Software Testing Practices

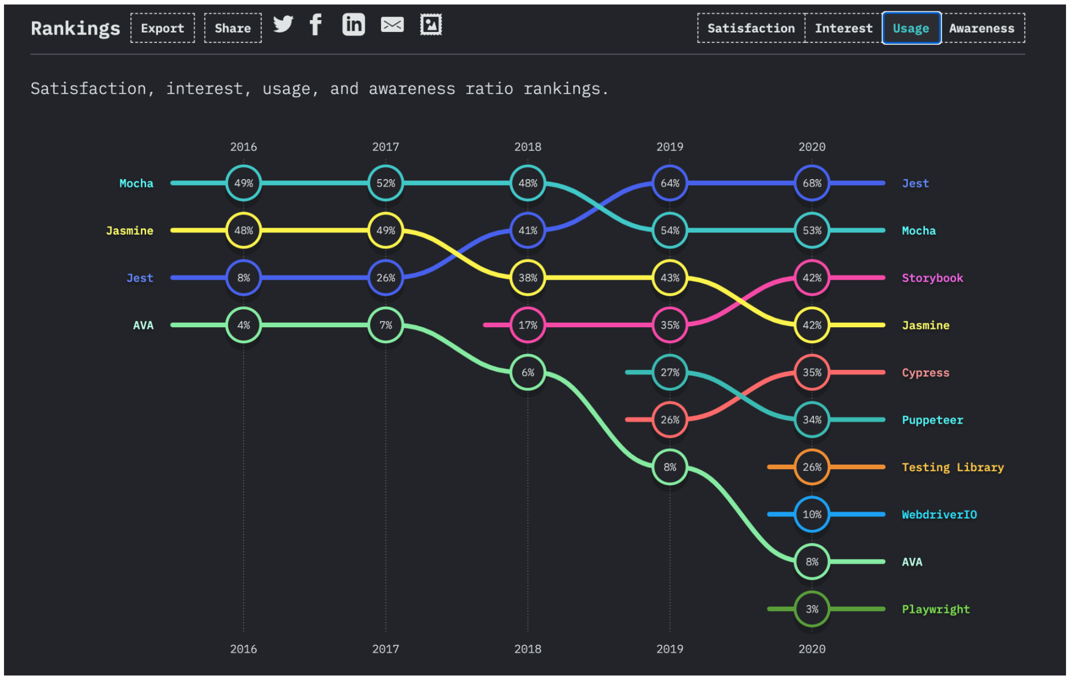
Bilge Deniz Koçak

Software testing is a method to evaluate and verify that a software product is working according to the expected requirements. As the bugs and issues get identified in the development process and get fixed prior to the product's launch, it is ensured that only quality products would be distributed to the customers. Software products can, and should be, tested on many levels. The goal of this Honors project is to explore different types of software testing more thoroughly and build upon the testing methods covered in CSC 4700: Software Engineering. After reading more about the use cases of different testing methods, I followed some tutorials and wrote tests on my own. All the code I have written for this project is available in this GitHub repository.

1. Unit Tests

1.1 What is Unit Testing?

Unit testing is a way of testing a unit, the smallest piece of code that can be logically isolated in the system. A unit test verifies a single unit of behavior, does it quickly, and does it in isolation from other tests. Unit testing is typically done by developers and with full knowledge of the source code, and they are typically automated tests. A popular Unit testing framework for Java is JUnit, which we covered in this class. For unit testing in JavaScript, some popular testing frameworks are Mocha, Jest, and Jasmine.



Satisfaction, interest, usage, and awareness ratio rankings.

Based on the StateofJS statistics, the JavaScript testing framework that is used the most is Jest. It also has the highest awareness, interest, and is the 2nd in satisfaction. Therefore, I wanted to learn more about this testing framework.

1.2 Jest

Jest is a JavaScript testing framework that is built on top of Jasmine, and it was built with a focus on simplicity and support for large web applications. It works with projects using Babel, TypeScript, Node.js, React, Angular, Vue.js and Svelte. Jest helps us create, run, and structure tests.

```
const { add, subtract, multiply } = require('../calculator');

describe("Testing with done", () => {

    test('Adding two numbers', (done) => {
        expect(add(5, 5)).toStrictEqual(10)
        expect(add(100, 200)).toStrictEqual(300)
        done()
    })

    test('Subtracting two numbers', (done) => {
        expect(subtract(10, 10)).toStrictEqual(0)
        expect(subtract(200, 100)).toStrictEqual(100)
        done()
    })

    test('Multiplying two numbers', (done) => {
        expect(multiply(10, 10)).toStrictEqual(100)
        expect(multiply(200, 100)).toStrictEqual(20000)
        done()
    })

})
```

Above, we can observe the syntax for a simple Jest unit test. describe is a Jest method that we use to group tests together, so it contains one or more related tests. Every time we are writing a new suite of tests for a functionality, we can wrap them in a describe block. Following the describe keyword, we can include a description for the following group of tests, and a callback function to wrap the tests. The test function includes a description for that specific test, and another callback function that generally has expect statements within it. This is similar to "assert" for Java. There are multiple methods we can utilize, like toStrictEqual, toBe, toHaveBeenCalled, toHaveBeenCalledTimes, or toHaveBeenCalledWith, to write our tests in accordance with the expected behavior.

1.3 Setting Up Jest for a Node.js Project

To install Jest, we can use the **npm install --save-dev jest** command. To initiate a Node.js project, we can run the **npm init** command in our folder. Then, to install the Jest package into our project, we can **run npm install --save-dev jest**. In our package.json file, we can add this script:

```
"scripts": {
"test": "jest"
      }
```

Then, we can write Jest tests with the appropriate syntax, and test them with **npm test**.

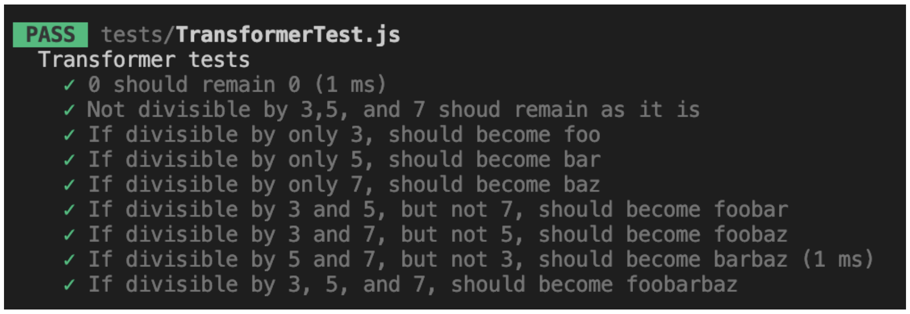
1.4 Jest Takeaways

To experiment with Jest, I coded the FooBarBaz game's console portion in JavaScript and wrote three Jest tests for the Transformer, InputValidator, and Console. This project is available on my GitHub repository.

An error that I ran into when I was first trying to run my Jest tests was Jest not being able to find my tests. I had a src and a test folder in my project, and my code was in src and tests were in the test folder. To make Jest find my tests, I created a jest.config.js file and added testMatch: ['**/tests/*.js?(x)'].

Another challenging aspect was testing the console log using Jest. For this purpose, I used Jest's spyOn method. Spying is to observe calls made to a method without changing the method itself. To check multiple arguments on multiple calls for jest spies, I compared the mockFunction.mock.calls to the expected results. I stored the list of expected arguments from each call in arrays and created an array of arrays to do the comparison with the mockFunction.mock.calls.

During this process, I noticed that naming the tests and describe block's description in a meaningful way is very important. As we are trying to write down the descriptions, we could better understand what to test. For someone who is not too familiar with a project and trying to understand the expected behavior, these descriptions could be extremely helpful.



```
PASS  tests/TransformerTest.js
  Transformer tests
    ✓ 0 should remain 0 (1 ms)
    ✓ Not divisible by 3,5, and 7 shoud remain as it is
    ✓ If divisible by only 3, should become foo
    ✓ If divisible by only 5, should become bar
    ✓ If divisible by only 7, should become baz
    ✓ If divisible by 3 and 5, but not 7, should become foobar
    ✓ If divisible by 3 and 7, but not 5, should become foobaz
    ✓ If divisible by 5 and 7, but not 3, should become barbaz (1 ms)
    ✓ If divisible by 3, 5, and 7, should become foobarbaz
```

For example, if only the tests with a number divisible by 3 were failing in this output, I would have known to check that implementation immediately since I named the tests very descriptively.

2. Acceptance Tests

2.1 What is Acceptance Testing?

Instead of a stakeholder passing down the requirements to the development team, the developer and stakeholder can collaborate to write automated tests that express the outcome the stakeholder wants. While unit tests ensure you build the thing right, acceptance tests ensure you build the right thing. A principle that could be followed with automated acceptance testing is Behavior Driven Development (BDD). Behavior driven development specifies tests of any unit of software in terms of the desired behavior of that unit. Taking in a user story, the team could discover and agree on details of what is expected to be done. Then, they could formulate them in a way that could be automated. Finally, the behaviors could be implemented based on the tests. Through this practice, writing redundant code would be hindered and the final product would match the provided requirements.

2.2 Cucumber

Cucumber is a software tool that supports behavior driven development. For each feature a team wants to implement, they can create a feature file. Each feature file consists of scenarios to test, which have steps to work through. Along with the features, the developers provide Cucumber with step definitions, which provide business readable language into code. For feature files to be understood by Cucumber, they have to follow a specific syntax called Gherkin. Through Gherkin, each test becomes an executable specification.

2.3 Gherkin

Each file should start with the Feature keyword. In general, features contain 5-20 scenarios. If there are a set of steps common to all scenarios in a feature file, they can be moved up after the Background keyword. The keywords Given, When, and Then help structure the user stories. The Given keyword sets up the context where the scenario happens, When interacts with the system, and Then checks if the outcome of the interaction is as expected. More steps can be added to Given, When, and Then statements through the addition of And and But statements. To create the step definitions, special Cucumber annotations like @Given, @When, and @Then can be used. If a scenario has undefined steps, Cucumber gives snippets of code developers can use to implement those steps, making working with the Gherkin syntax even simpler.

2.4 Running Cucumber on Java

Maven is a build tool that is primarily used for Java projects. Archetypes in Maven are project templates that users can utilize to set up a project as quickly as possible. To start a Cucumber project, users can make use of the cucumber-archetype Maven plugin. Running this command in the terminal will help initiate a Cucumber project:

```
mvn archetype:generate                      \
   "-DarchetypeGroupId=io.cucumber"           \
   "-DarchetypeArtifactId=cucumber-archetype" \
   "-DarchetypeVersion=7.11.2"               \
   "-DgroupId=hellocucumber"                  \
   "-DartifactId=hellocucumber"               \
   "-Dpackage=hellocucumber"                  \
   "-Dversion=1.0.0-SNAPSHOT"                 \
   "-DinteractiveMode=false"
```

The features should be added under src/test/resources/projectName, and they can be executed through the **mvn test** command. Cucumber provides a Java tutorial on their website. I followed this tutorial to get familiar with Cucumber, and the code for this is available on GitHub. This could be a good starting point for implementing other Cucumber + Java projects.

2.5 Running Cucumber on JavaScript

After installing Node.js and creating the project directory, the developer can run the **npm init** command to initialize a new Node project. Then, **npm install @cucumber/cucumber** command should be run to install the cucumber package from the Node package manager (npm). Then, the developer can create the features directory and write their features. Here is how the directories would look like:

```
project-name
├── features
|    └── something.feature
|    └── stepDefinitions.js
├── node_modules
├── cucumber.js
├── package-lock.json
├── package.json
```

Since the output for the tests can get too extensive, module.exports = { default: '--publish-quiet' } could be added in the cucumber.js file to not receive information about publishing reports. To see some other configuration options for cucumber, this site could be helpful. After this, the tests can be run with the **npx cucumber-js** command. With the code snippets given for the undefined steps, creating the stepDefinitions file and writing the definitions there will become easier. Once that is finalized, the **npx cucumber-js** command can be run again to see the test results.

I implemented the Java tutorial published by Cucumber in JavaScript as well, and the code for this is available on GitHub. I created an additional example for JavaScript, where I wrote a feature for the FooBarBaz game, and that is also available on my repository.

2.6 Interpreting Test Results

In Cucumber, a step that is executed can end up in any of these states: Failed, Pending, Undefined, Skipped, Passed.

- If Cucumber cannot find a step definition that matches a step, that step gets marked as undefined.
- If a step definition is halfway through being implemented, it gets marked as pending.
- If a step definition raises an exception, or uses an assertion but the assertion does not pass, the step gets marked as failed.
- If a step fails, the scenario will be stopped and the rest of the steps in that scenario will be skipped. If a step is undefined or pending, the rest of the steps in that scenario will either be skipped or get marked as undefined (if they also don't have a step definition).
- If neither of these are the case, the step will pass.

2.7 Additional Resources

To learn more about Cucumber, the Cucumber for Java Book on O'Reilly is a great source. The documentation of Cucumber could also be helpful. I found this video to be helpful for getting started with BDD testing in Javascript.

3. Integration Tests

3.1 What is Integration Testing?

Sometimes, despite all of the unit tests passing, an application might still not work. Aside from testing software components in isolation, we also need to test how those components work with each other and the external systems that we are using. The entry condition for integration testing is unit testing, meaning our code should pass the unit tests before writing our integration tests. In a complex software system, there are numerous interconnections between components for a particular feature, which makes writing integration tests complicated. This gave rise to different types of integration testing practices, such as top-down, bottom-up, integration, and big bang integration testing.

3.2 Big Bang Integration Testing

In big bang integration, all modules are linked at once, resulting in a complete system. It is very hard to isolate the errors found, and it is hard to cover all cases for integration testing without missing even a single one of them. This approach is typically used when there is a tight deadline for a project, and all development teams are working in parallel in their respective components. Top-down, bottom-up, or hybrid integration testing techniques, on the other hand, follow a more incremental approach.

3.3 Top-Down Integration Testing

This testing is done through integrating two modules by moving down from top to bottom through control flow of architecture structure. We test the higher level modules first, and then the lower level modules, and then integrate them together. We use *stubs*, a small piece of code that takes the place of another component during testing, if the invoked submodule is not developed yet. So, stubs work as a momentary replacement. If a significant defect occurs toward the top of the program, this testing approach would be more helpful.

3.4 Bottom-Up Integration Testing

This testing is done through integrating two modules by moving up from the bottom to top through control flow of architecture structure. We test the lower level modules first, and then the higher level modules, and then integrate them together. We use *drivers*, a module that acts as a temporary replacement for a calling module to simulate the behavior of the upper level modules that are not yet integrated. If there are crucial flaws toward the bottom of the program, this testing approach would be more helpful.

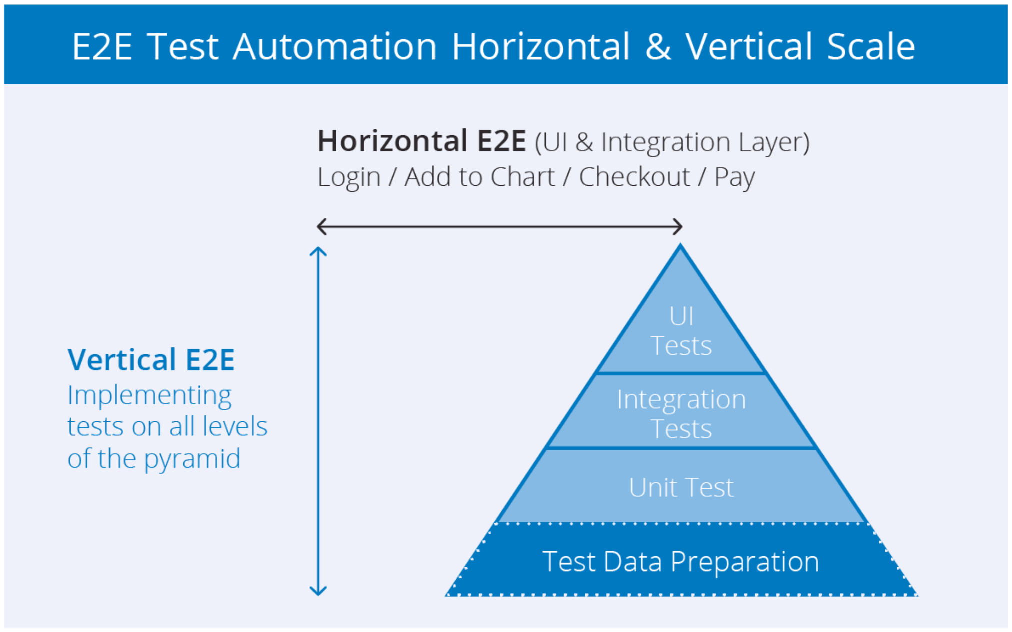
3.5 Hybrid Integration Testing

Hybrid integration testing takes the advantages of both bottom-up and top-down integration testing, and it is also known as sandwich testing. The product features are organized into a top, main, and bottom layer, and the testing is focused on the main, central layer. This makes testing more extensive and better suitable for a large group that has various sub-projects. If the goal is to have a basic working system in the earlier stages of the development cycle, hybrid integration testing would be a good fit. The testing can be carried out in both directions, making this method more time-efficient. However, implementing this approach is harder.

4. End-to-End (E2E) Tests

4.1 What are E2E Tests?

Through writing end to end tests, we automate a flow from start to finish and we simulate a real user scenario. But it is important to note that E2E testing is not just a happy-path testing. We can have our user enter invalid data, and have the tests for these cases as well. To write good E2E tests, we should have a good understanding of the problem domain. As the integration tests don't cover the UI, if we are wondering whether our UI delivers a pleasant experience, we can do E2E testing. Generally, E2E tests are executed on finished products and they improve the quality of our products by validating that our product is functional at every level. So, unlike integration testing that is used from the very beginning of the project, E2E tests are written for a finalized (or near completion) product.


E2E Test Automation Horizontal & Vertical Scale
Horizontal E2E (UI & Integration Layer)
Login / Add to Chart / Checkout / Pay
Vertical E2E
Implementing tests on all levels of the pyramid
UI Tests
Integration Tests
Unit Test
Test Data Preparation

4.2 Horizontal E2E Testing

A horizontal E2E test verifies each workflow through each individual application to ensure that related processes occur correctly. For example, if we have different parts of our application, like the login, adding to chart, checking out, and paying, we would be testing across all of these from the beginning to end. Therefore, we would need to have all applications functioning. If we are not too deep into development (not done with one of these applications), it might be better to not choose this methodology.

4.3 Vertical E2E Testing

A vertical E2E test consists of testing each layer of a single application's architecture from top to bottom. So, we can test each sub system independently of another through using unit tests, and then test the UI and API layers. A vertical test focuses on each layer of an individual application. As we work with just one application rather than many, we can test more quickly compared to horizontal E2E testing. In the example above, a vertical E2E test would be writing tests at all levels of the pyramid for one of the applications mentioned.

4.4 Getting Started with Cypress

Cypress is a JavaScript-based end-to-end testing tool designed for modern web test automation. It is based on Mocha and Chai, and uses Node.js while running on browsers.With Cypress, you can also run cross-browser testing and execute your tests on Chrome, Electron, Firefox, and Edge. To learn how to write a Cypress test, I followed this tutorial, although it was a bit outdated. Once again, to initiate a Node.js project, we can run the **npm init** command in our folder. To install the Cypress package as a dependency, we can run **npm install --save-dev cypress**. Then, we can alter the scripts portion of the package.json file:

```
"scripts": {
    "test": "npx cypress run",
    "cypress:open": "npx cypress open"
},
```

After this, we can create our cypress.config.js file:

```
const { defineConfig } = require('cypress')
module.exports = defineConfig({
    chromeWebSecurity: false,
    e2e: {
        setupNodeEvents(on, config) {},
        supportFile: false,
      },
})
```

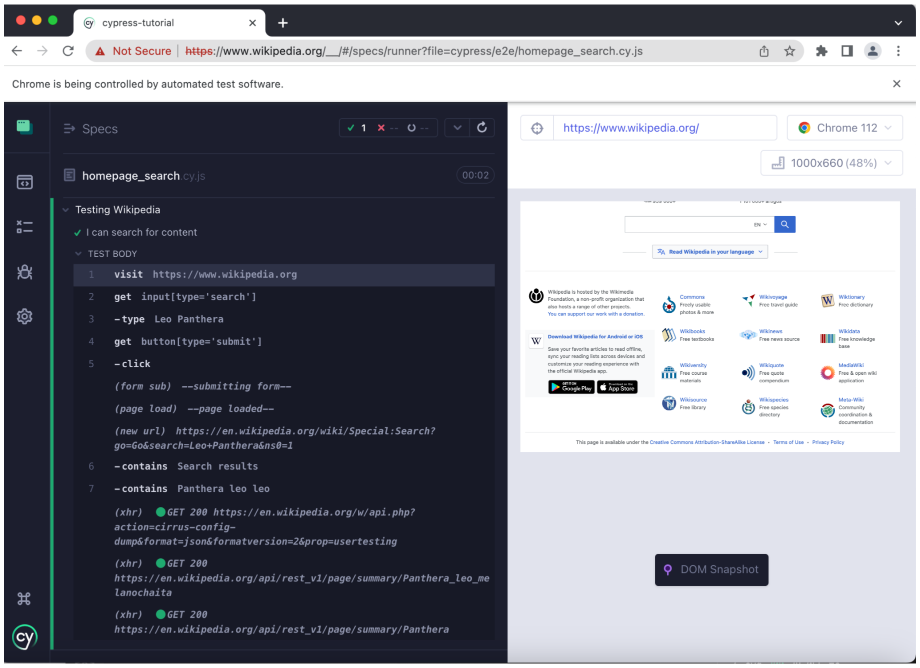
For Cypress to find the E2E tests we are writing, we should create a folder named cypress under our root directory, and put our tests under an e2e folder within this cypress folder. Since Cypress follows Mocha's syntax, it makes it easier for the developers/testers familiar with Mocha to get started with this tool. To run the Cypress tests, we can run **npm run cypress:open** on the terminal. Then, in the pop-up window, we can choose which browser we want to run our tests in.

4.5 Cypress Example

Here is an example of what a Cypress test could look like:

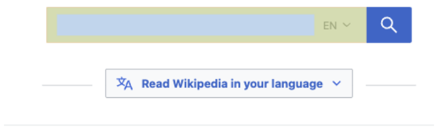
```
describe('Testing Wikipedia', () => {
        it('I can search for content', () => {
                cy.visit('https://www.wikipedia.org');
                cy.get("input[type='search']").type('Leo Panthera');
                cy.get("button[type='submit']").click();
                cy.contains('Search results');
                cy.contains('Panthera leo leo');
        });
});
```

With this test, we are looking forward to testing Wikipedia and seeing if we can search for content. Cypress visits Wikipedia's website, finds the search bar and inputs Leo Panthera, submits it, and then finds Search results and Panthera leo leo in the site displayed. Cypress makes it very easy to understand this process. Here is what we would see in the pop up browser:

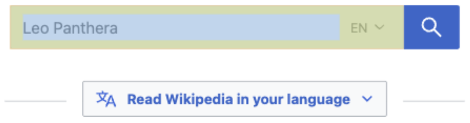

cypress-tutorial
Not Secure
~~https~~://www.wikipedia.org/__/#/specs/runner?file=cypress/e2e/homepage_search.cy.js
Chrome is being controlled by automated test software.
Specs
✓ 1 ✗ -- ○ --
homepage_search.cy.js
00:02
https://www.wikipedia.org/
Chrome 112
1000x660 (48%)
DOM Snapshot

```
Testing Wikipedia
  ✓ I can search for content
    TEST BODY
      visit  https://www.wikipedia.org
      get  input[type='search']
      -type  Leo Panthera
      get  button[type='submit']
      -click
      (form sub)  --submitting form--
      (page load)  --page loaded--
      (new url)  https://en.wikipedia.org/wiki/Special:Search?
      go=Go&search=Leo+Panthera&ns0=1
      -contains  Search results
      -contains  Panthera leo leo
      (xhr)  ● GET 200 https://en.wikipedia.org/w/api.php?
      action=cirrus-config-
      dump&format=json&formatversion=2&prop=usertesting
      (xhr)  ● GET 200
      https://en.wikipedia.org/api/rest_v1/page/summary/Panthera_leo_me
      lanochaita
      (xhr)  ● GET 200
      https://en.wikipedia.org/api/rest_v1/page/summary/Panthera
```

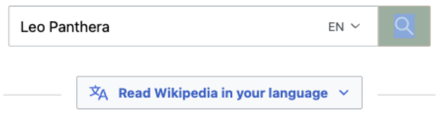
On the top, we can see that our test passed, since it has 1 next to the check mark. On the left, we can see the different steps that Cypress took. And as we scroll over those steps, we can see what Cypress has done on the website displayed on the right, with snapshots and before-after illustrations. Here is the process it took for this test after visiting the website:



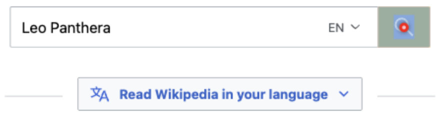
1. Find the search bar



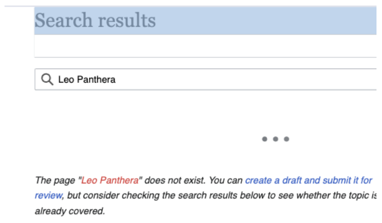
2. Type “Leo Panthera”



3. Find the submit button

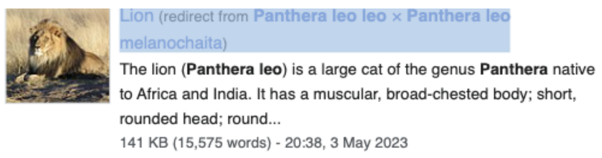


4. Click the submit button



The page "Leo Panthera" does not exist. You can create a draft and submit it for review, but consider checking the search results below to see whether the topic is already covered.

5. Loaded website contains “Search results”



Lion (redirect from **Panthera leo leo** × **Panthera leo melanochaita**)
The lion (**Panthera leo**) is a large cat of the genus **Panthera** native to Africa and India. It has a muscular, broad-chested body; short, rounded head; round...
141 KB (15,575 words) - 20:38, 3 May 2023

6. Loaded website contains “Panthera leo leo”

5. Reflection

In our CSC 4700: Software Engineering class, we talked about the importance of testing our software projects and had labs on how to write Java unit tests using JUnit. Through this Honors project, I was able to go beyond what we learned in the classroom. I read more about unit testing, and I learned about acceptance, integration, and E2E tests. I not only read about these tests but also practiced writing them, either through following tutorials and coming up with some examples on my own. While the Computer Science classes at Villanova mainly focus on Java, I wanted to learn more about JavaScript and increased my proficiency in JavaScript. I also had the chance to learn about different testing frameworks, like Jest, Cucumber, and Cypress, while becoming familiar with what else is in the market, like Mocha or Chai. To share a portion of what I learned by doing this project with my classmates, I gave a presentation on Cucumber & Acceptance Testing to our class as well.

Due to time constraints, I did not follow a tutorial on integration testing, but I know that I could have done it with Jest. Some other ideas I had while starting this project were writing load and concurrency tests if I had more time.

A special circumstance that I faced while working on this project was the February 6, 2023 earthquakes that hit my home country, Turkey. The area impacted was about the size of Germany, and 14 million people were impacted by these earthquakes. I was truly devastated by the news. This event impacted some of my family members and friends, and despite being so far away from home, I wanted to be there for them. I spent about three weeks raising awareness, collecting donations, and contributing to prayer services organized on campus. Writing these words, I once again remember the 50,000+ lives lost.

May their souls rest in eternal peace.