1. Implementation

The implementation can be found at: https://github.com/slashsorin/TS-QA

It requires ***Python Behave*** and ***Selenium*** with ***Chromedriver*** to run.

These can be installed on Mac OS using the terminal and the commands:

pip install selenium
pip install behave

Inside the **/qa** directory is enough to just run the command:

behave

2. Approach

From the technology perspective, next to Selenium, I chose to use Behave for implementing the test case.

Behave is behavior-driven development (Cucumber-like) for Python: http://pythonhosted.org/behave/

Behave is good for complex projects, making communication of requirements easier between the parties involved (devs, qa and bussines)

For this test case I designed one feature file that contains the full description of the business case we are trying to protect written in Gherkin.

Gherking is a domain-specific language (DSL) used to write scenarios using the *"given-when-then"* approach. The feature file contains scenarios and each scenario has a sequence of steps that describe it. In order to make the scenarios executable we need to implement steps definitions. Each step definition is annotated with the step description so Behave could figure out which definition belongs to with step (this is how binding steps to steps definitions is done).

Example of feature file:

```
Feature: Amazon shopping basket

  Scenario: View the shopping basket product(s)
    Given I visit a product page on amazon.com
    When I click on the Add to Cart button
    Then the product is added to the basket
    When I go to my shopping basket page
    Then I can view my product
```

Example of steps definitions bindings to steps descriptions:

```
@given('I visit a product page on amazon.com')
def step(context):
    context.window.visit(AmazonProductPage())


@when('I click on the Add to Cart button')
def step(context):
    context.window.page.add_to_cart()
    context.window.page.dismiss_protection_plan_popover()
    context.window.visit(AddedToBasketPage())


@then('the product is added to the basket')
def step(context):
    assert context.window.page.is_confirmed()
```

Each step definition function contains the code to be run for each step.

Behind the Behave layer this test case uses the Page Object Model, which is the recommended pattern for testing web UI. This pattern is very good for tests maintenance. There is clean separation between the test code and page specific code. It can also reduce the possibility of flaky tests.

My page object model implementation can be found in the **/pages** directory of the solution repository.

The file **page_object.py** contains the base page object class from which all the other pages will inherit.

The file **window.py** contains the *Window* class. This instance wraps the browser window.

3. True Negatives

1. It is possible to determine if a certain product from amazon was not added to the shopping cart.

2. Upon visiting the shopping cart page, if the product expected to be displayed is not visible then the business case could be considered broken.

4. False Positives

1. The test would fail due to bad or slow network connections. Currently, it's setup to wait up to 10 seconds before it can find an element on the web page and then make an assertion on it. Yet, if slower than that, the test would fail, the business case could be still safe but less efficient.

2. Checking (asserting) on the wrong things during testing:
- assert a different number of products have been added to the shopping cart
- even asserting on product name/description that doesn't match or fit with the real one.

5. False Negatives

1. If on the shopping cart page there are two elements where the product name is displayed, but only one is verified and passes correctly. The other could be wrong and we wouldn't know.

2. Errors in the tooling, framework setup or the test environment. If the automation tool returns true instead of false or the other way around.