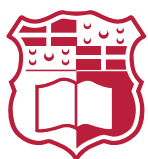# Web Test Automation, BDD and Model Based Testing

Andre' Vella

February 2024

Submitted in partial fulfilment of the requirements for the unit CPS3230.

L-Università ta' Malta
Faculty of Information & Communication Technology

Contents

1 Web Test Automation and BDD

1.1 Website of Choice

We consider the shopping website of a Maltese computers shop, `Klikk`. This shopping website satisfies the following criteria:

- ✓ Contains login and logout functionality.

- ✓ Contains product categories.

- ✓ Product search functionality.

- ✓ Allows for addition and removal of items from cart.

1.2 Scenarios

The site passes the following acceptance criteria. **NB:** The term "book" does not fit the domain of the chosen website, hence the term is replaced by "phone".

```
Scenario -> Reachability of product categories (at least 5 categories)
Given I am a user of the website
When I visit the news website
And I click on the [category-name] category
Then I should be taken to [category-name] category
And the category should show at least [num-products] products
When I click on the first product in the results
Then I should be taken to the details page for that product

Scenario -> Search functionality
Given I am a user of the website
When I search for a product using the term "phone"
Then I should see the search results
And there should be at least 5 products in the search results
When I click on the first product in the results
Then I should be taken to the details page for that product
```

Listing 1.1 Scenarios to test.

1.2.1 Feature Files

The scenarios in Listing 1.1 are defined in two separate *.feature* files which are stored in *src/test/resources/features*.

- *search_functionality.feature*

```
1  Feature: Search Functionality
2
3    In order to help me decide which phone to get
4    As a user of the technology shop
5    I want to be able to have multiple choices of phones that are
       available
6    on the market and be able to see the respective phone's specs/
       details
7
8    @SearchFunctionalityTest
9    Scenario: Search functionality:
10     Given I am a user of the website
11     When I search for a product using the term "phone"
12     Then I should see the search results
13     And there should be at least 5 products in the search results
14     When I click on the first product in the results
15     Then I should be taken to the details page for that product
```

Listing 1.2 Search Functionality Feature

- *reachability_of_product_categories.feature*

```
1  Feature: Reachability of Product Categories
2
3    In order to help me view items of a particular category
4    As a user of the technology shop
5    I want to be able to choose a category and see products related to
       that category
6
7    @CategoriesTest
8    Scenario Outline: Reachability of product categories:
9      Given I am a user of the website
10     When I visit the store website
11     And I click on the <category-name> category
12     Then I should be taken to <category-name> category
13     And the category should show at least <num-products> products
14     When I click on the first product in the results
15     Then I should be taken to the details page for that product
16
17     Examples:
18        | category-name       | num-products |
19        | DESKTOP_AND_LAPTOPS | 290          |
20        | PHONES_AND_TABLETS  | 1500         |
21        | COMPUTING           | 1300         |
22        | GAMING              | 340          |
23        | HOME_AND_LIFE       | 800          |
```

```
24        | ACCESORIES           | 900          |
25        | DEALS                | 200          |
```

Listing 1.3 Reachability of Product Categories

Scenarios level tags `@CategoriesTest` and `@SearchFunctionalityTest` are used to allow a test runner to run a select feature independent of the others (see section 1.5).

1.3 Page Object Design Pattern

1.3.1 Generic Class

The website being tested some exhibited erratic behavior which may be due to the choice of the front-end client framework used. I attempted to use `WebDriverWait` to wait for an element to become visible, clickable, etc. However, after a certain element became visible, attempting to perform an action on it resulted in it becoming stale again.

To address this issue and attempt to make the system under test more testable, a generic class named `WebDriverMethods` (located in package `edu.um.cps3230.pageobjects`) was created. This class handles the required interactions with web elements at the point in time when an element becomes visible and intractable, with an imposed wait timeout. Additionally, to handle error 504 (possibly caused by temporary request overload), the current page is refreshed a maximum of 3 times.

```java
package edu.um.cps3230.pageobjects;

import org.openqa.selenium.*;
import org.openqa.selenium.support.ui.WebDriverWait;

import java.time.Duration;

/**
 * Generic Class to interact with a website.
 *
 * @author Andre
 */
public class WebDriverMethods {
    Duration timeoutInSeconds = Duration.ofSeconds(40);
    private final WebDriver webDriver;
    private int count = 0;
    private final int maxTries = 3;
```

```java
18
19      /**
20       * Constructs a new WebDriverMethods objects to interact with a website
        given a WebDriver.
21       *
22       * @param webDriver
23       */
24      public WebDriverMethods(WebDriver webDriver) {
25          this.webDriver = webDriver;
26      }
27
28      private void _click(By element) {
29          new WebDriverWait(webDriver, timeoutInSeconds)
30                  .ignoring(StaleElementReferenceException.class,
        ElementNotInteractableException.class)
31                  .until((WebDriver d) -> {
32                      d.findElement(element).click();
33                      return true;
34                  });
35      }
36
37
38      private void _sendKeys(By element, String charSequence) {
39          new WebDriverWait(webDriver, timeoutInSeconds)
40                  .ignoring(StaleElementReferenceException.class,
        ElementNotInteractableException.class)
41                  .until((WebDriver d) -> {
42                      d.findElement(element).sendKeys(charSequence);
43                      return true;
44                  });
45      }
46
47
48      private String _getText(By element) {
49          return new WebDriverWait(webDriver, timeoutInSeconds)
50                  .ignoring(StaleElementReferenceException.class,
        ElementNotInteractableException.class)
51                  .until((WebDriver d) -> d.findElement(element).getText());
52      }
53
54      private String _getText(By element, int index) {
55          return new WebDriverWait(webDriver, timeoutInSeconds)
56                  .ignoring(StaleElementReferenceException.class,
        ElementNotInteractableException.class)
57                  .until((WebDriver d) -> d.findElements(element).get(index).
        getText());
58      }
```

4

```java
59
60     /**
61      * Locates an element and clicks it.
62      *
63      * @param element The element to be clicked.
64      */
65     public void click(By element) {
66         while (true) {
67             try {
68                 _click(element);
69                 break;
70             } catch (TimeoutException e) {
71                 //We try to refresh page in case of 504 gateway error
72                 if ((++count) == maxTries) {
73                     webDriver.navigate().refresh();
74                     count = 0; //reset counter
75                     throw e;
76                 }
77             }
78         }
79     }
80
81     /**
82      * Locates an element and inputs charSequence in it.
83      *
84      * @param element      The element located.
85      * @param charSequence The string sequence to be inputted in given
       element.
86      */
87     public void sendKeys(By element, String charSequence) {
88         while (true) {
89             try {
90                 _sendKeys(element, charSequence);
91                 break;
92             } catch (TimeoutException e) {
93                 //We try to refresh page in case of 504 gateway error
94                 if ((++count) == maxTries) {
95                     webDriver.navigate().refresh();
96                     count = 0; //reset counter
97                     throw e;
98                 }
99             }
100        }
101    }
102
103    /**
104     * Locates an elements and gets its visible text.
```

```
105        *
106        * @param element The element located.
107        * @return Returns the visible text of the given element.
108        */
109     public String getText(By element) {
110         while (true) {
111             try {
112                 return _getText(element);
113             } catch (TimeoutException e) {
114                 //We try to refresh page in case of 504 gateway error
115                 if ((++count) == maxTries) {
116                     webDriver.navigate().refresh();
117                     count = 0; //reset counter
118                     throw e;
119                 }
120             }
121         }
122     }
123
124     /**
125      * Gets the visible text of the (index+1) sub-element.
126      *
127      * @param element Element evaluated.
128      * @param index   Index of sub-element.
129      * @return Returns the visible text of the (index+1) sub-element of the
       given element.
130      */
131     public String getText(By element, int index) {
132         while (true) {
133             try {
134                 return _getText(element, index);
135             } catch (TimeoutException e) {
136                 //We try to refresh page in case of 504 gateway error
137                 if ((++count) == maxTries) {
138                     webDriver.navigate().refresh();
139                     count = 0; //reset counter
140                     throw e;
141                 }
142             }
143         }
144     }
145 }
```

Listing 1.4 Generic methods

To enable more understandable tests of the scenarios in Section 1.2, a number of page objects are identified. For each component, methods are constructed using the

`WebDriverMethods` methods, to interact with and query the respective component using generic actions shown in Listing 1.4. These methods render the tests more understandable since they eliminate the need for locating web elements in the test itself. Furthermore, if a component locator changes, then we can update a corresponding page object once instead of going through the tests that make use of the very same component and updating them one by one. These page objects are described in the following subsections.
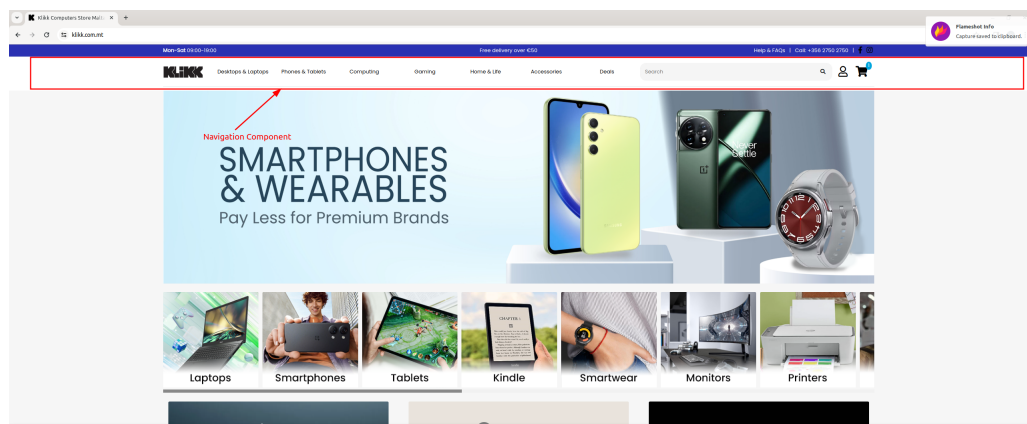
1.3.2   Navigation Component

Figure 1.1 Navigation component in red rectangle.

The following methods are written for the Navigation Component page object and are in class `NavigationComponent` (in package `edu.um.cps3230.pageobjects`):

- `void returnToHomePage()`: This method locates the home logo and clicks it to return to the home page.

- `void search(String searchVal)`: This method locates the search field and inputs the given string using the private methods `inputExpression(String expression)`.

- `void clickSectionById(Category category)`: This method locates a section in the navigation component and goes to its respective product view page. Each category has further categories as shown in Figure 1.2. We assume that the scenario test for the reachability of main categories; hence, this page driver method selects *ViewAllProducts* for a given main category, as shown in Figure 1.2.
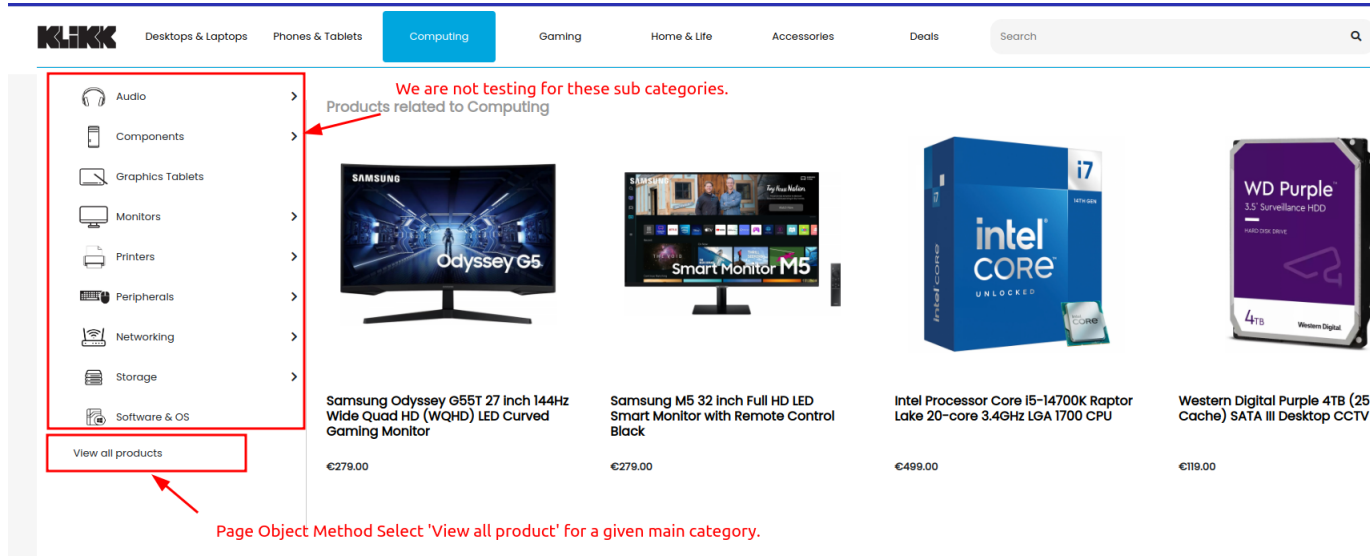
Figure 1.2 We assume that a scenario tests for the reachability of products of a main category such as the reachability of products related to Computing.

The parameter of this method is an enumerable (see Listing 1.5), where each enum instance represents a category with an associated button ID. This allows for easy updates without the need to modify the method itself.

```java
package edu.um.cps3230.pageobjects;

/**
 * Enum representing the main categories of the web store.
 */
public enum Category {
    /**
     * Represents desktop and laptops category.
     */
    DESKTOPS_AND_LAPTOPS("pills-_desktops-laptops-tab", "_desktops-laptops"),
    /**
     * Represents phones and tablets category.
     */
    PHONES_AND_TABLETS("pills-_phones-tablets-tab", "_phones-tablets"),
    /**
     * Represents category computing.
     */
    COMPUTING("pills-_computing-tab", "_computing"),
    /**
     * Represents computing category.
     */
    GAMING("pills-_gaming-tab", "_gaming"),
    /**
```

```
24      * Represents home and life category.
25      */
26   HOME_AND_LIFE("pills-_home-life-tab", "_home-life"),
27    /**
28     * Represents accessories category.
29     */
30   ACCESSORIES("pills-_accessories-tab", "_accessories"),
31    /**
32     * Represents deals category.
33     */
34   DEALS("pills-_deals-tab", "_deals");
35
36   public final String buttonId;
37   public final String queryParam;
38
39   private Category(String button_id, String queryParam) {
40       this.buttonId = button_id;
41       this.queryParam = queryParam;
42   }
43 }
```

Listing 1.5 Product categories

- `void login(String email, String password)`: This method locates the sign-in button, email, and password fields, and logs in to the account using the given email and password.

- `void logout()`: This method locates the logout icon and clicks it.

- `void goToPurchasePage()`: This method clicks on the shopping cart icon, leading to the purchase page.
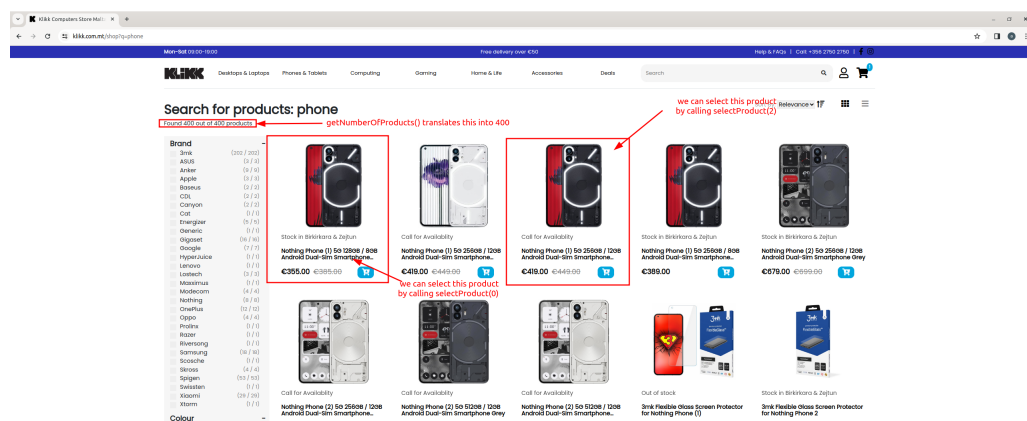
1.3.3 Product View Component

Figure 1.3 Products view component.

9

The page object identified in Figure 1.3, allows the user to see the products related to a category or search query. The following methods are written for the Products View Component page object and are in class `ProductViewComponent`(in package `edu.um.cps3230.pageobjects`):

- `int getNumberOfProducts()`: This method gets the number of products in this page object.

- `String selectProduct(int index)`: This method clicks on the $(index + 1)^{th}$ product in this page object. The name of the product clicked is returned.

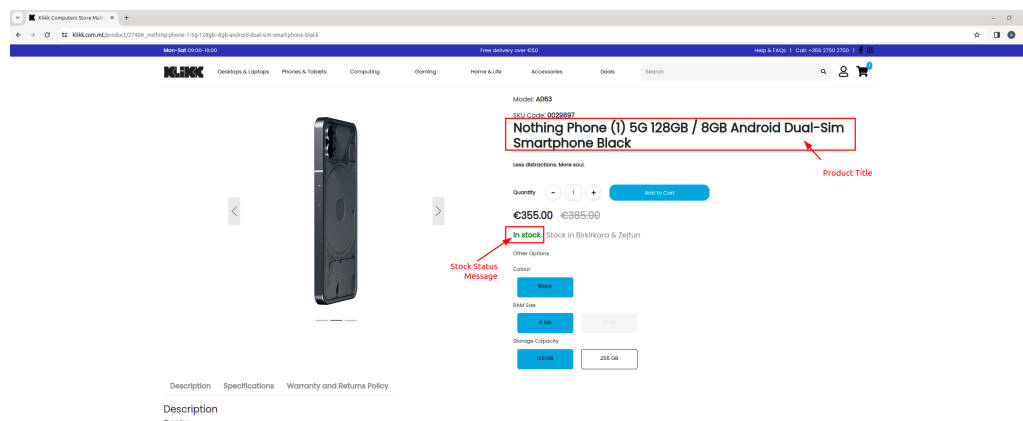### 1.3.4   Product Details Component

Figure 1.4 Product Details Component

The page object identified in Figure 1.4, allows the user to see the details of a selected product. The following methods are written for the Product Details Component page object and are in class *ProductDetailsComponent* in package *edu.um.cps3230.pageobjects*:

- `String getProductTitle()`:This method retrieves the title of a selected product by locating it using the class name. For instance, in the case of Figure 1.4, this method would return "Nothing Phone (1) 5G 128GB / 8GB Android Dual-Sim Smartphone Black".

- `StockStatus getStockStatus`: This method returns an enumerable instance of *StockStatus* (see Listing 1.6).

```
1  package  edu.um.cps3230.pageobjects;
2
3  /**
4   * Enum representing the possible stock statuses of the web store.
5   */
```

```java
6  public enum StockStatus {
7      /**
8       * Represents that a product is in stock.
9       */
10     IN_STOCK("In stock"),
11     /**
12      * Represents that a product is out of stock,
13      */
14     OUT_OF_STOCK("Out of stock"),
15     /**
16      * Represents that a product is coming soon.
17      */
18     COMING_SOON("Coming Soon."),
19     /**
20      * Represents that a product may be available.
21      */
22     CHECK_FOR_AVAILABILITY("Check for availability");
23     public final String stock_status_message;
24
25     private StockStatus(String stock_status_message) {
26         this.stock_status_message = stock_status_message;
27     }
28 }
```

Listing 1.6 All the possible stock statuses.

In the following section, we bridge these page object methods with steps of a scenario.

1.4   Step Definition

The code presented in this section can be found in class `StepDefinitions` (in package `test.store.webtestautomation`).

For example, the statement "Given I am a user of the website" is interpreted as the action of a user launching a web browser, and the statement "When I visit the store website" is interpreted as the user navigating to the website. The following actions are given in Listing 1.7.

```java
1  @Given("I am a user of the website")
2  public void iAmAUserOfTheWebsite() {
3      webDriver = new ChromeDriver();
4      //maximise to ensure that the page does not render the burger menu
5      webDriver.manage().window().maximize();
6      searchComponent = new NavigationComponent(webDriver);
7      navigationComponent = new NavigationComponent(webDriver);
```

```
8      productDetailsComponent = new ProductDetailsComponent(webDriver);
9      productsViewComponent = new ProductsViewComponent(webDriver);
10 }
11
12 @When("I visit the store website")
13 public void iVisitTheStoreWebsite() {
14     webDriver.get("https://www.klikk.com.mt");
15 }
```

Listing 1.7 Initial user behaviour.

1.4.1 Page Objects make test cleaner

Listings 1.8, 1.9, and 1.10, are examples of where the 3 identified page objects come in handy.

```
1 @When("I click on the {category} category")
2 public void iClickOnTheCategory(Category category) {
3     navigationComponent.clickSectionById(category);
4 }
```

Listing 1.8 Using navigation component in test.

```
1 @And("the category should show at least {int} products")
2 public void theCategoryShouldShowAtLeastProducts(int leastNumberOfProducts)
3 {
4     numberOfResults = productsViewComponent.getNumberOfProducts();
5     Assertions.assertTrue(leastNumberOfProducts <= numberOfResults);
6 }
```

Listing 1.9 Using product details component in test.

```
1 @Then("I should be taken to the details page for that product")
2 public void iShouldBeTakenToTheDetailsPageForThatProduct()
3 {
4     Assertions.assertEquals(nameOfFirstProduct, productDetailsComponent.
     getProductTitle());
5 }
```

Listing 1.10 Using products view component in test.

Note how assertions are used to verify the intended behavior in Listings 1.9 and 1.10. Listing 1.10 shows the use of a custom parameter type category which is discussed in the next subsection.

1.4.2 Parameterised type

In Listing 1.8, we introduce a type *category*. This allows Cucumber expressions to have more strongly typed values by using regex expressions. In the same test file, there is a

method shown in Listing 1.11 that matches values of the scenario's category from the examples in the scenario outline of the "Reachability of Product Categories" (see Listing 1.3) with a regex expression and returns the corresponding enumerable from Listing 1.5. If an unknown category is introduced in the examples, this will fail to match the regex expression and cucumber will raise a compilation error.

```
1 @ParameterType("DESKTOPS_AND_LAPTOPS|PHONES_AND_TABLETS|COMPUTING|GAMING|
      HOME_AND_LIFE|ACCESSORIES|DEALS")
2 public Category category(String categoryName) {
3     return Category.valueOf(categoryName);
4 }
```

Listing 1.11 Paremeter type in Cucumber

1.5   Test Runner

For running the features shown in Section 1.2.1, two independent runners are created. Each runner identifies the required feature to run by the use of feature tags.

```
1 package test.store.webtestautomation;
2
3 import io.cucumber.junit.Cucumber;
4 import io.cucumber.junit.CucumberOptions;
5 import org.junit.runner.RunWith;
6
7 @RunWith(Cucumber.class)
8 @CucumberOptions(
9         features = "src/test/resources/features",
10        tags = "@SearchFunctionalityTest"
11
12 )
13 public class TestSearchFunctionalityRunner {
14 }
```

Listing 1.12 Runner for Search Functionality Test

```
1 package test.store.webtestautomation;
2
3 import io.cucumber.junit.Cucumber;
4 import io.cucumber.junit.CucumberOptions;
5 import org.junit.runner.RunWith;
6
7 @RunWith(Cucumber.class)
8 @CucumberOptions(
9         features = "src/test/resources/features",
10        tags = "@CategoriesTest"
```

```
11
12 )
13 public class TestReachabilityOfProductCategoriesRunner {
14 }
```

Listing 1.13 Runner for Reachability of Product Test

1.6   Conclusion

Although the website may have displayed erratic behavior at times, such as elements becoming stale instantaneously upon locating, the use of `WebDriverWait` with `until` in Section 1.3.1 helped to address these issues. Overall, with a little experimentation, the test ran successfully, and the identification of the relevant page objects and their respective methods made the test more understandable.

1.6.1   Recordings

Link containing recodings for each test runner: `https://github.com/andimon/uni-cps3230-web-test-automation-and-model-based-testing/tree/main/recordings`.

2 Model Based Testing

In this section, an automaton is developed and implemented. The model demonstrates a practical journey that a user might go through when using the online store. The automaton is broken down into sub-automata to help facilitate implementation by taking a phased approach. Reachability of all the discussed sub-automata from the starting state was kept in mind during the design phase.

In the following section, we describe the starting state, alluding to the transitions and next states in the following sections.

2.1 Starting State

The following states and state variables are identified:

- $States = \{HOME\_PAGE\}$

- $Vars = \{\langle isCartEmpty \rangle, \langle productsInPage \rangle, \langle inStock \rangle, \langle loggedIn \rangle\}$

$$S \quad \begin{array}{c} \langle isCartEmpty \rangle, \\ \langle productsInPage \rangle, \\ \langle inStock \rangle, \\ \langle loggedIn \rangle \end{array}$$
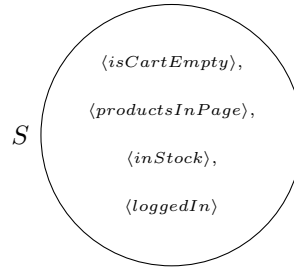
Figure 2.1 Representing a state where $S \in States$

It is assumed that the first page that the user lands on is the home page. Hence we set $HOME\_PAGE$ as the starting state. The initial state of the variables is:

- $\langle isCartEmpty \rangle =$ true: A fresh instance of the web store does not have products in the cart.

- $\langle productsInPage \rangle = 0$: It is assumed that there are no selectable products on the home page. Occasionally, the store may display selectable products on their homepage, but they are not really testable since the components are frequently changing. Hence, we opt out from considering these products in the model.

- $\langle inStock \rangle =$ false: Since we are not viewing a product, we set this variable to false.

- $\langle loggedIn \rangle$ = false: It is assumed that the first interaction with the website will be in a logged-out state.

Figure 2.2 Starting state

2.1.1 Implementation

We set up the current state by appropriately declaring the initial values of the model test class, as shown in Listing 2.1. The ENUM `CurrentPage` is used to enumerate all the possible states, as shown in Listing 2.2.

```
1  public class TestWebStoreTester implements FsmModel {
2
3      private final Random random = new Random();
4      private final int testSequenceLength = 1500;
5      private static WebDriver webDriver = new ChromeDriver();
6      WebStoreOperator systemUnderTest = new WebStoreOperator(webDriver);
7
8      // State and state variables
9      CurrentPage currentPage;
10     boolean isCartEmpty, inStock, loggedIn;
11     int productsInPage;
```

Listing 2.1 Initial declaration should represent starting state.

```
1  package test.store.modelbasedtesting;
2
3  public enum CurrentPage {
4      HOME_PAGE("https://www.klikk.com.mt/"),
5      PRODUCTS_VIEW_PAGE("https://www.klikk.com.mt/shop"),
6      PRODUCT_DETAILS_PAGE("https://www.klikk.com.mt/product"),
7      PURCHASE_PAGE("https://www.klikk.com.mt/cart");
8
9      public final String url;
10
11     private CurrentPage(String url) {
12         this.url = url;
13     }
14 }
```

Listing 2.2 ENUM representing all possible states.

16

2.1.2 Asserting State

Note that for each page we have an associated state we have a URL. This helps us to assert state by use of the following assertions:

- Asserting when a current page is known to be the homepage:
  ```
  Assertions.assertEquals(webDriver.getCurrentUrl(), currentPage.url);
  ```

- Asserting when a current page is not the homepage:
  ```
  Assertions.assertTrue(webDriver.getCurrentUrl().contains(currentPage.url));
  ```

Furthermore, for each method in the system under test, we keep track of variables to compare with the model. For instance, let us consider the login and logout methods in the SUT.

```
1 public class WebStoreOperator {
2 public boolean isCartEmpty, inStock, loggedIn;
3 public void login(String user, String pass) {
4     navigationComponent.login(user, pass);
5     loggedIn = true;
6 }
7 /**
8  * Logout
9  * Sometimes after a sequence of operations, when a user logs in he will
     remain logged out. Hence:
10  * If we cannot manage then check if user is already logged out.
11  * If the user is not logged out then this method will throw an exception.
12  */
13 public void logout() {
14     try {
15         //try to log out
16         navigationComponent.logout();
17     } catch (TimeoutException e) {
18         //confirm that user is already logged out
19         webDriver.findElement(By.id("login_icon_blue"));
20     }
21     loggedIn = false;
22 }
```

Listing 2.3 Updating state

Then, in the model action, we assert the SUT's current state with the variables.

```
1 Assertions.assertEquals(loggedIn, systemUnderTest.loggedIn);
```

Listing 2.4 Assert the state of logged in with the model.

2.1.3 Reset

The ModelJUnit interface requires us to implement a method called `reset(bool)`, which should randomly set the state to the initial state. Moreover, when the parameter is true, the system under test should also be set to its original state. Listing 2.1.3 is the implementation of the reset method for our scenario where the system under test is reset by quitting the current webdriver and initiating a new one.

```java
@Override
public void reset(boolean testing) {
    //update state and state variables
    webDriver.quit();
    currentPage = CurrentPage.HOME_PAGE;
    isCartEmpty = true;
    productsInPage = 0;
    inStock = false;
    loggedIn = false;
    webDriver = new ChromeDriver();
    webDriver.manage().window().maximize();
    webDriver.get("https://www.klikk.com.mt");
    systemUnderTest = new WebStoreOperator(webDriver);
    //reset SUT
    if (testing) {
        systemUnderTest = new WebStoreOperator(webDriver);
    }
}
```

2.2 Returning to Home

In any state S, excluding the home state, i.e., $S \in STATES \setminus HOME\_PAGE$, and for any state variables, the user can return to the home page. This is achieved by calling a method that locates the home button and clicks it. Since we are assuming that there are no products on the Home Page (see Section 2.1), we then update the value of any current state variable (represented by y in Figure 2.3), $\langle productsInPage \rangle$, to 0. Similarly, the value of $inStock$ (represented by w in Figure 2.3) is set to false, as the homepage does not specify details of a particular product. The value of the state variable $\langle loggedIn \rangle$ (represented by z in Figure 2.3) remains unchanged.
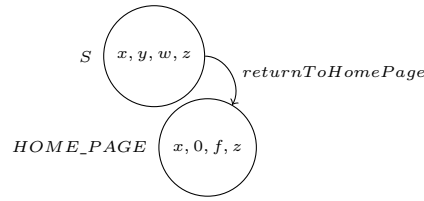
Figure 2.3 $S \in STATES \setminus \{HOME\_PAGE\}$ and $x, y \in \{t, f\}$

2.2.1 Implementation

To check if the current state is as just described, we use a guard.

```
public boolean returnToHomePageGuard() {
    //check expected current state and variables
    return currentPage != CurrentPage.HOME_PAGE;
}
```

The action is implemented as follows.

```
public @Action void returnToHomePage() {
    //action
    systemUnderTest.returnToHome();
    //update state and state variables
    currentPage = CurrentPage.HOME_PAGE;
    productsInPage = 0;
    inStock = false;
    //Assert State
    Assertions.assertEquals(webDriver.getCurrentUrl(), currentPage.url);
}
```

2.3 Selecting a product

Products can be selected from the products view page (if it is not empty). Upon selection, the user is redirected to the Product Details page of the selected product. The action we are modeling is the selection of a random product from a list of products. Furthermore, the store sometimes displays products on their homepage; however, this transition does not cover when the current state is the home page due to the dynamic nature of the homepage (hence not really testable). This action is represented by the sub-automaton shown in Figure 2.4, where the state variables are as follows:

- $y \in \{t, f\}$ represents the value of variable $\langle isCartEmpty \rangle$. The action of selecting a product does not change this variable.

- In order for a product to be selected then there should be products in the current page. Hence the variable $\langle productsInPage \rangle$ greater than 0. Since the next state does not have any selectable products in page the variable is set to 0.

19

- It is assumed that the products view page does not provide details about a particular product; hence, at the current state, the variable $\langle inStock \rangle$ is assumed to be false. This state variable is set to true in the next state if the product is out of stock; otherwise, it remains set to false.

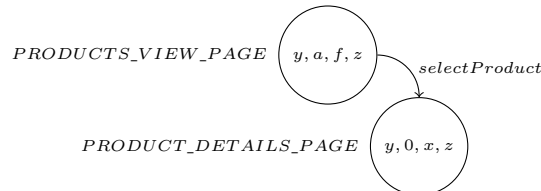- The variable $\langle loggedIn \rangle$ remains unchanged.

Figure 2.4 Selecting a product where $x, y, z \in \{t, f\}$, $a > 0$, ,and x is set to true if the select product is in stock and false otherwise

2.3.1 Implementation

To check the current state is as just described we make use of a guard.

```
public boolean selectProductGuard() {
    //check expected current state and variables
    return (currentPage==CurrentPage.PRODUCTS_VIEW_PAGE) && (productsInPage
    >0) && (!inStock);
}
```

We use **random testing technique** to select a random product randomly.

```
public @Action void selectProduct() {
    //action
    systemUnderTest.selectProduct(random.nextInt(productsInPage));
    //update state and state variables
    currentPage = CurrentPage.PRODUCT_DETAILS_PAGE;
    productsInPage = 0;
    inStock = systemUnderTest.getStockStatus() == StockStatus.IN_STOCK;
    //Assert State
    Assertions.assertTrue(webDriver.getCurrentUrl().contains(currentPage.
    url));
    Assertions.assertEquals(productsInPage,systemUnderTest.productsInPage);
    Assertions.assertEquals(inStock,systemUnderTest.inStock);
}
```

2.4 Adding Product To Cart

We can add a product in the current state $PRODUCT\_DETAILS\_PAGE$. Since there are no selectable products in $PRODUCT\_DETAILS\_PAGE$, the variable

$\langle productsInPage \rangle$ is expected to be 0. It is assumed that the product can be added to the cart if and only if $\langle inStock \rangle$ is true. The variable $\langle isCartEmpty \rangle$ is set to false after the action.
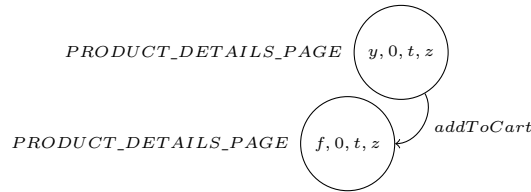

Figure 2.5 $y, z \in \{t, f\}$

2.4.1   Implementation

To check that the current state is as just described, we make use of a guard. We verify that the current page is the product details page. In the guard, we also check the value of $\langle productsInPage \rangle$ and $\langle inStock \rangle$.

```
public boolean addToCartGuard() {
    //check expected current state and variables
    return (currentPage == CurrentPage.PRODUCT_DETAILS_PAGE) && (
    productsInPage == 0) && inStock;
}
```

The action is implemented as follows.

```
public @Action void addToCart() {
    //action
    systemUnderTest.addToCart();
    //update state and state variables
    isCartEmpty = false;
    //Assert State
    Assertions.assertTrue(webDriver.getCurrentUrl().contains(currentPage.
    url));
    Assertions.assertEquals(isCartEmpty,systemUnderTest.isCartEmpty);
}
```

Listing 2.5 Add to cart

2.5   Going to Purchase Page

The user can go to a dedicated page for viewing the products in the cart to update the cart or purchase the products. The user can access this page in any given state and with any state variables. Since the purchase page does not include any selectable products and does not describe a specific product, the variables $\langle productsInPage \rangle$ and

21

$\langle inStock \rangle$ are set to 0 and $false$, respectively. The state variables $\langle isCartEmpty \rangle$ and $\langle loggedIn \rangle$ remain the same.
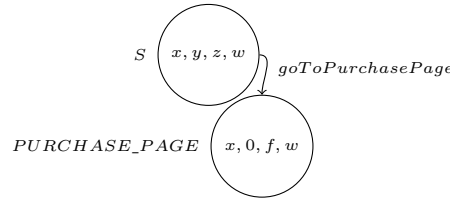

Figure 2.6 Action of going to purchase page where
$S \in STATES \setminus \{PURCHASE\_PAGE\}$, $x, z, w \in \{t, f\}$ and y is an integer

2.5.1 Implementation

To check that the current state is as just described, we make use of a guard.

```java
public boolean goToPurchasePageGuard(){
    return currentPage != CurrentPage.PURCHASE_PAGE;
}
```

The transition is implemented via a *modeljunit* action where the current state is assigned $HOME\_PAGE$, representing the next state. State variables $\langle productsInPage \rangle$ and $\langle inStock \rangle$ are set to 0 and false respectively, while the other variables are left untouched.

```java
public @Action void goToPurchasePage() {
    //action
    systemUnderTest.goToPurchasePage();
    //update state and state variables
    currentPage = CurrentPage.PURCHASE_PAGE;
    productsInPage = 0;
    inStock = false;
    //Assert State
    Assertions.assertTrue(webDriver.getCurrentUrl().contains(currentPage.
    url));
    Assertions.assertEquals(inStock, systemUnderTest.inStock);
    Assertions.assertEquals(productsInPage,systemUnderTest.productsInPage);
}
```

2.6 Clear Cart

For the sake of simplicity, we consider the action of deleting all products from the cart instead of any number of products. The shopping cart can be cleared from the purchase page. We consider the following values for a state variable:

- $\langle isCartEmpty \rangle$ is false in the current state and true in the next state.

- $\langle productsInPage \rangle$ is 0 in both the current and next state since there are no selectable products on the purchase page.

- $\langle inStock \rangle$ is false in both the current state and the next state.

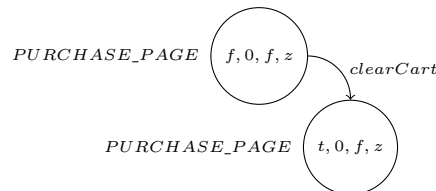- $\langle loggedIn \rangle$ remains unchanged.

$PURCHASE\_PAGE$ $\;f, 0, f, z\;$

$clearCart$

$PURCHASE\_PAGE$ $\;t, 0, f, z\;$

Figure 2.7 Action of deleting all items in a cart with $z \in \{t, f\}$.

2.6.1 Implementation

To check that the current state is as just described, we make use of a guard.

```
public boolean clearCartGuard() {
    //check expected current state and variables
    return currentPage == CurrentPage.PURCHASE_PAGE && !isCartEmpty && (
    productsInPage==0) && !inStock;
```

The transition is implemented via a *modeljunit* action where the current state is left untouched. $isCartEmpty$ is the only state variable that requires updating.

```
public @Action void clearCart() {
    //action
    systemUnderTest.clearCart();
    //update state and state variables
    isCartEmpty = true;
    //Assert State
    Assertions.assertTrue(webDriver.getCurrentUrl().contains(currentPage.
    url));
    Assertions.assertEquals(isCartEmpty,systemUnderTest.isCartEmpty);
}
```

2.7 Searching for a product

We will model the action of the user searching for a phone using the search field with the query `phone`. The user can search for a product from any webpage. Upon searching, the user is led to the products view page of the searched products. Since the products view page may contain products, the variable $\langle productsInPage \rangle$ is set to the number of products on the page so that the action `selectProduct` (see Section

2.3) can randomly select a product from the page. The variable $\langle inStock \rangle$ is set to false, since the $PRODUCTS_VIEW_PAGE$ is assumed to not have details about a particular product. All the other variables remain unchanged.
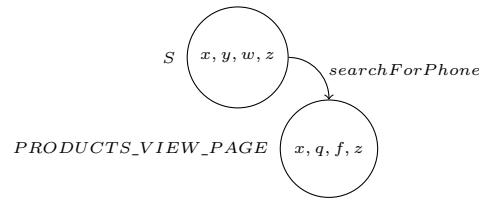

Figure 2.8 Action of search for a phone with $x, w, z \in \{t, f\}$, y, q are integers, and $s \in STATES$. q is set to the number of products in the current page.

2.7.1   Implementation

Unlike other transitions, this particular action does not need a guard. The action is implemented as follows.

```
public @Action void searchForPhone() {
    //action
    systemUnderTest.searchProduct("phone");
    //update state and state variables
    currentPage = CurrentPage.PRODUCTS_VIEW_PAGE;
    productsInPage = systemUnderTest.getNumberOfProducts();
    inStock = false;
    //Assert State
    Assertions.assertTrue(webDriver.getCurrentUrl().contains(currentPage.
    url));
}
```

2.8   Logging in and out

A user can log in when the state variable $\langle loggedIn \rangle$ is false and can log out when it is true. The other state variables remain unaffected. The log-in action can be carried out at any state; furthermore, the next state remains unchanged. The logout action is carried out from any state except $PRODUCT_DETAILS_PAGE$. When a users logs out from other states, they are returned to the $HOME_PAGE$ state. However, if the user logs out from a product details page, they end up on a product not found page, which is not handled by these tests.
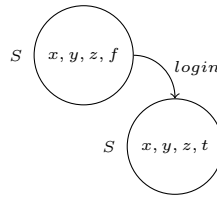
Figure 2.9 Action of logging in with $x, z \in \{t, f\}$, y is an integer, and $s \in STATES$.
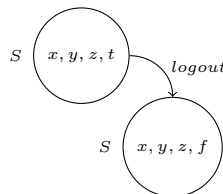

Figure 2.10 Action of logging out with $x, z \in \{t, f\}$ and $s \in STATES$.

The implementations of the guards and actions for logging in and logging out are given below.

```
1 public boolean loginGuard(){
2     return !loggedIn
3 }
```

Listing 2.6 login guard

```
1 public boolean logoutGuard(){
2     return loggedIn;
3 }
```

Listing 2.7 logout guard

A test user has been created and hardcoded. Another way to handle this is to set up a testing environment with testing credentials as environment variables.

```
1 public @Action void login() {
2     systemUnderTest.login("katijik879@grassdev.com", "test123!");
3     loggedIn = true;
4     //Assert State
5     Assertions.assertTrue(webDriver.getCurrentUrl().contains(currentPage.
      url));
6     Assertions.assertEquals(loggedIn, systemUnderTest.loggedIn);
7 }
```

Listing 2.8 login action

```
1 public @Action void logout() {
2     systemUnderTest.logout();
3     loggedIn = false;
4     currentPage = CurrentPage.HOME_PAGE;
5     //Assert State
```

```
6     Assertions.assertTrue(webDriver.getCurrentUrl().contains(currentPage.
   url));
7     Assertions.assertEquals(loggedIn, systemUnderTest.loggedIn);
8 }
```

Listing 2.9 logout action

2.9   Final Remarks

To ensure test compatibility with different browsers, the same test were tested again a chrome driver and a Firefox driver. Other actions could have been considered in the model, such as removing products individually, navigating to other pages such as the "About" page and career opportunities, etc. The combined implementation of the sub-automata discussed in previous sections culminates in the implementation of the automaton shown in Figure 2.11. It was ensured during the design phase that the states and actions are reachable from the current state. In fact, this is evident by the metrics provided by ModelJUnit (see Figure 2.11) after running the model using a greedy graph traversal approach (see Listing 2.11) with a test sequence length of 250. The greedy exploration method was also compared to the random exploration method.
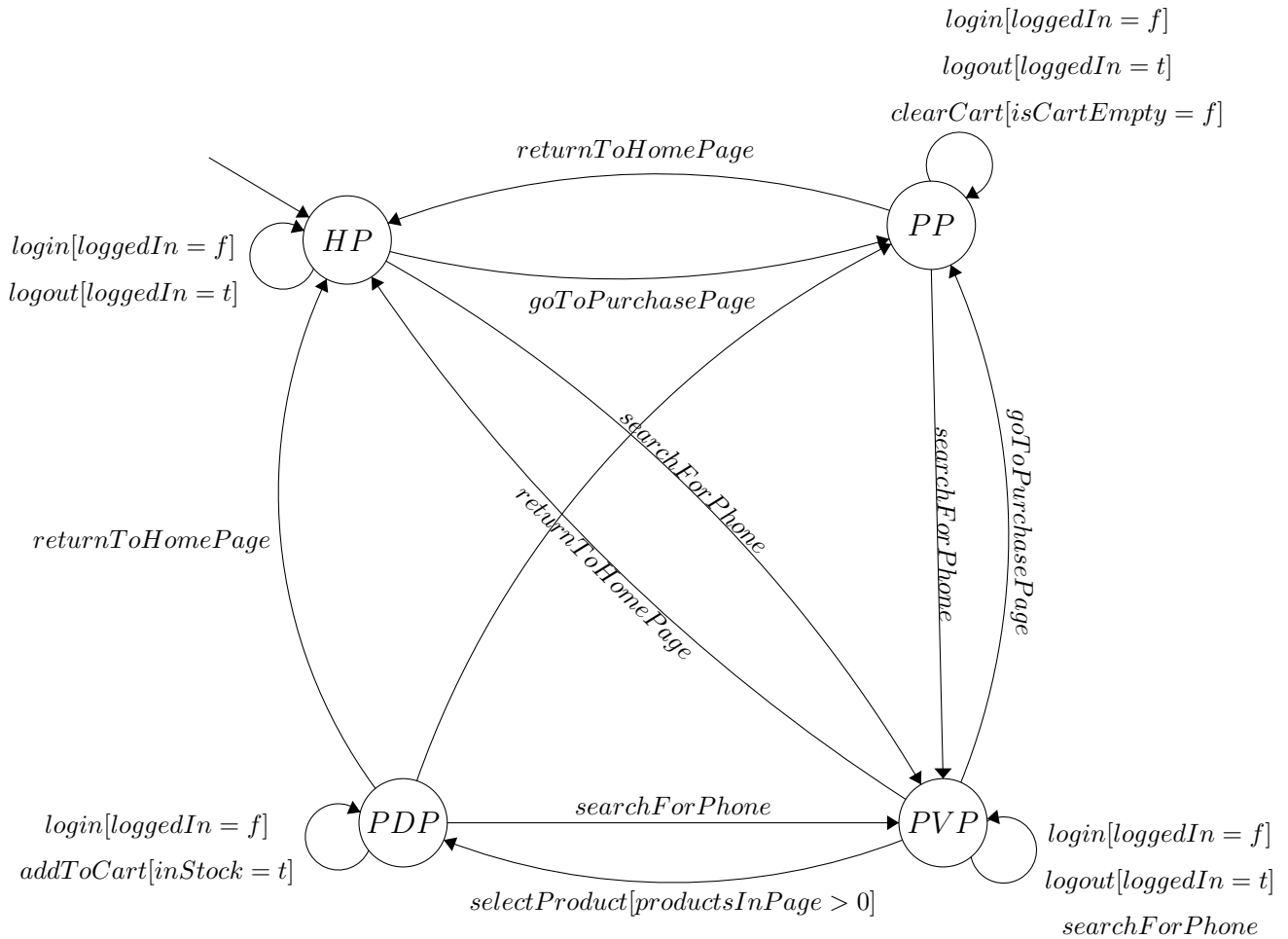
$$login[loggedIn = f]$$
$$logout[loggedIn = t]$$
$$clearCart[isCartEmpty = f]$$

Figure 2.11 The automaton covered by the sub automata in previous sections. States HP, PP, PDP, and PVP are $HOME\_PAGE$, $PRODUCT\_PAGE$, $PRODUCT\_DETAILS\_PAGE$, and $PRODUCTS\_VIEW\_PAGE$ respectively.

2.9.1 Test exploration

A greedy and a random model traversal were chosen as two approaches for exploring test sequences. Over a given number of n steps, the greedy approach covered more states, transitions, and edges at two different states. However, this came at the cost of slower test execution time. Counterintuitively, in my case, I prefer using the random approach to compensate for the slow execution time. However, if one wants to ensure that the test traverses all edge-pairs, then one should stick with the greedy approach. The slow loading of the web page components also contributed to the tests being slower. The recordings for each exploration method can be found in the following link: `https://github.com/andimon/` `uni-cps3230-web-test-automation-and-model-based-testing/tree/main/` `recordings`. The recordings are taken over the same number of test sequence length (`private final int testSequenceLength=60;`) for a fair comparison.

```
1 @Test
```

```
2  public void WebStoreTesterRunnerUsingGreedyTester() {
3      final Tester tester = new GreedyTester(new WebStoreTester());
4      tester.setRandom(new Random());
5      tester.buildGraph();
6      tester.addListener(new StopOnFailureListener());
7      tester.addListener("verbose");
8      tester.addCoverageMetric(new TransitionPairCoverage());
9      tester.addCoverageMetric(new StateCoverage());
10     tester.addCoverageMetric(new ActionCoverage());
11     tester.generate(testSequenceLength);
12     tester.printCoverage();
13 }
```

Listing 2.10 Running the model using a greedy graph walk algorithm

```
1  @Test
2  public void WebStoreTesterRunnerUsingRandomTester() {
3      final Tester tester = new RandomTester(new WebStoreTester());
4      tester.setRandom(new Random());
5      tester.addListener(new StopOnFailureListener());
6      tester.addListener("verbose");
7      tester.addCoverageMetric(new TransitionPairCoverage());
8      tester.addCoverageMetric(new StateCoverage());
9      tester.addCoverageMetric(new ActionCoverage());
10     tester.generate(testSequenceLength);
11     tester.printCoverage();
12 }
```

Listing 2.11 Running the model using a random walk algorithm


```
action coverage: 6/8
state coverage: 4/???
transition-pair coverage: 34/???
```

Figure 2.12 Metrics for Random Exploration (test sequence length of 60)


```
action coverage: 8/8
state coverage: 4/4
transition-pair coverage: 37/98
```
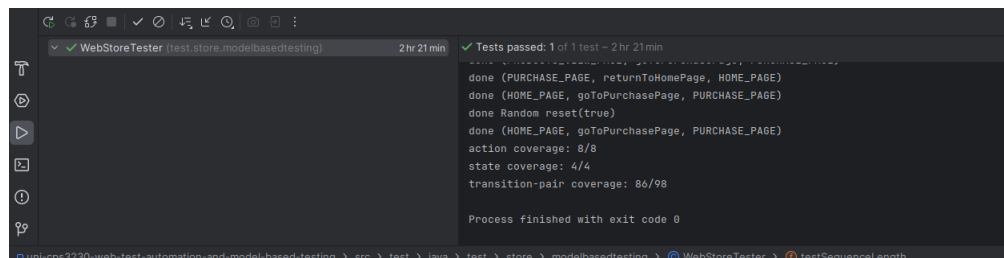
Figure 2.13 Metrics for Greedy Exploration (test sequence length of 60)

Figure 2.14 Metrics for Greedy Exploration (test sequence length of 250)

Figure 2.15 Metrics for Greedy Exploration (test sequence length of 400)

Figure 2.16 Metrics for Greedy Exploration (test sequence length of 1000)