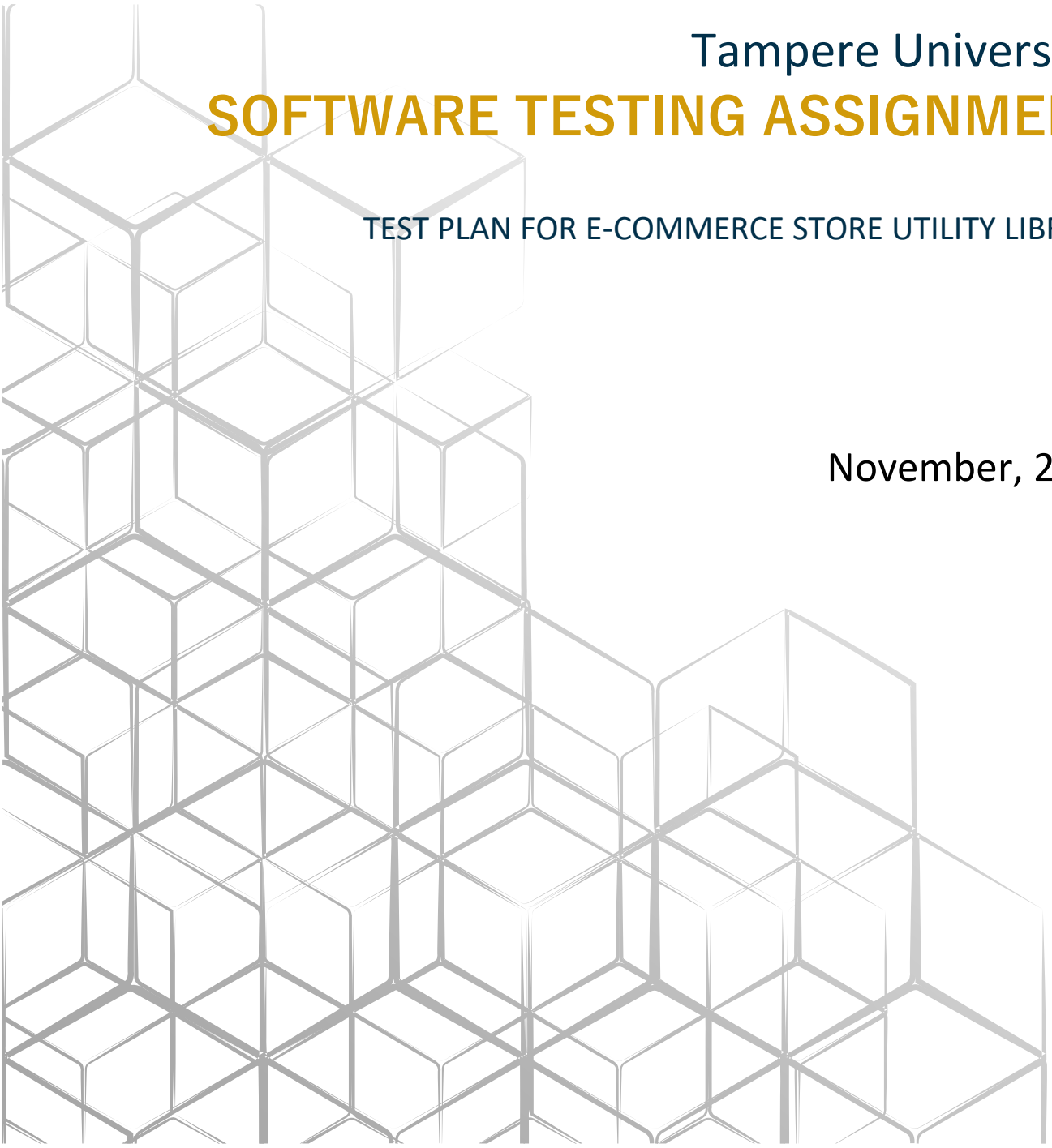
Md Ozayer Islam (151960099) & Sayem Rahman (152132363)

Software Testing

COMP.SE.200-2025-2026-1



Tampere University

SOFTWARE TESTING ASSIGNMENT

TEST PLAN FOR E-COMMERCE STORE UTILITY LIBRARY

November, 2025

Definitions, Acronyms and Abbreviations

CI/CD - Continuous Integration/Continuous Deployment, **E2E** - End-to-End, **UML** - Unified Modeling Language, **PR** - Pull Request, **XML** - eXtensible Markup Language

Utility Library - JavaScript functions for data processing, validation, and manipulation **Producer** - Food producers selling through the e-commerce platform **Consumer** - End users purchasing products **Predicate Function** - Function returning boolean values for filtering/validation **Test Coverage** - Percentage of code executed during testing

Introduction

This test plan is designed to provide a testing strategy for a utility library of an e-commerce store, selling food products from small vendors. We focused on 10 selected functions that we believe is extremely critical for this e-commerce operation.

Document Purpose

- Setting the testing scope and tools for the library functions
- Establish quality gates and success criteria
- Setup test implementation and execution plan
- Ensure reliable e-commerce functionality testing through comprehensive planning

Context of testing

Scenarios

Scenario 1: Customer Product Search and Purchase
Tpically a customer searches for different terms like "organic vegetables", “fruits”, applies price filter (i.e. 5e-15e), adds one or multiple products to cart, removes from the cart, adjust quantity, calculates the total amount and checkout.
Key Functions from utility library: words, capitalize, filter, clamp, add, reduce, isEmpty, get, toNumber

UML diagram: See appendix A: **Customer Purchase Flow Sequence Diagram**

Scenario 2: Producer Product Management
Producer normally adds products like “Apple”, enters price "2.50e/kg", enters available lot quantity, leaves category blank, validates all fields.
Key Functions from utility library: capitalize, toString, toNumber, clamp, defaultTo, isEmpty, get

UML diagram: See appendix A: **Producer Product Management Activity Diagram**

Scenario 3: Shopping Cart Management
Customer typically adds items, updates quantities, removes products, views formatted totals.

Key Functions from utility library: add, toNumber, map, reduce, clamp, filter, compact, toString, capitalize

UML diagram: See appendix A: **Shopping Cart Management Activity Diagram**

Testing scope

In Scope: Unit testing of 10 selected functions, input validation, edge cases, mathematical operations, string/array processing.

Out of Scope: .internal directory, React frontend, REST API, third-party services, performance/security testing that might be in the repository.

Selected 10 Functions (Risk-based prioritization):

1. **isEmpty** - Form validation (Critical)
2. **get** - Safe object access (Critical)
3. **filter** - Product search (Critical)
4. **add** - Price calculations (Critical)
5. **toNumber** - Type conversion (High)
6. **reduce** - Cart totals (High)
7. **map** - Data transformation (High)
8. **defaultTo** - Optional fields (High)
9. **clamp** - Range validation (Medium)
10. **capitalize** - Text formatting (Medium)

Testing related assumptions

- Functions operate in modern JavaScript ES6+ environment
- Input data types consistent with e-commerce scenarios
- Synthetic test data will simulate realistic e-commerce operations
- Integration testing simulated through comprehensive unit testing
- Functions handle typical e-commerce data volumes (hundreds of products, dozens of cart items)
- Error handling managed by consuming application layers
- Coverage tools accurately measure even if we exclude .internal directory

Tools and reporting

Tools and reporting

Selected Testing Tools

Jest - Primary testing framework

- Purpose: Test runner, assertion library, mocking capabilities
- Justification: Comprehensive solution with ES6 module support

- Role in E2E scenarios: Execute unit tests for all utility functions used in customer purchase, producer management, and cart operations

C8 - Code coverage measurement

- Purpose: Native V8 coverage reporting
- Justification: Accurate ES6 module coverage, easily excludes .internal directory
- Role in E2E scenarios: Measure coverage of critical functions (isEmpty, get, filter, add) ensuring business-critical paths are tested

Coveralls - Coverage service (this is mandatory)

- Purpose: Online coverage tracking and reporting
- Justification: Required by assignment specifications
- Role in E2E scenarios: Monitor coverage trends for e-commerce critical functions

GitHub Actions - CI/CD pipeline (this is mandatory)

- Purpose: Automated testing on code changes
- Justification: Required by specifications and a popular CI/CD tool in industry projects
- Role in E2E scenarios: Ensure all scenario-supporting functions pass tests before deployment

Environment Setup

- Node.js 16+ with ES6 module support
- Installation: npm install --save-dev jest c8 coveralls
- Configuration: package.json scripts, jest.config.js, GitHub workflow

Reporting

We should have the following reporting once we have completed implementing this testplan.

Test Progress Reporting

- Real-time console output during development
- GitHub Actions summary for each commit/PR
- Coverage percentage tracking in Coveralls dashboard

Test Results Documentation

- JUnit XML reports for CI/CD integration
- HTML coverage reports for detailed analysis
- Test execution logs with pass/fail status

Defect Tracking

- **Tool**: GitHub Issues
- **Lifecycle**: Open → In Progress → Testing → Closed
- **Priority Levels**: Critical (business-breaking), High (feature-breaking), Medium (minor issues), Low (cosmetic)

- **Criticality Levels**: Blocker, Major, Minor, Trivial
- **Templates**: Bug report template with steps to reproduce, expected vs actual results

Quality gates and metrics

Entry Criteria

- Testing environment setup complete (Jest, C8, GitHub Actions configured)
- All 10 selected functions stable and documented
- Test data prepared for e-commerce scenarios

Exit Criteria

- Minimum 80% line coverage for all selected functions
- Minimum 75% branch coverage
- All unit tests pass consistently
- No critical/high severity defects unresolved
- CI/CD pipeline executes successfully

Coverage Expectations

- **Line Coverage**: 80% minimum per function
- **Branch Coverage**: 75% minimum for conditional logic
- **Function Coverage**: 80% minimum overall
- **Exclusions**: .internal directory, system-level error handling

Success Metrics

- Test-to-code ratio: minimum 2:1
- Test execution time: under 30 seconds for full suite
- CI/CD pipeline duration: under 5 minutes
- Zero critical defects in production-ready functions

Tests

Testing Strategy

Unit Tests (Best possible way for our library)

- Purpose: Validate individual utility functions in isolation
- Importance: Critical for e-commerce reliability as it ensures data processing, validation, and calculations work correctly
- Depth: Comprehensive testing including normal cases, edge cases, and error conditions

Integration Tests (Simulated: Somehow anticipated flow)

- Purpose: Validate function interactions within the e-commerce workflows
- Implementation: Chain multiple utility functions as used typically in real life scenarios
- Examples: Product search workflow (words => capitalize => filter), cart calculation pipeline (add => reduce => clamp)

More Tests: good to have as complexity grows

- **Regression Testing**: Automated execution in CI/CD pipeline
- **Performance Testing**: Basic validation with typical e-commerce data volumes (1000+ products)
- **End-to-End Scenario Validation**: Simulate complete user journeys through comprehensive unit test sequences

Detailed Test Case Design

Function 1: isEmpty(input value) - Priority: Critical: This method will validate form fields and product data to ensure that required information is present such as checking name of the product, price, quantity etc. are empty before even processing those.

Normal Cases:

Test case number	IsEmpty input value	Expected output value
1	**null**	**true**
2	**“" - empty string**	**true**
3	**[] - empty array**	**true**
4	**undefined**	**true**
5	**{} - empty object**	**true**
6	**“any string”**	**false**
7	**[1, “Hello”]**	**false**

Edge and error Cases:

Test case number	IsEmpty input value	Expected output value
8	**false**	**true**
9	**0**	**true**
10	**“ “**	**false**
11	function () {}	**true**

Function 2: get(object, path, default value) - Priority: Critical: This method will be used to access objects and their properties such as product details, user data, cart information, etc.

Normal Cases:

Test case number	get input value	Expected output value
12	**{a: {b:2}}, ‘a, b’**	**2**
13	**{a:1}, ‘a’**	**1**
14	**{a: {b:2}}, [‘a, b’]**	**2**

15	**{}, 'missing value', 'default value'**	**'default value'**

Edge and error Cases:

Test case number	get input value	Expected output value
16	**null, 'a'**	**undefined**
17	**{a:1}, null**	**undefined**
18	**{a:1}, ''**	**undefined**

General guidance to be followed during test design

Mathematical Functions **(add, clamp, toNumber, reduce):**

- Tests valid numeric inputs and string conversions
- Tests invalid inputs (NaN, Infinity, null, undefined)
- Tests boundary conditions and precision handling

Array Functions **(filter, map):**

- Tests empty arrays, single elements, large datasets
- Tests various data types and invalid predicates

String Functions **(capitalize):**

- Tests normal strings, empty strings, special characters
- Tests unicode and very long strings

Utility Functions **(defaultTo):**

- Tests null/undefined values vs falsy values
- Tests with complex objects and arrays

Test Implementation Standards

- Arrange-Act-Assert pattern to be followed
- Descriptive test names indicating scenario and expected outcome
- Reusable test data fixtures for complex scenarios, separate json stubs as test data

AI and testing

AI Advantages in E-commerce Testing

Test Case Generation: AI can analyze any number of functions and generate comprehensive edge cases for various scenarios specific to food e-commerce.

Test Data Creation: AI can generate realistic product catalogs, user profiles, and transaction data that mirrors actual food producer inventories with edge cases like missing categories or extreme prices with ease and with a very less time-consuming manner

Bug Pattern Recognition: AI can identify potential issues with currency calculations, string formatting, or data validation commonly occurring in e-commerce applications.

AI Disadvantages in E-commerce Testing

Context Understanding: AI may not grasp e-commerce business nuances, creating technically valid but unrealistic test cases that don't reflect actual user behavior or business requirements, especially when e-commerce varies from time to time, case to case.

Over-reliance Risk: Testers might lose critical thinking skills for identifying subtle but critical issues like currency rounding errors or cultural considerations in text processing.

Quality vs Quantity: AI might generate numerous tests without meaningful coverage, focusing on quantity over business-relevant scenarios.

AI Usage Declaration

- Introduction and definition section: shortened the definition of terms for using less space.
- Testing scopes: Validated our scenario to find out if we missed any important case. We also learnt through AI what all these functions can do in our e-commerce product. This helped us prioritize library functions.
- Tools and reporting: AI has data to our previous projects or skill base; we selected some of the tools based on what we have already done previously or if some tool is like any of our previously used tools. We haven't used some of the tools yet, so we had to get help from AI about how those tools flow looks like.
  Also for quality gates and metrics, we had very little previous knowledge, so we gathered from different search results what they are and set up the criteria that is generally a practice in real world industry.
- Tests: We did a little bit of research into what tests we could include in our e-commerce platform and came up with a few options.
  In the test case design, no help was taken from AI in any way whatsoever. We have done a thorough study of these to make sure the tests are comprehensive.
- UML: The UMLs (Sequence and Activity diagrams) were polished using AI to make sure the flows and transitions look smooth and meaningful.
- References and Appendix: We took help in formatting these sections as these always seem to confuse us, especially how do we create a link properly.

Overall, we did a thorough grammar check to make sure we use simple and meaningful sentences and keep the formatting as clean as possible. We created a template of the required points first so that we do not forget any part of this assignment.

References

Jest Testing Framework https://jestjs.io/

C8 Code Coverage Tool https://www.npmjs.com/package/c8

Coveralls Test Coverage Service https://www.npmjs.com/package/coveralls

GitHub Actions Documentation https://github.com/features/actions

Web Sequence Diagrams https://www.websequencediagrams.com/

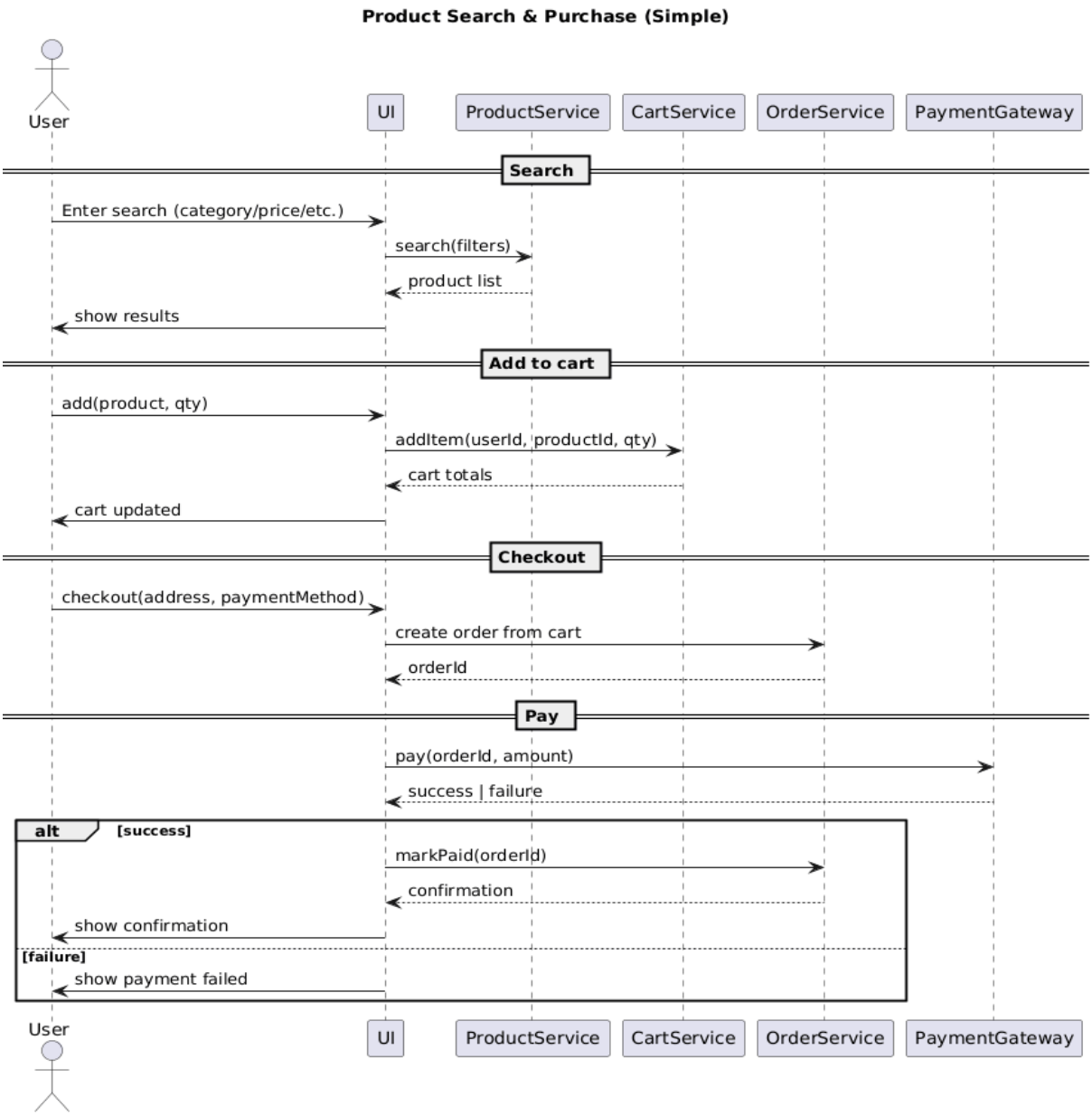
JavaScript Testing Best Practices https://github.com/goldbergyoni/javascript-testing-best-practices

Risk-Based Testing Approach https://www.guru99.com/risk-based-testing.html

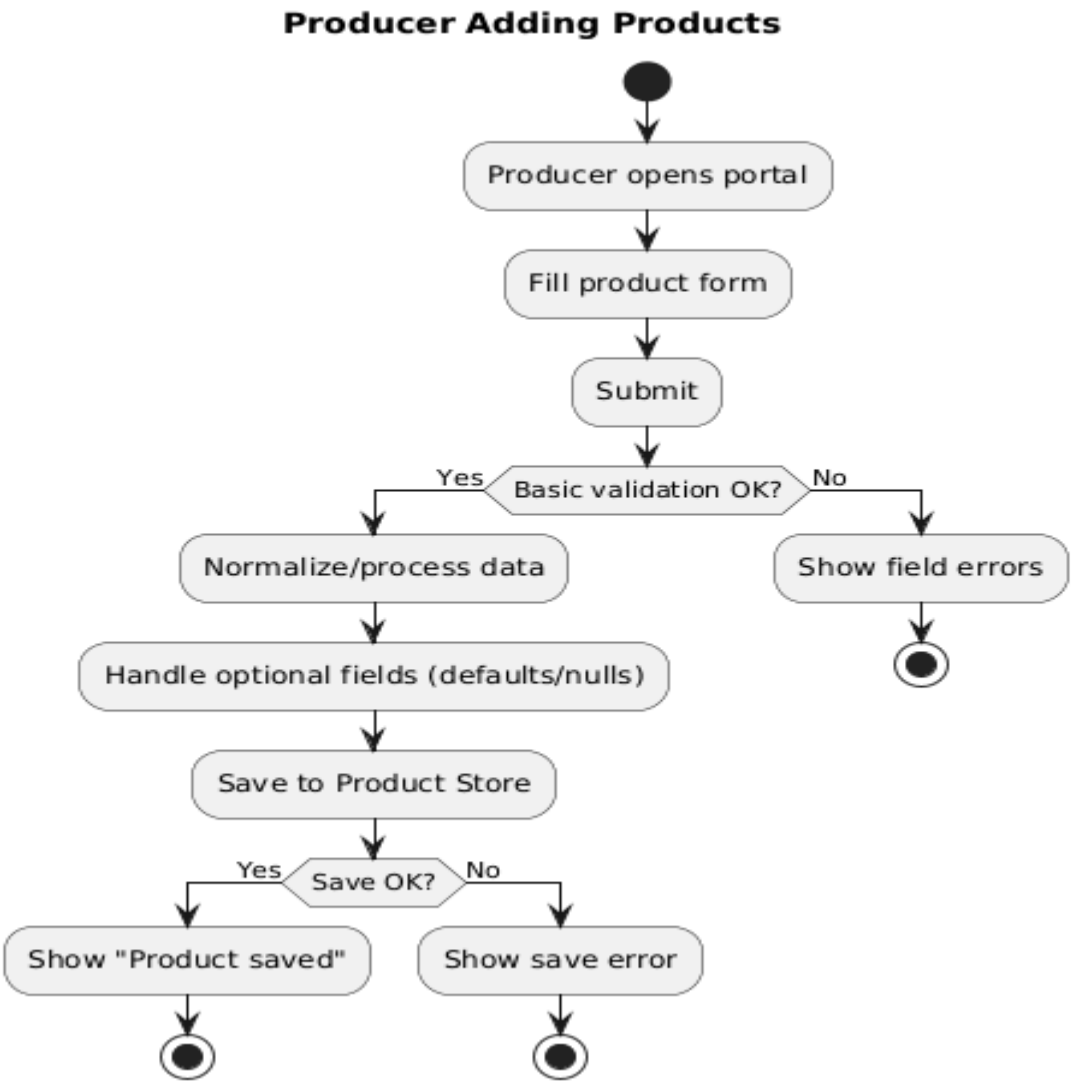
Appendix

Appendix A: UML Diagrams

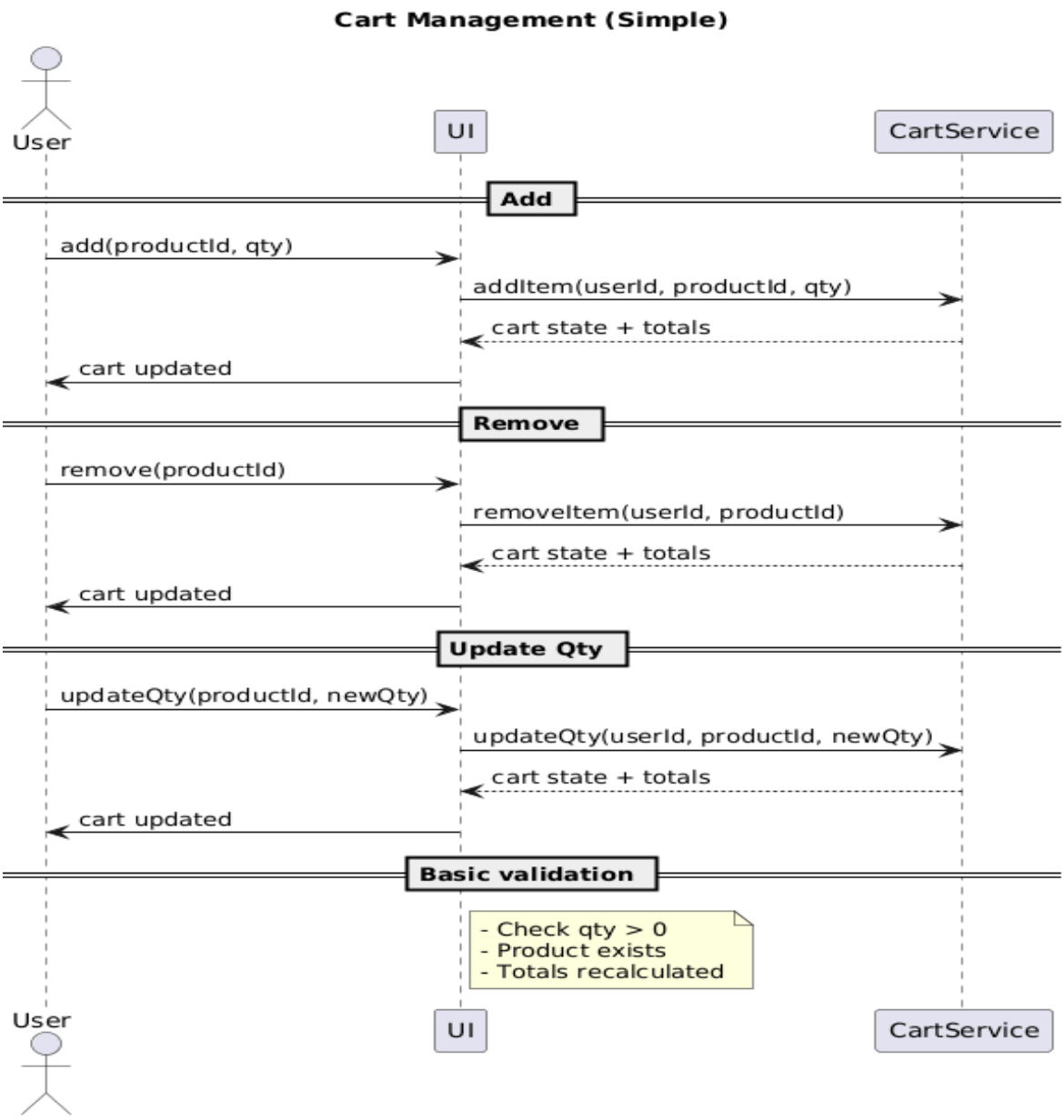
Scenario 1: Customer Purchase Flow Sequence Diagram



Scenario 2: Producer Product Management Activity Diagram



Scenario 3: Shopping Cart Management Activity Diagram



Appendix B: Configuration Examples

GitHub Actions Workflow: Name: Test, Triggers: push/pull_request Steps: checkout → setup-node → npm install → npm run test:coverage → coveralls

Appendix C: Test Data Samples

Product Test Data:

- Valid: {name: "Organic Honey", price: 12.50, category: "Sweeteners"}
- Missing fields: {name: "Artisan Bread", price: 8.99, category: null}
- Edge cases: {name: "", price: 0.01, category: ""}

Cart Test Data:

- Empty cart: []
- Single item: [{product: "Honey", quantity: 1, price: 12.50}]
- Multiple items: [{...}, {...}]