
```

Vous avez également parfois besoin de vérifier des contraintes qui s'appliquent à tous les éléments considérés. Le plus simple de ces cas de figure consiste à vérifier que toutes les valeurs prises par un champ particulier sont uniques. Vous pouvez faire cela en utilisant la méthode `each()`:

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

Vous pouvez également vérifier que le nombre d'éléments qui correspondent est conforme à ce que vous attendiez. Par exemple, pour vérifier qu'il n'y a qu'une seule personne dont le prénom est Bill, vous pourriez faire cela:

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

Vous pouvez également vérifier les valeurs minimum et maximum en utilisant les méthodes min() et max(). Par exemple, si la classe Person possède une méthode `getAge()`, nous pourrions nous assurer que chaque personne a plus de 21 ans et moins de 65 en faisant ce qui suit:

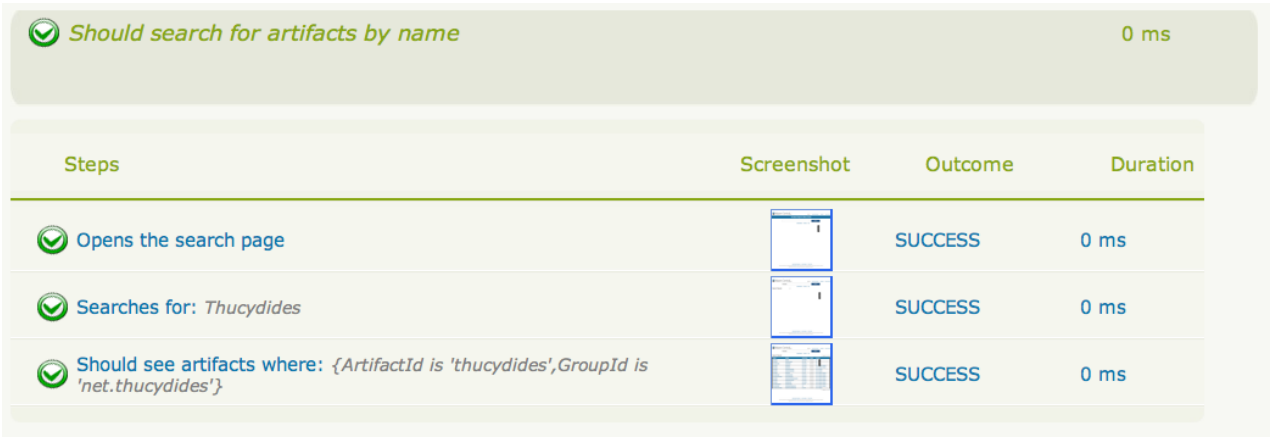
```
     shouldMatch(persons, min("age", greaterThanOrEqualTo(21)),
                          max("age", lessThanOrEqualTo(65)));
```

Ces méthodes fonctionnent avec les objets Java normaux mais aussi avec Maps. C'est pourquoi le code suivant fonctionne également:

```
    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"))
```

L'autre chose sympathique avec cette approche est que les matchers s'intègrent harmonieusement avec les rapports Thucydides. Ainsi, quand vous utilisez la classe BeanMatcher comme paramètre de vos étapes de test, les conditions exprimées dans l'étape seront affichées dans le rapport du test, comme montré dans Figure 8.2, "Les expressions de condition sont affichées dans les rapports de test".

Figure 8.2. Les expressions de condition sont affichées dans les rapports de test



Il existe deux canevas utilisés habituellement lors de la construction d'objets pages et d'étapes qui utilisent ce type de matcher. Le premier consiste à écrire une une méthode d'objet de page qui retourne la liste des objets du domaine (par exemple, les personnes) affichées dans la table. Par exemple, la méthode getSearchResults() utilisée dans l'étape should_see_artifacts_where() pourrait être implémentée comme suit:

```
    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;
    }
```

Le second consiste à accéder directement au contenu de la table HTML sans explicitement modéliser les données qui y sont contenues. Cette approche est plus rapide et plus efficace si vous ne prévoyez pas de réutiliser l'objet du domaine dans d'autres pages. Nous verrons comment faire ceci après.

8.7.1. Travailler avec les tables HTML

Puisque les tables HTML restent largement utilisées pour représenter des séries de données dans les applications web, Thucydides possède une classe HtmlTable qui fournit nombre de méthodes utiles qui facilitent l'écriture des objets page qui contiennent des tables. Par exemple, la méthode rowsFrom renvoie le contenu d'une table HTML sous forme d'une liste de Maps dans laquelle chaque map contient les valeurs des cellules pour une ligne, indexées par l'en-tête correspondant, comme montré ici:

```
...
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);
    }

}
```

Ceci économise beaucoup de saisie - notre méthode `getSearchResults()` ressemble maintenant à ceci:

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

Et puisque les matchers Thucydides fonctionnent à la fois avec les objets Java et les Maps, les expressions des matchers seront très semblables. La seule différence est que les Maps renvoyés sont indexés par les valeurs textuelles contenues dans les en-têtes de la table au lieu que ce soit dans les noms de propriété compatibles Java.

Vous pouvez également lire des tables sans en-tête (i.e, éléments <th>) en indiquant vos propres en-têtes en utilisant la méthode `withColumns`. Par exemple :

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

Vous pouvez également utiliser la classe HtmlTable pour choisir des lignes particulières dans une table pour travailler avec. Par exemple, un autre scénario de test pour la page de recherche Maven implique de cliquer sur un artefact et d'en afficher les détails. Le test pour ceux-ci ressemble à quelque chose comme ça:

```
    @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"))
    }
```

Maintenant la méthode open_artifact_where() nécessite de cliquer sur une ligne particulière de la table. Cette étape ressemble à quelque chose comme ça:

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

De cette façon, nous déléguons effectivement à l'objet Page qui effectue le vrai travail. La méthode correspondante de l'objet Page ressemble à ceci:

```
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();
    }
```

La partie intéressante ici est la première ligne de la méthode où nous utilisons la méthode filterRows(). Cette méthode va renvoyer une liste de WebElements qui correspondent au matcher que vous avez passé. Cette méthode rend vraiment facile la sélection de lignes qui vous intéressent pour un traitement particulier.

8.8. Exécuter plusieurs étapes en utilisant le même objet Page

Parfois, faire appel au navigateur peut être coûteux. Par exemple, si vous testez des tables avec un grand nombre d'éléments web (par exemple un élément web pour chaque cellule), les performances peuvent être basses et l'utilisation mémoire élevée. Normalement Thucydides va demander la page (et créer un objet Page) à chaque fois que vous appelez `Pages.get()` ou `Pages.currentPageAt()`. Si vous êtes certain que la page ne va pas changer (i.e vous n'allez exécuter que des opérations de lecture sur la page) vous pouvez utiliser la méthode onSamePage() de la classe ScenarioSteps pour vous assurer que tous les appels suivants à `Pages.get()` ou `Pages.currentPageAt()` renverront le même objet page:

```
@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. Basculer vers une autre page

La classe PageObject offre une méthode, switchToPage(), pour faciliter le retour vers un nouvel objet Page après avoir navigué depuis une méthode d'une classe PageObject. Par exemple,

```
@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. Intégration avec Spring

Si vous exécutez vos tests de recette sur un serveur web embarqué (par exemple en utilisant Jetty), il peut parfois s'avérer utile d'accéder aux couches de service directement pour le code de jointure ou lié à l'infrastructure. Par exemple, vous pouvez avoir un scénario dans lequel une action utilisateur doit, par effet de bord, enregistrer un journal d'audit dans une table de la base de données. Pour que vos tests restent simples et centrés sur le besoin, vous pourriez vouloir appeler la couche service directement pour vérifier le journal d'audit plutôt que de vous connecter en tant qu'administrateur et de naviguer dans l'écran d'audit des journaux.

Spring fournit une excellente gestion pour les tests d'intégration via le lanceur de tests SpringJUnit4ClassRunner. Malheureusement, si vous utilisez Thucydides, ce n'est pas envisageable car un test ne peut avoir deux lanceurs en même temps. Heureusement, cependant, il y a une solution ! Pour injecter des dépendances en utilisant un fichier de configuration Spring, il vous suffit d'inclure la règle SpringIntegration Thucydides dans votre classe de test. Vous instanciez cette variable comme illustré ici:

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

Ensuite vous utilisez l'annotation `@ContextConfiguration` pour définir le ou les fichiers de configuration à utiliser. Ensuite vous pouvez injecter des dépendances comme vous le feriez avec un test d'intégration Spring ordinaire en utilisant les annotations habituelles de Spring telles que `@Autowired` ou `@Resource`. Par exemple, supposons que nous utilisons le fichier de configuration Spring suivant appelé *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>
```

Nous pouvons utiliser ce fichier de configuration pour injecter des dépendances comme montré ici:

```
@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())));
    }
}
```

D'autres annotations liées au contexte telles que `@DirtiesContext` fonctionneront également comme elles le feraient dans un test d'intégration Spring traditionnel. Spring va créer un nouveau ApplicationContext pour chaque test mais il n'en n'utilisera qu'un seul pour toutes les méthodes de votre test. Si l'un de vos tests modifie un objet dans l'ApplicationContext vous pourriez vouloir dire à Spring qu'il réinitialise le contexte pour le test suivant. Vous ferez cela en utilisant l'annotation `@DirtiesContext`. Dans le cas de test suivant, par exemple, les tests échoueront sans l'annotation `@DirtiesContext`:

```
@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 10. Rapports Thucydides

Pour générer l'intégralité des rapports Thucydides, exécutez mvn thucydides:aggregate. Pour que cela fonctionne, vous devez ajouter le bon groupe de plugins à votre fichier settings.xml, comme montré ici:

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

Vous pouvez exécuter ceci dans la même commande que celle de vos tests en positionnant la propriété maven.test.failure.ignore à true. Si vous ne faites pas cela, Maven s'arrêtera si la moindre erreur se produit et ne réalisera pas la génération du rapport:

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

Vous pouvez également intégrer les rapports Thucydide dans les rapports standards Maven. Si vous utilisez Maven 2, ajoutez simplement le plugin Maven Thucydide à la section reporting:

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

Si vous utilisez Maven 3, vous devrez ajouter le rapport Maven Thucydides à la configuration maven-site-plugin comme illustré ici:

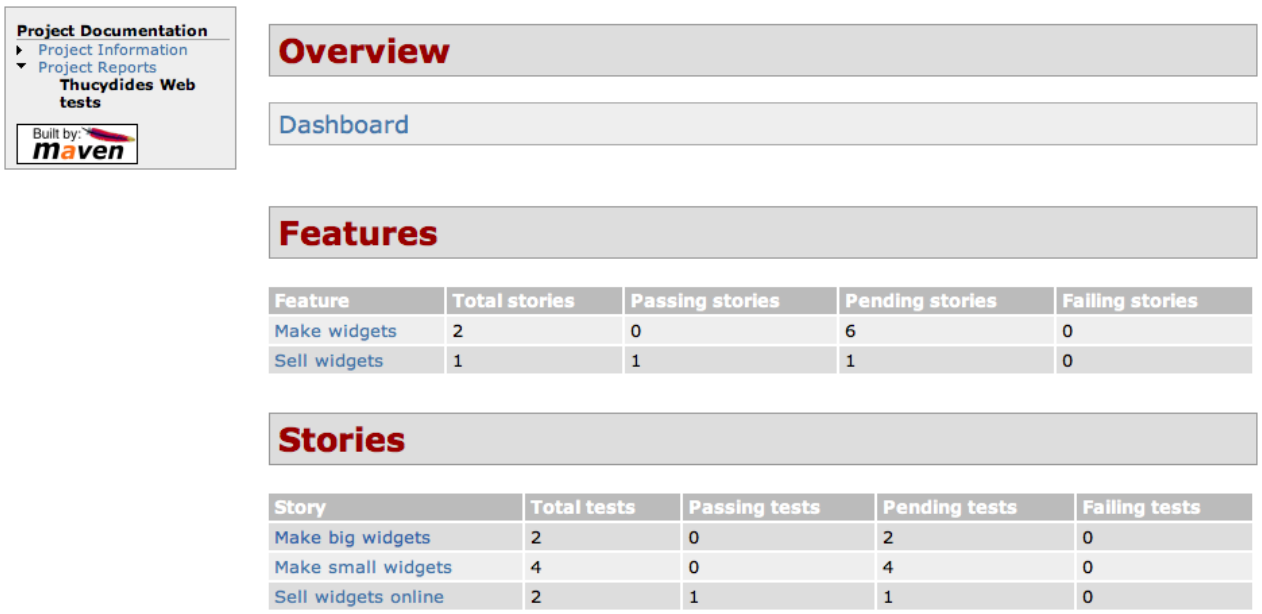
```
<build>
    <plugins>
        ...
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-site-plugin</artifactId>
            <version>3.0-beta-3</version>
            <configuration>
                <reportPlugins>
                    ...
        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

Pour générer ce rapport, lancez la commande mvn site après avoir exécuté mvn verify, par exemple.

```
$ mvn clean verify site
```

Ceci produira un rapport résumé dans la documentation de site Maven générée avec des liens vers les rapports Thucydides plus détaillées:

Figure 10.1. Rapports de tests Thucydides dans le site Maven



Chapter 11. Convertir les cas de tests xUnit, specFlow et Lettuce existant dans un rapport Thucydides

Une fois le plugin maven Thucydides ajouté comme décrit dans la section précédente, vous pouvez également importer des cas de tests existant specflow [http://www.specflow.org/] et Lettuce [http://lettuce.it/index.html] dans un rapport Thucydides. Pour cela, ajouter le répertoire `source` des cas de test et les paramètres `format` (xunit, specflow ou lettuce) à la configuration du plugin.

```
<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>
```

Chapter 12. Exécuter des tests Thucydides en ligne de commandes

Typiquement, vous exécuterez Thucydides comme une partie du processus de build (soit localement, soit sur un serveur d'intégration continue). En plus de l'option `webdriver.driver` déjà vue, vous pouvez également passer un certain nombre de paramètres en tant que propriétés système pour personnaliser la façon dont les tests sont exécutés. En voici la liste complète:

- **properties**: chemin absolu du fichier de propriétés dans lequel les valeurs par défaut des propriétés système Thucydides sont définies. `~/thucydides.properties` par défaut.
- **webdriver.driver**: le navigateur avec lequel vous voulez que vos tests soient exécutés : firefox, chrome, iexplorer, phantomjs, etc. Une prise en charge basique des pilotes iPhone et Android est également disponible.
- **webdriver.base.url**: L'URL de départ par défaut pour l'application, et l'URL de base pour les chemins relatifs.
- **webdriver.timeouts.implicitlywait**: Le temps que WebDriver attend par défaut qu'un élément apparaisse, en millisecondes.
- **thucydides.outputDirectory**: Où les rapports doivent être générés.
- **thucydides.only.save.failing.screenshots** : Est-ce que Thucydides doit n'enregistrer des captures d'écran que pour les tests en échec ? Ceci permet d'économiser de l'espace disque et augmente un peu la vitesse des tests. C'est très utile pour les tests dirigés par les données. Cette propriété est désormais obsolète. Utiliser `thucydides.take.screenshots` à la place.
- **thucydides.verbose.screenshots** : Mettez ceci à `true` pour enregistrer une capture d'écran pour chaque action d'un élément web (comme click(), typeAndEnter(), type(), typeAndTab() etc.). Cette propriété est maintenant obsolète. Utiliser à la place `thucydides.take.screenshots`.
- **thucydides.take.screenshots** : Configurer cette propriété pour avoir un contrôle plus fin sur la façon dont les captures d'écran sont prises. Cette propriété peut prendre les valeurs suivantes :
  - **FOR_EACH_ACTION** : identique à `thucydides.verbose.screenshots`
  - **BEFORE_AND_AFTER_EACH_STEP**,
  - **AFTER_EACH_STEP** et
  - **FOR_FAILURES** : identiques à `thucydides.only.save.failing.screenshots`
- **thucydides.verbose.steps** : Configurer cette propriété pour fournir une journalisation plus détaillée des étapes WebElementFacade quand les tests sont exécutés.
- **thucydides.restart.browser.frequency**: Lors de tests dirigés par les données, certains navigateurs (Firefox en particulier) peuvent ralentir au bout d'un moment à cause de fuites mémoire. Pour pallier cela, vous pouvez demander à Thucydides de démarrer une nouvelle session du navigateur à intervalles réguliers quand il exécute des tests dirigés par les données.

- **thucycides.step.delay**: Pause (en ms) entre chaque étape de test.

- **untrusted.certificates**: Utile si vous exécutez des tests Firefox sur un serveur de test HTTPS sans certificat valide. Ceci va faire que Thucydides va utiliser un profil doté de la propriété AssumeUntrustedCertificateIssuer.

- **thucydides.timeout**: Combien de temps le pilote attend les éléments qui ne sont pas immédiatement visibles.

- **thucydides.browser.width** and **thucydides.browser.height**: Re-dimensionne le navigateur à la taille indiquée de façon à prendre des captures d'écran plus grandes. Cela doit fonctionner avec Internet Explorer et Firefox mais pas avec Chrome.

- **thucydides.resized.image.width** : Valeur en pixels. Si elle est spécifiée, les captures d'écran sont redimensionnées à cette taille. Utile pour économiser de l'espace disque.

- **thucydides.keep.unscaled.screenshots** : À mettre à `true` si vous voulez enregistrer des captures d'écrans à la taille d'origine. Vaut `false` par défaut.

- **thucydides.store.html.source** : Mettre cette propriété à `true` pour enregistrer le code source HTML des pages web capturées. Elle vaut `false` par défaut.

- **thucydides.issue.tracker.url**: L'URL utilisée pour générer les liens vers le système de suivi des bugs.

- **thucydides.activate.firebugs** : Active les plugins Firebug et FireFinder pour Firefox quand des tests WebDriver sont exécutés. Ceci est utile pour déboguer mais pas recommandé quand les tests sont exécutés sur un serveur de build.

- **thucydides.batch.strategy** : Définit la stratégie de batch. Valeurs possibles - DIVIDE_EQUALLY (valeur par défaut) et DIVIDE_BY_TEST_COUNT. DIVIDE_EQUALLY va simplement répartir les tests équitablement entre tous les batchs. Ceci peut être inefficace si le nombre de tests varie beaucoup d'une classe de test à l'autre. La stratégie DIVIDE_BY_TEST_COUNT peut être beaucoup plus utile dans de tels cas puisqu'elle va créer des batchs en se basant sur le nombre de tests.

- **thucydides.batch.count**: Si le test de batch est utilisé, ceci est le nombre des batches exécutés.

- **thucydides.batch.number** : Si le test de batch est utilisé, ceci est le numéro du batch exécuté sur cette machine.

- **thcydides.reports.show.step.details** : Affiche des résultats détaillées d'étapes dans les tables de résultat des tests. Cette propriété est positionnée à false par défaut.

- **thucydides.use.unique.browser** : Positionner ceci pour exécuter tous les tests web dans un seul navigateur.

- **thucydides.locator.factory** : Positionner cette propriété pour surcharger la fabrique par défaut de localisateur (par ex., AjaxElementLocatorFactory ou DefaultElementLocatorFactory).
  Par défaut, Thucydides utilise une fabrique de localisateur personnalisée appelée DisplayedElementLocatorFactory.

- **thucydides.driver.capabilities** : Utiliser cette propriété pour passer une liste séparée par des virgules des capacités du pilote web.

- **thucydides.native.events** : Activer et désactiver les événements natifs de Firefox en positionnant cette propriété à `true` ou `false`.

- **security.enable_java** : Mettre ceci à vrai pour activer la prise en charge Java dans Firefox. Par défaut, c'est positionné à faux car ça ralentit le pilote web.

- **thucydides.test.requirements.basedir** : le répertoire de base du sous-module où les histoires jBehave sont conservées. Il est supposé que ce répertoire contient les sous répertoires src/test/resources. Si cette propriété est positionnée, les exigences sont lues depuis ce répertoire au lieu du classpath ou du répertoire de travail. Cette propriété est utilisée pour gérer des situations dans lesquelles votre répertoire de travail est différent du répertoire de base des exigences (par exemple on construit un projet multi modules du pom parent avec des exigences stockées dans un sous-module)

Un exemple d'utilisation de ces paramètres est montré ici:

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

Ceci exécutera les tests sur le serveur *staging* en utilisant Internet Explorer.

- **webdriver.firefox.profile**: Le chemin du répertoire du profil à utiliser quand on lance Firefox. Par défaut, WebDriver crée un profil anonyme. Ceci est utile si vous voulez exécutez des tests web en utilisant votre propre profil Firefox. Si vous n'êtes pas certain de la façon de trouver le chemin d'accès à votre profil, allez voir ici: http://support.mozilla.com/en-US/kb/Profiles. Par exemple, pour lancer le profil par défaut sur un système Mac OS X, vous feriez quelque chose comme ça:

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

Sur Windows, ce serait quelque chose comme:

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

- **firefox.preferences**: Une liste séparée par des virgules des paramètres de configuration de Firefox. Par exemple,

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

Les valeurs entières et booléennes seront converties dans les types correspondant dans les préférences Firefox; toutes les autres valeurs seront traitées comme des chaînes de caractères. Vous pouvez positionner une valeur booléenne à vrai simplement en indiquant le nom de la propriété, par exemple `-Dfirefox.preferences=app.update.silent`.

Un document de référence complet des paramètres de configuration de Firefox est disponible ici [http://kb.mozillazine.org/Firefox_:_FAQs_:_About:config_Entries].

- **thucydides.history**: Le répertoire dans lequel les données de résumé de l'historique des builds sont sauvegardées pour chaque projet. Chaque projet possède son propre sous-répertoire dans ce répertoire. Par défaut, c'est ~./thucydides.

Si vous voulez configurer des valeurs par défaut pour certaines de ces propriétés pour votre propre environnement de développement (par exemple, toujours activer le plugin Firebug sur votre machine

de développement), créez un fichier appelé `thucydides.properties` dans votre répertoire personnel (ou tout fichier de propriétés défini en utilisant la propriété système `properties`) et réglez-y les valeurs par défaut voulues. Ces valeurs seront toujours surchargées par celles définies dans les variables d'environnement. Un exemple est montré ici:

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

12.1. Fournir votre propre profil Firefox

Si vous avez besoin de configurer votre propre profil Firefox personnalisé, vous pouvez le faire en utilisant la méthode Thucydidies.useFirefoxProfile() avant de lancer vos tests. Par exemple :

```
@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**: Liste séparée par des deux points d'étiquettes. Si elle est fournie, seules les classes JUnit et/ou les méthodes avec des étiquettes présentes dans cette liste seront exécutées. Par exemple,

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

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

- **narrative.format**: Positionnez cette propriété à *asciidoc* pour activer l'utilisation du format Asciidoc [http://www.methods.co.nz/asciidoc/] dans les textes de narration.

Chapter 13. Intégrer avec les systèmes de suivi de bugs

13.1. Intégration de base avec un système de suivi de bugs

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

Pour faire cela dans Maven, vous devez passer la propriété système à JUnit en utilisant le plugin maven-surefire-plugin comme montré ici:

```
<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}</th
        </systemPropertyVariables>
    </configuration>
</plugin>
```

Thucydides fournit également un support spécial pour l'outil de suivi de bug JIRA de Atlassian. Si vous fournissez la propriété système `jira.url` au lieu de `jira.url` `thucydides.issue.tracker.url`, vous n'avez besoin de fournir que l'URL de base de votre instance JIRA et pas le chemin complet:

```
<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>
```

Vous devez fournir le numéro d'entrée. Vous pouvez le mettre dans le titre du test en le préfixant avec le caractère #. Pour les tests easyb, cela implique simplement de mentionner le numéro d'entrée (toujours en commençant par un caractère #) quelque part dans le nom du scénario. Pour les tests JUnit, vous utiliserez l'annotation @Title comme illustré ici:

```
@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");
    }
}
```

Une autre façon d'indiquer les entrées dans JUnit consiste à utiliser les annotations @Issue ou @Issues. Vous pouvez utiliser l'annotation @Issue pour associer un test donné avec une entrée particulière.

```
@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");
}
```

Vous pouvez également mettre l'annotation @Issue au niveau de la classe, auquel cas l'entrée sera associée à chaque test de la classe:

```
@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() {
            ...
        }
}
```

Si un test doit être associé à plusieurs entrées, vous pouvez utiliser à la place l'annotation @Issues:

```
@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");
}
```

Lorsque vous faites cela, les entrées apparaîtront dans les rapports ThucydidesRunner avec un hyperlien vers l'entrée correspondante de votre système de suivi de bugs.

Remplacer le répertoire par défaut des rapports

Par défaut, Thucydides génère ses rapports dans le répertoire `target/site/thucydides`. Il existe deux façons de surcharger ça, si c'est nécessaire. Avant tout, si vous remplacez le répertoire de sortie par défaut de Maven, vous pouvez remplacer l'élément `<directory>` dans la section `<build>` de votre fichier `pom.xml`. Cependant, puisque le plugin Maven Surefire exécute les tests dans une JVM forkée par défaut, vous devrez également passer la propriété `project.build.directory` aux tests unitaires en utilisant l'élément de configuration `<systemPropertyVariables>`, comme illustré ici:

```
     ...
    <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>
```

Ceci aura pour résultat que les rapports Thucydides seront générés dans 'build/site/thucydides` au lieu de `target/site/thucydides'.

Si vous voulez seulement remplacer le répertoire de sortie de Thucydides, vous pouvez utiliser les propriétés `thucydides.outputDirectory` et `thucydides.sourceDirectory`, soit dans le fichier `pom.xml`, soit à partir de la ligne de commandes. Par exemple, les propriétés suivantes généreront le rapport Thucydides dans le répertoire `build/thucydides-reports`:

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

Chapter 14. Utiliser les étiquettes Thucydides

Voir les résultats des tests est évidemment utile, mais, du point de vue d'une livraison et d'un déploiement, on ne fait qu'effleurer la surface. Il est beaucoup plus intéressant de pouvoir voir les résultats des tests en terme d'histoires, de fonctionnalités, de comportements, de scénarii ou de n'importe quelle autre organisation que vous pourriez trouver utile.

Les étiquettes (tags) Thucydides offrent un mécanisme très flexible pour catégoriser et rapporter les résultats de vos tests et qui sert d'alternative à la structure histoire/fonctionnalité décrite précédemment. Vous pouvez choisir un ensemble approprié de types d'étiquettes (telles que "fonctionnalité", "comportement", "épopée", "scénario", "exigences non fonctionnelles", etc.) puis affecter ces étiquettes de différents types à vos tests. Pour déclarer un type d'étiquette, vous utilisez simplement une étiquette du type indiqué - elle apparaîtra alors automatiquement dans les rapports Thucydides.

Vous pouvez affecter manuellement une étiquette à un test en utilisant l'annotation `@WithTag`:

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

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

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

Notez que ces étiquettes peuvent être affectées au niveau du test ou de la classe. Les noms d'étiquette et les types sont du texte libre -

14.1. Écrire un plugin d'étiquettes Thucydides

Les étiquettes Thucydides sont faciles à intégrer avec d'autres applications, telles que les systèmes de suivi d'incidents ou de gestion de projets agiles. Dans cette section, nous verrons comment écrire un plugin qui permet à Thucydides d'affecter automatiquement des étiquettes à vos tests en se basant sur vos propres exigences.

Dans Thucydides, les étiquettes sont des couples arbitraires de valeurs de chaînes de caractères (nom et type) représentés par la classe `TagType`. Vous pouvez créer une étiquette en utilisant la méthode `TagTest.withName()`, comme montré ici:

```
  TestTag specialFeatureTag = TestTag.withName("fonctionnalité spéciale").andType("fonct
```

Tous les types de fonctionnalité que vous fournissez seront affichés dans des onglets distincts en haut de l'écran des rapports et fourniront toutes les fonctionnalités d'agrégation et de filtrage qu'on trouve avec les rapports standards.

Pour définir vos propres étiquettes, vous devez écrire votre propre fournisseur d'étiquette en implémentant l'interface TagProvider, comme illustré ci-dessous:

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

L'unique méthode de cette interface, `getTagsFor()`, prend un objet `TestOutcome` et renvoie un ensemble d'étiquettes associée avec ce compte-rendu de tests. La classe `TestOutcome` fournit un grand nombre de champs décrivant le test et ses résultats. Par exemple, pour obtenir la liste des incidents indiqués pour ce test, utiliser la méthode `getIssues()`. Le code suivant est un exemple de fournisseur d'étiquette qui fournit une liste d'étiquettes en se basant sur les incidents associés (indiqués par les annotations `@Issue` and `@Issues`).

```
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) {...}
}
```

Vous devez également fournir une définition de service dans le répertoire `/META-INF/services` dans le classpath, de telle sorte que Thucydides puisse enregistrer et utiliser votre plugin. Un moyen simple de faire ceci est de créer un projet Maven avec un fichier appelé `net.thucydides.core.statistics.service.TagProvider` dans le répertoire `src/resources/META-INF/services`. Ce fichier est un fichier texte contenant le nom pleinement qualifié de votre fournisseur d'étiquette, par exemple:

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

Maintenant, incluez simplement le fichier JAR généré dans vos dépendances et Thucydides l'utilisera automatiquement pour inclure vos étiquettes personnalisées dans vos rapports.

14.2. Intégration bi-directionnelle avec JIRA

Une stratégie habituelle pour les organisations utilisant JIRA consiste à représenter les cartes d'histoire (story cards) et/ou les critères de validation associés comme des entrées JIRA. Il est utile de savoir quels tests automatisés ont été exécutés pour une story card JIRA et quelle histoire est testée par un test donné.

Vous pouvez ajouter ces deux fonctionnalités à votre projet Thucydides en utilisant le `thucydides-jira-plugin`. D'abord, vous devez ajouter le `thucydides-jira-plugin` à vos dépendances Maven. Les dépendances dont vous aurez besoin (incluant les dépendances normales de Thucydides) sont listées ici:

```
    <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>
    ...
```

Notez que l'intégration du cycle de vie JIRA requiert Groovy version 1.8 ou supérieure pour fonctionner correctement.

Vous aurez également besoin d'une implémentation de `slf4j`, par exemple 'slf4j-log4j12# (si vous utilisez Log4j) ou 'logback-classic' (si vous utilisez LogBack) (voir http://www.slf4j.org/codes.html#StaticLoggerBinder pour plus de détails). Si vous êtes bloqués, ajoutez simplement slf4j-simple:

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

Dans Thucydides, vous pouvez vous référer à une entrée JIRA en plaçant une référence au numéro correspondant de l'entrée JIRA soit dans le nom du test (en utilisant l'annotation @Title, par exemple) ou, plus simplement, en utilisant les annotations @Issue ou @Issues comme montré ici:

```
    @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 encyclopedi

        }
    }
```

Dans cet exemple, le test sera associé avec l'entrée WIKI-1.

Alternativement, vous pourriez vouloir associer une entrée (telle qu'une story card) avec toutes les histoires d'un cas de test en plaçant une annotation @Issue (ou @Issues) au niveau de la classe:

```
        @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_correspondi
                endUser.is_on_the_wikipedia_home_page();
                endUser.looks_up_cats();
                endUser.should_see_article_with_title("Cat - Wikipedia, the free encyclo

            }
        }
```

Thucydides peut utiliser ces annotations pour s'intégrer avec les entrées de JIRA. L'intégration la plus simple avec JIRA implique d'ajouter des liens vers les entrées correspondantes de JIRA dans les rapports Thucydides. Pour activer ceci, vous avez simplement à fournir l'option de ligne de commandes **jira.url**. Vous avez cependant à passer cette option à JUnit en utilisant le maven-surefire-plugin comme montré ici:

```
<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>
      ...
```

Pour une intégration plus fine et avec aller-retours, vous pouvez également utiliser le plugin thucydides-jira-plugin. Celui-ci n'inclura pas seulement les liens vers JIRA dans les rapports Thucydides mais il mettra également à jour les entrées JIRA avec les liens vers les pages des histoires correspondantes dans les rapports Thucydides. Pour configurer cela, ajoutez la dépendance thucydides-jira-plugin à votre projet.

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

Vous devez également fournir un nom d'utilisateur et un mot de passe pour la connexion à JIRA ainsi que l'URL à laquelle les rapports Thucydides seront publiés (par exemple, sur votre serveur d'intégration continue). Vous ferez cela en passant les paramètres systèmes jira.username, jira.password et thucydides.public.url.

```
<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>
                    <thucydides.public.url>http://localhost:9000</thucydides.public.url>
                </systemPropertyVariables>
            </configuration>
        </plugin>
        ...
```

Thucydides génère également des rapports agrégés regroupant les résultats des histoires et des fonctionnalités. Pour inclure les liens JIRA dans ces rapports, vous devez renseigner l'option de configuration `jiraUrl` dans `maven-thucydides-plugin`, comme illustré ici:

```
    <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>
```

Si vous ne souhaitez pas que Thucydides mette à jour les entrées JIRA pendant une exécution donnée (par exemple à des fins de test ou de débogage), vous pouvez également positionner `thucydides.skip.jira.updates` à true, par exemple:

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

Vous pouvez également configurer le plugin pour mettre à jour l'état des entrées JIRA.
Ceci est désactivé par défaut: pour utiliser cette option, vous devez positionner l'option `thucydides.jira.workflow.active` à 'true', par exemple:

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

La configuration par défaut fonctionnera avec le cycle de vie JIRA par défaut: les entrées ouvertes ou en cours associées avec des tests réussis seront résolues, et les entrées fermées ou résolues associées à des tests en échec seront réouvertes. Si vous utilisez un cycle de vie personnalisé, ou si vous voulez modifier la façon dont les transitions se comportent, vous pouvez écrire votre propre configuration de cycle de vie. La configuration de cycle de vie utilise un simple DSL Groovy. Ce qui suit est un exemple de fichier de configuration utilisé pour le cycle de vie par défaut:

```
    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'
    }
```

Vous pouvez écrire votre propre fichier de configuration et le mettre dans le classpath de votre projet de test (par exemple, dans le répertoire src/test/resources). Ensuite, vous pouvez surcharger la configuration par défaut en utilisant la propriété `thucydides.jira.workflow` dans le fichier `pom.xml` ou directement en ligne de commandes, par exemple:

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

Alternativement, vous pouvez simplement créer un fichier appelé `jira-workflow.groovy` et le placer quelque part dans votre classpath. Thucydides utilisera alors ce cycle de vie. Dans les deux cas, vous n'avez pas besoin de renseigner explicitement la propriété `thucydides.jira.workflow.active`.

Vous pouvez également intégrer les entrées JIRA dans vos histoires Thucydides easyb. Lorsque vous utilisez l'intégration easyb Thucydides, vous associez une ou plusieurs entrées avec l'histoire easyb comme un tout, mais pas avec des scénarios particuliers. Vous faites cela en utilisant la notation thucydides.tests_issue:

```
    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
        }
    }
```

Vous pouvez également associer plusieurs entrées en utilisant `thucydides.tests_issues`:

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

Pour utiliser easyb avec Thucydides, vous devez ajouter la dernière version de `thucydides-easyb-plugin` à vos dépendances, si ce n'est pas déjà fait:

```
    <dependency>
        <groupId>net.thucydides.easyb</groupId>
```

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

Comme avec JUnit, vous devez passer les paramètres appropriés à easyb pour que cela fonctionne. Vous devrez également utiliser la version 1.4 ou supérieure de maven-easyb-plugin, configuré pour passer les paramètres JIRA comme montré ici:

```
    <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>
```

Une fois que ceci est fait, Thucydides mettra automatiquement à jour les entrées JIRA appropriées à chaque fois que les tests seront exécutés.

Chapter 15. Gérer les captures d'écran

Par défaut, Thucydides enregistre une capture d'écran pour chaque étape exécutée pendant les tests. Thucydides peut être configuré pour contrôler quand enregistrer des captures d'écran.

15.1. Configurer quand des captures d'écran sont prises

La propriété `thucydides.take.screenshots` peut être utilisée pour configurer la fréquence à laquelle les captures d'écran sont faites. Cette propriété peut prendre les valeurs suivantes :

- `FOR_EACH_ACTION` : Enregistre une capture d'écran à chaque action d'un élément web (telle que click(), typeAndEnter(), type(), typeAndTab() etc.).
- `BEFORE_AND_AFTER_EACH_STEP` : Enregistre une capture d'écran avant et après chaque étape.
- `AFTER_EACH_STEP` : Enregistre une capture d'écran après chaque étape.
- `FOR_FAILURES` : Enregistre une capture d'écran seulement pour les étapes en échec. Ceci permet de préserver l'espace disque et accélère un peu les tests. C'est très utile pour les tests dirigés par les données.

15.2. Utiliser des annotations pour contrôler les captures d'écran

Il est possible d'utiliser un degré encore plus fin de contrôle en utilisant des annotations. vous pouvez annoter n'importe quelle méthode de test ou d'étape (ou n'importe quelle méthode utilisée par un test ou une méthode) avec l'annotation `@Screenshots` pour surcharger le nombre des captures d'écran prises pour cette étape (ou sous-étape). Voici quelques exemples :

```
@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() {…}
```

15.3. Prendre une capture d'écran à un moment arbitraire durant une étape

Il est possible d'avoir un contrôle encore plus fin sur les captures d'écran dans les tests. En utilisant la méthode `takeScreenshot`, vous pouvez indiquer à Thucydides de prendre une capture d'écran

à n’importe quel endroit de l'étape indépendemment du niveau de capture d'écran choisi via la configuration ou les annotations.

Appelez simplement `Thucydides.takeScreenshot()` dans les méthodes de test à chaque fois que vous voulez qu’une capture d'écran soit réalisée.

15.4. Augmenter la taille des captures d'écran

Parfois, la taille par défaut de la fenêtre est trop petite pour afficher tous les écrans de l’application dans les captures. Vous pouvez agrandir la taille de la fenêtre que Thucydides ouvre en fournissant les propriétés système `thucydides.browser.width` et `thucydides.browser.height`. Par exemple, pour utiliser une fenêtre de navigateur de dimensions 1200x1024, vous devriez faire ce qui suit:

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

Typiquement, le paramètre largeur (width) est le seul que vous aurez besoin d’indiquer, la hauteur étant déterminée par le contenu de la page du navigateur.

Si vous exécutez Thucydides avec JUnit, vous pouvez également indiquer ce paramètre (et n’importe quel autre qui serait concerné) directement dans votre fichier pom.xml, dans la configuration de maven-surefire-plugin, par exemple:

```
<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>
        ...
```

Lorsque la largeur du navigateur est supérieure à 1000px, la vue diaporama dans les rapports s’agrandira pour montrer les captures entières.

Notez qu’il existe quelques problèmes avec cette fonctionnalité. En particulier, elle ne fonctionnera pas du tout avec Chrome, car Chrome, par conception, ne gère pas le redimensionnement de la fenêtre. De plus, puisque WebDriver utilise un vrai navigateur, la taille maximum sera limitée par la taille physique du navigateur. Cette limitation s’applique à la largeur du navigateur, étant donné que la longueur verticale totale de l'écran sera toujours enregistrée dans la capture d'écran même si celle-ci dépasse la hauteur d’une seule page et nécessite un ascenseur.

15.4.1. Captures d'écran et problèmes de dépassement de mémoire

Selenium a besoin de mémoire pour prendre des captures d'écran, particulièrement si les écrans sont grands. Si Selenium n’a plus de mémoire quand il prend des captures d'écran, il journalise une erreur dans la sortie des tests. Dans ce cas, configurez le maven-surefire-plugin pour utiliser davantage de mémoire, comme illustré ici:

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

15.5. Enregistrer des captures d'écran brutes

Thucydides n’enregistre par défaut que des captures d'écran redimensionnées. Ceci est fait pour réduire l’espace disque utilisé par les rapports. Si vous avez besoin d’enregistrer des captures d'écran à la taille originale, ce réglage par défaut peut facilement être surchargé en réglant la propriété `thucydides.keep.unscaled.screenshots` à `true`.

15.6. Enregistrer des captures d'écran sous forme de fichiers source HTML

Il est possible d’enregistrer des fichiers source html pour les captures d'écran en positionnant la propriété `thucydides.store.html.source` à `true`. Les fichiers source Html ne sont pas enregistrés par défaut pour préserver l’espace disque.

15.7. Flouter des captures d'écran sensibles

Pour des raisons de sécurité ou de préservation de la vie privée, il peut être nécessaire de flouter des captures d'écran sensibles dans les rapports Thucydides. Ceci peut être fait en annotant les méthodes de test ou les étapes avec l’annotation `@BlurScreenshots`. Quand elle est définie sur un test, toutes les captures d'écran de ce test seront floutées. Quand elle est définie sur une étape, seules les captures d'écran de cette étape seront floutées. @BlurredScreenshot prend une chaîne de caractère en paramètre ayant la valeur `LIGHT, MEDIUM` ou `HEAVY` pour indiquer le niveau de flou. Par exemple,

```
@Test
@BlurScreenshots("HEAVY")
```

```
public void looking_up_the_definition_of_pineapple_should_display_the_corresponding_arti
    endUser.is_the_home_page();
    endUser.looks_for("pineapple");
    endUser.should_see_definition_containing_words("A thorny fruit");
}
```

Un écran à divers niveaux de flou est montré ci-dessous.

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
ລາວ
Latviešu
Magyar
Македонски
Malagasy
Nederlands
日本語
Occitan
Polski
Português
Română
Русский
Gagana Samoa
Српски / srpski
Suomi
Svenska
தமிழ்
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.

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

Pronunciation [edit]

- enPR: pīʹnăpəl, 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

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

Chapter 16. Gérer l'état entre les étapes

Quelquefois, il peut être utile d'être capable de passer des informations entre les étapes. Par exemple, vous pourriez avoir besoin de vérifier que les informations entrées dans un formulaire d'enregistrement d'un client apparaissent correctement sur une page de confirmation ultérieure.

Vous pouvez faire cela en passant des valeurs d'une étape à l'autre, cependant cela tend à polluer les étapes. Une autre approche consiste à utiliser la session de test Thucydides, qui est en fait un tableau de hachage dans lequel vous pouvez enregistrer des variables pendant un test donné. Vous pouvez obtenir ce hachage de session en utilisant la méthode statique `Thucydides.getCurrentSession()` comme illustré ici:

```
@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);
}
```

Puis, dans une étape invoquée ultérieurement dans le test, vous pouvez contrôler les valeurs sauvegardées dans la 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("pu

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

}
```

Si aucune variable correspondant au nom demandé n'est trouvée, le test échouera. La session de test est nettoyée au début de chaque test.

Chapter 17. Test dirigé par les données

17.1. Tests dirigés par les données en JUnit

Avec JUnit 4, vous pouvez utiliser le lanceur de test Parameterized pour exécuter des tests dirigés par les données. Dans Thucydides, vous utilisez `ThucydidesParameterizedRunner`. Ce lanceur est très semblable au lanceur de tests JUnit Parameterized, excepté que vous utilisez l'annotation TestData pour fournir les données de test et que vous pouvez utiliser toutes les autres annotations Thucydides (`@Managed`, `@ManagedPages`, `@Steps`, etc.). Ce lanceur de tests générera également des rapports spécifiques Thucydides HTML et XML pour les tests exécutés.

Un exemple de test Thucydides dirigé par les données est montré ci-dessous. Dans ce test, vous vérifiez que des valeurs correctes d'âge et de couleur préférée sont acceptées sur la page d'une application (imaginaire). Pour tester cela, nous utilisons plusieurs combinaisons d'âges et de couleur préférée indiquées par la méthode testData(). Ces valeurs sont représentées comme des variables d'instance dans la classe de test instanciée via le constructeur.

```
@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);
    }
}
```

17.2. Génération de rapport sur des tests web dirigés par les données

Quand vous générez des rapports sur des tests web dirigés par les données, ces rapports affichent la totalité des sorties et des captures d'écran pour chaque série de données. Un rapport global des histoires pour les tests dirigés par les données est affiché, avec un cas de test pour chaque ligne des données de test. Les données de test utilisées pour chaque test sont affichées dans le rapport.

17.3. Exécuter des tests dirigés par les données en parallèle

Les tests web dirigés par les données peuvent être longs, spécialement si vous devez naviguer dans une page particulière avant de tester une valeur de champ différente à chaque fois. Dans la plupart des cas, cependant, ceci est nécessaire car il n'est pas sûr de faire des suppositions sur l'état d'une page web après un test dirigé par les données précédemment effectué. Un moyen efficace de les accélérer, cependant, consiste à les exécuter en parallèle. Vous pouvez configurer `ThucydidesParameterizedRunner` pour que les tests s'exécutent en parallèle en utilisant l'annotation Concurrent.

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

Par défaut, ceci exécutera les tests de manière concurrente, en utilisant deux proccessus par coeur CPU. Si vous voulez régler finement le nombre de processus utilisés, vous pouvez le faire en précisant la propriété d'annotation *threads*.

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

Vous pouvez également exprimer cela comme valeur relative au nombre de processeurs disponibles. Par exemple, pour exécuter 4 processus par CPU, vous pourriez indiquer ce qui suit:

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

17.4. Tests dirigés par les données en utilisant des fichiers CSV

Thucydides vous permet d'effectuer des tests dirigés par les données en utilisant des données de test dans un fichier CSV. Vous enregistrez vos données de test dans un fichier CSV (par défaut avec des colonnes séparées par des virgules) et dont la première ligne se comporte comme un en-tête:

```
        NAME,AGE,PLACE OF BIRTH
        Jack Smith, 30, Smithville
        Joe Brown, 40, Brownville
```

```
        Mary Williams, 20, Williamsville
```

Ensuite, créez une classe de test contenant les propriétés qui correspondent aux colonnes des données de test. Chaque propriété doit être une propriété au sens JavaBean, avec un getter et un setter correspondant. La classe de test contiendra typiquement un ou plusieurs tests qui utiliseront ces propriétés comme paramètres pour l'étape de test ou les méthodes de l'objet page.

La classe contiendra également l'annotation **@UseTestDataFrom** pour indiquer où trouver le fichier CSV (celui-ci peut être soit un fichier dans le classpath, soit un chemin de fichier relatif ou absolu - mettre la série de données dans le classpath (par exemple dans `src/test/resources`) rend les tests davantage portables).

Vous pouvez également utiliser l'annotation **@RunWith** ou tout autre annotation Thucydides usuelle (`@Managed`, `@ManagedPages` et `@Steps`).

Un exemple de ce type de classes est montré ici:

```
        @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;
        }
    }
```

Vous pouvez également indiquer plusieurs chemins d'accès de fichiers séparés par des séparateurs de chemin - deux points, point virgule ou virgule. Par exemple :

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

Vous pouvez également configurer un répertoire arbitraire en utilisant la propriété système `thucydides.data.dir` puis vous y référer via la variable `$DATADIR` dans l'annotation.

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

Chaque ligne de données de test nécessite d'être distinguée dans les rapports générés. Par défaut, Thucydides appellera la méthode `toString()`. Si vous fournissez une méthode publique retournant une chaîne et annotée par `@Qualifier`, c'est cette méthode qui sera utilisée pour distinguer les séries de données. Elle doit renvoyer une valeur unique pour chacune des séries de données.

Le lanceur de tests créera une nouvelle instance de cette classe pour chaque ligne de données du fichier CSV, affectant les propriétés aux valeurs correspondantes des données de test. Ainsi, quand nous exécuterons ce test, nous obtiendrons une sortie telle que celle-ci:

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

Il y a quelques points à noter. Les colonnes du fichier CSV sont converties en nom de propriété en Camel Case (ainsi, "NAME" devient `name` et "PLACE OF BIRTH" devient `placeOfBirth`). Puisque nous testons des applications web, tous les champs devraient être des chaînes de caractères.

Si certaines des valeurs de champ contiennent des virgules, vous devrez utiliser un séparateur différent. Vous pouvez utiliser l'attribut **separator** de l'annotation **@UseTestDataFrom** pour indiquer ce séparateur alternatif. Par exemple, les données suivantes utilisent un point-virgule comme séparateur:

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

Pour exécuter nos tests avec ces données, nous devrons utiliser une classe de test telle que la suivante:

```
    @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;
            }
        }
```

Ceci produira une sortie telle que celle-ci:

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

Le support d’Excel sera ajouté dans une future version. Cependant, si vous enregistrez vos données sous forme CSV, il devient facile de suivre les modifications des données de test dans votre système de contrôle de version.

17.5. Utiliser des tests dirigés par les données pour des étapes individuelles

Parfois, vous voulez utiliser le test dirigé par les données au niveau d'une étape, plutôt qu'au niveau du test. Par exemple, vous pourriez vouloir naviguer dans un écran donné de l'application puis essayer différentes combinaisons de données ou boucler sur une séquence d'étapes avec des données provenant d'un fichier CSV. Ceci évite d'avoir à ré-ouvrir le navigateur à chaque ligne de données.

Vous pouvez faire cela en ajoutant des valeurs de propriété à vos fichiers d'étapes. Considérez le fichier d'étapes suivant:

```
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() {
                ...
        }
}
```

Le groupe d'étapes `enter_personal_details` utilise les champs étape pour exécuter les étapes `enter_name_and_age` et `enter_address`. Nous voulons récupérer ces données depuis un fichier CSV et boucler sur l'étape **enter_personal_details** pour chaque ligne de données.

Nous faisons cela en utilisant la méthode `withTestDataFrom()` de la classe **StepData**:

```
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_
    }
}
```

Ceci va appeler `data_driven_test_step()` plusieurs fois, injectant à chaque fois des données dans l'étape depuis le fichier `test-data/simple-data.csv`.

Vous pouvez également utiliser autant de fichiers que vous le souhaitez, y compris pour un même test. Vous pouvez également utiliser le même fichier de données pour plus d'une étape de test. Rappelez-vous que seules les propriétés correspondant aux colonnes du fichier CSV seront instanciées - les autres seront ignorées:

```
@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).en
        }
    }
```

Au fait, nous devons utiliser `ThucydidesRunner` pour les cas de tests au lieu de `ThucydidesParameterizedRunner`.

Notez que, comme raccourci, vous pouvez omettre les méthodes setters et ne déclarer que les champs nécessaires comme publics. De cette façon, la classe d'étapes montré ci-dessus pourra être réécrite comme suit:

```
    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 18. Exécuter les tests Thucydides dans des batchs parallèles

Les tests web sont de bons candidats pour les tests concurrents, en théorie du moins, mais l'implémentation peut être tordue. Par exemple, bien qu'il soit assez facile de configurer à la fois JUnit et easyb pour exécuter des tests en parallèle, exécuter plusieurs instances WebDriver de Firefox en parallèle sur le même affichage, par exemple, à tendance à devenir non fiable.

La solution naturelle dans ce cas est de diviser les tests web en batchs plus petits et d'exécuter chaque batch sur une machine différente et/ou sur des affichages virtuels différents. Quand chaque batch est terminé, les résultats peuvent être récupérés et agglomérés dans les rapports de test finaux.

Cependant, séparer les tests en batchs à la main à tendance à devenir pénible et non fiable - il est facile d'oublier d'ajouter un nouveau test à un batch, par exemple, ou d'avoir des batchs inégalement chargés.

La dernière version de Thucydides vous permet de faire ceci automatiquement en distribuant vos cas de test équitablement dans des batchs de taille donnée. En pratique, vous exécutez une tâche de build pour chaque batch. Vous devez indiquer deux paramètres quand vous exécutez chaque build: le nombre total de batchs à exécuter (`thucydides.batch.count`) et le numéro du batch exécuté dans ce build (`thucydides.batch.number`).

Par exemple, ce qui suit va diviser les cas de tests en 3 batchs (`thucydides.batch.count`) et n'exécutera que le premier test de chaque batch (`thucydides.batch.number`):

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

Ceci ne fonctionnera qu'avec le mode JUnit. Cependant, cette fonctionnalité est également gérée par easyb (à partir de la version 1.5) quoiqu'avec des paramètres différents. Lors de l'utilisation de l'intégration easyb Thucydides, vous devrez également fournir les options équivalentes pour easyb:

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

Si vous avez à la fois des tests Thucydides easyb et JUnit, vous devrez indiquer les deux types d'options.

18.1. Stratégie de batch basée sur le décompte des tests

Par défaut, les cas de test sont répartis également entre les batchs. Ceci peut être inefficace si certains cas de tests ont plus de tests que d'autres. Dans de telles situations, une stratégie de batch différente, `DIVIDE_BY_TEST_COUNT`, peut être définie en utilisant la propriété système `thucydides.batch.strategy`. Cette stratégie va distribuer équitablement les cas de tests entre les batchs en se basant sur le nombre de méthodes de test dans chacun des cas de test.

```
mvn verify -Dthucydides,batch.strategy=DIVIDE_BY_TEST_COUNT -Dthucydides.batch.count=3 -
```

Chapter 19. Fonctionnalités expérimentales

19.1. Intégration avec FluentLineum

Vous pouvez utiliser l'API agile de FluentLenium [https://github.com/FluentLenium/FluentLenium] avec Thucydides. La meilleure façon d'utiliser FluentLenium dans Thucydides est d'utiliser ThucydidesFluentAdapter qui est disponible dans le PageObject. Voici un exemple du même PageObject écrit en style traditionnel et avec 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();
            }
        };
    }
}
```

et avec 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();
    }
}
```

19.2. Raccourci pour la méthode element()

Une autre nouvelle fonctionnalité expérimentale introduit la possibilité de remplacer la méthode element() communément utilisée avec *$*, comme illustré dans les exemples suivants :

```
    ...
    @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()
    }
}
```

19.3. Réessayer des tests en échec

Il est parfois nécessaire de réessayer un test un échec. Ceci peut être réalisé en réglant la propriété système `max.retries` avec le nombre de fois que vous voulez que les tests en échec soient réessayés.

19.4. Utiliser des méthodes d'étape pour documenter des cas de test

Les méthodes dans la bibliothèque d'étape peuvent être utilisées pour fournir de la documentation complémentaire pour les scénarios de test dans les rapports Thucydides. Vous pouvez passer du texte HTML valide en paramètre aux méthodes @Step dans la bibliothèque d'étapes. Ce texte apparaîtra sous forme de texte mis en forme dans les rapports sur la page des détails de l'étape. La capture d'écran suivante illustre ceci.

Figure 19.1. du texte formaté HTML, s'il est passé à une méthode d'étape, sera affiché comme illustré ici. Ceci peut être utile pour annoter ou documenter les tests avec des informations utiles.

Ceci est obtenu en créant une méthode fictive @Step appelée description prenant un paramètre chaîne de caractères. À l'exécution, les tests fournissent à cette méthode du texte au format html en paramètre.

```
...
@Step
public void description(String html) {
    //ne fait rien
}

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 20. Lectures pour aller plus loin

Articles

- Dr. Dobb’s Journal. Project of the Month: Thucydides [http://drdobbs.com/open-source/232300277/], décembre 2011.
- JavaWorld. Acceptance test driven development for web applications [http://www.javaworld.com/javaworld/jw-08-2011/110823-atdd-for-web-apps.html], août 2011.
- JavaWorld. Selenium 2 and Thucydides for ATDD [http://www.javaworld.com/javaworld/jw-10-2011/111018-thucydides-for-atdd.html], août 2011.