University of Brighton

CI553: Object-Oriented Development and Testing

Modernising the Catshop

Jack Hughes

Table of Contents

Introduction

This project enhances the CatShop codebase, a catalog-based retail system, by introducing modern features and refinements that improve functionality, usability, and maintainability. The goal is to demonstrate how incremental improvements can evolve a legacy system into a more robust, user-centric solution while preserving its core architecture and separation of concerns.

The project focuses on four key improvements:

- **Bulk Purchase Discount**: Automatically applying discounts for qualifying orders to incentivize larger purchases and streamline order processing.
- **Stock Low Alert System**: Enabling proactive inventory management by notifying staff when stock levels fall below a predefined threshold.
- **Price Filtering**: Enhancing the customer experience by allowing users to filter products within a specified price range, offering greater control and convenience.
- **Dark Mode Toggle**: Modernizing the user interface to improve accessibility and user customization through light and dark theme options.

The design and implementation of these features were guided by comprehensive **behavioural**, **structural**, and **integration testing**, ensuring reliability and seamless integration into the existing system. While these tests were integral to development, the final validation and verification of the features were conducted manually. This involved observing and interacting with the Model-View-Controller (MVC) components to confirm that the new functionalities behaved as intended in real-world scenarios.

By addressing usability and operational challenges, this project demonstrates how targeted improvements can enhance the user experience while maintaining system maintainability. Through rigorous testing and iterative development, the project showcases a balanced approach to modernizing legacy systems, achieving innovation without compromising existing functionality.

Design

The design phase of this project aimed to integrate new features into the legacy CatShop codebase while preserving its core Model-View-Controller (MVC) architecture and ensuring adherence to software design principles such as separation of concerns (SOC), modularity, and maintainability. The development process was driven by a systematic testing approach that informed feature requirements and validated implementations.

---

1. Overarching Design Goals

The following principles guided the design of all improvements:

- **Preservation of Legacy Architecture**:
    - The CatShop system follows the MVC architecture:
        - **Model**: Handles core business logic and data manipulation (e.g., Basket, CustomerModel).
        - **View**: Displays information to the user (e.g., CustomerView, BackDoorView).
        - **Controller**: Manages user input and communication between models and views.
    - The new features were designed to respect this architecture by ensuring that logic remained confined to models, user-facing changes were implemented in views, and controllers acted as mediators.
- **Modularity**:
    - New features were implemented as independent methods or modules, minimizing dependencies and avoiding tight coupling with existing components.
    - Example: Basket.applyDiscount was introduced to handle discount calculations without affecting other basket operations.
- **Scalability and Maintainability**:
    - Features like filterProductsByPriceRange were designed with extensibility in mind, allowing for future enhancements such as advanced filtering options.
- **Testing-Driven Design**:
    - Behavioural, structural, and integration tests influenced the design by defining clear goals, ensuring individual components were reliable, and validating the seamless interaction of new and existing features.

---

2. Feature-Specific Design

Each feature was designed to address specific user and system requirements while adhering to the overarching principles.

2.1 Bulk Purchase Discount

- **Objective**:
  - Incentivize customers to make larger purchases by applying a 10% discount for baskets with 5 or more items.
- **Design Considerations**:
  - A new method, Basket.applyDiscount, was created to encapsulate discount logic.
  - The CashierModel.doBuy method was enhanced to call applyDiscount during the transaction process.
- **Testing Influence**:
  - Behavioural tests helped identify edge cases, such as handling mixed product types or ensuring no discount for fewer than 5 items.
  - Structural tests validated applyDiscount to ensure it worked independently and correctly calculated the discount.
- **Challenges and Solutions**:
  - **Challenge**: Prevent interference with existing basket operations.
  - **Solution**: Isolated discount logic within applyDiscount, maintaining SOC and preventing unexpected interactions.

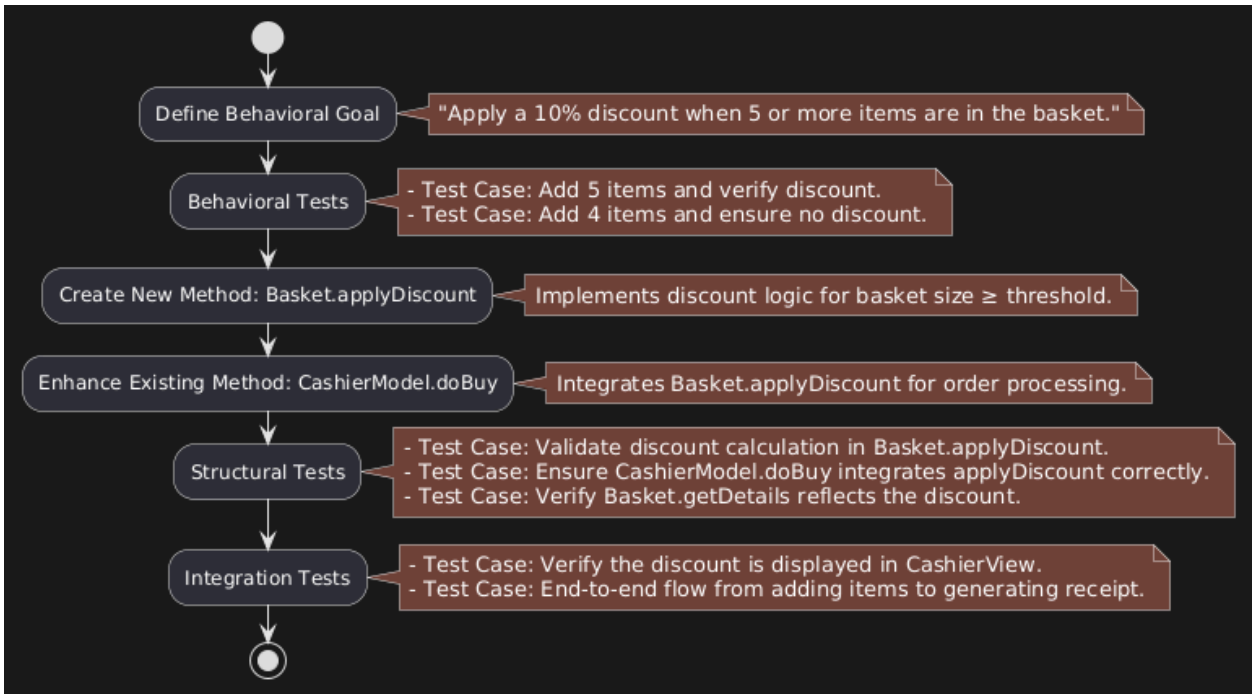

Figure 1 - Bulk Purchase Discount Workflow Diagram UML

2.2 Stock Low Alert System

- **Objective**:
  - Enable staff to manage inventory more effectively by notifying them when stock levels fall below a predefined threshold.
- **Design Considerations**:

- A new method, BackDoorModel.checkLowStock, was introduced to monitor stock levels and trigger alerts.
- Existing methods like doCheck and doRStock were enhanced to call checkLowStock when stock levels changed.
- Alerts were directed to BackDoorView for real-time display.

```
public void update(Observable modelC, Object arg) {
    BackDoorModel model = (BackDoorModel) modelC;
    String message = (String) arg;

    if (message.startsWith("LOW STOCK ALERT")) {
        // Highlight low stock alerts in theOutput
        theOutput.setText("\nALERT: " + message + "\n" + theOutput.getText());
    } else {
        // Display general messages
        theAction.setText(message);
    }

    theOutput.append(model.getBasket().getDetails() + "\n");
    theInput.requestFocus();
}
```

Figure 2 - BackDoorView.java - Low Stock Alert and General Message Handling

- **Testing Influence**:
  - Behavioural tests ensured alerts displayed correctly when stock levels dropped below the threshold.
  - Integration tests verified that stock updates in the model triggered notifications in the view.
- **Challenges and Solutions**:
  - **Challenge**: Avoid redundant alerts during frequent stock updates.
  - **Solution**: Used observer pattern refinements to ensure notifications were sent only once per update cycle.

Figure 3 - Low Stock Alert Workflow Diagram UML

2.3 Price Filtering

- **Objective**:
  - Improve the customer experience by enabling them to filter products within a user-specified price range, offering greater control and convenience.
- **Design Considerations**:
  - A new method, CustomerModel.filterProductsByPriceRange, was introduced to handle price filtering logic.
  - A complementary method, StockReader.getProductsByPriceRange, was created to fetch the relevant data from the backend.

- Filtered results were displayed in CustomerView, with clear error handling for invalid inputs (e.g., negative values or minPrice > maxPrice).

```
public Basket getBasket() {
    return theBasket;
}

/**
 * Filter products by price range.
 * @param minPrice The minimum price (inclusive).
 * @param maxPrice The maximum price (inclusive).
 */
public void filterProductsByPriceRange(double minPrice, double
    try {
        if (minPrice < 0 || maxPrice < 0) {
            setChanged();
            notifyObservers("Prices cannot be negative. Please
            return;
        }

        if (minPrice > maxPrice) {
```

Figure 4 - CashierModel.java - Bulk Purchase Discount Application Method

- **Testing Influence**:
  - Behavioural tests validated user-facing scenarios, such as ensuring only products within the range were displayed or showing an error for invalid inputs.
  - Structural tests focused on verifying the correctness of input validation and filtering logic in filterProductsByPriceRange.
  - Integration tests ensured that the filtering workflow—from input handling in CustomerView to data retrieval in StockReader—operated seamlessly.
- **Challenges and Solutions**:
  - **Challenge**: Handle invalid inputs gracefully without disrupting the user experience.
  - **Solution**: Implemented robust validation checks and ensured error messages were displayed clearly in the UI.

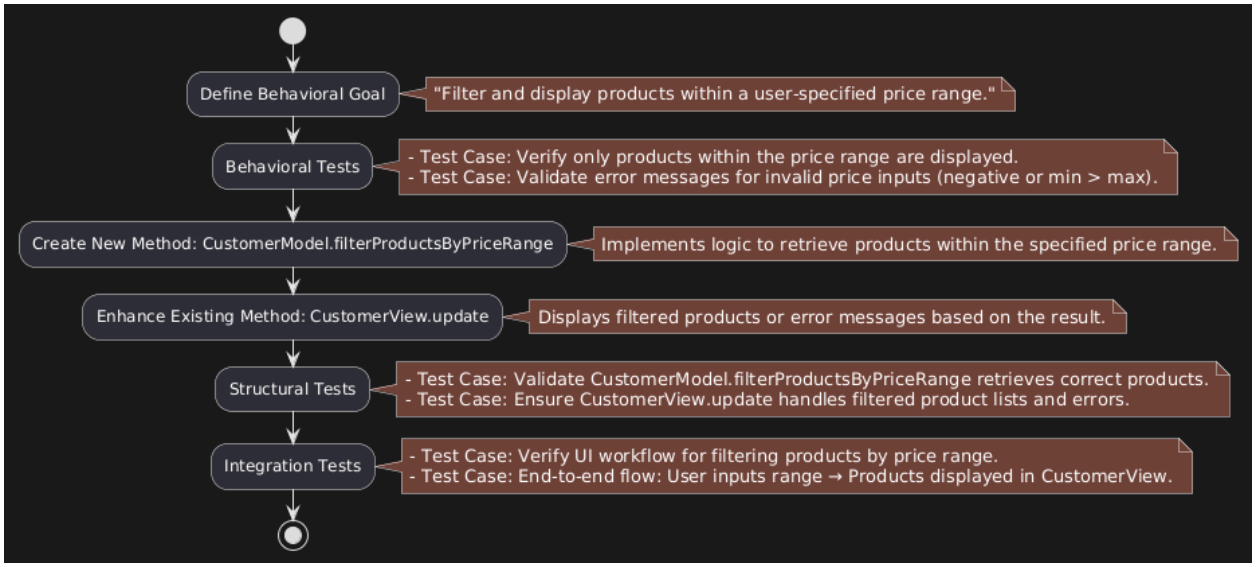

Figure 5 - Price Filtering Workflow Diagram UML

2.4 Dark Mode Toggle

- **Objective**:
  - Modernize the user interface by allowing users to toggle between light and dark themes for enhanced accessibility and customization.
- **Design Considerations**:
  - A new method, CustomerView.toggleTheme, was introduced to manage the state transition between light and dark themes.
  - Supporting methods:
    - updateTheme: Recursively updates the background and foreground colors for all UI components.
    - updateButtonTheme: Handles styling changes specific to buttons.
  - Theme state was maintained entirely within CustomerView to preserve SOC.
- **Testing Influence**:
  - Behavioural tests validated the visual consistency of components when toggling between themes.
  - Structural tests ensured the recursion logic in updateTheme worked correctly for deeply nested components.
  - Integration tests confirmed that all UI elements updated dynamically and that theme toggling did not disrupt other features like filtering or alerts.
- **Challenges and Solutions**:
  - **Challenge**: Ensure seamless updates across all UI components, including nested elements.
  - **Solution**: Designed updateTheme to iterate recursively through all child components and apply the theme dynamically.

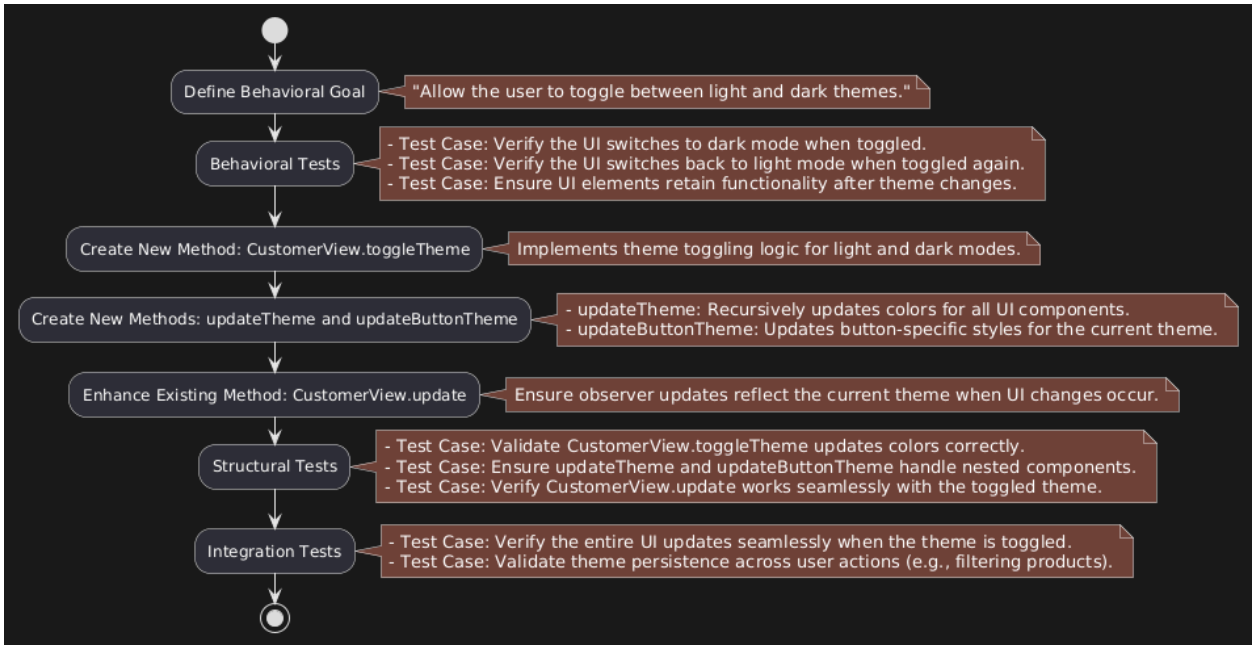

Figure 6 - Dark Mode UI Workflow Diagram UML

---

3. Testing-Driven Design and Validation

Testing was integral to this project, serving as both a guide for the iterative design process and a means of validating and verifying feature completion. By systematically applying behavioural, structural, and integration tests, the development process ensured that new features aligned with project goals, operated reliably, and interacted cohesively within the legacy system.

Iterative Test-Driven Design

The design process evolved iteratively, with each feature undergoing multiple stages of refinement informed by testing:

1. **Behavioural Tests**:
    - Defined high-level goals for each feature, focusing on what the system needed to achieve from a user's perspective.
    - Examples:
        - Ensuring a 10% discount applied correctly for orders with 5 or more items.
        - Verifying that stock alerts triggered when levels dropped below the threshold.
    - These tests shaped the design by clarifying desired outcomes and informing the functionality required.
2. **Structural Tests**:
    - Validated the functionality of individual methods and components in isolation to ensure logical correctness and reliability.
    - Examples:
        - Testing Basket.applyDiscount to calculate discounts accurately across edge cases (e.g., mixed product types or exactly 5 items).
        - Testing CustomerModel.filterProductsByPriceRange to handle invalid inputs gracefully (e.g., negative prices or minPrice > maxPrice).
3. **Integration Tests**:
    - Ensured new features interacted seamlessly with existing components across the Model-View-Controller (MVC) architecture.
    - Examples:
        - Verifying that stock updates in BackDoorModel triggered alerts displayed in BackDoorView.
        - Ensuring that toggling Dark Mode dynamically updated all UI components consistently.

---

4. Design Challenges

Throughout the design process, several challenges arose:

1. **Legacy Constraints**
   Incorporating new features into the legacy system without disrupting existing functionality required careful identification of dependencies and potential points of conflict.
2. **Maintaining Separation of Concerns**
   Balancing new functionality with the MVC architecture demanded careful design to ensure that responsibilities remained clearly delineated across models, views, and controllers.
3. **Supporting Scalability for Future Features**
   While implementing current improvements, the design had to remain flexible enough to support additional features in the future (e.g., dynamic discounts or enhanced product search).

These challenges were addressed through a combination of modular design, rigorous testing, and iterative refinement. Establishing clear distinctions between behavioral, structural, and integration tests was essential for defining completion criteria and providing direction before engaging with the legacy codebase. In this context, **tests** refer to the methods and approaches used to validate the system at various levels of abstraction:

- **Behavioral Tests**:
  Defined high-level goals for each feature, focusing on what we wanted the system to achieve from a user perspective. Example: Applying a 10% discount for qualifying orders or displaying low-stock alerts. These tests helped establish the overarching objectives for each improvement.
- **Structural Tests**:
  Validated the correctness and reliability of individual methods or components in isolation.
  Example: Ensuring Basket.applyDiscount correctly calculated discounts or CustomerModel.filterProductsByPriceRange handled invalid inputs gracefully.
- **Integration Tests**:
  Ensured that new features interacted seamlessly with existing components, verifying that changes to models, views, and controllers maintained the integrity of the system's architecture.

This structured approach to testing and validation provided a robust foundation for navigating the complexities of the legacy system while ensuring that new features were reliable, cohesive, and aligned with the project's goals.

Implementation and Workflow

This section focuses on the practical aspects of building new features, the tools used, and the iterative workflow that guided development. By integrating BlueJ and VSCode, each tool served a distinct purpose in implementation and manual validation. The testing-driven workflow provided structure and ensured the quality of the final system.

---

1. Implementation Details

The implementation process translated the design goals into functional features, ensuring modularity and alignment with the system's Model-View-Controller (MVC) architecture.

Practical Development

- Features were implemented incrementally, broken into manageable tasks:
    - Example: For the **Bulk Purchase Discount**, development began with the creation of the applyDiscountmethod in Basket. This method was isolated to calculate and apply discounts without disrupting other basket operations. Once verified, the CashierModel.doBuy method was updated to call applyDiscount as part of the transaction process.
    - Similarly, for **Price Filtering**, the CustomerModel.filterProductsByPriceRange method was written to handle filtering logic, while the StockReader.getProductsByPriceRange method handled backend queries.

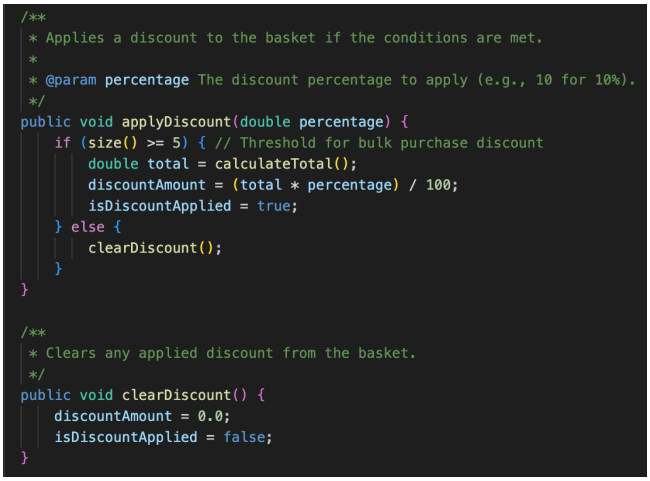

Figure 7 - Basket.java - Discount Application and Clearing Logic

Figure 8 - CashierModel.java - Method to Apply Bulk Discount to Basket

Key Tools for Implementation

- **VSCode**:

    - Preferred for writing and refining code due to its advanced editing capabilities, clean interface, and support for formatting.
    - Example: The filterProductsByPriceRange method was implemented in VSCode, where its formatting tools ensured readability and maintainability.
- **BlueJ**:
    - Used primarily for interacting with the MVC structure and validating UI behaviour.
    - Example: Testing the **Dark Mode Toggle** involved toggling the theme in CustomerView and observing updates across all nested UI components in BlueJ's visual environment.

Manual Code Validation

- Code validation was performed manually by inspecting key methods and running behavioural tests. This approach ensured that each feature was functionally correct before moving to the next development stage.
- Example: For the **Stock Low Alert**, the checkLowStock method in BackDoorModel was validated to confirm it triggered alerts correctly when stock thresholds were breached.

```
public void doCheck(String productNum) {
    pn = productNum.trim();                    // Product no.
    checkLowStock(pn);                         // Check for low stock
}
```

Figure 9 - BackDoorModel.java - Low Stock Check Method

---

2. Workflow and Task Management

The workflow was structured around an iterative process that relied on behavioural, structural, and integration tests to define tasks and ensure consistent progress. This simple yet effective framework provided natural checkpoints, reducing the need for external task management tools.

Iterative Workflow

Each feature followed a cyclical process that emphasized testing and refinement at every stage:

1. **Behavioural Tests**:
    - Established high-level goals for each feature.
    - Example: For the **Bulk Purchase Discount**, the behavioural goal was to ensure that adding 5 or more items to the basket automatically applied a 10% discount.
2. **Structural Tests**:
    - Validated individual methods and components in isolation, ensuring correctness and reliability.
    - Example: The toggleTheme method in CustomerView was tested to confirm it updated nested UI components correctly without impacting unrelated features.
3. **Integration Tests**:
    - Verified seamless interaction across the MVC architecture.
    - Example: For **Price Filtering**, integration tests ensured that CustomerModel.filterProductsByPriceRangeworked cohesively with StockReader and updated the CustomerView with filtered results.

By completing one type of test, the development process naturally advanced to the next stage, simplifying task management and ensuring iterative improvements.

Task Management

Task management was straightforward due to the simplicity and clarity of the design framework:

- **Testing as a Roadmap**:
    - Each test type acted as a clear checkpoint, guiding the transition from one task to the next.
    - The cyclical workflow (behavioural → structural → integration) created a predictable structure for tackling each feature.
- **Notes for Progress Tracking**:
    - Personal notes were used to organize tasks and track progress.
    - Example: Once behavioural tests for a feature like the **Dark Mode Toggle** were complete, the next steps—structural and integration tests—were already defined, removing ambiguity in task management.

PlantUML and Visual Planning

PlantUML was used to create sequence and activity diagrams that visualized feature workflows and interactions:

- **Role in Implementation**:
    - PlantUML diagrams provided a clear representation of how methods and components interacted within the MVC framework.
    - Example: For the **Bulk Purchase Discount**, a sequence diagram illustrated the flow from adding items to applying the discount and updating the receipt, helping to refine the Basket and CashierModel implementations.
- **Support for Task Management**:
    - These diagrams acted as visual guides, breaking complex tasks into smaller, actionable steps.

The combination of a structured workflow, practical testing framework, and tool-specific roles streamlined the implementation process. By leveraging BlueJ for UI testing and MVC interaction, VSCode for advanced coding, and PlantUML for diagramming, the project successfully balanced simplicity with effectiveness. This approach ensured that each feature was built incrementally, tested thoroughly, and seamlessly integrated into the legacy system.

Challenges and Limitations

1. **Version Control**
    - I chose not to use Git for version control, as it was a solo effort. Instead, tests and IDE-based methods (e.g., saving snapshots and manual reviews) were sufficient for tracking changes and ensuring progress.
    - **Reflection**: While this approach worked effectively for a single developer, incorporating Git in future projects could facilitate better collaboration, branching strategies, and detailed change tracking, especially for multi-developer teams.
2. **Manual Testing Over Automation**
    - The project relied entirely on manual testing to validate behavioural, structural, and integration aspects of the system. While this ensured rigorous, hands-on validation of features, automated testing could have provided additional efficiency and coverage.
    - **Reflection**: The focus on manual testing was a deliberate choice to align with the project's scope and resources. However, adding automated tests in future projects could reduce repetitive tasks and help catch edge cases more efficiently.
3. **Simplified Task Management**
    - Task management was streamlined using notes and a structured testing framework. This approach worked well given the project's simplicity, but it might become less effective in larger or more complex projects.
    - **Reflection**: The deliberate avoidance of formal tools like Trello or Jira was justified by the straightforward workflow. In future projects with larger teams or more complex requirements, adopting such tools could enhance tracking and collaboration.
4. **Balancing Testing Scope with Real-World Usability**
    - While the project's testing framework (behavioural → structural → integration) provided robust coverage, certain real-world scenarios or edge cases may not have been captured due to the reliance on manual validation.
    - **Reflection**: For the project I prioritized a rigorous manual testing process that aligned with its scope and goals. Expanding testing to include performance tests (e.g., ensuring the system remains responsive under high user load or with large datasets) or automated usability tests (e.g., validating UI responsiveness and dynamic

updates across various inputs) could enhance future projects by identifying edge cases and improving scalability.

5. **Feature Interdependencies**
    - Certain features, such as the Bulk Purchase Discount and the Stock Low Alert System, introduced interdependencies between existing components and newly added methods. For example, ensuring that Basket.applyDiscount did not interfere with other basket operations or that checkLowStock integrated smoothly with existing stock operations like doRStock.
    - **Reflection**: These challenges were addressed by maintaining clear boundaries between new and existing logic. Modular design principles ensured that new methods operated independently where possible, while integration tests verified that interactions between components remained seamless. This approach minimized disruptions to existing functionality while introducing new capabilities.

Future Enhancements

Although this project successfully delivered its objectives, several enhancements could further improve the development process, testing efficiency, and user experience in future projects. These recommendations focus on scaling the system and workflows for more complex or collaborative environments.

1. Automated Testing Integration

- **Opportunity**: While manual testing provided thorough validation, automated testing could enhance efficiency and expand test coverage.
- **Suggested Tools**:
    - **JUnit**: For validating structural and integration logic, ensuring key methods (e.g., applyDiscount, filterProductsByPriceRange) behave as expected under all scenarios.
    - **Selenium** or similar UI testing tools: To automate UI validation, reducing the time required for repetitive checks.
- **Impact**: Automated testing would complement manual validation, reduce the risk of human error, and free up resources for more complex development tasks.

2. Modular Feature Extensions

- **Opportunity:** The system's modular design offers opportunities for feature expansion without major structural changes.
- **Potential Extensions:**
    - **Dynamic Discount Logic:** Add support for seasonal discounts or customer loyalty rewards, extending the applyDiscount logic.
    - **Stock Monitoring Dashboard:** Provide staff with a visual overview of stock trends and low-stock alerts to enhance inventory management.
- **Impact:** These additions would leverage the existing architecture while increasing system functionality and user engagement.

3. Advanced UI and User Experience Features

- **Opportunity**: Building on the current UI improvements, future iterations could enhance customization and usability.
- **Potential Features**:
    - **Personalized Themes**: Allow users to select or create custom themes beyond light and dark modes.
    - **Advanced Filtering Options**: Expand price filtering to include additional criteria, such as product categories or stock availability.
    - **Responsive Design**: Optimize the UI for various screen sizes and devices, enhancing accessibility.
- **Impact**: These enhancements would modernize the user experience and make the system more versatile.

4. Improved Data Handling and Reporting

- **Opportunity:** Enhancing backend data processing and reporting features could provide users and staff with deeper insights into system performance and sales trends.
- **Potential Features:**
    - **Sales Reports:** Generate automated reports summarizing sales trends, popular products, and revenue growth.

    - **Inventory Analysis:** Predict restocking needs based on historical stock movement data.
- **Impact:** These tools would increase the system's value for decision-making and operational efficiency.

5. Expanded Testing Scope

- **Opportunity**: Future projects could incorporate stress tests and broader usability testing to ensure the system performs well under heavy load or unconventional scenarios.
- **Suggested Enhancements**:
    - **Stress Testing**: Simulate high user activity to test system performance and reliability.
    - **Edge Case Validation**: Explore rare or extreme inputs, ensuring robust error handling and stability.
- **Impact**: Expanding the testing scope would improve system robustness and ensure readiness for real-world use cases.

These enhancements are not intended to address gaps in the current project but rather to scale its processes and features for future, more complex requirements. By adopting automated testing, Git-based version control, and advanced UI features, future iterations can achieve greater efficiency, usability, and scalability.

Conclusion

This project successfully modernised the CatShop codebase by introducing carefully designed features that enhanced functionality, usability, and maintainability while preserving the legacy system's stability and core architecture. The implementation of the Bulk Purchase Discount, Stock Low Alert, Price Filtering, and Dark Mode features demonstrated a structured and user-focused approach to software evolution.

The development process was guided by a testing-driven design framework, using behavioural, structural, and integration tests to ensure feature reliability and seamless integration into the Model-View-Controller (MVC) architecture. Each stage of the workflow—defining high-level goals, validating individual methods, and confirming system-wide interactions—acted as a checkpoint for iterative refinement. While manual validation replaced automated testing, the rigorous approach ensured that features were robust and met real-world usability expectations.

Challenges, such as relying on manual testing and task management without formal tools, were effectively addressed through a simple and efficient workflow. The project's modular structure and testing-driven approach made traditional version control tools like Git unnecessary for a solo developer, demonstrating the adaptability of the methods used. The ability to navigate these challenges reflects the thoughtful planning and execution underpinning the project.

Key takeaways include the critical role of modularity and separation of concerns in maintaining a stable system, the value of iterative testing for guiding development, and the importance of user-focused design in delivering meaningful improvements. The project highlights how deliberate design choices and structured workflows can modernize legacy systems effectively, transforming them into scalable, maintainable, and user-friendly solutions.

Looking ahead, the scalable and modular design of the system opens opportunities for future enhancements, such as integrating automated testing frameworks, introducing advanced UI customization, and expanding data-driven reporting capabilities. These improvements could further elevate the system's functionality and efficiency, ensuring it remains adaptable to evolving needs.

In conclusion, this project demonstrates how a legacy system can be successfully modernized through a balanced approach that combines innovation with respect for existing functionality. By adhering to best practices in design, development, and testing, the project delivered a robust and maintainable solution that meets modern standards while laying a strong foundation for future growth.

GitHub Repository and Supporting Files

All project-related files, including the full report, screenshots, and UML diagrams, are available in the dedicated GitHub repository. The repository ensures version control and accessibility for all supporting materials used during this project. A **screenshots folder** is included, containing all relevant code snippets, UI captures, and UML diagrams that validate and illustrate the implementation process. The repository can be accessed at the following link:

GitHub Repository: https://github.com/JackHughesBrightonUni/CI553-Improved-Codebase

The screenshots folder contains the following files:

1. **Code Screenshots**:
    - `BackDoorModel.java-Screenshot.png`
    - `BackDoorModel.java-Snippet.png`
    - `BackDoorView.java-Screenshot.png`
    - `BackDoorView.java-Snippet.png`
    - `Basket.java-Screenshot.png`
    - `Basket.java-Snippet.png`
    - `CashierModel.java-Screenshot.png`
    - `CashierModel.java-Snippet.png`
    - `CustomerModel.java-Screenshot.png`
    - `CustomerModel.java-Snippet.png`
    - `CustomerView.java-Screenshot.png`
    - `R_StockRW.java-Screenshot.png`
    - `RemoteStockRW_I.java-Screenshot.png`
    - `StockR.java-Screenshot.png`
    - `StockReader.java-Screenshot.png`
2. **UI and Validation Screenshots**:
    - `BulkBuyDiscount-Screenshot.png`
    - `DarkModeUI-Screenshot.png`
    - `FilterByPrice-Screenshot.png`
    - `LowStockAlert-Screenshot.png`
    - `ReceiptDiscount-Screenshot.png`
3. **UML Diagrams**:
    - `UML-BulkBuyDiscount.png`
    - `UML-DarkModeUI.png`
    - `UML-LowStockAlert.png`
    - `UML-PriceFiltering.png`
    - `UsingPlantUMLtoMakeUMLDiagramForDiscount.png`

This folder provides detailed evidence of the development and implementation process, supporting the explanations and analysis provided in this report. These resources can be referenced to review the implementation of features such as the Bulk Purchase Discount, Stock Low Alert, Price Filtering, and Dark Mode.

ESTIMATED GRADE

- **Development**: A
    - The project demonstrates strong technical implementation with modular design, adherence to SOC, and seamless integration of features into the legacy system. Rigorous testing and manual validation ensured feature reliability and alignment with project goals.
- **Management**: A
    - The simple yet effective workflow, structured around behavioural, structural, and integration tests, provided clear checkpoints and guided task progression. The deliberate choice of tools and efficient task tracking with notes reflect strong project management tailored to a solo developer's needs.
- **Report**: A
    - The report is comprehensive, well-structured, and effectively communicates the project's design, implementation, challenges, and outcomes. It highlights the thoughtful decisions made throughout the process and provides detailed evidence to support the project's success.

References

PlantUML. (n.d.). *Documentation*. Available at: https://plantuml.com.

BlueJ. (n.d.). *BlueJ Documentation*. Available at: https://www.bluej.org/doc/documentation.html.

Visual Studio Code. (n.d.). *Code Editing. Redefined.* Available at: https://code.visualstudio.com.

Continuous Delivery. (n.d.). *Continuous Delivery YouTube Channel*. Available at: https://www.youtube.com/@ContinuousDelivery.

Farley, D. (2023). *Why Test Driven Development Goes Wrong*. [video] Available at: https://www.youtube.com/watch?v=UWtEVKVPBQ0&list=LL&index=80.

Farley, D. (2023). *The 3 Types of Unit Test in TDD*. [video] Available at: https://www.youtube.com/watch?v=W40mpZP9xQQ&list=LL&index=71.

Farley, D. (2023). *Don't Do E2E Testing!*. [video] Available at: https://www.youtube.com/watch?v=QFCHSEHgqFE&list=LL&index=60.

Farley, D. (2023). *5 Books That Can Change A Developer's Career*. [video] Available at: https://www.youtube.com/watch?v=RfOYWeu5pGk&list=LL&index=58.

Farley, D. (2023). *Coupling Is The Biggest Challenge In Software Engineering*. [video] Available at: https://www.youtube.com/watch?v=plMttQWztRM&list=LL&index=52.

Farley, D. (2023). *TDD Isn't Hard, It's Something Else*.... [video] Available at: https://www.youtube.com/watch?v=WDFN_u5FTyM&list=LL&index=50.

Farley, D. (2023). *Kent Beck On The FIRST Testing Frameworks, TDD, Waterfall & MORE | The Engineering Room Ep. 16*. [video] Available at: https://www.youtube.com/watch?v=guycIP56YeY&list=LL&index=47.

Farley, D. (2023). *Test Driven DESIGN - Step by Step*. [video] Available at: https://www.youtube.com/watch?v=-f_HgWbomCI&list=LL&index=46.

Farley, D. (2023). *Acceptance Testing with Executable Specifications*. [video] Available at: https://www.youtube.com/watch?v=knB4jBafR_M&list=LL&index=37.

Farley, D. (2023). *How To Manage Software Complexity | Martin Thompson In The Engineering Room Ep. 4*. [video] Available at: https://www.youtube.com/watch?v=sIBrFuzR3cs.

Wikimedia Commons. (n.d.). *Topgrade Logo*. Available at: https://commons.wikimedia.org/wiki/File:Topgrade_Logo.png