LINK TO PICTURE:
https://github.com/kasperpagh/TestAssignmentAutomatedUITests/blob/master/TestResults.PNG

LINK TO CODE:
https://github.com/kasperpagh/TestAssignmentAutomatedUITests/blob/master/src/test/java/carTest.java

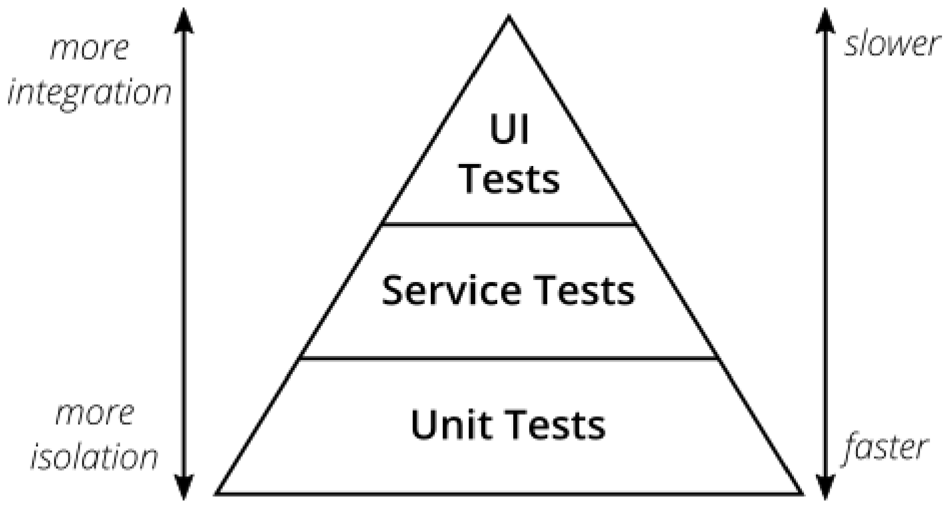
Pros and Cons with manual versus automated UI tests

I my own opinion, for writing excellent UI tests, you need both manual and automated UI tests in about equal measure.

The reason for this is the following: Manual tests are really good to determine which parts of the applications UI needs the heaviest automated testing. It also provides a way to examine the pattern of actions users take, when using the application. This information can be critical for constructing good automated tests.

When you have done some measure of manual testing, you can then, in theory, construct a test suite of automated tests, that can cover the most crucial parts of the user interface.

Automated Selenium Testing and the Test Pyramid



Source: https://martinfowler.com/articles/practical-test-pyramid.html

The Test Pyramid is a term coined in the book; *Succeeding with Agile* by Mike Cohn, and covers the idea that in automated testing, the testers focus should remain on the following overarching points:

- You should separate the subject of your tests into a multitude of layers (in the above image pictured as; Unit tests, Service test and UI tests). This assignment is an example of a test suite operating solely in the *UI test* sphere of the pyramid.
- As you scale the pyramid (from the bottom up) the number of tests you write should decrease. In other words; automated unit tests are more important that automated UI test.

As outlined above, this assignment most certainly do not follow the test pyramid, since ALL the tests are in the very topmost of the pyramid.

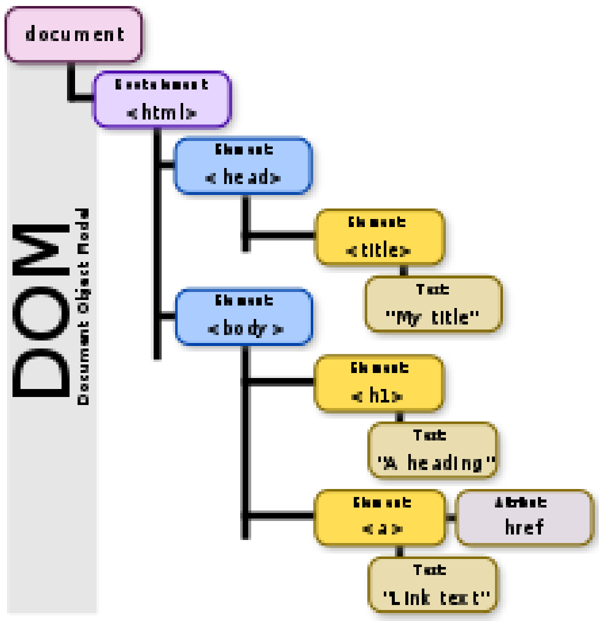
The Vulnerability of UI Tests

When you are doing automated UI testing, there is a couple of pitfalls to be aware of, namely:

- Your tests must happen in an isolated environment, since it is hard to test the UI if it’s tainted by input from random users. Therefore a test database will properly have to be maintained to ensure the validity of the tests.
- Automated tests can miss obvious (to humans, at least) errors.

DOM Manipulation

In practice automated UI tests of webpages function by interacting with the DOM (Document Object Model). The DOM is a document that can be rendered by a browser, and keeps its internal structure like a tree.



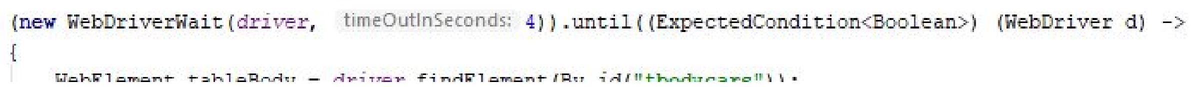
Source: https://en.wikipedia.org/wiki/Document_Object_Model

When we do Selenium testing, what we do is manipulate the DOM, and then assert that the new state of the DOM is changed (or unchanged) according with our design.

Document ready

A problem you often have when you do DOM manipulation, is trying to affect a DOM that have not properly been loaded yet.
For example: Many JQuery programmers will be familiar with doing all of their actual DOM read/write within the callback function of the *Document.ready* method.

In selenium testing we use the following construct that incorporates a build-in wait timer in our requests to the DOM.



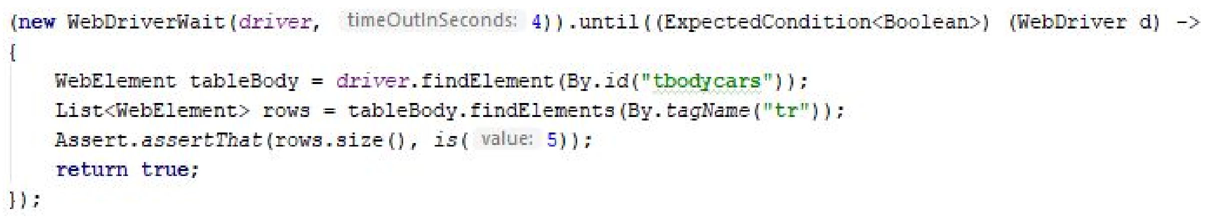
```
(new WebDriverWait(driver,  timeOutInSeconds: 4)).until((ExpectedCondition<Boolean>) (WebDriver d) ->
{
    WebElement tableBody = driver.findElement(By.id("tbodycars"));
```

An example of a call to the driver, including four seconds of wait time to allow the DOM to be constructed before we begin.

Exercise Implementation

In this exercise I implemented six different automated Selenium tests as outlined in the assignment. The approach used was much the same for all of them:

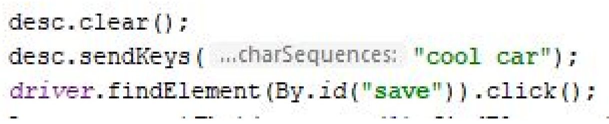
1. Go to the webpage and inspect the elements to figure out their ID (or tag).
2. Select the given DOM node in the following way:



```
(new WebDriverWait(driver,  timeOutInSeconds: 4)).until((ExpectedCondition<Boolean>) (WebDriver d) ->
{
    WebElement tableBody = driver.findElement(By.id("tbodycars"));
    List<WebElement> rows = tableBody.findElements(By.tagName("tr"));
    Assert.assertThat(rows.size(), is( value: 5));
    return true;
});
```

Find the table with the id; "tbodycars". Then find a list of all child nodes with the tag: "tr"

3. When we have the desired elements selected and saved in a variable, we can then manipulate them in a number of ways:



```
desc.clear();
desc.sendKeys( ...charSequences: "cool car");
driver.findElement(By.id("save")).click();
```

An example of manipulating a DOM node, in this case a textfield is cleared and the string: "cool car" is submitted. In the very last line a button with the id: "save" if found and then clicked.

Couple the above together with ordinary test practice, such as asserts, and it becomes possible to do automated UI testing.