Test report for the eCommerce application

COMP.SE.200 Ohjelmistojen testaus

Eetu Soronen (152830694) & Teemu Ranne (152934833)

https://github.com/soronen/COMP.SE.200-2024-2025-1

Definitions, acronyms and abbrevations

AI: Artificial Intelligence. Referring to LLMs like GitHub Copilot or ChatGPT

Branch coverage: A metric that indicates how many of the possible branches (e.g., if/else conditions) in the code have been tested.

Buffer overflow: A type of security vulnerability where more data is written to a buffer than it can hold, potentially causing unexpected behavior.

Integration tests: Tests that focus on verifying that different modules or systems work together as expected.

Line coverage: A metric that shows the percentage of lines of code executed by tests.

mocha: Unit testing framework for JavaScript / Node

mochawesome: An extension library to Mocha which generates html reports out of the test run console logs.

nyc: A JavaScript code coverage tool that works with Istanbul to generate reports of which parts of the code are covered by tests.

npm: Node Package Manager.

Pipeline: A sequence of automated steps in software development that includes tasks like building, testing, and deployment.

Robot Framework: A generic test automation framework for acceptance testing and robotic process automation.

Selenium: A popular tool for automating web browsers, often used for end-to-end or higher-level testing of web applications.

Unit tests: Tests that focus on individual components or units of the software, verifying that each part works correctly in isolation.

UI (User Interface): The part of the software that users interact with, including visual elements and controls.

Version control: A system that tracks changes to files, especially source code, and allows for collaboration and rollback of changes.

Introduction

We misunderstood the intended scope of the first document focused much more on higher level Acceptance tests, which in a real application are the most important level of tests and also generally what test plans focus on. However, in this document we test the "eCommerce" application library (COMP.SE.200-2024-2025-1 git repository) which only contains JavaScript modules, on a unit testing level. In addition to the tests, we also create an automatic testing and building pipeline, which runs our tests automatically when we push code into our GitHub repository. Code coverage is then analyzed by Coveralls cloud service.

After the tests have been written and run, any issues with the implementations will have bug reports written to the issues page on our GitHub (https://github.com/soronen/COMP.SE.200-2024-2025-1/issues).

Tests

This project uses JavaScript, which has no type-safety features. This means that even when a function input is supposed to accept only Number types for example, it may also accept "numberlike" inputs like string "5" with no error. This is almost never intended behavior, but protecting from this is outside the scope of unit tests. Instead, type checks could be gained to the project by using TypeScript instead, but for now we will just accept the regular JavaScript type behavior in our unit tests as "correct".

Tests can be run using after installing the dependencies from package.json with "npm run test", if you want a coverage report you can use "npm run coverage". These commands are also used in our GitHub Actions workflow.
As said in the introduction, though our test plan focused mainly on acceptance tests this report is about unit tests for the E-Commerce library. Generally, unit tests are so simple that we might as well test all of the modules, especially since we found so many errors at all points.

CI Pipeline

Getting tests themselves to run with GitHub actions proved no issue at all, GitHub provided ready example jobs which start up a virtual machine and run node tests as they are defined in package.json. Coveralls integration was a little trickier, but still doable with some time spent on troubleshooting various issues. The coverage report however was very difficult, for some reason the nyc coverage tool reported no line coverage, and the common fixes did nothing to rectify this. In the end we swapped from nyc coverage tool to c8, which worked immediately with the same configuration settings as nyc was supposed to. After this work our coverage reports work both locally and on Coveralls.

Also, a note on our pipeline, it only fails if there is an issue with creating the coveralls report. This is because our test run step is “npm run coverage || true”, which means it never fails. For production this is likely not desirable but right now we have many failing tests, and we don’t want them to crash our coverage reporting completely.

Our pipeline can be found at https://github.com/soronen/COMP.SE.200-2024-2025-1/actions.

Findings

We found several bugs with the tested modules, which we reported laconically at: https://github.com/soronen/COMP.SE.200-2024-2025-1/issues. For more complex tests it would perhaps be necessary to log, our environment, used version, etc. But for a basic bug in a side effectless function, which is even reproduced by our pipeline, it’s better to avoid unnecessary boilerplate templates and get to the point. More detailed (and automated) reports have been generated by mochawesome-library and are available at https://soronen.github.io/COMP.SE.200-2024-2025-1/.

Our test coverage is overall very good, nearly all modules have their own unit tests, with few exceptions where the untested module is a dependency for a tested module. Our line coverage for src/ directory (with .internal/ folder excluded) is 98%, with 80% branch coverage. Limit testing and such are not covered in these numbers, but attention has been paid to these tests as well. As a reminder, JavaScript is a memory safe language so buffer overflows or other security problems are not as much of a problem as in C++ for example.

We can consider the library fully unit tested. E-Commerce application itself will need acceptance tests at the minimum to be considered tested. Integration tests for the library and other components is also very much recommended.

Conclusions

The product is not ready for release. Some failures like camelCase lead to improper displays in the UI, but for example divide function is very likely to be used in business logic of the application. Prices, item quantities and other important numbers simply can’t be incorrect. Furthermore, finding issues in about ¼ of the modules is not an acceptable ratio for any commercial application. Furthermore, integration and acceptance tests are completely lacking at this point.

AI and testing

In part one our document focused much more on higher level tests, but on a unit level AI is very good, especially for JavaScript as basically an autocorrect / syntax checker. For creating the testing pipeline and troubleshooting the any errors with nyc, it was wrong in its suggestions until the very end. But its incorrect answers are still in mostly correct syntax, so fixing up the mistakes instead of writing the configuration files from scratch may have some value.

For actually writing the tests, we had GitHub Copilot enabled, which is intended to be a very fancy autocomplete, and just like the more traditional autocompletes sometimes it suggests the wrong thing. On the whole it's very good though.

Using AI for testing is much riskier than actual code writing, since testing is supposed to validate the code. AI can't be considered trustworthy enough to understand what the specification is and whether the implementation is correct. All AI output must be validated by human, and for testing especially using AI to design cases is not recommended. AI has a much faster output than a human though, and some benefit can be drawn from that in things like fixing missing brackets or typos.

Course feedback

The course overall was very good, I liked the lectures a lot and exercise sessions had plenty of different tools we got to try out, even though obviously there is only so much you can do in 2 hours. The assignment was a lot of report writing, which I guess is accurate to the testing profession. It's kind of difficult to keep track of what you should and shouldn't write when the subject is fairly large.

Creating unit tests for JavaScript/Node is not overly complicated, especially if you've created web applications before. I feel like this part of the assignment was a missed opportunity since unit testing is mostly a software developer responsibility. It would have been interesting to do some higher-level Robot or Selenium tests instead.

Creating the pipeline, choosing the libraries, and troubleshooting various things was a good learning experience. Unit test as a concept was already familiar to us though, so it wasn't a huge step to transfer those skills to the JavaScript world.

Appendix

Test Plan for eCommerce website

COMP.SE.200-2024-2025-1

Eetu Soronen (152830694) & Teemu Ranne (152934833)

Definitions, acronyms and abbreviations

MoSCoW method – test prioritization method based on the importance of the component and the chance as well as consequences of failure. Acronym stands for Must, Should, Could, and Won't test.

Introduction

This document contains a test plan for E-commerce web application. It is not a design document for the application itself, instead only providing the guidelines for designing and implementing tests for Unit, Integration, System, and Acceptance test cases, as well as the tools used to do so. Performance testing is also important for a commercial application and will have to be taken into account for all levels of tests.

The product is considered ready for shipping only when the acceptance tests run by the customer have been passed. Lower-level tests are used to help facilitate this goal and should be designed alongside the product and implemented throughout the development process. This document provides the guidelines for designing and implementing all tests throughout the product lifecycle.

Scenarios

Our test scenarios are based on the requirements and features of the web application. Testing tools for manual tests are a web browser, and for automated acceptance / UI testing we can use a framework like Selenium or Robot Framework. For lower-level tests like Unit tests and some integration tests we can use Unit testing frameworks like Mocha or Jest.

We have made test scenarios and sequence diagrams for the most important acceptance tests. Features and components used by these requirements should be tested by those diagrams should tested throughout the whole testing hierarchy, on unit, integration, system as well as acceptance testing level.

All components of the application should be tested. The payment provider is a 3rd party vendor, so that can be tested on integration level at the lowest, but otherwise testing should begin on unit testing level.

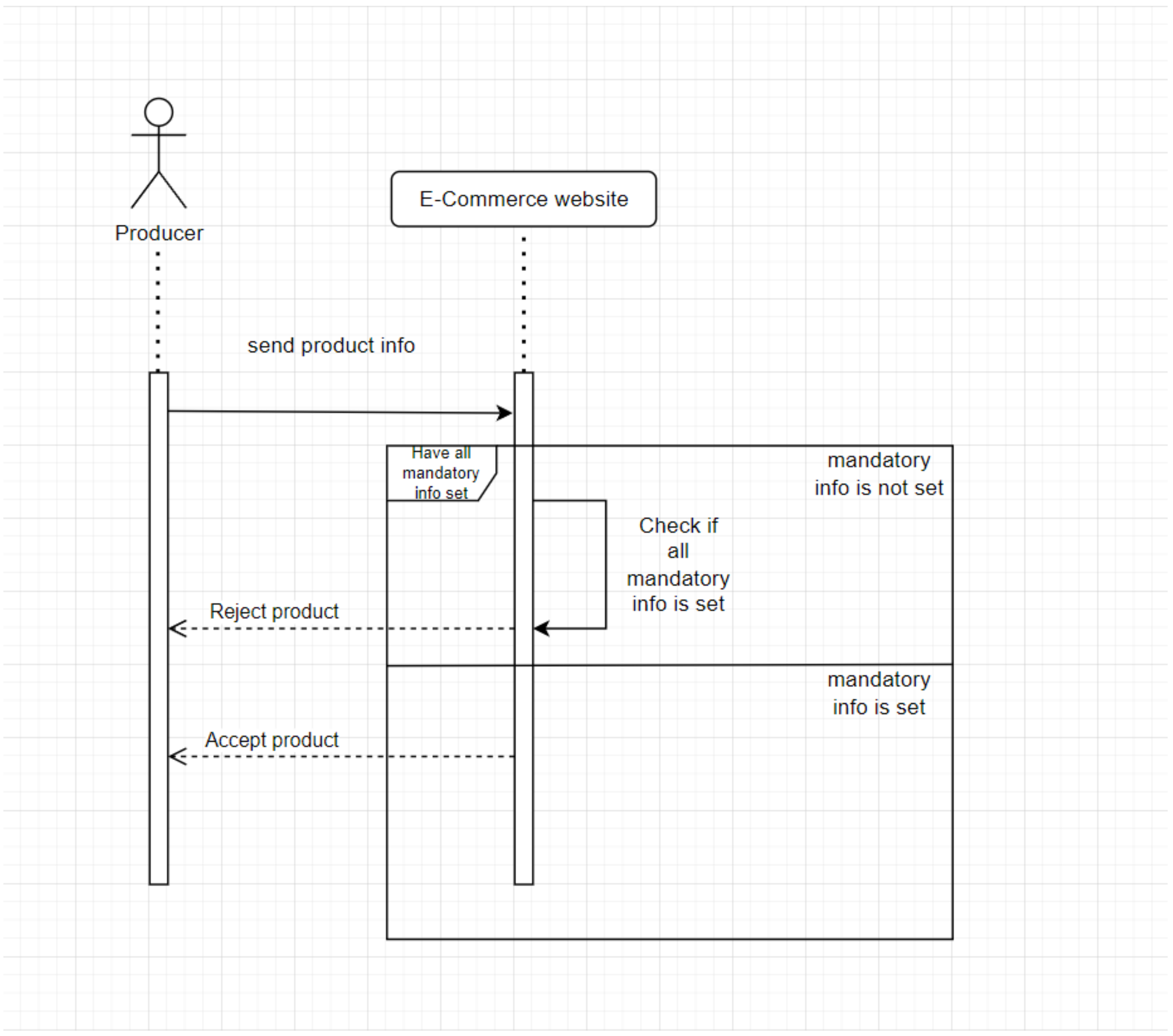
Our preliminary risk analysis based on the MoSCoW evaluation also makes the payment provider and our software's integration with it the most vital part of the application. Customers should never pay anything they didn't agree to, and inversely always pay what they expect to. This is why extra attention must be provided to integration between our store and the payment provider.

Our high-level scenarios can also double as acceptance tests, which are below. There are two main sets of scenarios, 0xx and 1xx tests. 0xx are for customer facing features, while 1xx are for vendor facing features. Our acceptance test cases, and their sequence diagrams are included below. Negative test scenarios, which test that our application fails in an expected way should also be made to test our scenarios.

Scenario name	Producer places product to eCommerce website

Preconditions	Producer is able to access the eCommerce website
Test Steps	**Expected Results**
1. Producer places product filling mandatory fields.	1. Product is only accepted when mandatory fields are filled.

Sequence diagram for producer facing scenario:



Scenario name	User searches products by category
Preconditions	User is able to access the eCommerce website
Test Steps	**Expected Results**
2. User selects the desired category from menu	1. User is able to view products of selected category. All other products are hidden from view.

Test Name	User searches products by product contents
Preconditions	User is able to access the eCommerce website
Test Steps	**Expected Results**
1. User searches for desired product contents (specific instructions tbd)	1. User is able to view products with selected contents. All other products are hidden from view.

Test Name	User searches products by product name
Preconditions	User is able to access the eCommerce website
Test Steps	**Expected Results**
1. User inputs the desired product name into the search bar.	1. Products with matching name are displayed. All other products are hidden from view.

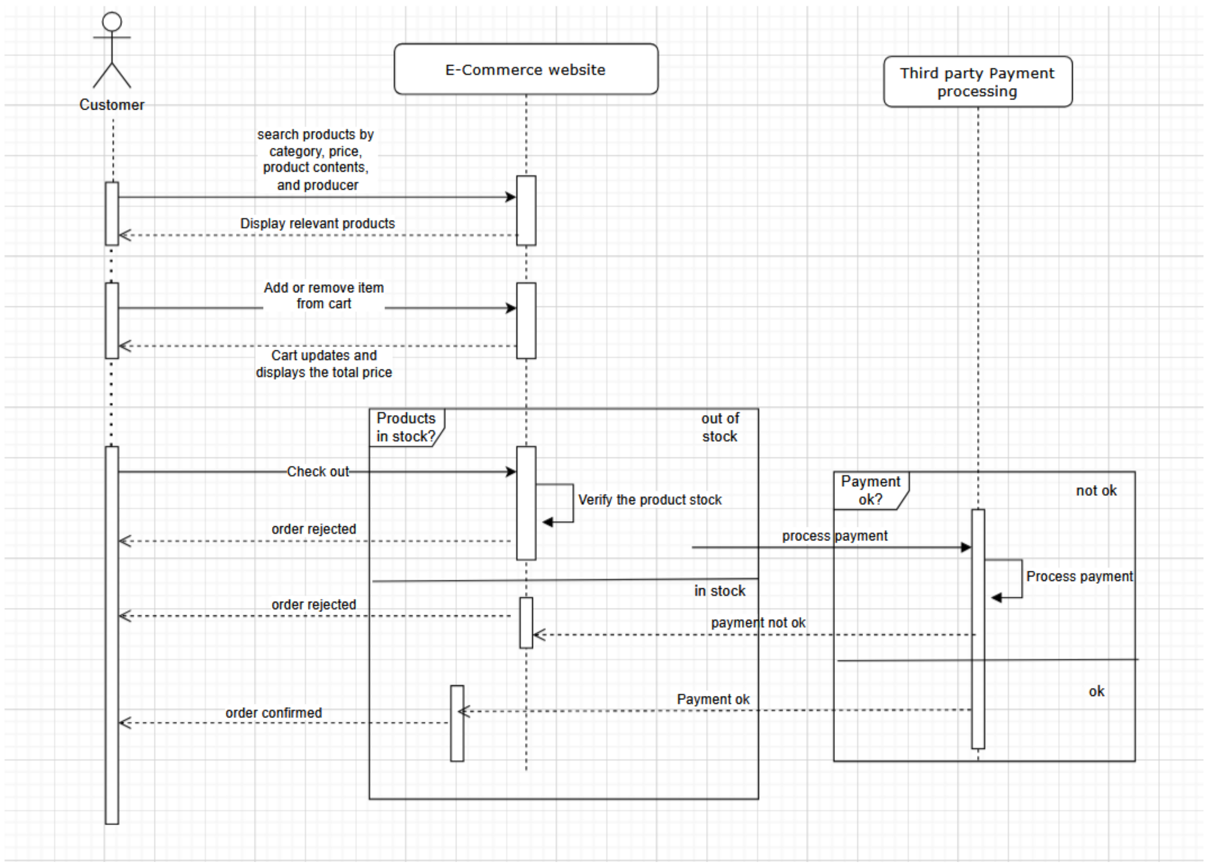
Scenario name	User is able to add or remove items from cart
Preconditions	User is able to access the eCommerce website
Test Steps	**Expected Results**
1. User adds a product to the shopping cart 2. User removes the added product to the shopping cart.	1. Product should be visible from the shopping cart after being added there. 2. Product disappears from cart upon removal.

Scenario name	User is able to check out and purchase products
Preconditions	User is able to access the eCommerce website User has products added in their shopping cart.(dependent on TC-004)

Test Steps	**Expected Results**
1. User enters checkout from shopping card view 2. User inputs payment information and checks out	1. Check out window opens, user is able to input payment information 2. Payment processor processes the payment. Payment is accepted and user sees “order confirmed” view.

Test Name	User is unable to purchase products with improper payment information
Preconditions	User is able to access the eCommerce website User has products added in their shopping cart.(dependent on TC-004)
Test Steps	**Expected Results**
1. User enters checkout from shopping card view 2. User inputs improper payment information and checks out	1. Check out window opens, user is able to input payment information 2. Payment processor processes and rejects the payment. The user is able to view feedback from the event.

Sequence diagram for user facing scenarios:



Tools

Since the product is a website implemented with React, we will use popular test automation tools for JavaScript. These include Mocha and Chai for unit testing, and Playwright or Robot Framework for automated testing involving the UI on system and acceptance testing levels. These tools are selected for their popularity, which makes learning and utilizing them easier than less popular choices. Note that not all tests should or even are possible to automate, especially on higher levels. Tools can still be used to, for example, collect results and generate reports.
Automated test reporting is produced by these frameworks, but collecting the test results into a database may also be desirable. Excel will work fine on a project of this scale.
Besides test frameworks and reports, there are many static code analysis tools included and available as extensions for VS Code, which is likely the IDE of choice for developing automated tests. AI tools such as GitHub copilot could also provide benefit in creating these.
Measurements for progress and test coverage are also reliant on tools. Most test frameworks are able to show things like code or function coverage. These are simple ways to measure how much of the application is tested and remains untested. Classifying features on their criticality means that not all tests are of equal importance.

This means that manual oversight and analysis of automated measurements is necessary.

Tests

Testing should be done throughout the project, with an emphasis on *negative* test cases. Around 80% of created tests should be verifying absence of errors, with remainder for positive test cases that verify the functional requirements. This makes sure that the application works also in the real world with users that will not read an operating manual to buy shoes online.

Since this is an eCommerce site with a variable number of concurrent users, it is also important to do performance and stress testing to determine that the application scales and performs well and according to requirements. Stress testing to find the maximum load until the breaking point is also very important. This is complicated enough that an external consultant could be considered for the first few test runs.

The focus of the component testing will be on the following key components: the product search function, the product creation function, the payment processing function, user authentication, and the communication between the frontend and backend. These areas will be thoroughly tested to ensure system stability and functionality.

In the product search function, special attention will be paid to the relevance of search results and performance, while in the product creation function, the focus will be on usability and error handling. For payment processing, emphasis will be placed on security, the functionality of various payment methods, and the handling of errors.

In terms of user authentication, the security of the login process will be tested along with the functionality of different login methods, such as usernames and passwords. The communication between the frontend and backend will be tested with a focus on data transfer speed, error handling, and data integrity.

Less critical functions, such as links redirecting to the product creator's homepage or social media pages, will be tested only at a minimal level. For these features, it will be sufficient to check that the links work as expected, but they will not be analyzed as deeply as the core functions.

The test report will include the following information: test name/code, tested function, input used in the test, and, in the case of a failure, error, or bug, a brief description of the issue. Test results will be documented, pending approval from teacher, by utilizing AI tools such as ChatGPT. This approach will allow us to quickly generate clear and readable test result documents by inputting the relevant data. This will allow us to use more time to design better test cases, ensuring the best possible outcomes in the final results.

Bugs and issues will be categorized using a prioritization method similar to the MoSCoW framework: Won't, Could, Should, and Must fix. Won't fix bugs are issues that do not significantly affect the user's ability to use the software. For example, a visual element slightly out of place would fall into this category. Bugs categorized as Could fix, while not critical, can impair the user experience or make navigation more difficult.

An example might be a text element displayed in the wrong language. Should fix bugs are more serious issues that hinder software functionality or prevent users from using specific features. Must fix bugs are critical issues that could result in physical harm, financial loss, or prevent users from accessing the software's core functions.

To guarantee comprehensive code coverage in our testing process, we will implement a combination of coverage types. We will utilize function coverage to confirm that all functions within the codebase are invoked at least once during testing. Key components will be tested using branch coverage to ensure all decision points are evaluated for both true and false outcomes. Or in cases if the software is small, we will utilize path coverage instead to ensure that all possible execution paths are exercised.

AI and testing

AI can be used as help in implementing the test cases and support for learning the tools involved. For low level unit testing it may also be used to design scenarios, but higher-level tests verifying functionality (Acceptance Tests) will be designed without AI tools. AI can also be used for general guidance and shorthand for Google or Bing search, but ultimately its results must be verified externally.
Another application of AI is to automate the generation of test reports documents, providing brief descriptions of issues and bugs while consolidating the writing style and structure of the documents. Not everyone excels at writing in a way that keeps readers or developers engaged, particularly those who may have attention disorders. This could ensure that reports are clear and concise, making them more accessible to most if not all readers.