

C868 – Software Capstone Project Summary

Task 2 – Section C



Capstone Proposal Project Name: Parts and Products Manager Improvement

Student Name: Jonathan Kleve

Table of Contents

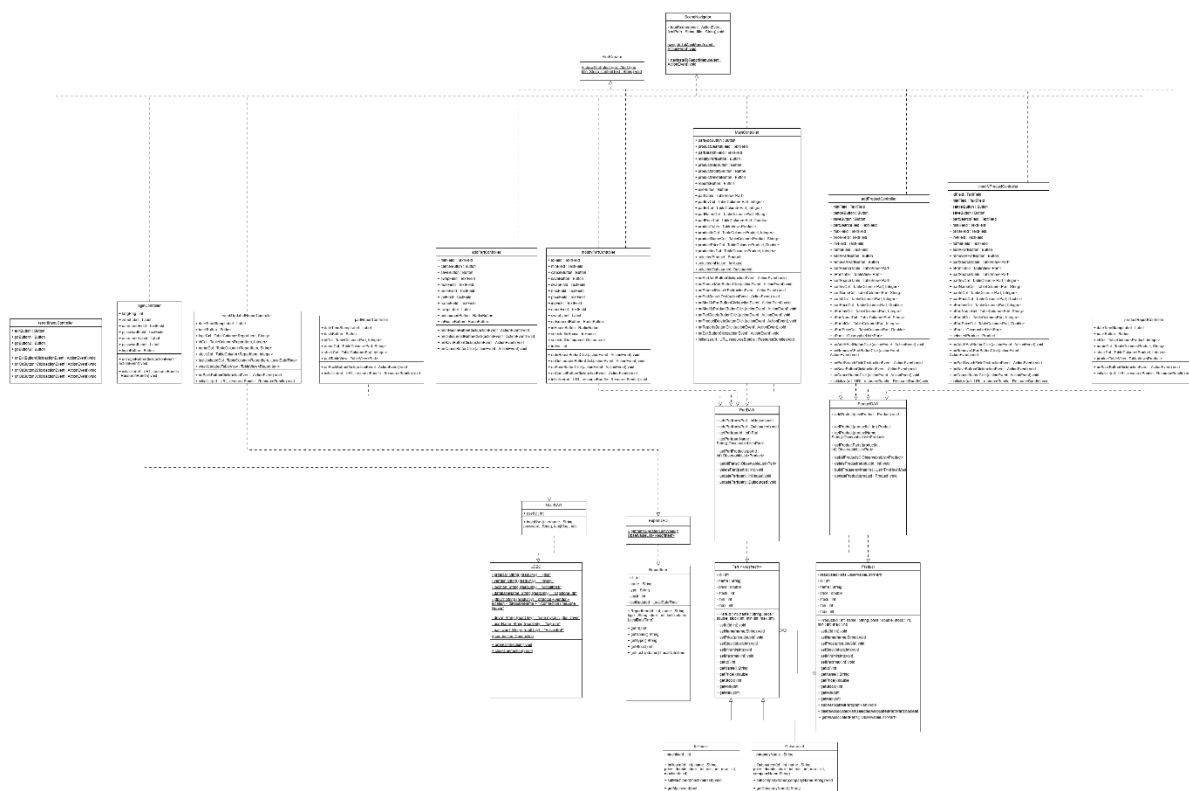
<i>Table of Contents.....</i>	<i>2</i>
<i>Design Documents</i>	<i>4</i>
Class Design	4
UI Design	5
<i>Unit Test Plan.....</i>	<i>5</i>
Introduction	5
Purpose	6
Overview	6
Test Plan	7
Items	7
Features.....	8
Deliverables	8
Tasks	9
Needs	9
Pass/Fail Criteria.....	10
Specifications.....	12
Procedures.....	12
Results.....	15
<i>C4. Source Code.....</i>	<i>16</i>
<i>Parts and Products Manager - Developer & Maintenance Guide.....</i>	<i>17</i>
Introduction	17
Prerequisites.....	17

Project Setup	17
Database Setup	18
This application requires a MySQL database.	18
Create the Database	18
Create Tables and Populate Initial Data:.....	18
Database Connection Configuration	18
Setting up the Project in IntelliJ IDEA	19
Building the Project	19
Running the Application from IntelliJ IDEA	20
Running Unit/Integration Tests	20
Troubleshooting	21
<i>Parts and Products Manager – User Manual</i>	22
Introduction	22
System Requirements	22
Getting Started	22
Login	22
Using the application	23
Troubleshooting	24
Support	24

Design Documents

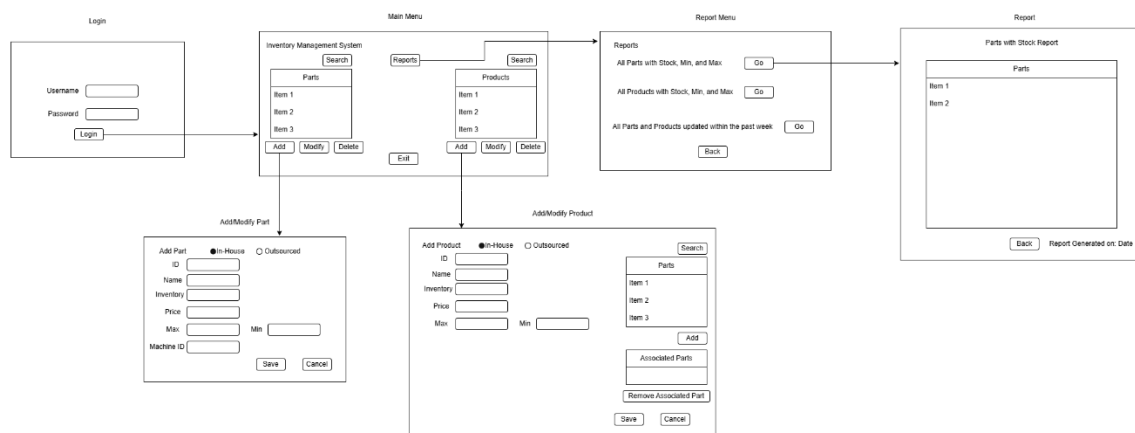
Class Design

This UML class diagram provides a static, high-level view of the core components within the Products and Parts Manager application. It details the various classes that make up the system, showcasing their attributes (data) and operations (behaviors). Crucially, the diagram also illustrates the fundamental relationships between these classes, such as associations, dependencies, and inheritance, offering a comprehensive representation of the application's structural design. The full-size image file, "Capstone UML Diagram.png," is included in this submission for easier viewing.



UI Design

This section provides a low-fidelity UI design diagram illustrating the various menus and views within the Part and Product Manager application. It visually represents the user interface screens and uses arrows to indicate the navigation paths and flow between them, offering a comprehensive overview of the application's user experience structure. The full-size image file, "Capstone UI Diagram.png," is included in this submission for easier viewing.



Unit Test Plan

Introduction

This section details the unit test performed on the PartDAO class, a core component of the inventory management application. The PartDAO handles all Create, Read, Update, and Delete (CRUD) operations for Part objects within the system's persistent storage, specifically a MySQL database. This testing was conducted to ensure the reliability and correctness of data interactions, which are fundamental to the application's functionality.

Purpose

The primary purpose of testing the PartDAO was to validate its integrity and functionality in managing Part data. This process involved verifying that Part objects could be accurately added to, retrieved from, updated within, and deleted from the database.

The testing method employed was integration testing, leveraging the JUnit 5 framework. This approach allowed for a comprehensive assessment of the PartDAO's interactions with the actual database via the JDBC utility class. The results successfully validated all tested PartDAO operations, indicating that the class performs its intended data persistence tasks correctly. No immediate remediation was required for the core functionalities tested, as all assertions passed during the initial test execution. If failures had occurred, remediation would involve debugging the specific PartDAO method or the underlying JDBC interaction, correcting the code, and re-running the affected test cases until they passed.

Overview

The PartDAO test represents a crucial segment of the overall project's quality assurance. As the data access layer, the PartDAO bridges the application's business logic (managed by controllers) and the persistent storage (the MySQL database). Ensuring the correctness of PartDAO is paramount because any data corruption or retrieval errors at this level would directly impact the entire application's functionality and data integrity.

This testing method, focusing on CRUD operations via DAOs, is a standard and repeatable approach used across the application's persistence layer. A similar testing methodology would be (and should be) applied to the ProductDAO and any other data access objects to ensure consistent data handling. This approach is not unique to PartDAO but exemplifies the standard practice for validating all DAO components.

Specifically, the following functions of PartDAO were tested:

- Adding Parts (addPart()): Verify that new Part objects (including subclasses like InHouse and Outsourced) have been successfully inserted into the database.
- Retrieving Parts by ID (lookupPart(int id)): Confirmation that a Part can be accurately fetched using its unique identifier.
- Retrieving Parts by Name (lookupPart(String name)): Validation that a Part (or list of Parts) can be found using its name.
- Updating Parts (updatePart(int id, Part part)): Ensure that existing Part records can be modified in the database with new data.
- Deleting Parts (deletePart(int id)): Proof that Part records can be permanently removed from the database.
- Retrieving All Parts (getAllParts()): Confirmation that all Part records currently in the database can be retrieved correctly.

Tests were conducted by programmatically invoking the PartDAO methods and then using JUnit 5 assertions to compare the actual database state or method returns against expected outcomes. Errors (such as SQLExceptions) encountered during test execution were caught and printed to the console. The JUnit fail() assertion was used to mark the test as failed, providing immediate feedback on database connectivity or SQL issues. This direct feedback mechanism facilitates rapid identification and debugging of persistence-related problems.

Test Plan

Items

The following are required to complete these tests:

- Integrated Development Environment (IDE): IntelliJ IDEA, Eclipse, or similar.

- Java Development Kit (JDK): Version 11 or higher (as typically required for JavaFX and modern JUnit).
- Maven: as a build automation and dependency management tool
- JUnit 5 Library: org.junit.jupiter:junit-jupiter-api and org.junit.jupiter:junit-jupiter-engine.
- MySQL Database: A running MySQL server instance.
- MySQL JDBC Driver: The com.mysql.cj.jdbc.driver accessible by the application.
- Database Schema: The capstone_db database must have a parts table correctly structured to store Part, InHouse, and Outsourced data.

Features

Each test method in PartDAOTest focuses on a specific PartDAO function/feature:

- testAddInHousePart(): Tests the addPart() feature for InHouse type.
- testLookupPartById(): Tests the lookupPart(int) feature.
- testLookupPartByNameExact(): Tests the lookupPart(String) feature.
- testUpdatePart(): Tests the updatePart() feature.
- testDeletePart(): Tests the deletePart() feature.
- testGetAllParts(): Tests the getAllParts() feature.

Deliverables

Upon completion, the test(s) would produce the following deliverables:

- JUnit Test Report: Generated by the IDE or build tool, indicating pass/fail status for each test method.
- Console Output: Logs from the test execution, including system print statements for connection status and any error stack traces.

- Updated Codebase: The PartDAOTest.java source file itself serves as documented test cases for future reference and regression testing.

Tasks

The following tasks were required to complete the testing process:

1. JUnit 5 Integration: Add JUnit 5 dependencies to the project's build configuration (e.g., pom.xml for Maven).
2. Database Configuration: Ensure the MySQL database is running and accessible as configured in the JDBC class. Verify that the capstone_db and parts table exist.
3. Test Class Creation: Create PartDAOTest.java within the src/test/java directory.
4. Test Setup (@BeforeAll, @BeforeEach): Implement methods to open the database connection once and clear the parts table before each test to ensure test independence.
5. Test Case Implementation (@Test): Write individual test methods for each PartDAO function, following the Arrange-Act-Assert pattern.
6. Assertion Implementation: Use JUnit 5 Assertions to verify expected outcomes.
7. Test Teardown (@AfterAll): Implement a method to close the database connection after all tests are complete.
8. Test Execution: Run the tests via the IDE or build tool.
9. Results Review: Analyze the JUnit test report and console output for pass/fail status and any errors.

The outcomes identified were that all implemented test cases for PartDAO passed successfully on the initial run, indicating robust functionality for the primary data operations.

Needs

To perform these tests, the following support items and technical requirements had to be in place:

- Java 11+: The application is built with Java, and JUnit 5 requires Java 8 or higher. Version consistency is crucial.
- MySQL Database Server: An instance of MySQL server (e.g., MySQL 8.0) running and accessible from the development environment.
- capstone_db Database: A database named capstone_db was created on the MySQL server, along with a parts table that matches the schema expected by the PartDAO and Part model classes.
- MySQL Connector/J: The JDBC driver for MySQL (com.mysql.cj.jdbc.driver).
- JUnit 5 Libraries: As specified under "Items," the junit-jupiter-api and junit-jupiter-engine Maven dependencies were explicitly identified and employed.

Pass/Fail Criteria

The criteria used to determine the success of each test were based on JUnit 5 assertions.

1. Positive Result Protocol (Pass): A test was considered successful if all assertions within its method executed without throwing an AssertionError and no uncaught exceptions occurred during its execution. A green checkmark or a "Passed" status indicates a positive result in the JUnit test runner's interface. These indicators signify that the tested PartDAO method performed exactly as expected, correctly manipulating or retrieving data from the database.
2. Recourse for Failure (Fail): If any assertion failed or an unexpected exception was thrown, the test was immediately marked as "Failed" (typically indicated by a red 'X' or "Failed" status).
 - Remediation Strategies:

1. Analyze Error Message & Stack Trace: The first step was to examine the detailed error message provided by JUnit and the stack trace to pinpoint the exact line of code where the failure occurred.
 2. Inspect Database State: For database-related failures, verifying the actual state of the parts table directly in MySQL was critical.
 3. Debugging: Use the IDE's debugger to step through the failing test method and the PartDAO code, observing variable values and execution flow.
 4. Code Correction: Implement fixes in the PartDAO logic or the underlying SQL queries.
 5. Re-test: Re-run the specific failed test (and potentially related tests) to confirm the fix.
- Documentation Requirements: Any failure would necessitate updating the test report, including a description of the failure, the root cause identified, and the remediation steps taken. This documentation is crucial for regression analysis and understanding system behavior.

Specifications

Below is a sample of the testing code used for PartDAOTest, specifically demonstrating the structure and a few key test cases. The full testing code is included in the source code file included in this submission at `/src/test/java/kleve/testtwo/DAO/PartDAOTest.java`.

```

20  class PartDAOTest {
56
57      /**
58       * Test case for adding an InHouse part.
59       * Annotated with @Test for JUnit 5.
60       */
61      @Test
62      @DisplayName("1. Test adding an InHouse part successfully")
63      void testAddInHousePart() {
64          System.out.println("Running testAddInHousePart...");
65          PartDAO.addPart(testInHousePart);
66
67          ObservableList<Part> allParts = PartDAO.getAllParts();
68          assertNotNull(allParts, message: "getAllParts should not return null");
69          assertFalse(allParts.isEmpty(), message: "Part list should not be empty after adding");
70          assertEquals(expected: 1, allParts.size(), message: "Should contain exactly one part after adding");
71
72          Part retrievedPart = allParts.get(0); // Get the first (and only) part, assuming sequential IDs or just one part
73          // testPartId = retrievedPart.getId(); // Storing ID if needed for subsequent (chained) tests, but @BeforeEach cleans up.
74
75          // Assert that the retrieved part's details match the added part's details
76          assertEquals(testInHousePart.getName(), retrievedPart.getName(), message: "Retrieved part name should match");
77          assertEquals(testInHousePart.getPrice(), retrievedPart.getPrice(), message: "Retrieved part price should match");
78          assertTrue(retrievedPart instanceof InHouse, message: "Retrieved part should be an InHouse instance");
79          assertEquals(testInHousePart.getMachineId(), ((InHouse) retrievedPart).getMachineId(), message: "Retrieved machine ID should match");
80
81          System.out.println("TestAddInHousePart Passed.");
82      }

```

Procedures

The following detailed steps were used to complete the testing process:

1. Configure Project for JUnit 5:
 - Opened the project's pom.xml
 - Added the junit-jupiter-api and junit-jupiter-engine dependencies with test scope.
 - Ensured the maven-surefire-plugin was configured for Maven
2. Database Readiness Check:
 - Verified that the MySQL server was running.
 - Confirmed that the capstone_db database existed.
 - Ensured the parts table had the correct schema.

- Confirmed that JDBC.java had the correct username and password for the database.
3. Create Test Class:
- Created PartDAOTest.java in the src/test/java/kleve/testtwo/DAO directory.
4. Implement `@BeforeAll` Setup:
- Defined the static `setUpAll()` method.
 - Called `JDBC.openConnection()` within `setUpAll()` to establish a single database connection for the entire test suite.
5. Implement `@BeforeEach` Setup:
- Defined the non-static `setUpEach()` method.
 - Added code within `setUpEach()` to execute a `DELETE FROM parts` SQL statement to ensure that the parts table is empty before *each* test method runs, preventing test contamination and ensuring tests are independent.
 - Initialized fresh InHouse and Outsourced test Part objects for use in the current test.
6. Implement Test Cases (`@Test` Methods):
- For each function to be tested (`addPart`, `lookupPart` by ID/name, `updatePart`, `deletePart`, `getAllParts`), a separate void method was created and annotated with `@Test`.
 - Each test method was given a descriptive name using `@DisplayName`.
 - Inside each test method, the Arrange-Act-Assert pattern was followed:
 - Arrange: Set up any specific data needed for that test (e.g., creating a new Part object).

- Act: Call the PartDAO method being tested.
- Assert: Use JUnit 5 Assertions (e.g., assertEquals, assertNotNull, assertTrue, assertNull, assertFalse) to verify the outcome against the expected result.

7. Implement @AfterAll Teardown:

- Defined the static tearDownAll() method.
- Called JDBC.closeConnection() within tearDownAll() to close the database connection after all tests in the class have finished.

8. Execute Tests:

- Accessed the test runner functionality within the IDE (e.g., right-clicking PartDAOTest.java and selecting "Run 'PartDAOTest'").
- The test runner automatically discovered and executed all @Test methods.

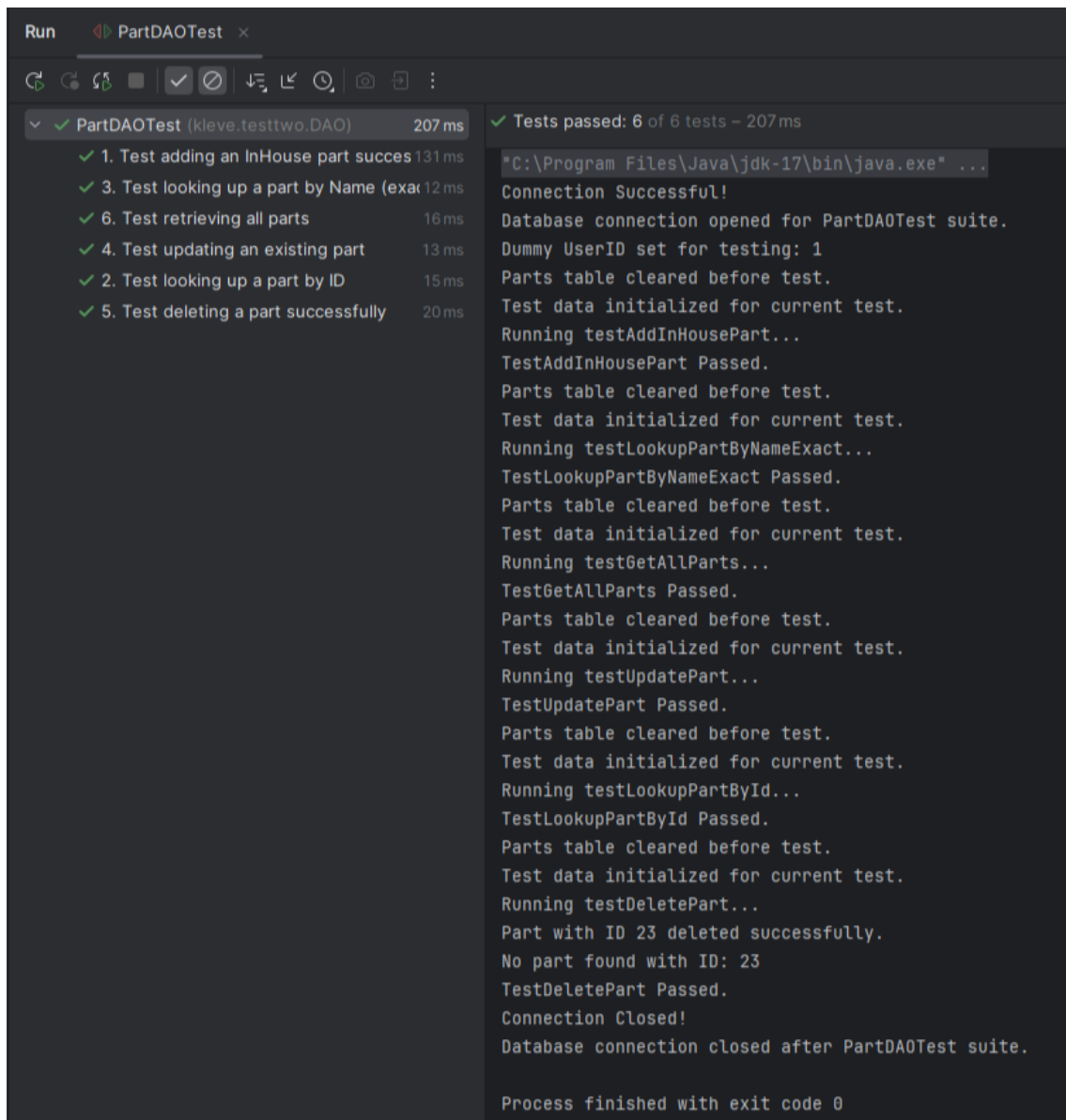
9. Review Results:

- Observed the JUnit test report provided by the IDE.
- Checked the console output for any System.out.println messages or error stack traces.

This process involves iterations during development. Initially, tests might fail due to bugs in the PartDAO or incorrect assumptions in the test itself. Each failure prompts a debugging cycle (as described in Pass/Fail Criteria) and code correction, followed by re-execution of the relevant tests until they pass. Pass/fail results are provided immediately after each test run by the JUnit test runner, allowing for rapid feedback and continuous refinement.

Results

Initial results indicated an error with the test code as the user ID value was not initialized, but this was quickly remedied by assigning it in the setupAll method. Subsequently, the unit tests for PartDAO yielded successful results across all tested functionalities. These results indicate that Part objects can be reliably created, retrieved, updated, and deleted from the database through the PartDAO interface.



```
Run PartDAOTest x
[Icons] [Check] [Cancel] [Run] [Debug] [Test] [Exit] [Close] [Maximize] [Full Screen] [Help]
✓ PartDAOTest (kleve.testtwo.DAO) 207 ms
  ✓ 1. Test adding an InHouse part succes 131 ms
  ✓ 3. Test looking up a part by Name (exa 12 ms
  ✓ 6. Test retrieving all parts 16 ms
  ✓ 4. Test updating an existing part 13 ms
  ✓ 2. Test looking up a part by ID 15 ms
  ✓ 5. Test deleting a part successfully 20 ms
✓ Tests passed: 6 of 6 tests - 207ms
"C:\Program Files\Java\jdk-17\bin\java.exe" ...
Connection Successful!
Database connection opened for PartDAOTest suite.
Dummy UserID set for testing: 1
Parts table cleared before test.
Test data initialized for current test.
Running testAddInHousePart...
TestAddInHousePart Passed.
Parts table cleared before test.
Test data initialized for current test.
Running testLookupPartByNameExact...
TestLookupPartByNameExact Passed.
Parts table cleared before test.
Test data initialized for current test.
Running testGetAllParts...
TestGetAllParts Passed.
Parts table cleared before test.
Test data initialized for current test.
Running testUpdatePart...
TestUpdatePart Passed.
Parts table cleared before test.
Test data initialized for current test.
Running testLookupPartById...
TestLookupPartById Passed.
Parts table cleared before test.
Test data initialized for current test.
Running testDeletePart...
Part with ID 23 deleted successfully.
No part found with ID: 23
TestDeletePart Passed.
Connection Closed!
Database connection closed after PartDAOTest suite.
Process finished with exit code 0
```

The green checkmarks next to each test method confirm successful execution, and the summary indicates that all six tests passed. The console output verifies the setup and teardown procedures, confirming that the database connection was managed correctly, and that the table was cleared before each test. This robust set of passing tests instills confidence in the PartDAO's ability to handle part data reliably.

C4. Source Code

The source code is included in this submission inside the TestTwo.zip file.

Parts and Products Manager - Developer & Maintenance Guide

Version: 1.0 Date: June 11, 2025

Introduction

This guide provides instructions for setting up, building, and running the Parts and Products Manager application from a developer or maintenance perspective. It covers the necessary prerequisites, database configuration, IDE setup, and execution steps required to maintain, debug, or extend the application.

Prerequisites

Before proceeding, ensure you have the following software installed on your system:

- Java Development Kit (JDK): Version 17 or higher.
 - Verify by running `Java -version` and `javac -version` in your terminal.
- Apache Maven: Version 3.8.x or higher.
 - Verify by running `mvn -v` in your terminal.
- MySQL Server: Version 8.0 or higher.
 - Ensure the MySQL server is running and accessible.
- MySQL Workbench (or similar SQL Client): For database management and schema setup.
- IntelliJ IDEA (Community or Ultimate Edition): Recommended Integrated Development Environment.

Project Setup

1. Download as Zip
2. Unzip the archive to your desired development directory
3. Navigate into the unzipped project folder

Database Setup

This application requires a MySQL database.

Create the Database

- Open MySQL Workbench (or your preferred SQL client).
- Connect to your MySQL server.
- Execute the following SQL command to create the database:
 1. `CREATE DATABASE IF NOT EXISTS capstone_db; -- Or your specific database name`
- Switch to the newly created database via the following command:
 1. `USE capstone_db;`

Create Tables and Populate Initial Data:

- Locate the SQL script containing the table schemas and initial data. This is typically found in a database or sql folder within the project, e.g., `src/main/resources/database/schema.sql`.
- Execute the contents of this SQL script within your `capstone_db` database. This will create the parts, products, users, etc., tables and populate them with any necessary starting data.
- Verify: Run a simple `SELECT * FROM parts;` to confirm the tables exist and data is present.

Database Connection Configuration

The application connects to the MySQL database via JDBC. The connection parameters are defined in your `JDBC.java` utility class (e.g., `kleve.testtwo.DAO.JDBC.java`).

- Open JDBC.java: Navigate to src/main/java/kleve/testtwo/DAO/JDBC.java (or your equivalent path).
- Verify/Update Connection Details: Ensure the dbURL, user, and password variables match your MySQL server configuration.

Setting up the Project in IntelliJ IDEA

- Open IntelliJ IDEA
- Import Project:
 - Select File -> Open...
 - Navigate to the root directory of your project (the folder containing pom.xml).
 - Click Open.
- Maven Import: IntelliJ IDEA should automatically detect the pom.xml file and prompt you to import the Maven project. If not, click the "Reload All Maven Projects" button in the Maven tool window (usually on the right side, a circular blue arrow icon).
 - Allow IntelliJ IDEA to download all necessary dependencies. This may take some time depending on your internet connection.
- Verify Project Structure
 - Ensure src/main/java and src/test/java are correctly marked as Source Root (blue) and Test Source Root (green), respectively. If not, right-click the folder -> Mark Directory as -> [Source/Test Sources Root].

Building the Project

- From IntelliJ IDEA:
 - Open the Maven tool window.
 - Expand Lifecycle.

- Double-click clean to clear previous builds.
- Double-click install to compile the code, run tests, and package the application into a .jar file.
- The compiled .jar file will be located in the target/ directory (e.g., target/TestTwo-1.0-SNAPSHOT.jar).
- From Terminal:
 - Navigate to the project root directory in your terminal.
 - Run: mvn clean install

Running the Application from IntelliJ IDEA

- Locate Main Class: In your src/main/java directory, find your main application class (e.g., kleve.testtwo.HelloApplication). This class will have a main method.
- Right-click on the main application class file (e.g., HelloApplication.java).
- Select Run 'HelloApplication.main()'.
- Alternatively, if you already have a Run Configuration, select it from the top toolbar and click the green play button.

Running Unit/Integration Tests

- Locate Test Class: Navigate to src/test/java/kleve/testtwo/DAO/PartDAOTest.java.
- Right-click on the PartDAOTest.java file.
- Select Run 'PartDAOTest' to execute all tests in the class.
 - You can also click the green play icon next to individual test methods to run them separately.
- Review Results: The "Run" tool window at the bottom will display test results (green for pass, red for fail) and console output.

Troubleshooting

- "No value specified for parameter 1" or other SQLException
 - Verify your database is running.
 - Check JDBC.java for the correct dbURL, user, and password.
 - Ensure database schema (parts, products tables, etc.) is correctly loaded.
 - Review SQL queries in your DAO classes for correct syntax and parameter binding (PreparedStatement.setString(), setInt(), etc. must be called *before* executeQuery() or executeUpdate()).
- NullPointerException related to UserDao.userId:
 - As identified during development, the test environment bypasses the login process. Ensure your test setup (PartDAOTest.@BeforeAll method) explicitly sets a dummy user ID (e.g., UserDao.userId = 1;) if your DAOs depend on it.
- Maven Errors / Dependencies Not Found:
 - Click "Reload All Maven Projects" in IntelliJ IDEA.
 - Check your internet connection.
 - Ensure your pom.xml has correct dependency versions and no typos.
 - Try File -> Invalidate Caches / Restart... in IntelliJ IDEA.
- Application UI Not Launching:
 - Verify your mainClass in javafx-maven-plugin configuration in pom.xml is correct.
 - Check the console for JavaFX-specific errors.

Parts and Products Manager – User Manual

Version: 1.0 Date: June 11, 2025

Introduction

Parts and Products Manager is a desktop application designed to assist small business owners in managing their shop inventories. This guide will help you get started and use its core features.

System Requirements

To run this application, you need:

- Java Runtime Environment (JRE): Version 17 or higher installed on your computer
 - If you do not have it already installed or are unsure, you can download it from <https://www.java.com/en/download/manual.jsp>
- Windows 10/11
 - older versions may work but are untested and unsupported

Getting Started

1. Locate the application file: It will be provided to you as PartsAndProductsManager.jar.
2. Save the file: Save or copy this jar file to a convenient location on your computer.
3. To run the application, double-click the jar file, and the application should open.
 - If nothing happens, see the troubleshooting section below.

Login

- The login menu should appear on your screen now that the application is running. Enter the username and password provided to you by your employer or the database administrator, and click the login button.
- If successful, you will now be taken to the application's main menu.

- If login fails, please re-enter your username and password carefully, noting that they are case-sensitive.

Using the application

- The main menu will have sections for Parts and Products. Be sure to use the associated buttons, i.e., next to the tables, to perform operations respectively.
- On the left should be a table containing all the parts currently saved, while on the right should contain a table containing all the products currently saved.
- You can search either table by entering a name or an ID number into the associated search field and pressing enter.
- To add a new part or product, click the associated Add button, which will take you to a separate menu. Complete all the fields on that screen and click the Save button to add the Part or Product to the database. Clicking the cancel button will return you to the main menu without adding a new part or product.
- To modify an existing part or product, select the desired item from the table by clicking on it and then clicking the associated modify button. Perform all revisions using the fields in the menu on the following screen, then click save when complete. Clicking the cancel button will return you to the main menu without performing any revisions.
- To delete an existing part or product, select the desired item from the table by clicking on it and then clicking the associated delete button. Then, you must click the okay button on the confirmation popup to proceed with the deletion. If you do not wish to delete the part or product, click the cancel button to close the popup without deleting it.
- To generate a report, click the reports button, then click the go button next to the type of report you wish to generate. Clicking the back button on the report screen will return you

to the report menu, and clicking the exit button on the report menu will return you to the main menu.

- To close the application, click the exit button near the bottom of the main menu or the close button (x) in the top right corner.

Troubleshooting

- If nothing happens when attempting to run the application, this may be due to Java not being correctly associated with .jar files. You can work around this by opening your system's Command Prompt, navigating to the directory (folder) where you saved the jar file using the cd command (for example, cd C:\Users\YourUser\Documents\ PartsAndProductsManager.jar.) Then execute the following command: Java -jar PartsAndProductsManager.jar.
- An "Error: A JNI error has occurred..." or "Java not found" message indicates that Java is not installed or correctly configured on your system. Please download the latest Java Runtime Environment (JRE) from here: <https://www.java.com/en/download/manual.jsp>
- If you encounter "Access Denied" or Database Errors, contact your IT support or employer.

Support

If you encounter any issues not covered in this guide or require further assistance, please contact Jonathan Kleve by email at jkleve3@wgu.edu