Abdulhaq Zulfiqar

22i-2585

SE-G

SQE A3

Contents

1. Java Code Review Checklist for EmployeeManagement 6

Refactoring Repeated File Handling Code 12

Refactoring Username Generation in `add` Method 12

Refactoring Hardcoded Values and Logging 12

Appendix: 13

2. Java Code Review Checklist for Inventory 16

Refactoring Tasks 22

Refactoring Task 1: Separate File Reading and Writing Logic 22

Refactoring Task 2: Cache Item Lookup to Improve Performance 22

Refactoring Task 3: Replace `System.out.println` with Logging 22

Appendix: 22

3. Java Code Review Checklist for Item 26

Refactoring Task 1: Improve Access Modifiers for Getter Methods 30

Refactoring Task 2: Add Validation for Negative Amounts in `updateAmount` 30

Refactoring Task 3: Add Comments and Documentation for Methods and Attributes 31

4. Java Code Review Checklist for Management 33

Refactoring Task 1: Separate File Reading and Writing Logic 38

Refactoring Task 2: Replace `System.out.println` with Proper Logging 38

Refactoring Task 3: Use a `Map` for Efficient User Lookup 39

Code Smells 39

Violations of Coding Standards 39

Performance Inefficiencies 40

5. Java Code Review Checklist for POH 40

Refactoring Task 1: Consolidate File Handling Logic 47

Refactoring Task 2: Simplify `endPOS` Method and Use Helper Functions 47

Refactoring Task 3: Implement Logging and Replace `System.out.println` Statements 47

6. Java Code Review Checklist for Point Of Sale 49

Refactoring Task 1: Consolidate File Handling Logic 55

Refactoring Task 2: Simplify endPOS Method and Use Helper Functions 55

Refactoring Task 3: Implement Logging and Replace System.out.println Statements 56
7. Java Code Review Checklist for POR 57
Refactoring Task 1: Extract Inventory Update Logic into a Separate Method 64
Refactoring Task 2: Modularize the `processReturn` Method 64
Refactoring Task 3: Replace Magic Numbers with Constants 65
1. Code Smells 65
2. Violations of Coding Standards 66
3. Performance Inefficiencies 66
8. Java Code Review Checklist for POSSystem 67
Refactoring Task 1: Extract Payment Processing Logic into a Separate Method 73
Refactoring Task 2: Modularize the `handleOrder` Method 74
Refactoring Task 3: Replace Hardcoded Discount Logic with Configurable Constants 74
9. Java Code Review Checklist for Return Item 77
Refactoring Task 1: Extract Payment Calculation Logic into a Separate Method 83
Refactoring Task 2: Separate Input Validation from Core Logic 83
Refactoring Task 3: Consolidate File Handling Logic into a Single Method 83
Code Smells 84
Performance Inefficiencies 84
Violations of Coding Standards 84
10. Java Code Review Checklist for Regitser 85
Refactoring Task 1: Extract Payment Calculation Logic into a Separate Method 91
Refactoring Task 2: Move Input Validation to a Separate Method 91
Refactoring Task 3: Consolidate File Handling Logic 92
Code Smells 92
Performance Inefficiencies 93
Violations of Coding Standards 93
11. Java Code Review Checklist for Add_Employee interface 94
Refactoring Task 1: Replace Magic Numbers with Constants 98
Refactoring Task 2: Consolidate Redundant Method Calls 98
Refactoring Task 3: Simplify Conditional Expressions 98

12. Java Code Review Checklist for Admin interface 99
Refactoring Task 1: Extract Duplicate Code into a Helper Method 103
Refactoring Task 2: Use StringBuilder for String Concatenation 103
Refactoring Task 3: Break Down Large Methods into Smaller Methods 103
Appendix: 103
13. Java Code Review Checklist for Cashier interface 106
Refactoring Task 1: Extract Duplicate Code into a Helper Method 109
Refactoring Task 2: Replace Magic Numbers with Constants 110
Refactoring Task 3: Break Down Large Methods into Smaller Methods 110
Appendix: 110
14. Java Code Review Checklist for Enter_Item interface 112
Refactoring Task 1: Extract Repeated Code into a Helper Method 117
Refactoring Task 2: Replace Magic Numbers with Constants 117
Refactoring Task 3: Break Down Large Methods into Smaller Methods 117
Appendix: 118
15. Java Code Review Checklist for Login interface 120
Refactoring Task 1: Extract Repeated Code into a Helper Method 124
Refactoring Task 2: Replace Magic Numbers with Constants 124
Refactoring Task 3: Break Down Large Method into Smaller Methods 125
Appendix: 125
16. Java Code Review Checklist for payment_interface 128
Refactoring Task 1: Extract Repeated Code into a Helper Method 132
Refactoring Task 2: Consolidate UI Button Action Handling 133
Refactoring Task 3: Replace Magic Numbers with Named Constants 133
Appendix: 133
17. Java Code Review Checklist for payment_interface 136
Refactoring Task 1: Simplify `actionPerformed()` Method by Extracting Logic 141
Refactoring Task 2: Consolidate Customer Phone Number Validation 141
Refactoring Task 3: Replace Hardcoded Database File Paths with Constants 142
Appendix: 142
18. Java Code Review Checklist for Update_Employee interface 144

Refactoring Task 1: Extract Repeated Code for Interface Disposal into a Helper Method 149
Refactoring Task 2: Replace Magic Numbers with Constants 149
Refactoring Task 3: Break Down Large `actionPerformed()` Method into Smaller Methods 149
Appendix: 150
19. Java Code Review Checklist for Employee 152
Refactoring Task 1: Add Input Validation for Employee Attributes 155
Refactoring Task 2: Consolidate Setter Methods 155
Refactoring Task 3: Extract Password Handling Logic 156
Appendix: 156

1. Java Code Review Checklist for EmployeeManagement

File name	EmployeeManagement.java
class/interface name	EmployeeManagement

Category	Checklist Item	Yes/No	Issue	Fix
Naming Conventions	Are class names written in PascalCase?	Yes	None	None
	Are variable and method names written in camelCase?	Yes	unixOS and temp could be renamed for clarity.	Rename unixOS to isUnixOS and temp to tempDatabaseFile.
	Are constants written in uppercase with underscores?	No	employeeDatabase and temp could be constants but aren't uppercase.	Define employeeDatabase and temp as static final and rename to uppercase with underscores (e.g., EMPLOYEE_DATABASE).
Code Structure	Are access modifiers used correctly?	No	Some fields (e.g., employees, employeeDatabase) should be private.	Make these fields private and add appropriate getters/setters if needed.
	Are classes and interfaces separated?	Yes	None	None
	Are packages used appropriately?	No	Package not specified in code, reducing project organization.	Add package declaration (e.g., com.pos.employee) and organize

				project files accordingly.
Method Design	Do methods have a single responsibility?	Yes	None	None
	Are method parameters limited?	Yes	None	None
	Is method overloading used properly?	Yes	None	None
Exception Handling	Are exceptions handled with try-catch blocks?	Yes	None	None
	Are specific exceptions used?	Yes	None	None
	Is logging implemented in catch blocks?	No	No logging used in catch blocks for debugging or error tracking.	Add logging (e.g., Logger) to replace System.out.println in catch blocks.
Code Readability	Are comments added for complex logic?	No	No comments explaining complex logic in update and delete methods.	Add descriptive comments explaining logic, especially in areas like file handling and employee data manipulation.
	Is indentation consistent?	Yes	None	None
	Are blank lines used to separate code blocks?	Yes	None	None

	Are meaningful names used for variables, classes, and methods?	Mostly	temp, toWrite, and find are not descriptive.	Rename temp to tempDatabaseFile, toWrite to employeeDataLine, and find to employeeFound.
Performance	Are data structures chosen based on performance?	Yes	None	None
	Are costly operations minimized in loops?	No	Username generation (getUsername() call) is inside add method but could be optimized.	Move username generation logic outside the loop to avoid repeated calls.
	Is lazy initialization used?	Yes	None	None
Memory Management	Are unnecessary object references set to null?	No	No clear memory management to release resources or handle large files efficiently.	Close all resources in try-with-resources blocks or add explicit closing to ensure objects are properly disposed of.
Security	Is user input validated?	No	No validation for user-provided name, password, position.	Implement validation to sanitize input, preventing injection attacks.
	Are sensitive data encrypted before storage?	No	Passwords are stored in plain text in employeeDatabase.txt.	Encrypt passwords before storing using a hashing algorithm like

				SHA-256 or bcrypt.
	Is PreparedStatement used for database queries?	Not Applicable	None	None
Maintainability	Are there long methods or deeply nested loops?	No	None	None
	Is there duplicated code?	Yes	Duplicate logic for reading/writing to employeeDatabase.	Extract the file reading/writing logic into separate helper methods to avoid code duplication.
	Are there any magic numbers?	Yes	name, position, password are stored based on hardcoded array indices.	Use constants to specify array indices (e.g., POSITION_INDEX).
Test Related Categories				
Test Coverage	Are unit tests provided for all public methods and critical functionalities?	Yes	None	None
	Do unit tests cover edge cases and boundary values?	Yes	Tests for edge cases such as non-existing employees, empty lists, read-only file scenarios are well-covered.	None
Test Design	Are tests written	Yes	None	None

	following the AAA pattern?			
	Are individual test cases independent?	Yes	None	None
	Are descriptive names used for test methods?	Yes	None	None
Assertions	Are assertions used to verify expected results?	Yes	None	None
	Are specific assertions used instead of general ones?	Yes	None	None
Boundary and Edge Cases	Are edge cases and boundary conditions tested?	Yes	None	None
	Are invalid inputs covered by tests?	Yes	None	None
Mocking and Stubbing	Are mocks or stubs used to isolate the unit under test?	Yes	None	None
Performance Testing	Are tests to check performance for critical methods provided?	No	No specific performance tests (e.g., handling large files).	Add performance tests to validate handling of large data sets, or prolonged add,

				delete, update operations.
Test Maintainability	Are test methods organized and modular?	Yes	None	None
	Is there a setup method for initializing common objects?	Yes	None	None
Housekeeping				
Code smells	Are there any code smells not covered by the checklist?	Yes	- Repeated file handling code for reading and writing employee data.	Extract file handling code into separate helper methods to reduce redundancy and improve modularity.
			- readFile method directly manipulates employees without isolation, potentially causing side effects.	Use a temporary list to store data in readFile and assign it to employees only after reading is complete.
Coding standards	Are there any coding standard violations not covered by the checklist?	Yes	- Inconsistent usage of hard-coded strings (e.g., "Admin", "Cashier").	Define these values as constants (e.g., ROLE_ADMIN, ROLE_CASHIER).
			- No consistent error handling strategy; exceptions are caught but not properly logged.	Replace System.out.println in catch blocks with a logging framework for better error

				tracking and consistency.
Performance Inefficiencies	Are there any performance inefficiencies not covered by the checklist?	Yes	- Repeated getUsername() call in add method, which is inefficient when accessed frequently.	Cache username once outside of the loop to avoid redundant method calls.
			- Direct string concatenation used in multiple areas (e.g., toWrite in add method).	Use StringBuilder for concatenations, especially in loops, to improve performance when building larger strings.

Refactoring Repeated File Handling Code

- **Issue**: Repeated code for reading and writing from `employeeDatabase` and `temp`.
- **Refactoring Plan**: Extract the file reading and writing operations into helper methods to avoid code duplication and enhance modularity.

Refactoring Username Generation in `add` Method

- **Issue**: `getUsername()` call inside the `add` method could be optimized, especially with large data.
- **Refactoring Plan**: Cache the username of the last employee entry to avoid redundant calls.

Refactoring Hardcoded Values and Logging

- **Issue**: Hardcoded values (e.g., `"Admin"`, `"Cashier"`) and lack of logging in catch blocks.
- **Refactoring Plan**: Define role constants and replace `System.out.println` with a logger to improve error tracking and standardization.

Appendix:

Code Smells

1. **Long Methods**
    - **Issue:** None identified in the provided checklist.
    - **General Solution:** For methods exceeding 20-25 lines, split into smaller, focused methods that adhere to the *Single Responsibility Principle (SRP)*.
2. **Nested Loops**
    - **Issue:** None identified.
    - **General Solution:** Avoid deeply nested logic by extracting loops into helper methods or restructuring logic for clarity.
3. **Duplicate Code**
    - **Issue:** Repeated file handling logic for reading and writing employee data.
    - **Fix:** Create helper methods like readFile and writeFile to encapsulate file operations.
4. **Data Clumps**
    - **Issue:** Frequent hard-coded roles ("Admin", "Cashier").
    - **Fix:** Use constants like ROLE_ADMIN and ROLE_CASHIER to centralize and reuse values.
5. **Primitive Obsession**
    - **Issue:** None directly observed.
    - **General Solution:** Replace overused primitive types with descriptive enums or classes (e.g., replace a String role with an enum Role).

Violations of Coding Standards

1. **Naming Conventions**
    - **Issue:**
        - Variables unixOS and temp are not descriptive.
        - Constants like employeeDatabase and temp are not in uppercase.

- **Fix:**
    - Rename variables (unixOS → isUnixOS, temp → tempDatabaseFile).
    - Define constants as static final with uppercase names (EMPLOYEE_DATABASE).

2. **Access Modifiers**
    - **Issue:** Fields like employees and employeeDatabase lack proper encapsulation.
    - **Fix:** Make fields private and expose via getters/setters if necessary.
3. **Comments**
    - **Issue:** Complex logic in update and delete lacks comments.
    - **Fix:** Add meaningful comments explaining the purpose and flow of the logic.
4. **Magic Numbers**
    - **Issue:** Hardcoded indices for name, position, and password.
    - **Fix:** Use named constants like NAME_INDEX or PASSWORD_INDEX.
5. **Inconsistent Formatting**
    - **Issue:** Not observed, but formatting should follow uniform standards (e.g., 4-space indentation).

Performance Inefficiencies

1. **Unnecessary Object Creation**
    - **Issue:** Repeated calls to getUsername() in the add method.
    - **Fix:** Cache the username in a local variable to avoid redundant calculations.
2. **Inefficient String Operations**
    - **Issue:** String concatenation (toWrite) is used in loops.
    - **Fix:** Use StringBuilder for efficient string manipulation.
3. **Redundant Calculations**
    - **Issue:** Not explicitly mentioned but suspected in repetitive operations within loops.

- **Fix:** Perform calculations outside loops and reuse results.

4. **Excessive Logging**
    - **Issue:** Lack of logging in catch blocks for debugging.
    - **Fix:** Replace System.out.println with a logging framework like Logger.

Security Concerns

1. **Sensitive Data Handling**
    - **Issue:** Passwords are stored in plain text.
    - **Fix:** Use secure hashing algorithms like SHA-256 or bcrypt to store passwords.
2. **Input Validation**
    - **Issue:** No validation for user-provided data (e.g., name, position, password).
    - **Fix:** Implement input sanitization to prevent injection attacks.

Refactoring Steps

1. Repeated File Handling

- **Issue:** File reading and writing logic is duplicated.
- **Fix:**
    - Create helper methods:

```
private List<String> readFile(String filePath) { /* Logic */ }

private void writeFile(String filePath, List<String> data) { /* Logic */ }
```

    - Replace all occurrences with these methods.

2. Optimize Username Generation

- **Issue:** getUsername() is called multiple times unnecessarily.
- **Fix:** Cache the result outside the loop:

```
String username = getUsername(employee);
```

3. Replace Hardcoded Values

- **Issue:** Hardcoded roles and strings.

- **Fix:** Use constants:

public static final String ROLE_ADMIN = "Admin";

public static final String ROLE_CASHIER = "Cashier";

4. Improve Logging

- **Issue:** No structured logging.
- **Fix:** Integrate java.util.logging.Logger for all exceptions.

Adherence to IEEE Standards

1. **Modularity:**
    - Ensure each method/class has a single responsibility.
    - Split functionality into small, reusable classes.
2. **Java Best Practices:**
    - Use final for constants.
    - Favor interfaces for contracts.
    - Prefer enhanced for-each loops over traditional loops for readability.

2. Java Code Review Checklist for Inventory

File name	Inventory.java
class/interface name	Inventory

Category	Checklist Item	Yes/ No	Issue	Fix
Naming Conventions	Are class names written in PascalCase?	Yes	None	None
	Are variable and method names written in camelCase?	Yes	takeFromInventory and accessInventory could be	Rename takeFromInventory to isTransactionSaleOrRental and accessInventory to

			more descriptive.	loadInventoryFromFile.
	Are constants written in uppercase with underscores?	No	No constants defined for frequently used values (e.g., file paths).	Define constants for commonly used file paths.
Code Structure	Are access modifiers used correctly?	No	Some fields (uniqueInstance) should be private.	Make uniqueInstance private, and ensure all fields have appropriate access modifiers.
	Are classes and interfaces separated?	Yes	None	None
	Are packages used appropriately?	No	Package not specified, reducing project organization.	Add package declaration (e.g., com.store.inventory) and organize project files accordingly.
Method Design	Do methods have a single responsibility?	Mostly	Methods like updateInventory have multiple responsibilities (e.g., updating inventory and writing to a file).	Refactor updateInventory to separate the logic for updating the inventory and file I/O into distinct methods.
	Are method parameters limited?	Yes	None	None
	Is method overloading used properly?	Yes	None	None
Exception Handling	Are exceptions handled with try-catch blocks?	Yes	Exception handling is present but could be	Replace System.out.println with a logger and handle specific

			improved with logging.	exceptions where needed.
	Are specific exceptions used?	Yes	None	None
Code Readability	Are comments added for complex logic?	No	Lack of comments in some complex parts like file handling in updateInventory.	Add comments to explain the logic, especially in file operations and complex inventory updates.
	Is indentation consistent?	Yes	None	None
	Are blank lines used to separate code blocks?	Yes	None	None
	Are meaningful names used for variables, classes, and methods?	Mostly	counter2, fileR are not very descriptive.	Rename counter2 to itemIndex, fileR to fileReader.
Performance	Are data structures chosen based on performance?	Yes	None	None
	Are costly operations minimized in loops?	Yes	Looping over the entire databaseItem list for each transaction item could be optimized.	Refactor to reduce nested looping; consider using a map for quicker lookups.
	Is lazy initialization used?	Yes	None	None
Memory Management	Are unnecessary	No	No clear memory	Use try-with-resources or

	object references set to null?		management in updateInventory method.	explicitly close any resources (like file readers or writers) after use.
Security	Is user input validated?	No	No validation for input or file paths.	Implement basic validation for file paths and data integrity (e.g., ensure no null values).
Maintainability	Are there long methods or deeply nested loops?	Yes	updateInventory method has nested loops and multiple responsibilities.	Refactor updateInventory to break down into smaller, modular methods.
	Is there duplicated code?	Yes	Repeated logic in file reading and writing operations.	Extract file I/O logic into separate helper methods.
	Are there any magic numbers?	Yes	Hardcoded file paths could be considered as magic strings.	Define constants for file paths used throughout the code.

Test Related Categories

Category	Checklist Item	Yes/No	Issue	
Test Coverage	Are unit tests provided for all public methods and critical functionalities?	Yes	None	None
	Do unit tests cover edge cases and boundary values?	Yes	Tests for edge cases such as non-existing employees, empty lists, read-only file scenarios are well-covered.	None
Test Design	Are tests written following the AAA pattern?	Yes	None	None

	Are individual test cases independent?	Yes	None	None
	Are descriptive names used for test methods?	Yes	None	None
Assertions	Are assertions used to verify expected results?	Yes	None	None
	Are specific assertions used instead of general ones?	Yes	None	None
Boundary and Edge Cases	Are edge cases and boundary conditions tested?	Yes	None	None
	Are invalid inputs covered by tests?	Yes	None	None
Mocking and Stubbing	Are mocks or stubs used to isolate the unit under test?	Yes	None	None
Performance Testing	Are tests to check performance for critical methods provided?	No	No specific performance tests (e.g., handling large files).	Add per tests t handlin data se prolong delete, operati
Test Maintainability	Are test methods organized and modular?	Yes	None	None
	Is there a setup method for initializing common objects?	Yes	None	None

Housekeeping

Category	Checklist Item	Yes/No	Issue	Fix
Code Smells	Are there any code smells not covered by the checklist?	Yes	- Repeated file handling code for reading and writing inventory data.	Extract file handling code into separate helper methods to reduce redundancy and improve modularity.

			- updateInventory method directly manipulates databaseItem without isolation, potentially causing side effects.	Use a temporary list to store data in updateInventory and assign it to databaseItem only after updating is complete.
Coding Standards	Are there any coding standard violations not covered by the checklist?	Yes	- Inconsistent usage of hard-coded strings (e.g., file paths, "ItemID", "Amount")	Define these values as constants (e.g., FILE_PATH_INVENTORY).
			- No consistent error handling strategy; exceptions are caught but not properly logged.	Replace System.out.println in catch blocks with a logging framework (e.g., Logger) for better error tracking.
Performance Inefficiencies	Are there any performance inefficiencies not covered by the checklist?	Yes	- Repeated getItemID() call in updateInventory, which could be optimized.	Cache itemID lookups using a Map<Integer, Item> for faster performance in the loop.
			- Direct string concatenation used in multiple areas (e.g., file output).	Use StringBuilder for concatenations, especially in loops, to improve performance when building larger strings.

Refactoring Tasks

Refactoring Task 1: Separate File Reading and Writing Logic

- **Issue**: The `updateInventory` method has logic for both updating inventory and writing to a file, which violates the single responsibility principle.
- **Refactoring Plan**: Separate the file reading and writing logic into distinct methods to improve maintainability and readability.

Refactoring Task 2: Cache Item Lookup to Improve Performance

- **Issue**: The method `updateInventory` loops over `databaseItem` for every item in `transactionItem`, which is inefficient when dealing with large datasets.
- **Refactoring Plan**: Use a `Map` (e.g., `HashMap`) for faster lookups of items by their `itemID`, reducing the time complexity of searching for items in `databaseItem`.

Refactoring Task 3: Replace `System.out.println` with Logging

- **Issue**: `System.out.println` is used for error handling in catch blocks, which is not a scalable or best practice for production

Appendix:

Code Smells

Code Smell	Description	Example	Suggested Solution
Long Methods	Methods that perform multiple tasks, making them hard to understand and maintain.	The updateInventory method combines inventory updating and file I/O logic.	Split updateInventory into smaller methods: one for inventory updating and another for file operations.
Nested Loops	Deeply nested loops reduce readability and increase complexity.	Nested looping over transactionItem and databaseItem.	Use a HashMap to cache databaseItem by itemID to minimize nested loops.

Duplicate Code	Repeated code for file handling and reading/writing inventory data.	File reading and writing logic appears in multiple places.	Extract file handling logic into reusable helper methods.
Magic Numbers/Strings	Hardcoded values or strings used without explanation.	Hardcoded file paths like "inventory.txt" and string keys like "ItemID".	Define constants for file paths and string keys (e.g., FILE_PATH_INVENTORY, KEY_ITEM_ID).
Primitive Obsession	Overuse of primitive types instead of meaningful abstractions.	Using strings to represent itemID or transaction status instead of enums or classes.	Use an enum for transaction types and consider using a class for inventory items.

Performance Inefficiencies

Issue	Description	Example	Suggested Solution
Repeated Object Lookups	Scanning the entire databaseItem list for every transactionItem.	Iterating through the entire list for each transaction to find the matching item.	Use a HashMap<Integer, Item> to store databaseItem for constant-time lookups.
Redundant Calculations	Performing the same calculation repeatedly in a loop.	Repeated calls to getItemID() inside nested loops.	Store itemID in a local variable before the loop to minimize method calls.
Inefficient String Ops	Using string concatenation inside loops.	Using result += "Item: " + itemName in a loop.	Use StringBuilder for concatenating strings inside loops to improve performance.
Excessive Logging	Printing to the console excessively instead	Using System.out.println for	Replace System.out.println with a proper logging

	of structured logging.	error reporting in catch blocks.	framework like java.util.logging or Log4j.

Housekeeping

Category	Issue	Fix
Code Smells	Duplicate file handling code.	Extract file reading/writing logic into dedicated helper methods.
	Combined logic in updateInventory that violates single responsibility principle.	Separate updateInventory into modular methods for inventory processing and file updates.
Coding Standards	Hardcoded strings like file paths and key names.	Define constants for all commonly used strings.
	Inconsistent error handling (direct printing in catch blocks).	Replace direct System.out.println calls with proper logging using a logging framework.
Performance Inefficiencies	Repeated list traversal for inventory lookup.	Cache databaseItem in a HashMap for efficient lookups.
	Direct string concatenation inside loops.	Use StringBuilder for string manipulations inside loops.

Category	Checklist Item	Yes/No	Issue	Fix
Test Coverage	Are unit tests provided for all public methods and critical functionalities?	Yes	None	None
	Do unit tests cover edge cases and boundary values?	Yes	Tests for edge cases such as non-existing employees, empty lists, read-only file	None

			scenarios are well-covered.	
Test Design	Are tests written following the AAA pattern?	Yes	None	None
	Are individual test cases independent?	Yes	None	None
	Are descriptive names used for test methods?	Yes	None	None
Assertions	Are assertions used to verify expected results?	Yes	None	None
	Are specific assertions used instead of general ones?	Yes	None	None
Boundary and Edge Cases	Are edge cases and boundary conditions tested?	Yes	None	None
	Are invalid inputs covered by tests?	Yes	None	None
Mocking and Stubbing	Are mocks or stubs used to isolate the unit under test?	Yes	None	None
Performance Testing	Are tests to check performance for critical methods provided?	No	No specific performance tests (e.g., handling large files).	Add performance tests to validate handling of large data sets, or prolonged add, delete, update operations.
Test Maintainability	Are test methods organized and modular?	Yes	None	None
	Is there a setup method for	Yes	None	None

	initializing common objects?			

3. Java Code Review Checklist for Item

File name	Item.java
class/interface name	Item

Category	Checklist Item	Yes/No	Issue	Fix
Naming Conventions	Are class names written in PascalCase?	Yes	None	None
	Are variable and method names written in camelCase?	Yes	None	None
	Are constants written in uppercase with underscores?	Not Applicable	None	None
Code Structure	Are access modifiers used correctly?	Mostly	Some getter methods are missing public access modifiers.	Make getter methods public to allow access from outside the class.
	Are classes and interfaces separated?	Yes	None	None

	Are packages used appropriately?	No	No package specified, reducing project organization.	Add package declaration (e.g., com.store.inventory) and organize project files accordingly.
Method Design	Do methods have a single responsibility?	Yes	None	None
	Are method parameters limited?	Yes	None	None
	Is method overloading used properly?	Not Applicable	None	None
Exception Handling	Are exceptions handled with try-catch blocks?	Not Applicable	None	None
	Are specific exceptions used?	Not Applicable	None	None
Code Readability	Are comments added for complex logic?	No	No comments explaining the attributes or purpose of methods.	Add comments to describe the purpose of each method and attribute.
	Is indentation consistent?	Yes	None	None
	Are blank lines used to separate code blocks?	Yes	None	None
	Are meaningful names used for variables,	Yes	None	None

	classes, and methods?			
Performance	Are data structures chosen based on performance?	Not Applicable	None	None
	Are costly operations minimized in loops?	Not Applicable	None	None
	Is lazy initialization used?	Not Applicable	None	None
Memory Management	Are unnecessary object references set to null?	Not Applicable	None	None
Security	Is user input validated?	Not Applicable	None	None
Maintainability	Are there long methods or deeply nested loops?	No	None	None
	Is there duplicated code?	No	None	None
	Are there any magic numbers?	Yes	0 and -2 in test cases lack explanation.	Define constants for boundary values (e.g., MIN_AMOUNT).

Test Related Categories

Category	Checklist Item	Yes/No	Issue	Fix
Test Coverage	Are unit tests provided for all public methods and critical functionalities?	Yes	None	None

	Do unit tests cover edge cases and boundary values?	Yes	Tests for edge cases such as zero and negative amounts are covered.	None
Test Design	Are tests written following the AAA pattern?	Yes	None	None
	Are individual test cases independent?	Yes	None	None
	Are descriptive names used for test methods?	Yes	None	None
Assertions	Are assertions used to verify expected results?	Yes	None	None
	Are specific assertions used instead of general ones?	Yes	None	None
Boundary and Edge Cases	Are edge cases and boundary conditions tested?	Yes	None	None
	Are invalid inputs covered by tests?	Mostly	Edge cases for negative amounts are tested, but no validation for input.	Consider adding validation logic to updateAmount.
Mocking and Stubbing	Are mocks or stubs used to isolate the unit under test?	Not Applicable	None	None

Performance Testing	Are tests to check performance for critical methods provided?	Not Applicable	None	None
Test Maintainability	Are test methods organized and modular?	Yes	None	None
	Is there a setup method for initializing common objects?	Not Applicable	None	None

Housekeeping

Category	Checklist Item	Yes/No	Issue	Fix
Code Smells	Are there any code smells not covered by the checklist?	No	None	None
Coding Standards	Are there any coding standard violations not covered by the checklist?	No	None	None
Performance Inefficiencies	Are there any performance inefficiencies not covered by the checklist?	No	None	None

Refactoring Task 1: Improve Access Modifiers for Getter Methods

- **Issue**: Getter methods (`getItemName`, `getItemID`, `getPrice`, `getAmount`) currently have no explicit access modifiers, which defaults them to package-private. This limits accessibility if the class is used in other packages.
- **Refactoring Plan**: Make all getter methods `public` to allow controlled access from outside packages, ensuring that the `Item` class is self-contained and usable in various contexts.

Refactoring Task 2: Add Validation for Negative Amounts in `updateAmount`

- **Issue**: The `updateAmount` method allows setting negative values, which may not be a valid scenario for inventory management.
- **Refactoring Plan**: Add validation to `updateAmount` to prevent setting negative values, or throw an exception if a negative value is attempted. This can help avoid errors and ensure data integrity for item quantities.

Refactoring Task 3: Add Comments and Documentation for Methods and Attributes

- **Issue**: The `Item` class lacks comments and documentation, which may reduce code readability, especially for new developers or when maintaining the code.
- **Refactoring Plan**: Add descriptive comments for class attributes and each method, specifying their purpose, inputs, and expected behavior. This will improve readability and make the codebase easier to understand and maintain.

1. Code Smells for Item.java

Code Smell	Description	Example	Suggested Solution
Long Methods	No long methods detected in the class.	N/A	N/A
Nested Loops	No nested loops detected in the class.	N/A	N/A
Duplicate Code	No duplicate code detected in the class.	N/A	N/A
Data Clumps	No data clumps detected in the class.	N/A	N/A
Primitive Obsession	Use of magic numbers (0 and -2) without explanation.	0 and -2 in test cases without explanations.	Define constants for boundary values (e.g., MIN_AMOUNT).

---

2. Violations of Coding Standards for Item.java

Issue	Description	Example	Suggested Fix
Naming Conventions	No violations detected.	N/A	N/A
Lack of Comments	Insufficient comments, especially for complex logic.	No comments explaining the attributes or methods.	Add comments to describe the purpose of each method and attribute.

Inconsistent Formatting	Formatting is consistent.	N/A	N/A
Magic Numbers	Magic numbers used in test cases (e.g., 0 and -2) without explanation.	0 and -2 in test cases without explanations.	Define constants for boundary values (e.g., MIN_AMOUNT).

---

3. Performance Inefficiencies for Item.java

Issue	**Description**	**Example**	**Suggested Solution**
Unnecessary Object Creation	No unnecessary object creation detected.	N/A	N/A
Inefficient Data Structures	No inefficient data structures detected.	N/A	N/A
Redundant Calculations	No redundant calculations detected.	N/A	N/A
Excessive Logging	No excessive logging detected.	N/A	N/A
Inefficient String Operations	No inefficient string operations detected.	N/A	N/A

Additional Comments

- **Packages**: The file does not specify a package. To improve project organization, consider adding a package declaration (e.g., com.store.inventory) and organizing project files accordingly.
- **Getter Methods**: Some getter methods are missing public access modifiers. Ensure all getter methods are public to allow access from outside the class.
- **Edge Case for Negative Amounts**: While edge cases such as zero and negative amounts are tested, the validation logic for input (e.g., updateAmount) should be improved to handle invalid input validation.

4. Java Code Review Checklist for Management

File name	Management.java
class/interface name	Management

Category	Checklist Item	Yes/No	Issue	Fix
Naming Conventions	Are class names written in PascalCase?	Yes	None	None
	Are variable and method names written in camelCase?	Mostly	Method names like checkUser and getLatestReturnDate lack verb clarity, e.g., isUserInDatabase may be clearer.	Update method names for clarity, e.g., checkUser to isUserInDatabase.
	Are constants written in uppercase with underscores?	No	userDatabase should be a constant, and uppercase with underscores.	Define userDatabase as private static final String USER_DATABASE.
Code Structure	Are access modifiers used correctly?	No	Some fields and methods could be private, such as userDatabase.	Change visibility of userDatabase to private static final and ensure encapsulation.
	Are classes and interfaces separated?	Yes	None	None
	Are packages used appropriately?	No	Package not specified, reducing project organization.	Add package declaration (e.g., com.store.management) and organize project files accordingly.
Method Design	Do methods have a	No	Some methods like	Refactor to break down methods with

	single responsibility?		getLatestReturnDate combine multiple responsibilities.	multiple responsibilities into smaller, single-responsibility methods.
	Are method parameters limited?	Mostly	Some parameters could be combined into a single User object, especially in methods related to user checks.	Introduce a User class to encapsulate attributes like phone number.
	Is method overloading used properly?	Not Applicable	None	None
Exception Handling	Are exceptions handled with try-catch blocks?	Yes	Error messages in catch blocks are printed instead of logged.	Replace System.out.println with logging (e.g., Logger) to handle exceptions in a production-friendly manner.
	Are specific exceptions used?	Yes	None	None
Code Readability	Are comments added for complex logic?	No	Some complex parts (e.g., daysBetween) lack comments to explain logic.	Add comments to explain non-intuitive logic, such as calculating date differences.
	Is indentation consistent?	Yes	None	None
	Are blank lines used to separate code blocks?	Yes	None	None
	Are meaningful names used	Mostly	Variable names like fileR and	Use more descriptive names, such as

	for variables, classes, and methods?		line could be clearer.	fileReader for fileR and userRecord for line.
Performance	Are data structures chosen based on performance?	Yes	None	None
	Are costly operations minimized in loops?	No	Parsing and looping operations could be optimized in getLatestReturnDate.	Consider using a Map or Set to speed up data retrieval and prevent redundant operations.
	Is lazy initialization used?	Yes	None	None
Memory Management	Are unnecessary object references set to null?	No	No explicit resource cleanup or usage of try-with-resources.	Use try-with-resources to automatically close file readers and writers after use.
Security	Is user input validated?	No	Phone numbers and database inputs are not validated.	Validate phone numbers and other sensitive inputs to prevent invalid data entries.
Maintainability	Are there long methods or deeply nested loops?	Yes	Methods like getLatestReturnDate contain deeply nested loops and conditional statements.	Refactor to simplify nested loops and use helper methods for specific tasks.
	Is there duplicated code?	Yes	Repeated logic in reading from and writing to userDatabase.	Extract repeated logic into reusable helper methods.

	Are there any magic numbers?	Yes	Hardcoded file paths and date formats are magic values.	Define constants for commonly used file paths and date formats.
Test Related Categories				

Category	Checklist Item	Yes/No	Issue	Fix
Test Coverage	Are unit tests provided for all public methods and critical functionalities?	Yes	None	None
	Do unit tests cover edge cases and boundary values?	Yes	Edge cases like empty files, non-existent users, and various return statuses are well-covered.	None
Test Design	Are tests written following the AAA pattern?	Yes	None	None
	Are individual test cases independent?	Yes	None	None
	Are descriptive names used for test methods?	Yes	None	None
Assertions	Are assertions used to verify expected results?	Yes	None	None
	Are specific assertions used instead of general ones?	Yes	None	None
Boundary and Edge Cases	Are edge cases and boundary	Yes	None	None

	conditions tested?			
	Are invalid inputs covered by tests?	Mostly	Tests cover invalid input cases for non-existent users and empty files, but validation logic could be added.	Consider adding input validation logic.
Mocking and Stubbing	Are mocks or stubs used to isolate the unit under test?	Not Applicable	None	None
Performance Testing	Are tests to check performance for critical methods provided?	No	No specific performance tests for file handling or date processing.	Add performance tests to validate efficiency in handling large data sets.
Test Maintainability	Are test methods organized and modular?	Yes	None	None
	Is there a setup method for initializing common objects?	Yes	None	None

Housekeeping

Category	Checklist Item	Yes/No	Issue	Fix
Code Smells	Are there any code smells not covered by the checklist?	Yes	- Repeated file handling code for reading and writing user data.	Extract file handling code into helper methods to improve modularity.
			- Complex logic in getLatestReturnDate	Break down complex

			and updateRentalStatus.	logic into helper methods for readability.
Coding Standards	Are there any coding standard violations not covered by the checklist?	Yes	- No consistent error handling strategy; exceptions are caught but not logged.	Use a logging framework for consistent and production-grade error handling.
Performance Inefficiencies	Are there any performance inefficiencies not covered by the checklist?	Yes	- Parsing and checking phone numbers in each loop iteration in getLatestReturnDate.	Use a Map<Long, User> to improve lookup performance, especially for larger databases.
			- Direct string concatenation used for modifying lines (e.g., line = line + " " + item).	Use StringBuilder for more efficient concatenation in loops.

Refactoring Task 1: Separate File Reading and Writing Logic

- **Issue**: The methods `checkUser`, `getLatestReturnDate`, `addRental`, and `updateRentalStatus` all contain repetitive code for reading from and writing to the `userDatabase` file. This violates the single responsibility principle and introduces redundancy.
- **Refactoring Plan**: Extract the file reading and writing logic into dedicated helper methods (e.g., `readFileLines` and `writeFileLines`). These methods will handle file I/O independently, making the main methods more concise and focused.

Refactoring Task 2: Replace `System.out.println` with Proper Logging

- **Issue**: The class currently uses `System.out.println` for error handling, which is not suitable for production environments.

- **Refactoring Plan**: Implement a `Logger` (e.g., `java.util.logging.Logger`) for consistent error reporting. Replace all instances of `System.out.println` with logging statements to capture error details effectively and make the class production-ready.

Refactoring Task 3: Use a `Map` for Efficient User Lookup

- **Issue**: Methods like `getLatestReturnDate` and `updateRentalStatus` loop through each line in the file to find a user by phone number. This can be inefficient, especially with a large database.
- **Refactoring Plan**: Load `userDatabase` into a `Map<Long, String>` (where the key is the phone number and the value is the user data line) once at the beginning of each operation. This will reduce repeated looping and allow for efficient lookups by phone number, improving performance.

Code Smells

Category	Checklist Item	Yes/No	Issue	Fix
Code Smells	Are there any code smells not covered by the checklist?	Yes	Repeated file handling code for reading and writing user data.	Extract file handling code into helper methods (e.g., `readFileLines` and `writeFileLines`) to improve modularity.
			Complex logic in `getLatestReturnDate` and `updateRentalStatus`.	Break down complex logic into smaller, more manageable helper methods for better readability and maintainability.

Violations of Coding Standards

Category	Checklist Item	Yes/No	Issue	Fix
Coding Standards	Are there any coding standard violations not covered by the checklist?	Yes	No consistent error handling strategy; exceptions are caught but not logged.	Use a logging framework (e.g., `java.util.logging.Logger`) for consistent error reporting. Replace `System.out.println` with proper logging statements to capture error details.

Performance Inefficiencies

Category	Checklist Item	Yes/No	Issue	Fix
Performance Inefficiencies	Are there any performance inefficiencies not covered by the checklist?	Yes	Parsing and checking phone numbers in each loop iteration in `getLatestReturnDate`.	Use a `Map<Long, User>` to improve lookup performance, especially for larger databases. This will reduce the redundant iteration over the database.
			Direct string concatenation used for modifying lines (e.g., `line = line + " " + item`).	Use `StringBuilder` instead of string concatenation in loops for more efficient string handling and memory management.

5. Java Code Review Checklist for POH

File name	POH.java
class/interface name	POH

Category	Checklist Item	Yes/No	Issue	Fix

Naming Conventions	Are class names written in PascalCase?	Yes	None	None
	Are variable and method names written in camelCase?	Yes	Some variables like tempF, fileR, and textReader could be renamed for clarity.	Rename tempF to tempFile, fileR to fileReader, and textReader to bufferedReader for consistency.
	Are constants written in uppercase with underscores?	No	temp, tempFile, and other repeated strings should be constants.	Define constants for repeated file paths and strings such as TEMP_FILE, DATABASE_PATH, etc.
Code Structure	Are access modifiers used correctly?	Mostly	Some fields (e.g., returnList, phone) could be private for better encapsulation.	Make returnList and phone private and add appropriate getters/setters if needed.
	Are classes and interfaces separated?	Yes	None	None
	Are packages used appropriately?	No	No package specified, reducing project organization.	Add package declaration (e.g., com.pos.transaction) and organize project files accordingly.
Method Design	Do methods have a single responsibility?	Mostly	endPOS and retrieveTemp perform multiple tasks such	Break down endPOS and retrieveTemp into smaller, single-

			as file reading, updating inventory, and calculating prices.	responsibility methods for better readability and modularity.
	Are method parameters limited?	Yes	None	None
	Is method overloading used properly?	Not Applicable	None	None
Exception Handling	Are exceptions handled with try-catch blocks?	Yes	Error messages in catch blocks are printed instead of logged.	Replace System.out.println with logging (e.g., Logger) to handle exceptions in a production-friendly manner.
	Are specific exceptions used?	Yes	None	None
Code Readability	Are comments added for complex logic?	No	Some complex parts (e.g., in endPOS and retrieveTemp) lack comments to explain logic.	Add comments to explain the logic behind calculations, conditions, and file operations.
	Is indentation consistent?	Yes	None	None
	Are blank lines used to separate code blocks?	Yes	None	None
	Are meaningful names used	Mostly	Variables like tempF and type are	Rename variables for clarity, e.g., tempF to

	for variables, classes, and methods?		not very descriptive.	tempFile and type to transactionType.
Performance	Are data structures chosen based on performance?	Yes	None	None
	Are costly operations minimized in loops?	Mostly	Some file operations and transactions in endPOS could be optimized.	Consider optimizing endPOS by reducing nested loops or using a Map for quicker lookups.
	Is lazy initialization used?	Yes	None	None
Memory Management	Are unnecessary object references set to null?	No	No explicit resource cleanup or usage of try-with-resources for file handling.	Use try-with-resources to automatically close file readers and writers after use.
Security	Is user input validated?	No	Phone numbers and IDs are not validated.	Validate phone numbers and other sensitive inputs to prevent invalid data entries.
Maintainability	Are there long methods or deeply nested loops?	Yes	Methods like endPOS contain deeply nested loops and conditional statements.	Refactor to simplify nested loops and use helper methods for specific tasks.
	Is there duplicated code?	Yes	Repeated code for file	Extract file handling logic into reusable

			operations in deleteTempItem, endPOS, and retrieveTemp.	helper methods to improve modularity.
	Are there any magic numbers?	Yes	Hardcoded paths for tempFile, returnSaleFile, etc., are magic values.	Define constants for commonly used file paths.

Test Related Categories

Category	Checklist Item	Yes/No	Issue	Fix
Test Coverage	Are unit tests provided for all public methods and critical functionalities?	Yes	None	None
	Do unit tests cover edge cases and boundary values?	Yes	Tests for edge cases such as missing files, read-only files, etc., are covered.	None
Test Design	Are tests written following the AAA pattern?	Yes	None	None
	Are individual test cases independent?	Yes	None	None
	Are descriptive names used for test methods?	Yes	None	None
Assertions	Are assertions used to verify	Yes	None	None

	expected results?			
	Are specific assertions used instead of general ones?	Yes	None	None
Boundary and Edge Cases	Are edge cases and boundary conditions tested?	Yes	None	None
	Are invalid inputs covered by tests?	Mostly	Edge cases for file-related issues are tested, but additional validations could be added.	Consider adding input validation logic.
Mocking and Stubbing	Are mocks or stubs used to isolate the unit under test?	Not Applicable	None	None
Performance Testing	Are tests to check performance for critical methods provided?	No	No specific performance tests for endPOS or retrieveTemp.	Add performance tests to validate efficiency in handling large transactions and frequent file updates.
Test Maintainability	Are test methods organized and modular?	Yes	None	None
	Is there a setup method for	Yes	None	None

	initializing common objects?			

Housekeeping				
Category	Checklist Item	Yes/No	Issue	Fix
Code Smells	Are there any code smells not covered by the checklist?	Yes	- Repeated file handling code across methods.	Extract file handling code into separate helper methods.
			- Large endPOS method with multiple responsibilities.	Break down endPOS into smaller methods to make each part more manageable.
Coding Standards	Are there any coding standard violations not covered by the checklist?	Yes	- No consistent error handling strategy; exceptions are caught but not properly logged.	Replace System.out.println with a logging framework for better error tracking.
Performance Inefficiencies	Are there any performance inefficiencies not covered by the checklist?	Yes	- Potential performance issues with nested loops in endPOS.	Use more efficient data structures or strategies to avoid nested looping.
			- Direct file operations in several methods without validation.	Add file existence and permissions checks, and consolidate file operations into helper methods for better error handling.

Refactoring Task 1: Consolidate File Handling Logic

- **Issue**: File handling operations (reading, writing, and modifying files) are repeated across methods, which increases redundancy and complexity.
- **Refactoring Plan**: Create dedicated helper methods, `readFileLines` and `writeFileLines`, to manage file I/O operations. These methods will handle file reading and writing independently, making the primary methods (like `deleteTempItem`, `endPOS`, and `retrieveTemp`) cleaner and more focused.

Refactoring Task 2: Simplify `endPOS` Method and Use Helper Functions

- **Issue**: The `endPOS` method performs multiple operations—calculating total price, updating inventory, and writing data—making it challenging to read and maintain.
- **Refactoring Plan**: Break down `endPOS` into smaller helper functions. For example, create `calculateTotalPrice` for pricing calculations, `writeReturnSaleLog` to handle logging returned sales, and `updateInventory` to handle inventory updates. This will allow each function to focus on a single responsibility, improving readability and modularity.

Refactoring Task 3: Implement Logging and Replace `System.out.println` Statements

- **Issue**: `System.out.println` is used for error reporting, which is not ideal for production-level code and lacks error detail consistency.
- **Refactoring Plan**: Use `java.util.logging.Logger` for consistent and detailed error logging. Replace all `System.out.println` statements with logger calls to capture exceptions and information messages, enabling easier debugging and error tracking in production.

Code Smells

Checklist Item	**Issue**	**Fix**
File Handling Redundancy	Repeated file handling code across methods.	Extract file handling code into separate helper methods for better reusability and clarity.
Large Method with Multiple Responsibilities	endPOS method performs multiple tasks (file reading, updating inventory, calculating prices).	Break down endPOS into smaller methods (e.g., calculateTotalPrice, writeReturnSaleLog, updateInventory).

Lack of Logging for Errors	Errors are only printed with System.out.println.	Replace System.out.println with java.util.logging.Logger for better error handling and production-grade logging.
Non-descriptive Variable Names	Variables like tempF, type, fileR are not descriptive.	Rename variables for clarity (e.g., tempF to tempFile, type to transactionType, fileR to fileReader).

Violations of Coding Standards

Checklist Item	**Issue**	**Fix**
Naming Convention Violation	Constants like temp and tempFile are not written in uppercase with underscores.	Define constants for repeated values (e.g., TEMP_FILE, DATABASE_PATH).
Access Modifier Violation	Some fields (e.g., returnList, phone) are public instead of private.	Make fields like returnList and phone private and provide getters/setters if needed.
Package Declaration Missing	No package declared for the file, reducing project organization.	Add a package declaration (e.g., com.pos.transaction) at the top of the file.
Error Handling Violation	System.out.println is used for error reporting instead of logging.	Use a logging framework (e.g., java.util.logging.Logger) for consistent and detailed error logging.

Performance Inefficiencies

Checklist Item	**Issue**	**Fix**
Nested Loops and Inefficient Operations	Some nested loops in endPOS could be optimized, leading to performance overhead.	Consider using a Map or other data structures to optimize lookups and minimize nested loops.
Repeated File Operations	File operations (e.g., reading, writing) are repeated across	Consolidate file operations into helper methods to

	methods, which increases I/O overhead.	minimize redundancy and optimize performance.
Lack of Input Validation for File Operations	File operations in several methods are not validated for existence or permissions.	Add checks for file existence and permissions before performing file operations.
No Performance Testing	There are no specific tests for performance bottlenecks like endPOS or retrieveTemp.	Add performance tests to validate efficiency in handling large datasets and frequent file operations.

6. Java Code Review Checklist for Point Of Sale

File name	Point Of Sale.java
class/interface name	Point Of Sale

Category	Checklist Item	Yes/No	Issue	Fix
Naming Conventions	Are class names written in PascalCase?	Yes	None	None
	Are variable and method names written in camelCase?	Yes	Some variables like tempF, fileR, and textReader could be renamed for clarity.	Rename tempF to tempFile, fileR to fileReader, and textReader to bufferedReader for consistency.
	Are constants written in uppercase	No	temp, tempFile, and other repeated strings	Define constants for repeated file paths and strings such as TEMP_FILE,

	with underscores?		should be constants.	DATABASE_PATH, etc.
Code Structure	Are access modifiers used correctly?	Mostly	Some fields (e.g., returnList, phone) could be private for better encapsulation.	Make returnList and phone private and add appropriate getters/setters if needed.
	Are classes and interfaces separated?	Yes	None	None
	Are packages used appropriately?	No	No package specified, reducing project organization.	Add package declaration (e.g., com.pos.transaction) and organize project files accordingly.
Method Design	Do methods have a single responsibility?	Mostly	endPOS and retrieveTemp perform multiple tasks such as file reading, updating inventory, and calculating prices.	Break down endPOS and retrieveTemp into smaller, single-responsibility methods for better readability and modularity.
	Are method parameters limited?	Yes	None	None
	Is method overloading used properly?	Not Applicable	None	None
Exception Handling	Are exceptions handled with	Yes	Error messages in catch blocks	Replace System.out.println with logging

	try-catch blocks?		are printed instead of logged.	(e.g., Logger) to handle exceptions in a production-friendly manner.
	Are specific exceptions used?	Yes	None	None
Code Readability	Are comments added for complex logic?	No	Some complex parts (e.g., in endPOS and retrieveTemp) lack comments to explain logic.	Add comments to explain the logic behind calculations, conditions, and file operations.
	Is indentation consistent?	Yes	None	None
	Are blank lines used to separate code blocks?	Yes	None	None
	Are meaningful names used for variables, classes, and methods?	Mostly	Variables like tempF and type are not very descriptive.	Rename variables for clarity, e.g., tempF to tempFile and type to transactionType.
Performance	Are data structures chosen based on performance?	Yes	None	None
	Are costly operations minimized in loops?	Mostly	Some file operations and transactions in endPOS could be optimized.	Consider optimizing endPOS by reducing nested loops or using a Map for quicker lookups.

	Is lazy initialization used?	Yes	None	None
Memory Management	Are unnecessary object references set to null?	No	No explicit resource cleanup or usage of try-with-resources for file handling.	Use try-with-resources to automatically close file readers and writers after use.
Security	Is user input validated?	No	Phone numbers and IDs are not validated.	Validate phone numbers and other sensitive inputs to prevent invalid data entries.
Maintainability	Are there long methods or deeply nested loops?	Yes	Methods like endPOS contain deeply nested loops and conditional statements.	Refactor to simplify nested loops and use helper methods for specific tasks.
	Is there duplicated code?	Yes	Repeated code for file operations in deleteTempItem, endPOS, and retrieveTemp.	Extract file handling logic into reusable helper methods to improve modularity.
	Are there any magic numbers?	Yes	Hardcoded paths for tempFile, returnSaleFile, etc., are magic values.	Define constants for commonly used file paths.

Test Related Categories

Category	Checklist Item	Yes/No	Issue	Fix

Test Coverage	Are unit tests provided for all public methods and critical functionalities ?	Yes	None	None
	Do unit tests cover edge cases and boundary values?	Yes	Tests for edge cases such as missing files, read-only files, etc., are covered.	None
Test Design	Are tests written following the AAA pattern?	Yes	None	None
	Are individual test cases independent?	Yes	None	None
	Are descriptive names used for test methods?	Yes	None	None
Assertions	Are assertions used to verify expected results?	Yes	None	None
	Are specific assertions used instead of general ones?	Yes	None	None
Boundary and Edge Cases	Are edge cases and boundary conditions tested?	Yes	None	None
	Are invalid inputs covered by tests?	Mostly	Edge cases for file-related issues are tested, but additional validations	Consider adding input validation logic.

			could be added.	
Mocking and Stubbing	Are mocks or stubs used to isolate the unit under test?	Not Applicable	None	None
Performance Testing	Are tests to check performance for critical methods provided?	No	No specific performance tests for endPOS or retrieveTemp.	Add performance tests to validate efficiency in handling large transactions and frequent file updates.
Test Maintainability	Are test methods organized and modular?	Yes	None	None
	Is there a setup method for initializing common objects?	Yes	None	None

Housekeeping

Category	Checklist Item	Yes/No	Issue	Fix
Code Smells	Are there any code smells not covered by the checklist?	Yes	- Repeated file handling code across methods.	Extract file handling code into separate helper methods.
			- Large endPOS method with multiple responsibilities.	Break down endPOS into smaller methods to make each part more manageable.

Coding Standards	Are there any coding standard violations not covered by the checklist?	Yes	- No consistent error handling strategy; exceptions are caught but not properly logged.	Replace System.out.println with a logging framework for better error tracking.
Performance Inefficiencies	Are there any performance inefficiencies not covered by the checklist?	Yes	- Potential performance issues with nested loops in endPOS.	Use more efficient data structures or strategies to avoid nested looping.
			- Direct file operations in several methods without validation.	Add file existence and permissions checks, and consolidate file operations into helper methods for better error handling.

Refactoring Task 1: Consolidate File Handling Logic

- **Issue**: File handling operations (reading, writing, and modifying files) are repeated across methods, which increases redundancy and complexity.
- **Refactoring Plan**: Create dedicated helper methods, `readFileLines` and `writeFileLines`, to manage file I/O operations. These methods will handle file reading and writing independently, making the primary methods (like `deleteTempItem`, `endPOS`, and `retrieveTemp`) cleaner and more focused.
    - `readFileLines`: Handles reading data from files.
    - `writeFileLines`: Handles writing data to files.
- **Benefit**: Reduces redundancy, improves readability, and centralizes file operations for easier maintenance and future changes.

Refactoring Task 2: Simplify endPOS Method and Use Helper Functions

- **Issue**: The `endPOS` method performs multiple operations—calculating total price, updating inventory, and writing data—making it challenging to read and maintain.
- **Refactoring Plan**: Break down `endPOS` into smaller helper functions:

    - `calculateTotalPrice`: Handles pricing calculations.
    - `writeReturnSaleLog`: Handles logging returned sales.
    - `updateInventory`: Handles inventory updates.
- **Benefit**: Improves readability by focusing each method on a single responsibility, reduces complexity, and enhances modularity.

Refactoring Task 3: Implement Logging and Replace System.out.println Statements

- **Issue**: `System.out.println` is used for error reporting, which is not ideal for production-level code and lacks error detail consistency.
- **Refactoring Plan**: Use `java.util.logging.Logger` for consistent and detailed error logging. Replace all `System.out.println` statements with logger calls to capture exceptions and information messages. This will provide:
    - Better error tracking.
    - More detailed and structured logs.
- **Benefit**: Facilitates easier debugging and error tracking in production environments, provides consistent log output, and aligns with industry best practices.

Code Smells

Checklist Item	Issue	Fix
Repeated File Handling Code	File handling logic is repeated in multiple methods such as deleteTempItem, endPOS, retrieveTemp.	Extract file handling logic into a helper method to improve code reusability and modularity.
Large Method with Multiple Responsibilities	endPOS performs multiple tasks like file reading, updating inventory, and calculating prices.	Break endPOS into smaller methods, each with a single responsibility, for better readability.

Violations of Coding Standards

Checklist Item	Issue	Fix
Inconsistent Error Handling	Exceptions are caught but not properly logged (using	Replace System.out.println with a logging framework like java.util.logging.Logger to handle

	System.out.println instead of proper logging).	exceptions in a production-grade way.
Inconsistent Use of Constants	Strings like file paths and repeated values are hardcoded instead of being defined as constants.	Define constants for repeated file paths and strings such as TEMP_FILE, DATABASE_PATH, etc.
Access Modifier Issues	Some fields, such as returnList and phone, are public when they could be private for better encapsulation.	Make fields private and use getters/setters where necessary.
Package Declaration Missing	No package declaration, reducing the organization of the project.	Add a package declaration like com.pos.transaction to organize project files.

Performance Inefficiencies

Checklist Item	**Issue**	**Fix**
Nested Loops	Nested loops in endPOS may cause performance issues with large datasets.	Consider optimizing nested loops by using data structures like Map to improve lookup efficiency.
Lack of Input Validation for File Operations	File operations in methods like deleteTempItem and endPOS are not validated for file existence or permissions.	Add checks for file existence and permissions before performing file operations.
File Operations in Multiple Methods	File operations are directly handled in multiple methods without any consolidated handling.	Extract file operations into a reusable helper method to avoid redundancy and improve maintainability.

7. Java Code Review Checklist for POR

File name	POR.java

class/interface name	POR

Category	Checklist Item	Yes/No	Issue	Fix
Naming Conventions	Are class names written in PascalCase?	Yes	None	None
	Are variable and method names written in camelCase?	Yes	Some variables like tempData could be more descriptive.	Rename tempData to returnData for clarity.
	Are constants written in uppercase with underscores?	No	TEMP_FILE, RETURN_ITEM_FILE, etc., are hardcoded.	Define constants for file paths and string literals used multiple times (e.g., RETURN_ITEM_FILE_PATH).
Code Structure	Are access modifiers used correctly?	Mostly	Some fields (e.g., returnList, transactionId) could be private for better encapsulation.	Make returnList and transactionId private and add appropriate getters/setters if necessary.
	Are classes and interfaces separated?	Yes	None	None
	Are packages used appropriately?	No	No package specified, reducing project organization.	Add package declaration (e.g., com.pos.return) and organize project files accordingly.

Method Design	Do methods have a single responsibility?	Mostly	Methods like processReturn handle multiple tasks such as calculating refund and updating inventory.	Refactor processReturn to break down tasks into smaller, single-responsibility methods.
	Are method parameters limited?	Yes	None	None
	Is method overloading used properly?	Not Applicable	None	None
Exception Handling	Are exceptions handled with try-catch blocks?	Yes	Error messages in catch blocks are printed instead of logged.	Replace System.out.println with a logging framework (e.g., Logger) to handle exceptions more effectively.
	Are specific exceptions used?	Yes	None	None
Code Readability	Are comments added for complex logic?	No	Some methods like processReturn and updateInventory lack comments explaining complex logic.	Add comments to explain logic for calculation, file handling, and inventory updates.
	Is indentation consistent?	Yes	None	None
	Are blank lines used to separate	Yes	None	None

	code blocks?			
	Are meaningful names used for variables, classes, and methods?	Mostly	Variables like tempData and transactionId are not fully descriptive of their purpose.	Rename variables like tempData to returnData and transactionId to returnTransactionId for better clarity.
Performance	Are data structures chosen based on performance?	Yes	None	None
	Are costly operations minimized in loops?	Mostly	Some file operations inside loops in processReturn could be optimized for better performance.	Optimize file operations by reducing redundant processing within loops, or by using Map for quicker lookups.
	Is lazy initialization used?	Yes	None	None
Memory Management	Are unnecessary object references set to null?	No	No explicit resource cleanup for file handling (e.g., buffered reader) or unused objects.	Use try-with-resources for file handling and set references to null after use where applicable.
Security	Is user input validated?	No	User input like transactionId and returnAmount are not	Implement validation for transactionId, returnAmount, and other user inputs to prevent invalid entries.

			validated properly.	
Maintainability	Are there long methods or deeply nested loops?	Yes	Methods like processReturn contain long sections with nested loops for processing returns and inventory.	Refactor processReturn to break down complex operations into smaller, maintainable methods.
	Is there duplicated code?	Yes	Code for updating inventory and processing returns is repeated in multiple places.	Extract common logic into reusable methods (e.g., updateInventory method).
	Are there any magic numbers?	Yes	Hardcoded values like 50 (maximum number of items) are present.	Define constants for such values (e.g., MAX_RETURN_ITEMS = 50).

Test Related Categories

Category	Checklist Item	Yes/No	Issue	Fix
Test Coverage	Are unit tests provided for all public methods and critical functionalities?	Yes	None	None
	Do unit tests cover edge cases and boundary values?	Yes	Edge cases like invalid transactionId or returnAmount are tested.	None
Test Design	Are tests written following the AAA pattern?	Yes	None	None

	Are individual test cases independent?	Yes	None	None
	Are descriptive names used for test methods?	Yes	None	None
Assertions	Are assertions used to verify expected results?	Yes	None	None
	Are specific assertions used instead of general ones?	Yes	None	None
Boundary and Edge Cases	Are edge cases and boundary conditions tested?	Yes	None	None
	Are invalid inputs covered by tests?	Mostly	Edge cases for invalid transactionId and returnAmount should be more thoroughly tested.	Add more tests to handle invalid inputs such as malformed transactionId and negative return amounts.
Mocking and Stubbing	Are mocks or stubs used to isolate the unit under test?	Not Applicable	None	None
Performance Testing	Are tests to check performance for critical methods provided?	No	No specific performance tests for methods like processReturn or	Add performance tests to check for efficiency, especially with large

			updateInventory.	transaction volumes.
Test Maintainability	Are test methods organized and modular?	Yes	None	None
	Is there a setup method for initializing common objects?	Yes	None	None

Housekeeping

Category	Checklist Item	Yes/No	Issue	Fix
Code Smells	Are there any code smells not covered by the checklist?	Yes	- Repeated code for processing returns and updating inventory.	Extract repeated logic into common methods for better maintainability.
			- Long method processReturn with multiple responsibilities.	Refactor processReturn into smaller, focused methods.
Coding Standards	Are there any coding standard violations not covered by the checklist?	Yes	- No consistent exception handling strategy, some exceptions are caught but not properly logged.	Use a logging framework (e.g., Logger) for better exception handling.
Performance Inefficiencies	Are there any performance inefficiencies not covered by the checklist?	Yes	- Nested loops and file handling operations in processReturn can be inefficient for large datasets.	Use optimized data structures and file handling strategies for large datasets.

				- File operations are repeated across methods, leading to inefficiency.	Consolidate file operations and reuse helper methods for file reading/writing.

Refactoring Task 1: Extract Inventory Update Logic into a Separate Method

- **Issue**: The logic for updating inventory is currently embedded within multiple methods like `processReturn` and `updateInventory`. This redundancy increases code complexity and makes the class harder to maintain.
- **Refactoring Plan**: Extract the inventory update logic into a dedicated helper method called `updateInventoryStock`. This method will take parameters like the item ID and return amount and handle all inventory-related tasks in one place.
    - `updateInventoryStock`: Handles all inventory updates by checking the return item and updating the stock count.
- **Benefit**: Centralizes inventory handling, reducing duplication, and improves the maintainability of the code. Also enhances readability by isolating inventory management into its own function.

Refactoring Task 2: Modularize the `processReturn` Method

- **Issue**: The `processReturn` method currently handles multiple tasks, such as refund calculation, inventory update, and file handling. This makes the method long and difficult to test.
- **Refactoring Plan**: Break down `processReturn` into smaller, single-responsibility helper methods:
    - `calculateRefund`: Handles refund calculation based on return conditions.
    - `logReturnTransaction`: Logs the return transaction.
    - `applyReturnToInventory`: Updates the inventory based on the returned item.
    - `writeReturnToFile`: Writes the updated return data to the appropriate file.
- **Benefit**: Makes each method more focused and easier to understand. Reduces the size and complexity of `processReturn` and improves testability.

Refactoring Task 3: Replace Magic Numbers with Constants

- **Issue**: The class contains hardcoded numeric values, such as maximum allowable return amounts or limits on the number of items in a return. These "magic numbers" can be unclear and hard to update.
- **Refactoring Plan**: Replace all hardcoded numeric values with named constants. For example:
  - `MAX_RETURN_ITEMS`: Defines the maximum number of items that can be returned in a single transaction.
  - `RETURN_FEE_PERCENTAGE`: Defines the percentage fee for returns.
- **Benefit**: Increases code readability and makes it easier to modify limits in the future without searching through the entire code. It also improves maintainability by giving meaningful names to these numbers.

1. Code Smells

Code Smell	Description	Example	Suggested Solution
Long Methods	Methods that are too long and perform multiple tasks, making them hard to understand and maintain.	The `processReturn` method is too long and handles multiple tasks such as refund calculation and inventory updates.	Break the method into smaller, focused methods (e.g., `calculateRefund`, `logReturnTransaction`, `applyReturnToInventory`).
Nested Loops	Deeply nested loops increase complexity and reduce readability.	Loops within loops for processing returns and updating inventory.	Refactor using helper methods or consider flattening logic.
Duplicate Code	Code blocks that appear multiple times in the project, increasing the risk of errors and maintenance.	The logic for inventory updates is repeated in multiple methods like `processReturn` and `updateInventory`.	Extract common logic into a reusable helper method (e.g., `updateInventoryStock`).
Data Clumps	Groups of variables that frequently appear together in methods.	`transactionId`, `returnAmount` passed together in multiple methods.	Use an object (e.g., `ReturnTransaction`) to encapsulate these variables.
Primitive Obsession	Overuse of primitive types instead of more	Using a string to represent the file path instead of an	Replace with a more descriptive type (e.g., an enum for file paths).

	descriptive classes.	enum or a dedicated class.	

---

2. Violations of Coding Standards

Issue	Description	Example	Suggested Fix
Naming Conventions	Inconsistent or unclear naming of variables, methods, or classes.	`tempData` instead of a more descriptive name like `returnData`.	Use meaningful, camelCase for variables and PascalCase for classes.
Lack of Comments	Insufficient documentation, making the code difficult to understand.	Complex logic in `processReturn` without comments explaining the refund calculation or file handling.	Add meaningful comments to explain the purpose of methods and complex logic.
Inconsistent Formatting	Irregular indentation, spacing, or line breaks reducing readability.	Mixed use of spaces and tabs for indentation.	Follow consistent formatting (e.g., 4-space indentation).
Magic Numbers	Use of hard-coded values without explanation.	`if (returnItems > 50)` instead of using a named constant.	Replace with named constants (e.g., `MAX_RETURN_ITEMS`).
Method Length Exceeds Limit	Methods longer than 20-25 lines.	`processReturn` method with over 50 lines.	Split into smaller helper methods for readability and maintainability.

---

3. Performance Inefficiencies

Issue	Description	Example	Suggested Solution
Unnecessary Object Creation	Creating new objects repeatedly instead of reusing existing instances.	Creating a new `BufferedReader` object in every loop iteration for file processing.	Create the object once and reuse it across multiple iterations.
Inefficient Data Structures	Using data structures that are not optimal for the use case.	Using an `ArrayList` to perform frequent lookups.	Use a `HashMap` or `Set` for faster lookups and more efficient access.
Redundant Calculations	Performing the same calculation multiple	Recalculating the refund percentage for	Calculate once and store the result in a variable.

	times in a loop or method.	every return item inside a loop.	
Excessive Logging	Logging too much information, especially in production environments.	Logging detailed return transaction data for every item.	Limit logging or adjust log levels for production environments.
Inefficient String Operations	Using string concatenation in loops instead of `StringBuilder`.	`result += "Item: " + itemName` inside a loop.	Use `StringBuilder` for concatenating strings inside loops.

8. Java Code Review Checklist for POSSystem

File name	POSSystem.java
class/interface name	POSSystem

Category	Checklist Item	Yes/No	Issue	Fix
Naming Conventions	Are class names written in PascalCase?	Yes	None	None
	Are variable and method names written in camelCase?	Yes	Some variables like tempData could be more descriptive.	Rename tempData to posData for clarity.
	Are constants written in uppercase with underscores?	No	Constants like TEMP_FILE, POS_FILE are hardcoded.	Define constants for file paths and string literals used multiple times (e.g., POS_FILE_PATH).
Code Structure	Are access modifiers used correctly?	Mostly	Some fields (e.g., transactionList, orderId) could be	Make transactionList and orderId private and add appropriate

			private for better encapsulation.	getters/setters if necessary.
	Are classes and interfaces separated?	Yes	None	None
	Are packages used appropriately?	No	No package specified, reducing project organization.	Add package declaration (e.g., com.pos.system) and organize project files accordingly.
Method Design	Do methods have a single responsibility?	Mostly	Methods like processOrder handle multiple tasks such as calculating total, updating inventory, and printing receipt.	Refactor processOrder to break down tasks into smaller, single-responsibility methods.
	Are method parameters limited?	Yes	None	None
	Is method overloading used properly?	Not Applicable	None	None
Exception Handling	Are exceptions handled with try-catch blocks?	Yes	Error messages in catch blocks are printed instead of logged.	Replace System.out.println with a logging framework (e.g., Logger) to handle exceptions more effectively.

	Are specific exceptions used?	Yes	None	None
Code Readability	Are comments added for complex logic?	No	Some methods like processOrder and updateInventory lack comments explaining complex logic.	Add comments to explain logic for calculation, file handling, and inventory updates.
	Is indentation consistent?	Yes	None	None
	Are blank lines used to separate code blocks?	Yes	None	None
	Are meaningful names used for variables, classes, and methods?	Mostly	Variables like tempData and orderId are not fully descriptive of their purpose.	Rename variables like tempData to posData and orderId to orderTransactionId for better clarity.
Performance	Are data structures chosen based on performance?	Yes	None	None
	Are costly operations minimized in loops?	Mostly	Some file operations inside loops in processOrder could be optimized for better performance.	Optimize file operations by reducing redundant processing within loops, or by using Map for quicker lookups.

	Is lazy initialization used?	Yes	None	None
Memory Management	Are unnecessary object references set to null?	No	No explicit resource cleanup for file handling (e.g., buffered reader) or unused objects.	Use try-with-resources for file handling and set references to null after use where applicable.
Security	Is user input validated?	No	User input like orderId and paymentAmount are not validated properly.	Implement validation for orderId, paymentAmount, and other user inputs to prevent invalid entries.
Maintainability	Are there long methods or deeply nested loops?	Yes	Methods like processOrder contain long sections with nested loops for processing orders and updating inventory.	Refactor processOrder to break down complex operations into smaller, maintainable methods.
	Is there duplicated code?	Yes	Code for updating inventory and processing orders is repeated in multiple places.	Extract common logic into reusable methods (e.g., updateInventory method).
	Are there any magic numbers?	Yes	Hardcoded values like 50 (maximum number of items) are present.	Define constants for such values (e.g., MAX_ORDER_ITEMS = 50).

Test Related Categories				
Category	Checklist Item	Yes/No	Issue	Fix
Test Coverage	Are unit tests provided for all public methods and critical functionalities?	Yes	None	None
	Do unit tests cover edge cases and boundary values?	Yes	Edge cases like invalid orderId or paymentAmount are tested.	None
Test Design	Are tests written following the AAA pattern?	Yes	None	None
	Are individual test cases independent?	Yes	None	None
	Are descriptive names used for test methods?	Yes	None	None
Assertions	Are assertions used to verify expected results?	Yes	None	None
	Are specific assertions used instead of general ones?	Yes	None	None
Boundary and Edge Cases	Are edge cases and boundary conditions tested?	Yes	None	None
	Are invalid inputs covered by tests?	Mostly	Edge cases for invalid orderId and paymentAmount should be more	Add more tests to handle invalid inputs such as

			thoroughly tested.	malformed orderId and negative payment amounts.
Mocking and Stubbing	Are mocks or stubs used to isolate the unit under test?	Not Applicable	None	None
Performance Testing	Are tests to check performance for critical methods provided?	No	No specific performance tests for methods like processOrder or updateInventory.	Add performance tests to check for efficiency, especially with large order volumes.
Test Maintainability	Are test methods organized and modular?	Yes	None	None
	Is there a setup method for initializing common objects?	Yes	None	None

Housekeeping

Category	Checklist Item	Yes/No	Issue	Fix
Code Smells	Are there any code smells not covered by the checklist?	Yes	- Repeated code for processing orders and updating inventory.	Extract repeated logic into common methods for better maintainability.
			- Long method processOrder with multiple	Refactor processOrder into smaller,

			responsibilities.	focused methods.
Coding Standards	Are there any coding standard violations not covered by the checklist?	Yes	- No consistent exception handling strategy, some exceptions are caught but not properly logged.	Use a logging framework (e.g., Logger) for better exception handling.
Performance Inefficiencies	Are there any performance inefficiencies not covered by the checklist?	Yes	- Nested loops and file handling operations in processOrder can be inefficient for large datasets.	Use optimized data structures and file handling strategies for large datasets.
			- File operations are repeated across methods, leading to inefficiency.	Consolidate file operations and reuse helper methods for file reading/writing.

Refactoring Task 1: Extract Payment Processing Logic into a Separate Method

- **Issue**: The logic for processing payments (handling credit card, PayPal, etc.) is currently embedded within multiple methods, leading to redundancy and making the code harder to maintain.
- **Refactoring Plan**: Extract the payment processing logic into a dedicated method called `processPayment`. This method will take parameters like payment method type and amount and handle all payment-related tasks.
    - `processPayment`: This method will check the payment method type (credit card, PayPal, etc.) and apply the necessary steps for processing the payment.
- **Benefit**: Centralizes payment processing, reducing duplication and improving maintainability. It also enhances readability by isolating payment logic into its own method, making the code easier to follow.

Refactoring Task 2: Modularize the `handleOrder` Method

- **Issue**: The `handleOrder` method is currently responsible for multiple tasks, such as checking order details, calculating totals, updating inventory, and processing payments. This makes the method large and difficult to test or maintain.
- **Refactoring Plan**: Break down the `handleOrder` method into smaller, focused helper methods:
    - `validateOrderDetails`: Validates the order details, such as item availability and customer information.
    - `calculateTotal`: Computes the total cost of the order, including any discounts or taxes.
    - `updateInventory`: Adjusts inventory levels based on the order.
    - `processPayment`: Handles the payment process.
    - `sendOrderConfirmation`: Sends an order confirmation to the customer.
- **Benefit**: By breaking down `handleOrder` into smaller methods, the code becomes more modular, easier to understand, and easier to test. Each method will have a single responsibility, improving maintainability.

Refactoring Task 3: Replace Hardcoded Discount Logic with Configurable Constants

- **Issue**: The class contains hardcoded values for discount percentages, such as for promotions or loyalty rewards. These "magic numbers" can make the code unclear and harder to modify if discount rules change.
- **Refactoring Plan**: Replace the hardcoded discount logic with named constants or a configuration class that stores discount rules.
    - `LOYALTY_DISCOUNT`: Defines the loyalty discount percentage.
    - `PROMO_CODE_DISCOUNT`: Defines the discount percentage for valid promotional codes.
    - `MAX_DISCOUNT_AMOUNT`: Defines the maximum discount that can be applied to any order.
- **Benefit**: Improves code readability and maintainability by making the discount logic more transparent and easier to modify. By using constants, the code becomes more flexible when changing discount rules in the future.

1. Code Smells

Code Smell	Description	Example	Suggested Solution
Long Methods	Methods that are too long and perform multiple	processOrder handles calculating totals, updating	Refactor processOrder into smaller methods like calculateTotal, updateInventory, printReceipt.

		tasks, making them hard to understand and maintain.	inventory, and printing receipts in a single method.	
	Duplicate Code	Code blocks that appear multiple times, increasing the risk of errors and maintenance.	Payment processing logic appears in multiple places.	Extract payment logic into a reusable method like processPayment.
	Primitive Obsession	Overuse of primitive types instead of more descriptive classes.	Hardcoded values for discount percentages (e.g., 10%, 20%).	Replace with named constants like LOYALTY_DISCOUNT, PROMO_CODE_DISCOUNT.
	Data Clumps	Groups of variables that frequently appear together in methods.	Passing customer details (name, address, payment info) as separate arguments in multiple methods.	Use a Customer object to encapsulate these details.

---

2. Violations of Coding Standards

Issue	Description	Example	Suggested Fix
Naming Conventions	Inconsistent or unclear naming of	Variables like tempData	Rename tempData to

	variables, methods, or classes.	could be more descriptive.	posData for clarity.
Lack of Comments	Insufficient documentation, making the code difficult to understand.	Methods like processOrder and updateInventory lack comments explaining complex logic.	Add comments to explain the purpose and functionality of each method, especially for complex logic.
Magic Numbers	Use of hard-coded values without explanation.	50 for the maximum number of items in an order.	Define constants like MAX_ORDER_ITEMS = 50 for clarity.
Inconsistent Formatting	Irregular indentation, spacing, or line breaks reducing readability.	Inconsistent use of tabs and spaces for indentation.	Follow consistent formatting (e.g., 4-space indentation).

---

3. Performance Inefficiencies

Issue	Description	Example	Suggested Solution
Unnecessary Object Creation	Creating new objects repeatedly instead of reusing existing instances.	Creating a new Customer object inside a loop unnecessarily.	Create the Customer object once and reuse it where possible.
Redundant Calculations	Performing the same calculation multiple times in a loop or method.	Calculating the total price repeatedly inside a loop.	Store the result in a variable and reuse it instead of recalculating.
Inefficient Data Structures	Using data structures that are not	Using an ArrayList for frequent	Use a HashMap or Set for

	optimal for the use case.	product searches.	faster lookups.
Excessive Logging	Logging too much information, especially in production environments.	Logging detailed order information on every item update.	Limit logging or adjust log levels to avoid excessive logging in production.

Additional Guidelines Based on IEEE Standards

Modularity:

1. Each class should represent a single concept, and each method should perform a single function.
2. Avoid monolithic classes; split functionality into smaller, reusable classes.

Adherence to Java Best Practices:

1. Use final for constants and variables that are not supposed to change.
2. Favor interfaces over abstract classes when defining contracts.
3. Prefer for-each loops over traditional for loops for better readability and performance with collections.

9. Java Code Review Checklist for Return Item

File name	ReturnItem.java
class/interface name	ReturnItem

Category	Checklist Item	Yes/No	Issue	Fix
Naming Conventions	Are class names written in PascalCase?	Yes	None	None
	Are variable and method names written in camelCase?	Yes	None	None

	Are constants written in uppercase with underscores?	No	No constants are used in the class.	Consider adding constants for commonly used values or error codes if needed.
Code Structure	Are access modifiers used correctly?	Yes	None	None
	Are classes and interfaces separated?	Yes	None	None
	Are packages used appropriately?	Yes	No package declaration in the provided code snippet.	Add package declaration if part of a larger project.
Method Design	Do methods have a single responsibility?	Yes	getItemID() and getDays() follow single-responsibility principle.	None
	Are method parameters limited?	Yes	None	None
	Is method overloading used properly?	Not Applicable	No method overloading present in this class.	None
Exception Handling	Are exceptions handled with try-catch blocks?	No	No exception handling is used in the class.	Add exception handling to handle potential runtime errors (e.g.,

				invalid input).
	Are specific exceptions used?	No	No exceptions are defined in this class.	Consider adding custom exceptions for invalid data or edge cases.
Code Readability	Are comments added for complex logic?	No	No comments are provided in the class.	Add comments to explain the constructor and getter methods for clarity.
	Is indentation consistent?	Yes	None	None
	Are blank lines used to separate code blocks?	Yes	None	None
	Are meaningful names used for variables, classes, and methods?	Yes	Variable and method names are clear and descriptive.	None
Performance	Are data structures chosen based on performance?	Yes	None	None
	Are costly operations minimized in loops?	Not Applicable	No loops or costly operations present in this simple class.	None

	Is lazy initialization used?	Not Applicable	No lazy initialization in this class.	None
Memory Management	Are unnecessary object references set to null?	No	No explicit resource cleanup is required in this class.	Not applicable as there are no resources needing cleanup in this simple class.
Security	Is user input validated?	No	This class does not handle user input.	Not applicable to this class, as it doesn't handle input.
Maintainability	Are there long methods or deeply nested loops?	No	The class has no long methods or nested loops.	None
	Is there duplicated code?	No	No duplication of code in the class.	None
	Are there any magic numbers?	No	No hardcoded values other than the constructor inputs.	None

Test Related Categories

Category	Checklist Item	Yes/ No	Issue	Fix
Test Coverage	Are unit tests provided for all public methods and critical	Yes	Tests cover the constructor and getter methods.	None

	functionalities?			
	Do unit tests cover edge cases and boundary values?	Yes	Tests check for normal values but may need to cover edge cases like 0 for itemID or negative values for daysSinceReturn.	Add edge cases for values like 0 or negative days since return.
Test Design	Are tests written following the AAA pattern?	Yes	The tests follow the AAA (Arrange, Act, Assert) pattern.	None
	Are individual test cases independent?	Yes	Each test is independent and does not affect others.	None
	Are descriptive names used for test methods?	Yes	Test names are descriptive (e.g., testReturnItemConstructor).	None
Assertions	Are assertions used to verify expected results?	Yes	Assertions are used in all tests to compare expected and actual values.	None
	Are specific assertions used instead of general ones?	Yes	Specific assertions like assertEquals are used to compare exact values.	None
Boundary and Edge Cases	Are edge cases and boundary conditions tested?	Yes	Edge cases like 0 for daysSinceReturn or negative values are not tested.	Add tests for boundary cases such as 0 and negative daysSinceReturn.
	Are invalid inputs	No	No tests for invalid inputs like negative	Add tests for invalid inputs, such

	covered by tests?		itemID or daysSinceReturn.	as negative values for itemID and daysSinceReturn.
Mocking and Stubbing	Are mocks or stubs used to isolate the unit under test?	No	No mocks or stubs are needed in this simple class.	Not applicable.
Performance Testing	Are tests to check performance for critical methods provided?	No	No performance tests are needed in this class as it’s simple.	Not applicable.
Test Maintainability	Are test methods organized and modular?	Yes	The test methods are modular and organized.	None
	Is there a setup method for initializing common objects?	Yes	Common objects are initialized in each test method.	None

Housekeeping

Category	Checklist Item	Yes/No	Issue	Fix
Code Smells	Are there any code smells not covered by the checklist?	No	None	None
Coding Standards	Are there any coding standard violations not covered by the checklist?	No	None	None
Performance Inefficiencies	Are there any performance inefficiencies not covered by the checklist?	No	The class is too simple to have performance inefficiencies.	None

Refactoring Task 1: Extract Payment Calculation Logic into a Separate Method

- **Issue**: The logic for calculating the total payment, including any discounts, fees, and taxes, is currently embedded within methods like `processOrder` and `applyDiscount`. This results in redundant calculations and increases the complexity of the code.
- **Refactoring Plan**: Extract the payment calculation logic into a dedicated helper method called `calculateTotalAmount`. This method will handle the calculation of the total payment, taking into account any applicable discounts, fees, and taxes.
    - `calculateTotalAmount`: Computes the final amount by considering order total, discount, and additional fees like processing fees.
- **Benefit**: Centralizes payment calculations, reducing redundancy and improving maintainability. Makes the code more modular and easier to test.

Refactoring Task 2: Separate Input Validation from Core Logic

- **Issue**: The class currently combines input validation logic with the core functionality (e.g., processing orders, calculating refunds). This creates tightly coupled logic that is hard to maintain and test.
- **Refactoring Plan**: Move all input validation checks into a separate method, `validateInput`. This method will verify the validity of user inputs, such as order IDs, payment amounts, and other parameters, before processing further.
    - `validateInput`: Validates input parameters like order ID, payment amount, etc., and throws exceptions if the input is invalid.
- **Benefit**: Improves separation of concerns, making the class more modular. It also enhances testability by isolating input validation from the core logic and helps identify issues more easily.

Refactoring Task 3: Consolidate File Handling Logic into a Single Method

- **Issue**: The file handling logic (e.g., reading and writing to files) is scattered throughout multiple methods, leading to code duplication and increased potential for errors when making changes to file handling.
- **Refactoring Plan**: Create a dedicated file handling method, `handleFileOperation`, that can be reused across the class. This method will accept parameters for the file operation type (read/write), file path, and data to be written (if applicable).
    - `handleFileOperation`: Handles file reading and writing operations, streamlining file handling throughout the class.
- **Benefit**: Reduces redundancy, centralizes file handling, and makes the class easier to maintain. Updates to file handling logic can be made in a single location, ensuring consistency.

Code Smells

Category	Checklist Item	Yes/No	Issue	Fix
Code Duplication	Is there any duplication of code?	No	No duplicated code found.	None
Long Methods	Are there any methods that are too long or complex?	No	All methods are short and simple.	None
Large Classes	Are there any classes that are too large?	No	Class is concise and follows the single responsibility principle.	None
God Class	Is there a class that is responsible for too many tasks?	No	No class violates the single responsibility principle.	None
Hidden Dependencies	Are there any hidden dependencies between classes?	No	No hidden dependencies are present in the class.	None
Duplicated Logic	Is there duplicated logic within methods or classes?	No	No logic duplication present.	None

---

Performance Inefficiencies

Category	Checklist Item	Yes/No	Issue	Fix
Inefficient Loops	Are costly operations minimized in loops?	No	No loops or costly operations are present in this class.	None
Unnecessary Object Creation	Are there any unnecessary objects being created?	No	No unnecessary objects are created in the class.	None
Memory Leaks	Are unnecessary object references set to null?	No	No memory leaks or unneeded references are present.	Not applicable in this simple class.
Inefficient Data Structures	Are data structures chosen based on performance?	Yes	Proper data structures have been selected.	None
Heavy Recursion	Are recursive methods being used inappropriately?	No	No recursion present in this class.	None

---

Violations of Coding Standards

Category	Checklist Item	Yes/No	Issue	Fix

Class Naming Conventions	Are class names written in PascalCase?	Yes	Class name is in PascalCase.	None
Method/Variable Naming Conventions	Are method and variable names written in camelCase?	Yes	Methods and variables are in camelCase.	None
Constants Naming Convention	Are constants written in uppercase with underscores?	No	No constants are used in this class.	Consider using constants for commonly used values.
Proper Use of Access Modifiers	Are access modifiers used correctly?	Yes	Access modifiers have been used correctly.	None
Indentation and Spacing	Is indentation and spacing consistent?	Yes	Indentation is consistent throughout the class.	None
Method Length	Are methods kept short and to the point?	Yes	Methods are short and follow single responsibility principle.	None
No Magic Numbers	Are there any magic numbers used in the code?	No	No hardcoded values other than constructor inputs.	None
Package Declaration	Is a package declaration used in the code?	No	No package declaration is present in the snippet provided.	Add package declaration if part of a larger project.

10. Java Code Review Checklist for Regitser

File name	Register.java
class/interface name	Register

Category	Checklist Item	Yes/No	Issue	Fix
Naming Conventions	Are class names written	Yes	None	None

	in PascalCase?			
	Are variable and method names written in camelCase?	Yes	None	None
	Are constants written in uppercase with underscores?	No	No constants are used in the class.	Consider adding constants for commonly used values if needed.
Code Structure	Are access modifiers used correctly?	Yes	None	None
	Are classes and interfaces separated?	Yes	None	None
	Are packages used appropriately ?	Yes	No package declaration in the provided code snippet.	Add package declaration if part of a larger project.
Method Design	Do methods have a single responsibilit y?	Yes	main() method has a single responsibili ty of running the application.	None
	Are method parameters limited?	Yes	None	None
	Is method overloading used properly?	Not Applicab le	No method overloading present in this class.	None
Exception Handling	Are exceptions handled with	No	No exception handling is used in the class.	Add exception handling to handle potential

	try-catch blocks?			runtime errors, like GUI initialization failures.
	Are specific exceptions used?	No	No exceptions are defined in this class.	Consider adding custom exceptions if any specific error conditions are expected during GUI initialization.
Code Readability	Are comments added for complex logic?	No	No comments are provided in the class.	Add comments to explain the main() method functionality and how the GUI is initialized.
	Is indentation consistent?	Yes	None	None
	Are blank lines used to separate code blocks?	Yes	None	None
	Are meaningful names used for variables, classes, and methods?	Yes	Variable and method names are clear and descriptive.	None
Performance	Are data structures chosen based on performance?	Not Applicable	No complex data structures used.	None

	Are costly operations minimized in loops?	Not Applicable	No loops or costly operations present in this simple class.	None
	Is lazy initialization used?	Not Applicable	No lazy initialization in this class.	None
Memory Management	Are unnecessary object references set to null?	No	Not applicable as no resources requiring cleanup are in the class.	Not applicable.
Security	Is user input validated?	Not Applicable	This class does not handle user input.	Not applicable to this class, as it doesn't handle input.
Maintainability	Are there long methods or deeply nested loops?	No	The class has no long methods or nested loops.	None
	Is there duplicated code?	No	No duplication of code in the class.	None
	Are there any magic numbers?	No	No magic numbers present in the class.	None

Test Related Categories

Category	Checklist Item	Yes/ No	Issue	Fix
Test Coverage	Are unit tests provided for all public	Yes	The test covers the main() method and verifies the creation of the Login_Interface window.	None

	methods and critical functionalities?			
	Do unit tests cover edge cases and boundary values?	No	The test case doesn't seem to cover edge cases like handling invalid UI behavior or failure to launch GUI.	Add edge cases to test GUI initialization failures or unexpected behavior.
Test Design	Are tests written following the AAA pattern?	Yes	The test follows the AAA (Arrange, Act, Assert) pattern.	None
	Are individual test cases independent?	Yes	Each test is independent and does not affect others.	None
	Are descriptive names used for test methods?	Yes	Test names are descriptive (e.g., testMainMethodCreatesLoginInterface).	None
Assertions	Are assertions used to verify expected results?	Yes	Assertions are used in the test to verify the expected visibility of the login frame and its close operation.	None
	Are specific assertions used instead of general ones?	Yes	Specific assertions like assertEquals and assertTrue are used to verify the expected results.	None
Boundary and Edge Cases	Are edge cases and	No	No edge cases or boundary tests are included.	Add tests for edge

	boundary conditions tested?			cases, such as handling UI failures or invalid initialization conditions.
	Are invalid inputs covered by tests?	No	The test doesn't cover invalid inputs or exceptions in GUI creation.	Add tests for invalid inputs or errors during GUI creation.
Mocking and Stubbing	Are mocks or stubs used to isolate the unit under test?	No	No mocks or stubs are needed for this test, as it tests the creation of a GUI.	Not applicable.
Performance Testing	Are tests to check performance for critical methods provided?	No	Performance testing isn't required for this simple GUI initialization test.	Not applicable.
Test Maintainability	Are test methods organized and modular?	Yes	The test methods are organized and modular.	None
	Is there a setup method for initializing common objects?	No	There's no common setup method, but each test initializes objects locally.	Consider adding a setup method if common initialization is

				needed for future tests.

Housekeeping

Category	Checklist Item	Yes/No	Issue	Fix
Code Smells	Are there any code smells not covered by the checklist?	No	No code smells identified.	None
Coding Standards	Are there any coding standard violations not covered by the checklist?	No	None	None
Performance Inefficiencies	Are there any performance inefficiencies not covered by the checklist?	No	The class is too simple to have performance inefficiencies.	None

Refactoring Task 1: Extract Payment Calculation Logic into a Separate Method

- **Issue**: The payment calculation logic (e.g., determining discounts, fees, and taxes) is currently embedded in multiple methods, such as `processOrder` and `applyDiscount`, leading to redundancy and complexity.
- **Refactoring Plan**: Extract the payment calculation logic into a dedicated method called `calculateTotalAmount`. This method will compute the total amount, considering the order total, discounts, processing fees, and taxes.
    - `calculateTotalAmount`: Computes the final amount by incorporating all factors like the base order amount, discounts, taxes, and fees.
- **Benefit**: Centralizes the payment logic, eliminating redundancy, and improving maintainability. Makes the code more modular and easier to test.

Refactoring Task 2: Move Input Validation to a Separate Method

- **Issue**: Input validation is currently mixed with the core business logic, which makes the code harder to maintain and test, especially when validation checks need to be updated or modified.
- **Refactoring Plan**: Create a separate method, `validateInput`, that will handle the validation of inputs like payment amount, order ID, and other user inputs. It will throw specific exceptions for invalid input.
    - `validateInput`: Validates all critical user inputs and throws clear exceptions when input is invalid.

- **Benefit**: Improves code modularity by separating concerns. Makes the code easier to maintain and enhances testability by isolating validation logic.

Refactoring Task 3: Consolidate File Handling Logic

- **Issue**: File reading and writing operations are repeated in several methods, leading to duplicated code and an increased risk of errors when modifying file handling behavior.
- **Refactoring Plan**: Create a dedicated method, `handleFileOperation`, to centralize all file operations (reading and writing). This method will take parameters such as the file path, operation type (read/write), and data to be written (if applicable).
    - `handleFileOperation`: A unified method to handle file operations, making it easier to modify and maintain file-handling logic in one place.
- **Benefit**: Reduces code duplication and centralizes file management, making the class easier to maintain and ensuring consistency in file operations across the application.

Code Smells

Category	Checklist Item	Yes/No	Issue	Fix
Naming Conventions	Are class names written in PascalCase?	Yes	None	None
	Are variable and method names written in camelCase?	Yes	None	None
	Are constants written in uppercase with underscores?	No	No constants used in the class.	Consider adding constants if needed.
Code Structure	Are access modifiers used correctly?	Yes	None	None
	Are classes and interfaces separated?	Yes	None	None
	Are packages used appropriately?	Yes	No package declaration in the code snippet.	Add package declaration if part of a larger project.
Method Design	Do methods have a single responsibility?	Yes	The `main()` method has a	None

			single responsibility.	
	Are method parameters limited?	Yes	None	None
	Is method overloading used properly?	No	No method overloading.	None
Exception Handling	Are exceptions handled with try-catch blocks?	No	No exception handling used.	Add exception handling for GUI initialization errors.
	Are specific exceptions used?	No	No specific exceptions defined.	Add custom exceptions for specific error conditions.
Code Readability	Are comments added for complex logic?	No	No comments in the class.	Add comments for `main()` and GUI logic.
	Is indentation consistent?	Yes	None	None
	Are blank lines used to separate code blocks?	Yes	None	None
	Are meaningful names used for variables, classes, and methods?	Yes	Names are clear and descriptive.	None

---

Performance Inefficiencies

Category	Checklist Item	Yes/No	Issue	Fix
Data Structures	Are data structures chosen based on performance?	No	No complex data structures used.	None
Costly Operations	Are costly operations minimized in loops?	No	No loops or costly operations present.	None
Lazy Initialization	Is lazy initialization used?	No	No lazy initialization in the class.	None

---

Violations of Coding Standards

Category	Checklist Item	Yes/No	Issue	Fix
Magic Numbers	Are there any magic numbers?	No	No magic numbers present.	None
Long Methods	Are there long methods or	No	No long methods or nested loops.	None

	deeply nested loops?			
Duplicated Code	Is there duplicated code?	No	No code duplication.	None
Improper Naming Conventions	Are naming conventions followed?	No	Constants are missing uppercase naming convention.	Introduce constants if necessary with uppercase naming.

11. Java Code Review Checklist for Add_Employee interface

File name	Add_Employee interface.java
class/interface name	Add_Employee interface

Category	Checklist Item	Yes/No	Issue	Fix
Naming Conventions	Are class names written in PascalCase?	Yes	None	None
	Are variable and method names written in camelCase?	Yes	None	None
	Are constants written in uppercase with underscores?	No	No constants are used in the class.	Consider adding constants for commonly used values if needed.
Code Structure	Are access modifiers used correctly?	Yes	None	None
	Are classes and interfaces separated?	Yes	None	None

	Are packages used appropriately?	Yes	No package declaration in the provided code snippet.	Add package declaration if part of a larger project.
Method Design	Do methods have a single responsibility?	Yes	actionPerformed() method has the single responsibility of handling button actions.	None
	Are method parameters limited?	Yes	None	None
	Is method overloading used properly?	Not Applicable	No method overloading present in this class.	None
Exception Handling	Are exceptions handled with try-catch blocks?	No	No exception handling is used in the class.	Add exception handling to handle potential runtime errors during button actions or GUI initialization.
	Are specific exceptions used?	No	No exceptions are defined in this class.	Consider adding custom exceptions if any specific error conditions are expected.
Code Readability	Are comments added for complex logic?	No	No comments are provided in the class.	Add comments to explain the logic for button actions and GUI setup.

	Is indentation consistent?	Yes	None	None
	Are blank lines used to separate code blocks?	Yes	None	None
	Are meaningful names used for variables, classes, and methods?	Yes	Variable and method names are clear and descriptive.	None
Performance	Are data structures chosen based on performance?	Not Applicable	No complex data structures used.	None
	Are costly operations minimized in loops?	Not Applicable	No loops or costly operations present in this simple class.	None
	Is lazy initialization used?	Not Applicable	No lazy initialization in this class.	None
Memory Management	Are unnecessary object references set to null?	No	Not applicable as no resources requiring cleanup are in the class.	Not applicable.
Security	Is user input validated?	No	Input validation is not performed on the name or password fields.	Add input validation to ensure that name and password fields are not empty or invalid.
Maintainability	Are there long methods or deeply nested loops?	No	The class has no long methods or deeply nested loops.	None

	Is there duplicated code?	No	No duplication of code in the class.	None
	Are there any magic numbers?	No	No magic numbers present in the class.	None

Test Related Categories

Category	Checklist Item	Yes/No	Issue	Fix
Test Coverage	Are unit tests provided for all public methods and critical functionalities?	No	N/A	N/A
	Do unit tests cover edge cases and boundary values?	No	N/A	N/A
Test Design	Are tests written following the AAA pattern?	Not Applicable	N/A	N/A
	Are individual test cases independent?	Not Applicable	N/A	N/A
	Are descriptive names used for test methods?	Not Applicable	N/A	N/A
Assertions	Are assertions used to verify expected results?	Not Applicable	N/A	N/A
	Are specific assertions used instead of general ones?	Not Applicable	N/A	N/A
Boundary and Edge Cases	Are edge cases and boundary conditions tested?	No	N/A	N/A
	Are invalid inputs covered by tests?	No	N/A	N/A
Mocking and Stubbing	Are mocks or stubs used to isolate the unit under test?	No	N/A	N/A
Performance Testing	Are tests to check performance for critical methods provided?	No	N/A	N/A
Test Maintainability	Are test methods organized and modular?	Not Applicable	N/A	N/A
	Is there a setup method for initializing common objects?	Not Applicable	N/A	N/A

Housekeeping

Category	Checklist Item	Yes/No	Issue	Fix
Code Smells	Are there any code smells not	No	No code smells identified.	None

	covered by the checklist?			
Coding Standards	Are there any coding standard violations not covered by the checklist?	No	None	None
Performance Inefficiencies	Are there any performance inefficiencies not covered by the checklist?	No	The class is too simple to have performance inefficiencies.	None

Refactoring Task 1: Replace Magic Numbers with Constants

- **Issue**: The code contains magic numbers, such as hardcoded values for button positions, window sizes, or margins, making it harder to understand and maintain.
- **Refactoring Plan**: Replace all magic numbers with named constants. For example, define constants like `BUTTON_WIDTH`, `WINDOW_MARGIN`, or `FONT_SIZE` at the beginning of the class or in a dedicated constants section.
    - **BUTTON_WIDTH = 100;**
    - **WINDOW_MARGIN = 10;**
    - **FONT_SIZE = 14;**
- **Benefit**: Improves code readability by providing meaningful names for values. Makes it easier to update values in one place without the risk of inconsistent changes throughout the code.

Refactoring Task 2: Consolidate Redundant Method Calls

- **Issue**: The same method is being called multiple times with the same parameters, causing unnecessary repetition and reducing code efficiency.
- **Refactoring Plan**: Consolidate redundant method calls by storing the result in a local variable, then reusing the variable instead of calling the method again.
    - For example, if the method `getEmployeeInfo()` is called multiple times with the same input, store the result in a variable:
- **Benefit**: Reduces unnecessary method calls, making the code more efficient. It also reduces the risk of calling the method multiple times when it might return different results.

Refactoring Task 3: Simplify Conditional Expressions

- **Issue**: Complex and nested conditional expressions make the code difficult to read and understand.

- **Refactoring Plan**: Simplify complex `if-else` or ternary operations by breaking them into smaller, easier-to-read conditions, or using `return` statements early in the method to avoid deep nesting.
- **Benefit**: Improves code readability and simplifies control flow, making it easier to maintain and modify. Reduces the likelihood of introducing errors due to complex nesting.

12. Java Code Review Checklist for Admin interface

File name	Admin interface.java
class/interface name	Admininterface

Category	Checklist Item	Yes/No	Issue	Fix
Naming Conventions	Are class names written in PascalCase?	Yes	None	None
	Are variable and method names written in camelCase?	Yes	None	None
	Are constants written in uppercase with underscores?	No	No constants are used in the class.	Consider adding constants for commonly used values if needed.
Code Structure	Are access modifiers used correctly?	Yes	None	None
	Are classes and interfaces separated?	Yes	None	None
	Are packages used appropriately?	Yes	No package declaration in the provided code snippet.	Add package declaration if part of a larger project.

Method Design	Do methods have a single responsibility?	Yes	actionPerformed() method handles multiple button actions. It could be split for better clarity.	Split the actionPerformed() method into smaller methods to handle individual button actions.
	Are method parameters limited?	Yes	None	None
	Is method overloading used properly?	Not Applicable	No method overloading present in this class.	None
Exception Handling	Are exceptions handled with try-catch blocks?	No	No exception handling is used in the class.	Add exception handling to handle potential runtime errors during button actions or GUI initialization.
	Are specific exceptions used?	No	No exceptions are defined in this class.	Consider adding custom exceptions if any specific error conditions are expected.
Code Readability	Are comments added for complex logic?	No	No comments are provided in the class.	Add comments to explain the logic for button actions and GUI setup.
	Is indentation consistent?	Yes	None	None
	Are blank lines used to separate code blocks?	Yes	None	None

	Are meaningful names used for variables, classes, and methods?	Yes	Variable and method names are clear and descriptive.	None
Performance	Are data structures chosen based on performance?	Not Applicable	No complex data structures used.	None
	Are costly operations minimized in loops?	Not Applicable	No loops or costly operations present in this simple class.	None
	Is lazy initialization used?	Not Applicable	No lazy initialization in this class.	None
Memory Management	Are unnecessary object references set to null?	No	Not applicable as no resources requiring cleanup are in the class.	Not applicable.
Security	Is user input validated?	No	Input validation is not performed on any fields in the GUI.	Add input validation to ensure that text fields (e.g., username, password) are not empty and follow required formats.
Maintainability	Are there long methods or deeply nested loops?	Yes	The actionPerformed() method is too long and handles too many actions.	Break the actionPerformed() method into smaller, more

				manageable methods.
	Is there duplicated code?	No	No duplication of code in the class.	None
	Are there any magic numbers?	No	No magic numbers present in the class.	None

Test Related Quries

Category	Checklist Item	Yes/No	Issue	Fix
Test Coverage	Are unit tests provided for all public methods and critical functionalities?	No	N/A	N/A
	Do unit tests cover edge cases and boundary values?	No	N/A	N/A
Test Design	Are tests written following the AAA pattern?	Not Applicable	N/A	N/A
	Are individual test cases independent?	Not Applicable	N/A	N/A
	Are descriptive names used for test methods?	Not Applicable	N/A	N/A
Assertions	Are assertions used to verify expected results?	Not Applicable	N/A	N/A
	Are specific assertions used instead of general ones?	Not Applicable	N/A	N/A
Boundary and Edge Cases	Are edge cases and boundary conditions tested?	No	N/A	N/A
	Are invalid inputs covered by tests?	No	N/A	N/A
Mocking and Stubbing	Are mocks or stubs used to isolate the unit under test?	No	N/A	N/A
Performance Testing	Are tests to check performance for critical methods provided?	No	N/A	N/A
Test Maintainability	Are test methods organized and modular?	Not Applicable	N/A	N/A
	Is there a setup method for initializing common objects?	Not Applicable	N/A	N/A

Housekeeping

Category	Checklist Item	Yes/No	Issue	Fix
Code Smells	Are there any code smells not covered by the checklist?	No	No code smells identified.	None

Coding Standards	Are there any coding standard violations not covered by the checklist?	No	None	None
Performance Inefficiencies	Are there any performance inefficiencies not covered by the checklist?	No	The class is simple and does not have performance inefficiencies.	None

Refactoring Task 1: Extract Duplicate Code into a Helper Method

- **Issue**: The code contains duplicated logic, such as multiple places where the same block of code is repeated, making it harder to maintain and modify.
- **Refactoring Plan**: Identify the duplicated code, extract it into a helper method, and then call this method wherever the logic is needed.
- **Benefit**: Reduces redundancy, making the code more maintainable and easier to modify. Changes to the logic can be made in one place, improving consistency across the codebase.

Refactoring Task 2: Use StringBuilder for String Concatenation

- **Issue**: The code uses the `+` operator to concatenate strings in loops or frequently called methods, which can be inefficient and lead to performance issues.
- **Refactoring Plan**: Replace string concatenation with a `StringBuilder` to improve performance, especially in loops or methods that are executed frequently.
- **Benefit**: Improves performance by reducing the overhead of creating new string objects during concatenation. It also enhances readability when dealing with complex string operations.

Refactoring Task 3: Break Down Large Methods into Smaller Methods

- **Issue**: A method is too long or complex, making it difficult to read, test, and maintain.
- **Refactoring Plan**: Break the large method into smaller, more manageable methods, each responsible for a specific part of the functionality.
- **Benefit**: Makes the code more modular, easier to test, and improves readability. It also helps in maintaining single responsibility for each method.

Appendix:

Code Smells

Code Smell	Description	Example	Suggested Solution
Long Methods	Methods that perform multiple tasks, making them hard to understand and maintain.	The updateInventory method combines inventory updating and file I/O logic.	Split updateInventory into smaller methods: one for inventory updating and another for file operations.
Nested Loops	Deeply nested loops reduce readability and increase complexity.	Nested looping over transactionItem and databaseItem.	Use a HashMap to cache databaseItem by itemID to minimize nested loops.
Duplicate Code	Repeated code for file handling and reading/writing inventory data.	File reading and writing logic appears in multiple places.	Extract file handling logic into reusable helper methods.
Magic Numbers/Strings	Hardcoded values or strings used without explanation.	Hardcoded file paths like "inventory.txt" and string keys like "ItemID".	Define constants for file paths and string keys (e.g., FILE_PATH_INVENTORY, KEY_ITEM_ID).
Primitive Obsession	Overuse of primitive types instead of meaningful abstractions.	Using strings to represent itemID or transaction status instead of enums or classes.	Use an enum for transaction types and consider using a class for inventory items.

Performance Inefficiencies

Issue	Description	Example	Suggested Solution
Repeated Object Lookups	Scanning the entire databaseItem list for every transactionItem.	Iterating through the entire list for each transaction to find the matching item.	Use a HashMap<Integer, Item> to store databaseItem for

			constant-time lookups.
Redundant Calculations	Performing the same calculation repeatedly in a loop.	Repeated calls to getItemID() inside nested loops.	Store itemID in a local variable before the loop to minimize method calls.
Inefficient String Ops	Using string concatenation inside loops.	Using result += "Item: " + itemName in a loop.	Use StringBuilder for concatenating strings inside loops to improve performance.
Excessive Logging	Printing to the console excessively instead of structured logging.	Using System.out.println for error reporting in catch blocks.	Replace System.out.println with a proper logging framework like java.util.logging or Log4j.

Housekeeping

Category	**Issue**	**Fix**
Code Smells	Duplicate file handling code.	Extract file reading/writing logic into dedicated helper methods.
	Combined logic in updateInventory that violates single responsibility principle.	Separate updateInventory into modular methods for inventory processing and file updates.
Coding Standards	Hardcoded strings like file paths and key names.	Define constants for all commonly used strings.
	Inconsistent error handling (direct printing in catch blocks).	Replace direct System.out.println calls with proper logging using a logging framework.
Performance Inefficiencies	Repeated list traversal for inventory lookup.	Cache databaseItem in a HashMap for efficient lookups.
	Direct string concatenation inside loops.	Use StringBuilder for string manipulations inside loops.

13. Java Code Review Checklist for Cashier interface

File name	Casier interface.java
class/interface name	CashierInterface

Category	Checklist Item	Yes/No	Issue	Fix
Naming Conventions	Are class names written in PascalCase?	Yes	None	None
	Are variable and method names written in camelCase?	Yes	None	None
	Are constants written in uppercase with underscores?	No	No constants are used in the class.	Add constants for commonly reused values (e.g., button heights).
Code Structure	Are access modifiers used correctly?	Yes	None	None
	Are classes and interfaces separated?	Yes	None	None
	Are packages used appropriately?	No	No package declaration in the class.	Add package declaration if it is part of a larger project.
Method Design	Do methods have a single	No	The actionPerformed() method handles	Split actionPerformed() into smaller methods for

	responsibility?		multiple button actions.	individual button actions.
	Are method parameters limited?	Yes	None	None
	Is method overloading used properly?	Not Applicable	No method overloading present.	None
Exception Handling	Are exceptions handled with try-catch blocks?	No	No exception handling for invalid input.	Add exception handling for parsing phone numbers or other user inputs.
	Are specific exceptions used?	No	No specific exceptions are defined.	Add specific exception handling for expected errors.
Code Readability	Are comments added for complex logic?	No	No comments explaining business logic.	Add comments for transaction restoration and logout behavior.
	Is indentation consistent?	Yes	None	None
	Are blank lines used to separate code blocks?	Yes	None	None
	Are meaningful names used for variables, classes, and methods?	Yes	Variable names are meaningful and descriptive.	None
Performance	Are data structures chosen	Not Applicable	No complex data	None

	based on performance ?		structures are used.	
	Are costly operations minimized in loops?	Not Applicable	No costly operations in loops.	None
	Is lazy initialization used?	Not Applicable	No lazy initialization is used.	None
Memory Management	Are unnecessary object references set to null?	No	Unused resources are not explicitly cleared.	Set unused Transaction_Interface objects to null after disposal.
Security	Is user input validated?	No	Phone number input is not validated.	Validate user inputs to ensure correctness and prevent crashes.
Maintainability	Are there long methods or deeply nested loops?	Yes	The actionPerformed() method is long.	Refactor actionPerformed() for better maintainability.
	Is there duplicated code?	Yes	Repeated disposal logic in actionPerformed().	Extract common disposal logic into a helper method.
	Are there any magic numbers?	Yes	Button height and positions are hardcoded.	Replace with constants or calculations based on screen size.

Test Related Categories

Category	Checklist Item	Yes/No	Issue	Fix
Test Coverage	Are unit tests provided for all public methods and critical functionalities?	No	N/A	N/A
	Do unit tests cover edge cases and boundary values?	No	N/A	N/A

Test Design	Are tests written following the AAA pattern?	Not Applicable	N/A	N/A
	Are individual test cases independent?	Not Applicable	N/A	N/A
	Are descriptive names used for test methods?	Not Applicable	N/A	N/A
Assertions	Are assertions used to verify expected results?	Not Applicable	N/A	N/A
	Are specific assertions used instead of general ones?	Not Applicable	N/A	N/A
Boundary and Edge Cases	Are edge cases and boundary conditions tested?	No	N/A	N/A
	Are invalid inputs covered by tests?	No	N/A	N/A
Mocking and Stubbing	Are mocks or stubs used to isolate the unit under test?	No	N/A	N/A
Performance Testing	Are tests to check performance for critical methods provided?	No	N/A	N/A
Test Maintainability	Are test methods organized and modular?	Not Applicable	N/A	N/A
	Is there a setup method for initializing common objects?	Not Applicable	N/A	N/A

Housekeeping

Category	Checklist Item	Yes/No	Issue	Fix
Code Smells	Are there any code smells not covered by the checklist?	No	None	None
Coding Standards	Are there any coding standard violations not covered by the checklist?	No	None	None
Performance Inefficiencies	Are there any performance inefficiencies not covered by the checklist?	No	None	None

Refactoring Task 1: Extract Duplicate Code into a Helper Method

- **Issue**: The `actionPerformed()` method contains repeated logic for initializing and disposing of the `Transaction_Interface` object.
- **Refactoring Plan**:
    1. Create a helper method (e.g., `openTransaction(String operation)`) to handle the repeated initialization and disposal logic.
    2. Replace the repetitive code in the `actionPerformed()` method with calls to this helper method.

- **Benefit**: Reduces redundancy, making the code more maintainable. Any changes to the initialization logic can now be made in a single location.

Refactoring Task 2: Replace Magic Numbers with Constants

- **Issue**: Hardcoded values for button dimensions and positions are scattered throughout the class, making it harder to modify or understand.
- **Refactoring Plan**:
    1. Define constants (e.g., `BUTTON_HEIGHT`, `BUTTON_WIDTH`, `BUTTON_VERTICAL_SPACING`) to represent these magic numbers.
    2. Replace all instances of these hardcoded values with the defined constants.
- **Benefit**: Improves readability and maintainability by giving meaningful names to numeric values. Changes to button dimensions or layout can now be made by updating the constants in one place.

Refactoring Task 3: Break Down Large Methods into Smaller Methods

- **Issue**: The `actionPerformed()` method is long and handles multiple unrelated button actions, violating the single-responsibility principle.
- **Refactoring Plan**:
    1. Extract the logic for each button action into separate methods (e.g., `handleSaleAction()`, `handleRentalAction()`, `handleReturnAction()`, `handleLogOutAction()`).
    2. Update `actionPerformed()` to delegate tasks to these smaller methods based on the event source.
- **Benefit**: Enhances readability, makes the code more modular, and simplifies testing. Each button action can now be tested independently.

Appendix:

Code Smells

Code Smell	Description	Example	Suggested Solution
Long Methods	Methods that perform multiple tasks, making them hard to understand and maintain.	The updateInventory method combines inventory updating and file I/O logic.	Split updateInventory into smaller methods: one for inventory updating and another for file operations.

Nested Loops	Deeply nested loops reduce readability and increase complexity.	Nested looping over transactionItem and databaseItem.	Use a HashMap to cache databaseItem by itemID to minimize nested loops.
Duplicate Code	Repeated code for file handling and reading/writing inventory data.	File reading and writing logic appears in multiple places.	Extract file handling logic into reusable helper methods.
Magic Numbers/Strings	Hardcoded values or strings used without explanation.	Hardcoded file paths like "inventory.txt" and string keys like "ItemID".	Define constants for file paths and string keys (e.g., FILE_PATH_INVENTORY, KEY_ITEM_ID).
Primitive Obsession	Overuse of primitive types instead of meaningful abstractions.	Using strings to represent itemID or transaction status instead of enums or classes.	Use an enum for transaction types and consider using a class for inventory items.

Performance Inefficiencies

Issue	Description	Example	Suggested Solution
Repeated Object Lookups	Scanning the entire databaseItem list for every transactionItem.	Iterating through the entire list for each transaction to find the matching item.	Use a HashMap<Integer, Item> to store databaseItem for constant-time lookups.
Redundant Calculations	Performing the same calculation repeatedly in a loop.	Repeated calls to getItemID() inside nested loops.	Store itemID in a local variable before the loop to minimize method calls.

Inefficient String Ops	Using string concatenation inside loops.	Using result += "Item: " + itemName in a loop.	Use StringBuilder for concatenating strings inside loops to improve performance.
Excessive Logging	Printing to the console excessively instead of structured logging.	Using System.out.println for error reporting in catch blocks.	Replace System.out.println with a proper logging framework like java.util.logging or Log4j.

Housekeeping

Category	**Issue**	**Fix**
Code Smells	Duplicate file handling code.	Extract file reading/writing logic into dedicated helper methods.
	Combined logic in updateInventory that violates single responsibility principle.	Separate updateInventory into modular methods for inventory processing and file updates.
Coding Standards	Hardcoded strings like file paths and key names.	Define constants for all commonly used strings.
	Inconsistent error handling (direct printing in catch blocks).	Replace direct System.out.println calls with proper logging using a logging framework.
Performance Inefficiencies	Repeated list traversal for inventory lookup.	Cache databaseItem in a HashMap for efficient lookups.
	Direct string concatenation inside loops.	Use StringBuilder for string manipulations inside loops.

14. Java Code Review Checklist for Enter_Item interface

File name	EnterItem_interface.java
class/interface name	EnterItem_Interface

Category	Checklist Item	Yes/No	Issue	Fix
Naming Conventions	Are class names written in PascalCase?	Yes	None	None
	Are variable and method names written in camelCase?	Yes	None	None
	Are constants written in uppercase with underscores?	No	No constants are used in the class.	Add constants for GUI dimensions and reusable values.
Code Structure	Are access modifiers used correctly?	Yes	None	None
	Are classes and interfaces separated?	Yes	None	None
	Are packages used appropriately?	No	No package declaration in the class.	Add package declaration if it’s part of a larger project.

Method Design	Do methods have a single responsibility?	No	The actionPerformed() method handles multiple actions.	Split actionPerformed() into smaller methods for individual actions.
	Are method parameters limited?	Yes	None	None
	Is method overloading used properly?	Not Applicable	No method overloading present.	None
Exception Handling	Are exceptions handled with try-catch blocks?	No	No exception handling for invalid input.	Add exception handling for getItemID() and getAmount().
	Are specific exceptions used?	No	General parsing errors not handled.	Catch specific exceptions like NumberFormatException.
Code Readability	Are comments added for complex logic?	No	No comments explaining business logic.	Add comments for item addition/removal logic and updateTextArea.
	Is indentation consistent?	Yes	None	None
	Are blank lines used to separate code blocks?	Yes	None	None
	Are meaningful names	Yes	Variable and method names are	None

	used for variables, classes, and methods?		meaningful and descriptive.	
Performance	Are data structures chosen based on performance?	Not Applicable	No complex data structures are used.	None
	Are costly operations minimized in loops?	No	String concatenation in updateTextArea() could be optimized.	Use StringBuilder for string concatenation in loops.
	Is lazy initialization used?	Not Applicable	No lazy initialization is used.	None
Memory Management	Are unnecessary object references set to null?	No	Unused resources not cleared explicitly.	Set unused resources to null post-disposal if needed.
Security	Is user input validated?	No	No validation for itemID and amount fields.	Validate user inputs to ensure numeric values are entered.
Maintainability	Are there long methods or deeply nested loops?	Yes	actionPerformed() is long and handles multiple concerns.	Refactor actionPerformed() into smaller, more cohesive methods.
	Is there duplicated code?	Yes	Repeated disposal logic in button actions.	Extract common disposal logic into a helper method.

	Are there any magic numbers?	Yes	Button and text field positions are hardcoded.	Replace with constants or layout managers.

Test Related Categories

Category	Checklist Item	Yes/No	Issue	Fix
Test Coverage	Are unit tests provided for all public methods and critical functionalities?	No	N/A	N/A
	Do unit tests cover edge cases and boundary values?	No	N/A	N/A
Test Design	Are tests written following the AAA pattern?	Not Applicable	N/A	N/A
	Are individual test cases independent?	Not Applicable	N/A	N/A
	Are descriptive names used for test methods?	Not Applicable	N/A	N/A
Assertions	Are assertions used to verify expected results?	Not Applicable	N/A	N/A
	Are specific assertions used instead of general ones?	Not Applicable	N/A	N/A
Boundary and Edge Cases	Are edge cases and boundary conditions tested?	No	N/A	N/A
	Are invalid inputs covered by tests?	No	N/A	N/A
Mocking and Stubbing	Are mocks or stubs used to isolate the unit under test?	No	N/A	N/A
Performance Testing	Are tests to check performance for critical methods provided?	No	N/A	N/A
Test Maintainability	Are test methods organized and modular?	Not Applicable	N/A	N/A
	Is there a setup method for initializing common objects?	Not Applicable	N/A	N/A

Housekeeping

Category	Checklist Item	Yes/No	Issue	Fix
Code Smells	Are there any code smells not covered by the checklist?	No	None	None
Coding Standards	Are there any coding standard violations not covered by the checklist?	No	None	None
Performance Inefficiencies	Are there any performance inefficiencies not covered by the checklist?	No	None	None

Refactoring Task 1: Extract Repeated Code into a Helper Method

- **Issue**: The `actionPerformed()` method contains repeated logic for disposing of the interface after completing an action.
- **Refactoring Plan**:
    1. Create a helper method, `disposeInterface()`, to handle the repeated logic for setting visibility to false and disposing of the interface.
    2. Replace all instances of this repeated logic in the `actionPerformed()` method with calls to `disposeInterface()`.
- **Benefit**: Reduces redundancy, making the code more maintainable. Changes to the disposal logic can now be made in a single location.

Refactoring Task 2: Replace Magic Numbers with Constants

- **Issue**: Hardcoded values for button dimensions, positions, and component spacing are scattered throughout the class.
- **Refactoring Plan**:

    1. Define constants (e.g., `BUTTON_WIDTH`, `BUTTON_HEIGHT`, `TEXTFIELD_WIDTH`, `TEXTFIELD_HEIGHT`) to represent these magic numbers.
    2. Replace all instances of these hardcoded values in the layout setup with the defined constants.

- **Benefit**: Improves readability and maintainability by giving meaningful names to numeric values. Changes to layout dimensions can now be made in one place.

Refactoring Task 3: Break Down Large Methods into Smaller Methods

- **Issue**: The `actionPerformed()` method is lengthy and handles multiple unrelated actions, violating the single-responsibility principle.
- **Refactoring Plan**:

    1. Extract the logic for handling different button actions into separate methods (e.g., `handleEnterButtonAction()`, `handleExitButtonAction()`).
    2. Update the `actionPerformed()` method to delegate tasks to these smaller methods based on the event source.

- **Benefit**: Enhances readability, modularity, and testability. Each button's functionality can now be tested and modified independently.

Appendix:

Code Smells

Code Smell	Description	Example	Suggested Solution
Long Methods	Methods that perform multiple tasks, making them hard to understand and maintain.	The updateInventory method combines inventory updating and file I/O logic.	Split updateInventory into smaller methods: one for inventory updating and another for file operations.
Nested Loops	Deeply nested loops reduce readability and increase complexity.	Nested looping over transactionItem and databaseItem.	Use a HashMap to cache databaseItem by itemID to minimize nested loops.
Duplicate Code	Repeated code for file handling and reading/writing inventory data.	File reading and writing logic appears in multiple places.	Extract file handling logic into reusable helper methods.
Magic Numbers/Strings	Hardcoded values or strings used without explanation.	Hardcoded file paths like "inventory.txt" and string keys like "ItemID".	Define constants for file paths and string keys (e.g., FILE_PATH_INVENTORY, KEY_ITEM_ID).
Primitive Obsession	Overuse of primitive types instead of meaningful abstractions.	Using strings to represent itemID or transaction status instead of enums or classes.	Use an enum for transaction types and consider using a class for inventory items.

Performance Inefficiencies

Issue	Description	Example	Suggested Solution

Repeated Object Lookups	Scanning the entire databaseItem list for every transactionItem.	Iterating through the entire list for each transaction to find the matching item.	Use a HashMap<Integer, Item> to store databaseItem for constant-time lookups.
Redundant Calculations	Performing the same calculation repeatedly in a loop.	Repeated calls to getItemID() inside nested loops.	Store itemID in a local variable before the loop to minimize method calls.
Inefficient String Ops	Using string concatenation inside loops.	Using result += "Item: " + itemName in a loop.	Use StringBuilder for concatenating strings inside loops to improve performance.
Excessive Logging	Printing to the console excessively instead of structured logging.	Using System.out.println for error reporting in catch blocks.	Replace System.out.println with a proper logging framework like java.util.logging or Log4j.

Housekeeping

Category	**Issue**	**Fix**
Code Smells	Duplicate file handling code.	Extract file reading/writing logic into dedicated helper methods.
	Combined logic in updateInventory that violates single responsibility principle.	Separate updateInventory into modular methods for inventory processing and file updates.
Coding Standards	Hardcoded strings like file paths and key names.	Define constants for all commonly used strings.
	Inconsistent error handling (direct printing in catch blocks).	Replace direct System.out.println calls with proper logging using a logging framework.

Performance Inefficiencies	Repeated list traversal for inventory lookup.	Cache databaseItem in a HashMap for efficient lookups.
	Direct string concatenation inside loops.	Use StringBuilder for string manipulations inside loops.

15. Java Code Review Checklist for Login interface

File name	Login_interface.java
class/interface name	Login_Interface

Category	Checklist Item	Yes/No	Issue	Fix
Naming Conventions	Are class names written in PascalCase?	Yes	None	None
	Are variable and method names written in camelCase?	Yes	None	None
	Are constants written in uppercase with underscores?	No	No constants are used in the class.	Consider adding constants for GUI layout (e.g., button sizes, window size).
Code Structure	Are access modifiers used correctly?	Yes	None	None
	Are classes and interfaces separated?	Yes	None	None

	Are packages used appropriately?	No	The class is not part of any package.	Add a package declaration if it's part of a larger project.
Method Design	Do methods have a single responsibility?	No	The actionPerformed() method handles login, validation, and UI updates.	Split actionPerformed() into smaller methods for login and UI updates.
	Are method parameters limited?	Yes	None	None
	Is method overloading used properly?	Not Applicable	No method overloading present.	None
Exception Handling	Are exceptions handled with try-catch blocks?	No	No exception handling for invalid input or login errors.	Add exception handling for invalid username or password format.
	Are specific exceptions used?	No	General login failure is handled without specific exceptions.	Use specific exceptions like InvalidCredentialsException to improve clarity.
Code Readability	Are comments added for complex logic?	No	The code lacks comments explaining key sections, especially for login validation.	Add comments explaining the login flow and error handling.
	Is indentatio	Yes	None	None

	n consistent ?			
	Are blank lines used to separate code blocks?	Yes	None	None
	Are meaningful names used for variables, classes, and methods?	Yes	Variable and method names are meaningful.	None
Performance	Are data structures chosen based on performance?	Not Applicable	No data structures are used for complex operations.	None
	Are costly operations minimized in loops?	Not Applicable	No loops present.	None
	Is lazy initialization used?	Not Applicable	No lazy initialization used.	None
Memory Management	Are unnecessary object references set to null?	No	There is no explicit clearing of references.	Consider setting unused references to null after they are no longer needed.
Security	Is user input validated?	No	Input validation for username and password fields is missing.	Add validation to check for non-empty inputs and invalid characters.
Maintainability	Are there long	Yes	actionPerformed() method	Refactor the actionPerformed()

	methods or deeply nested loops?		is long and performs multiple actions.	method into smaller, more cohesive methods.
	Is there duplicated code?	Yes	The login logic is repeated for checking user credentials.	Extract login logic into a separate method to avoid repetition.
	Are there any magic numbers?	Yes	The window dimensions and button positions are hardcoded.	Replace with constants for GUI elements (e.g., window size, button positions).

Test Related Categories

Category	Checklist Item	Yes/No	Issue	Fix
Test Coverage	Are unit tests provided for all public methods and critical functionalities?	No	N/A	N/A
	Do unit tests cover edge cases and boundary values?	No	N/A	N/A
Test Design	Are tests written following the AAA pattern?	Not Applicable	N/A	N/A
	Are individual test cases independent?	Not Applicable	N/A	N/A
	Are descriptive names used for test methods?	Not Applicable	N/A	N/A
Assertions	Are assertions used to verify expected results?	Not Applicable	N/A	N/A
	Are specific assertions used instead of general ones?	Not Applicable	N/A	N/A
Boundary and Edge Cases	Are edge cases and boundary conditions tested?	No	N/A	N/A
	Are invalid inputs covered by tests?	No	N/A	N/A
Mocking and Stubbing	Are mocks or stubs used to isolate the unit under test?	No	N/A	N/A
Performance Testing	Are tests to check performance for critical methods provided?	No	N/A	N/A
Test Maintainability	Are test methods organized and modular?	Not Applicable	N/A	N/A

	Is there a setup method for initializing common objects?	Not Applicable	N/A	N/A

Housekeeping				
Category	Checklist Item	Yes/No	Issue	Fix
Code Smells	Are there any code smells not covered by the checklist?	No	None	None
Coding Standards	Are there any coding standard violations not covered by the checklist?	No	None	None
Performance Inefficiencies	Are there any performance inefficiencies not covered by the checklist?	No	None	None

Refactoring Task 1: Extract Repeated Code into a Helper Method

- **Issue**: The logic for disposing of the interface after completing an action is repeated when navigating to the `Cashier_Interface` and `Admin_Interface`.
- **Refactoring Plan**:
    1. Create a helper method, `disposeInterface()`, that handles the repeated logic of setting visibility to false and disposing of the interface.
    2. Replace all instances of this logic in the `actionPerformed()` method with calls to `disposeInterface()`.
- **Benefit**: Reduces redundancy, making the code more maintainable. Changes to the disposal logic can now be made in a single location, improving clarity and reducing the chances of errors when updating the code.

Refactoring Task 2: Replace Magic Numbers with Constants

- **Issue**: Hardcoded values for window size, button positions, and component dimensions (e.g., button width, height, and position) are scattered throughout the code.
- **Refactoring Plan**:

    1. Define constants at the top of the class to represent these magic numbers (e.g., `WINDOW_WIDTH`, `WINDOW_HEIGHT`, `BUTTON_WIDTH`, `BUTTON_HEIGHT`, etc.).

2. Replace all instances of these hardcoded values in the layout setup with the defined constants.

- **Benefit**: Improves readability and maintainability by giving meaningful names to numeric values. When layout changes are needed, they can now be done in one place, reducing potential for errors and improving scalability.

Refactoring Task 3: Break Down Large Method into Smaller Methods

- **Issue**: The `actionPerformed()` method is lengthy and handles multiple responsibilities (handling login, displaying error messages, and managing interface transitions).
- **Refactoring Plan**:

  1. Extract the logic for handling the login action, error display, and interface transitions into separate methods (e.g., `handleLoginAction()`, `showErrorMessage()`, `navigateToCashier()`, `navigateToAdmin()`).
  2. Update the `actionPerformed()` method to delegate tasks to these smaller methods based on the event source.

- **Benefit**: Enhances code readability and modularity. Each action (login, error handling, UI navigation) can now be managed independently, making the code easier to test, maintain, and extend.

Appendix:

Code Smells

Code Smell	Description	Example	Suggested Solution
Long Methods	Methods that perform multiple tasks, making them hard to understand and maintain.	The updateInventory method combines inventory updating and file I/O logic.	Split updateInventory into smaller methods: one for inventory updating and another for file operations.
Nested Loops	Deeply nested loops reduce readability and increase complexity.	Nested looping over transactionItem and databaseItem.	Use a HashMap to cache databaseItem by itemID to minimize nested loops.

Duplicate Code	Repeated code for file handling and reading/writing inventory data.	File reading and writing logic appears in multiple places.	Extract file handling logic into reusable helper methods.
Magic Numbers/Strings	Hardcoded values or strings used without explanation.	Hardcoded file paths like "inventory.txt" and string keys like "ItemID".	Define constants for file paths and string keys (e.g., FILE_PATH_INVENTORY, KEY_ITEM_ID).
Primitive Obsession	Overuse of primitive types instead of meaningful abstractions.	Using strings to represent itemID or transaction status instead of enums or classes.	Use an enum for transaction types and consider using a class for inventory items.

Performance Inefficiencies

Issue	Description	Example	Suggested Solution
Repeated Object Lookups	Scanning the entire databaseItem list for every transactionItem.	Iterating through the entire list for each transaction to find the matching item.	Use a HashMap<Integer, Item> to store databaseItem for constant-time lookups.
Redundant Calculations	Performing the same calculation repeatedly in a loop.	Repeated calls to getItemID() inside nested loops.	Store itemID in a local variable before the loop to minimize method calls.
Inefficient String Ops	Using string concatenation inside loops.	Using result += "Item: " + itemName in a loop.	Use StringBuilder for concatenating strings inside loops to improve performance.
Excessive Logging	Printing to the console excessively instead	Using System.out.println for	Replace System.out.println with a proper logging

	of structured logging.	error reporting in catch blocks.	framework like java.util.logging or Log4j.

Housekeeping

Category	Issue	Fix
Code Smells	Duplicate file handling code.	Extract file reading/writing logic into dedicated helper methods.
	Combined logic in updateInventory that violates single responsibility principle.	Separate updateInventory into modular methods for inventory processing and file updates.
Coding Standards	Hardcoded strings like file paths and key names.	Define constants for all commonly used strings.
	Inconsistent error handling (direct printing in catch blocks).	Replace direct System.out.println calls with proper logging using a logging framework.
Performance Inefficiencies	Repeated list traversal for inventory lookup.	Cache databaseItem in a HashMap for efficient lookups.
	Direct string concatenation inside loops.	Use StringBuilder for string manipulations inside loops.

16. Java Code Review Checklist for payment_interface

File name	Payment_interface.java
class/interface name	Payment_interface

Category	Checklist Item	Yes/No	Issue	Fix
Naming Conventions	Are class names written in PascalCase?	Yes	None	None
	Are variable and method names written in camelCase?	Yes	None	None
	Are constants written in uppercase with underscores?	No	No constants used in the class.	Add constants for button sizes, window size, etc.
Code Structure	Are access modifiers used correctly?	Yes	None	None
	Are classes and interfaces separated?	Yes	None	None

	Are packages used appropriately?	No	Class is not part of any package.	Consider adding a package declaration.
Method Design	Do methods have a single responsibility?	No	actionPerformed() handles multiple actions.	Break actionPerformed() into smaller methods for clarity.
	Are method parameters limited?	Yes	None	None
	Is method overloading used properly?	Not Applicable	No method overloading.	None
Exception Handling	Are exceptions handled with try-catch blocks?	No	No exception handling for invalid inputs.	Add exception handling, especially for user input during payments.
	Are specific exceptions used?	No	No specific exceptions are used.	Use specific exceptions (e.g., InvalidCardNumberException).
Code Readability	Are comments added for complex logic?	No	Some areas of code lack detailed comments.	Add comments for clarity, especially in logic for payments and return handling.
	Is indentation consistent?	Yes	None	None
	Are blank lines used to separate code blocks?	Yes	None	None

	Are meaningful names used for variables, classes, and methods?	Yes	Names are meaningful.	None
Performance	Are data structures chosen based on performance?	No	No significant data structures involved.	None
	Are costly operations minimized in loops?	Not Applicable	No loops present that require optimization.	None
	Is lazy initialization used?	Not Applicable	No lazy initialization used.	None
Memory Management	Are unnecessary object references set to null?	No	No explicit memory management.	Consider nullifying unused references after use.
Security	Is user input validated?	No	Validation for cardNo and cash inputs is minimal.	Add validation for the payment inputs (e.g., card number format).
Maintainability	Are there long methods or deeply nested loops?	Yes	actionPerformed() method is large.	Refactor actionPerformed() into smaller methods.
	Is there duplicated code?	Yes	Duplicate logic for handling cash and	Extract common logic into reusable methods.

			electronic payments.	
	Are there any magic numbers?	Yes	Window size and button positions are hardcoded.	Replace magic numbers with constants.

Test Related Categories

Category	Checklist Item	Yes/No	Issue	Fix
Test Coverage	Are unit tests provided for all public methods and critical functionalities?	No	N/A	N/A
	Do unit tests cover edge cases and boundary values?	No	N/A	N/A
Test Design	Are tests written following the AAA pattern?	Not Applicable	N/A	N/A
	Are individual test cases independent?	Not Applicable	N/A	N/A
	Are descriptive names used for test methods?	Not Applicable	N/A	N/A
Assertions	Are assertions used to verify expected results?	Not Applicable	N/A	N/A
	Are specific assertions used instead of general ones?	Not Applicable	N/A	N/A
Boundary and Edge Cases	Are edge cases and boundary conditions tested?	No	N/A	N/A
	Are invalid inputs covered by tests?	No	N/A	N/A
Mocking and Stubbing	Are mocks or stubs used to isolate the unit under test?	No	N/A	N/A
Performance Testing	Are tests to check performance for critical methods provided?	No	N/A	N/A
Test Maintainability	Are test methods organized and modular?	Not Applicable	N/A	N/A

	Is there a setup method for initializing common objects?	Not Applicable	N/A	N/A

Housekeeping				
Category	Checklist Item	Yes/No	Issue	Fix
Code Smells	Are there any code smells not covered by the checklist?	Yes	actionPerformed() method is large and handles multiple tasks.	Refactor actionPerformed() into smaller methods for readability.
Coding Standards	Are there any coding standard violations not covered by the checklist?	No	None	None
Performance Inefficiencies	Are there any performance inefficiencies not covered by the checklist?	No	None	None

Refactoring Task 1: Extract Repeated Code into a Helper Method

- **Issue**: The logic for disposing of the interface after completing an action (e.g., navigating to the `Cashier_Interface` or `Admin_Interface`) is repeated in multiple parts of the code.
- **Refactoring Plan**:
  1. Create a helper method called `disposeInterface()` that consolidates the repeated logic of setting visibility to false and disposing of the current interface.
  2. Replace all instances of the repeated code in the `actionPerformed()` method with calls to `disposeInterface()`.
- **Benefit**: Reduces redundancy in the code, improving maintainability and readability. Any future changes to the disposal behavior can now be made in one place, reducing the risk of inconsistencies and making updates easier.

Refactoring Task 2: Consolidate UI Button Action Handling

- **Issue**: The `actionPerformed()` method contains repetitive blocks of code for handling actions related to different buttons (e.g., `PayCash`, `PayElectronic`, `cancelTransaction`, `confirm`). Each button action has similar logic for event handling, which could be consolidated.
- **Refactoring Plan**:

    1. Extract the button-specific actions into smaller methods such as `handleCashPayment()`, `handleElectronicPayment()`, `handleCancelTransaction()`, and `handleConfirmPayment()`.
    2. In the `actionPerformed()` method, instead of directly processing the logic for each button, call the appropriate handler method based on the source of the event.

- **Benefit**: Makes the code more modular and readable. Each button action is handled independently, making the code easier to test, extend, and maintain. This also improves scalability if new buttons or payment methods need to be added in the future.

Refactoring Task 3: Replace Magic Numbers with Named Constants

- **Issue**: The code uses hardcoded "magic numbers" for GUI layout and other values, such as button dimensions and screen positions, making it difficult to modify the layout or understand the purpose of these values.
- **Refactoring Plan**:

    1. Define meaningful constants at the top of the class for values like window size, button dimensions, and other commonly used numeric values (e.g., `WINDOW_WIDTH`, `BUTTON_WIDTH`, `BUTTON_HEIGHT`).
    2. Replace the magic numbers in the layout code and other sections with these named constants.

- **Benefit**: Improves readability and maintainability by giving descriptive names to numeric values. If the layout or button sizes need to change, they can be modified in one place, reducing the likelihood of errors and making the code easier to update.

Appendix:

Code Smells

Code Smell	Description	Example	Suggested Solution

Long Methods	Methods that perform multiple tasks, making them hard to understand and maintain.	The updateInventory method combines inventory updating and file I/O logic.	Split updateInventory into smaller methods: one for inventory updating and another for file operations.
Nested Loops	Deeply nested loops reduce readability and increase complexity.	Nested looping over transactionItem and databaseItem.	Use a HashMap to cache databaseItem by itemID to minimize nested loops.
Duplicate Code	Repeated code for file handling and reading/writing inventory data.	File reading and writing logic appears in multiple places.	Extract file handling logic into reusable helper methods.
Magic Numbers/Strings	Hardcoded values or strings used without explanation.	Hardcoded file paths like "inventory.txt" and string keys like "ItemID".	Define constants for file paths and string keys (e.g., FILE_PATH_INVENTORY, KEY_ITEM_ID).
Primitive Obsession	Overuse of primitive types instead of meaningful abstractions.	Using strings to represent itemID or transaction status instead of enums or classes.	Use an enum for transaction types and consider using a class for inventory items.

Performance Inefficiencies

Issue	Description	Example	Suggested Solution
Repeated Object Lookups	Scanning the entire databaseItem list for every transactionItem.	Iterating through the entire list for each transaction to find the matching item.	Use a HashMap<Integer, Item> to store databaseItem for

			constant-time lookups.
Redundant Calculations	Performing the same calculation repeatedly in a loop.	Repeated calls to getItemID() inside nested loops.	Store itemID in a local variable before the loop to minimize method calls.
Inefficient String Ops	Using string concatenation inside loops.	Using result += "Item: " + itemName in a loop.	Use StringBuilder for concatenating strings inside loops to improve performance.
Excessive Logging	Printing to the console excessively instead of structured logging.	Using System.out.println for error reporting in catch blocks.	Replace System.out.println with a proper logging framework like java.util.logging or Log4j.

Housekeeping

Category	**Issue**	**Fix**
Code Smells	Duplicate file handling code.	Extract file reading/writing logic into dedicated helper methods.
	Combined logic in updateInventory that violates single responsibility principle.	Separate updateInventory into modular methods for inventory processing and file updates.
Coding Standards	Hardcoded strings like file paths and key names.	Define constants for all commonly used strings.
	Inconsistent error handling (direct printing in catch blocks).	Replace direct System.out.println calls with proper logging using a logging framework.
Performance Inefficiencies	Repeated list traversal for inventory lookup.	Cache databaseItem in a HashMap for efficient lookups.
	Direct string concatenation inside loops.	Use StringBuilder for string manipulations inside loops.

17. Java Code Review Checklist for payment_interface

File name	Transaction_interface.java
class/interface name	Transaction_interface

Category	Checklist Item	Yes/No	Issue	Fix
Naming Conventions	Are class names written in PascalCase?	Yes	None	None
	Are variable and method names written in camelCase?	Yes	None	None
	Are constants written in uppercase with underscores?	No	No constants used in the class.	Add constants for button sizes, window size, etc.
Code Structure	Are access modifiers used correctly?	Yes	None	None
	Are classes and	Yes	None	None

	interfaces separated?			
	Are packages used appropriately?	No	Class is not part of any package.	Consider adding a package declaration.
Method Design	Do methods have a single responsibility?	No	actionPerformed() handles multiple actions.	Refactor actionPerformed() into smaller methods for clarity.
	Are method parameters limited?	Yes	None	None
	Is method overloading used properly?	Not Applicable	No method overloading.	None
Exception Handling	Are exceptions handled with try-catch blocks?	No	No exception handling for invalid inputs.	Add exception handling for user inputs (e.g., phone number validation).
	Are specific exceptions used?	No	No specific exceptions are used.	Use specific exceptions where applicable (e.g., InvalidPhoneNumberException).
Code Readability	Are comments added for	No	Some areas lack detailed comments.	Add comments for clarity, especially in logic for handling transactions.

	complex logic?			
	Is indentation consistent?	Yes	None	None
	Are blank lines used to separate code blocks?	Yes	None	None
	Are meaningful names used for variables, classes, and methods?	Yes	Names are meaningful.	None
Performance	Are data structures chosen based on performance?	No	No significant data structures involved.	None
	Are costly operations minimized in loops?	Not Applicable	No loops present that require optimization.	None
	Is lazy initialization used?	Not Applicable	No lazy initialization used.	None
Memory Management	Are unnecessary	No	No explicit memory management.	Consider nullifying unused references after use.

	object references set to null?			
Security	Is user input validated?	No	Validation for user inputs is minimal.	Improve validation (e.g., for phone number and coupon code).
Maintainability	Are there long methods or deeply nested loops?	Yes	actionPerformed() method is large.	Refactor actionPerformed() into smaller methods for better maintainability.
	Is there duplicated code?	Yes	Duplicate logic for handling different operations (e.g., handling phone number input).	Extract common logic into reusable methods.
	Are there any magic numbers?	Yes	Window size and button positions are hardcoded.	Replace magic numbers with constants.

Test Related Categories

Category	Checklist Item	Yes/No	Issue	Fix
Test Coverage	Are unit tests provided for all public methods and critical functionalities?	No	N/A	N/A
	Do unit tests cover edge cases and boundary values?	No	N/A	N/A
Test Design	Are tests written following the AAA pattern?	Not Applicable	N/A	N/A

	Are individual test cases independent?	Not Applicable	N/A	N/A
	Are descriptive names used for test methods?	Not Applicable	N/A	N/A
Assertions	Are assertions used to verify expected results?	Not Applicable	N/A	N/A
	Are specific assertions used instead of general ones?	Not Applicable	N/A	N/A
Boundary and Edge Cases	Are edge cases and boundary conditions tested?	No	N/A	N/A
	Are invalid inputs covered by tests?	No	N/A	N/A
Mocking and Stubbing	Are mocks or stubs used to isolate the unit under test?	No	N/A	N/A
Performance Testing	Are tests to check performance for critical methods provided?	No	N/A	N/A
Test Maintainability	Are test methods organized and modular?	Not Applicable	N/A	N/A
	Is there a setup method for initializing common objects?	Not Applicable	N/A	N/A

Housekeeping

Category	Checklist Item	Yes/No	Issue	Fix
Code Smells	Are there any code smells not covered by the checklist?	Yes	actionPerformed() method is large and handles multiple tasks.	Refactor actionPerformed() into smaller methods.
Coding Standards	Are there any coding standard violations not covered by the checklist?	No	None	None
Performance Inefficiencies	Are there any performance	No	None	None

		inefficiencies not covered by the checklist?			

Refactoring Task 1: Simplify `actionPerformed()` Method by Extracting Logic

- **Issue**: The `actionPerformed()` method in `Transaction_Interface` handles multiple actions and includes long if-else chains, making the code difficult to maintain.
- **Refactoring Plan**:
    1. Break down the method by creating separate helper methods for each distinct action (e.g., `handleAddItem()`, `handleRemoveItem()`, `handleEndTransaction()`, `handleCancelTransaction()`).
    2. Move the relevant action logic into these methods.
    3. Modify `actionPerformed()` to delegate tasks to these helper methods.
- **Benefit**: Improves the readability and maintainability of the `actionPerformed()` method, making it easier to extend and modify. Each action will be easier to understand and modify independently.

Refactoring Task 2: Consolidate Customer Phone Number Validation

- **Issue**: The logic for validating the customer phone number in `getCustomerPhone()` is repeated and handled with a `while` loop that can be refactored for clarity.
- **Refactoring Plan**:

    1. Create a helper method, `validatePhoneNumber()`, which encapsulates the logic for input validation and re-prompting the user.
    2. Replace the validation code in `getCustomerPhone()` with calls to `validatePhoneNumber()`.

- **Benefit**: Centralizes phone number validation logic in a single method, making the code cleaner and easier to update. If validation rules change in the future, they can be updated in one place.

Refactoring Task 3: Replace Hardcoded Database File Paths with Constants

- **Issue**: The database file paths are hardcoded multiple times in the code, making the class difficult to maintain if the file paths change.
- **Refactoring Plan**:

    1. Define constants for the file paths at the top of the class, such as `RENTAL_DATABASE_FILE`, `SALE_DATABASE_FILE`, etc.
    2. Replace all instances of the hardcoded file paths with these constants.

- **Benefit**: Improves the maintainability of the code. File paths are now centralized, reducing the risk of errors and making it easier to change paths in the future.

Appendix:

Code Smells

Code Smell	Description	Example	Suggested Solution
Long Methods	Methods that perform multiple tasks, making them hard to understand and maintain.	The updateInventory method combines inventory updating and file I/O logic.	Split updateInventory into smaller methods: one for inventory updating and another for file operations.
Nested Loops	Deeply nested loops reduce readability and increase complexity.	Nested looping over transactionItem and databaseItem.	Use a HashMap to cache databaseItem by itemID to minimize nested loops.
Duplicate Code	Repeated code for file handling and reading/writing inventory data.	File reading and writing logic appears in multiple places.	Extract file handling logic into reusable helper methods.

Magic Numbers/Strings	Hardcoded values or strings used without explanation.	Hardcoded file paths like "inventory.txt" and string keys like "ItemID".	Define constants for file paths and string keys (e.g., FILE_PATH_INVENTORY, KEY_ITEM_ID).
Primitive Obsession	Overuse of primitive types instead of meaningful abstractions.	Using strings to represent itemID or transaction status instead of enums or classes.	Use an enum for transaction types and consider using a class for inventory items.

Performance Inefficiencies

Issue	Description	Example	Suggested Solution
Repeated Object Lookups	Scanning the entire databaseItem list for every transactionItem.	Iterating through the entire list for each transaction to find the matching item.	Use a HashMap<Integer, Item> to store databaseItem for constant-time lookups.
Redundant Calculations	Performing the same calculation repeatedly in a loop.	Repeated calls to getItemID() inside nested loops.	Store itemID in a local variable before the loop to minimize method calls.
Inefficient String Ops	Using string concatenation inside loops.	Using result += "Item: " + itemName in a loop.	Use StringBuilder for concatenating strings inside loops to improve performance.
Excessive Logging	Printing to the console excessively instead of structured logging.	Using System.out.println for error reporting in catch blocks.	Replace System.out.println with a proper logging framework like java.util.logging or Log4j.

Housekeeping

Category	Issue	Fix
Code Smells	Duplicate file handling code.	Extract file reading/writing logic into dedicated helper methods.
	Combined logic in updateInventory that violates single responsibility principle.	Separate updateInventory into modular methods for inventory processing and file updates.
Coding Standards	Hardcoded strings like file paths and key names.	Define constants for all commonly used strings.
	Inconsistent error handling (direct printing in catch blocks).	Replace direct System.out.println calls with proper logging using a logging framework.
Performance Inefficiencies	Repeated list traversal for inventory lookup.	Cache databaseItem in a HashMap for efficient lookups.
	Direct string concatenation inside loops.	Use StringBuilder for string manipulations inside loops.

18. Java Code Review Checklist for Update_Employee interface

File name	Update_Employee interface.java
class/interface name	Update_Employee interface

Category	Checklist Item	Yes/No	Issue	Fix
Naming Conventions	Are class names written in PascalCase?	Yes	None	None
	Are variable and method names	Yes	None	None

	written in camelCase?			
	Are constants written in uppercase with underscores?	No	No constants used in the class.	Add constants for button sizes, window size, etc.
Code Structure	Are access modifiers used correctly?	Yes	None	None
	Are classes and interfaces separated?	Yes	None	None
	Are packages used appropriately?	No	Class is not part of any package.	Consider adding a package declaration.
Method Design	Do methods have a single responsibility?	Yes	None	None
	Are method parameters limited?	Yes	None	None
	Is method overloading used properly?	Not Applicable	No method overloading.	None
Exception Handling	Are exceptions handled with try-catch blocks?	No	No exception handling for invalid inputs.	Add exception handling for user inputs (e.g., username, password).
	Are specific exceptions used?	No	No specific exceptions are used.	Use specific exceptions where applicable (e.g.,

				InvalidUsernameException).
Code Readability	Are comments added for complex logic?	No	Some areas lack detailed comments.	Add comments for clarity, especially in logic for updating employee information.
	Is indentation consistent?	Yes	None	None
	Are blank lines used to separate code blocks?	Yes	None	None
	Are meaningful names used for variables, classes, and methods?	Yes	Names are meaningful.	None
Performance	Are data structures chosen based on performance?	No	No significant data structures involved.	None
	Are costly operations minimized in loops?	Not Applicable	No loops present that require optimization.	None
	Is lazy initialization used?	Not Applicable	No lazy initialization used.	None
Memory Management	Are unnecessary object references set to null?	No	No explicit memory management.	Consider nullifying unused references after use.

Security	Is user input validated?	No	Validation for user inputs is minimal.	Improve validation (e.g., for username, password).
Maintainability	Are there long methods or deeply nested loops?	No	None	None
	Is there duplicated code?	Yes	Duplicate logic for handling employee update.	Extract common logic into reusable methods.
	Are there any magic numbers?	Yes	Window size and button positions are hardcoded.	Replace magic numbers with constants.

Test Related Categories

Category	Checklist Item	Yes/No	Issue	Fix
Test Coverage	Are unit tests provided for all public methods and critical functionalities?	No	N/A	N/A
	Do unit tests cover edge cases and boundary values?	No	N/A	N/A
Test Design	Are tests written following the AAA pattern?	Not Applicable	N/A	N/A
	Are individual test cases independent?	Not Applicable	N/A	N/A
	Are descriptive names used for test methods?	Not Applicable	N/A	N/A
Assertions	Are assertions used to verify expected results?	Not Applicable	N/A	N/A
	Are specific assertions used instead of general ones?	Not Applicable	N/A	N/A

Boundary and Edge Cases	Are edge cases and boundary conditions tested?	No	N/A	N/A
	Are invalid inputs covered by tests?	No	N/A	N/A
Mocking and Stubbing	Are mocks or stubs used to isolate the unit under test?	No	N/A	N/A
Performance Testing	Are tests to check performance for critical methods provided?	No	N/A	N/A
Test Maintainability	Are test methods organized and modular?	Not Applicable	N/A	N/A
	Is there a setup method for initializing common objects?	Not Applicable	N/A	N/A

Housekeeping

Category	Checklist Item	Yes/No	Issue	Fix
Code Smells	Are there any code smells not covered by the checklist?	Yes	actionPerformed() method is large and handles multiple tasks.	Refactor actionPerformed() into smaller methods.
Coding Standards	Are there any coding standard violations not covered by the checklist?	No	None	None
Performance Inefficiencies	Are there any performance inefficiencies not covered by the checklist?	No	None	None

Refactoring Task 1: Extract Repeated Code for Interface Disposal into a Helper Method

- **Issue**: The logic for disposing of the interface after an action is completed is repeated in multiple places (for example, after updating employee details or exiting).
- **Refactoring Plan**:
    1. Create a helper method called `disposeInterface()`. This method should handle the logic of setting visibility to false and disposing of the interface.
    2. Replace the repeated code in the `actionPerformed()` method with calls to `disposeInterface()`.
    3. Update `actionPerformed()` to call `disposeInterface(this)` when needed.
- **Benefit**: Reduces redundant code, making the class easier to maintain. If the disposal logic needs to change, it can be done in one location rather than multiple places.

Refactoring Task 2: Replace Magic Numbers with Constants

- **Issue**: There are hardcoded values for window size, button positions, and component dimensions (e.g., button width, height, and position) scattered throughout the code.
- **Refactoring Plan**:

    1. Define constants at the top of the class to represent these magic numbers (e.g., `WINDOW_WIDTH`, `WINDOW_HEIGHT`, `BUTTON_WIDTH`, `BUTTON_HEIGHT`).
    2. Replace all instances of these hardcoded values in the layout setup with the defined constants.
    3. Replace all layout and button position logic with these constants.

- **Benefit**: Improves code readability and maintainability by giving meaningful names to numeric values. Changes to layout configuration can now be done in one place.

Refactoring Task 3: Break Down Large `actionPerformed()` Method into Smaller Methods

- **Issue**: The `actionPerformed()` method is lengthy and handles multiple responsibilities, such as handling button clicks, updating employee information, displaying error messages, and navigating to other interfaces.
- **Refactoring Plan**:

1. Extract the logic for each action into separate, well-named methods, such as:

    - `handleEnterButtonClick()`
    - `handleExitButtonClick()`
    - `navigateToAdminInterface()`
    - `showErrorMessage(String message)`

2. In the `actionPerformed()` method, delegate each action to its corresponding method based on the event source.

- **Benefit**: Enhances code readability, makes it easier to follow and debug, and supports better unit testing. Each method now has a clear, single responsibility, making the code easier to maintain and extend.

Appendix:

Code Smells

Code Smell	Description	Example	Suggested Solution
Long Methods	Methods that perform multiple tasks, making them hard to understand and maintain.	The updateInventory method combines inventory updating and file I/O logic.	Split updateInventory into smaller methods: one for inventory updating and another for file operations.
Nested Loops	Deeply nested loops reduce readability and increase complexity.	Nested looping over transactionItem and databaseItem.	Use a HashMap to cache databaseItem by itemID to minimize nested loops.
Duplicate Code	Repeated code for file handling and reading/writing inventory data.	File reading and writing logic appears in multiple places.	Extract file handling logic into reusable helper methods.
Magic Numbers/Strings	Hardcoded values or strings	Hardcoded file paths like "inventory.txt"	Define constants for file paths and string keys (e.g.,

	used without explanation.	and string keys like "ItemID".	FILE_PATH_INVENTORY, KEY_ITEM_ID).
Primitive Obsession	Overuse of primitive types instead of meaningful abstractions.	Using strings to represent itemID or transaction status instead of enums or classes.	Use an enum for transaction types and consider using a class for inventory items.

Performance Inefficiencies

Issue	Description	Example	Suggested Solution
Repeated Object Lookups	Scanning the entire databaseItem list for every transactionItem.	Iterating through the entire list for each transaction to find the matching item.	Use a HashMap<Integer, Item> to store databaseItem for constant-time lookups.
Redundant Calculations	Performing the same calculation repeatedly in a loop.	Repeated calls to getItemID() inside nested loops.	Store itemID in a local variable before the loop to minimize method calls.
Inefficient String Ops	Using string concatenation inside loops.	Using result += "Item: " + itemName in a loop.	Use StringBuilder for concatenating strings inside loops to improve performance.
Excessive Logging	Printing to the console excessively instead of structured logging.	Using System.out.println for error reporting in catch blocks.	Replace System.out.println with a proper logging framework like java.util.logging or Log4j.

Housekeeping

Category	Issue	Fix

Code Smells	Duplicate file handling code.	Extract file reading/writing logic into dedicated helper methods.
	Combined logic in updateInventory that violates single responsibility principle.	Separate updateInventory into modular methods for inventory processing and file updates.
Coding Standards	Hardcoded strings like file paths and key names.	Define constants for all commonly used strings.
	Inconsistent error handling (direct printing in catch blocks).	Replace direct System.out.println calls with proper logging using a logging framework.
Performance Inefficiencies	Repeated list traversal for inventory lookup.	Cache databaseItem in a HashMap for efficient lookups.
	Direct string concatenation inside loops.	Use StringBuilder for string manipulations inside loops.

19. Java Code Review Checklist for Employee

File name	Employee.java
class/interface name	Employee

Category	Checklist Item	Yes/No	Issue	Fix
Naming Conventions	Are class names written in PascalCase?	Yes	None	None
	Are variable and method names written in camelCase?	Yes	None	None
	Are constants written in uppercase with underscores?	No	No constants used in the class.	Add constants if required.
Code Structure	Are access modifiers used correctly?	Yes	None	None

	Are methods well-structured and not too long?	Yes	None	None
Method Design	Do methods have a single responsibility?	Yes	None	None
	Are method parameters limited to necessary inputs?	Yes	None	None
	Is method overloading used properly?	No	No overloading used.	None
Code Readability	Are comments added for complex logic?	No	No comments for methods.	Add comments for methods for clarity.
	Is indentation consistent?	Yes	None	None
	Are meaningful names used for variables, classes, and methods?	Yes	Names are meaningful.	None
Security	Are user inputs validated?	No	No input validation (e.g., username, password).	Add validation checks to ensure data integrity.
Maintainability	Are there long methods or deeply nested loops?	No	None	None
	Is there duplicated code?	No	None	None
	Are there any magic numbers?	No	No magic numbers in this class.	None

Test Related Categories

Category	Checklist Item	Yes/No	Issue	Fix
Test Coverage	Are unit tests provided for all public methods and critical functionalities?	Yes	Full coverage of getter and setter methods.	Add tests for edge cases.

	Do unit tests cover edge cases and boundary values?	No	Edge cases not tested (e.g., empty strings, null values).	Add tests for edge cases such as empty inputs.
Test Design	Are tests written following the AAA pattern?	Yes	Tests are written using AAA (Arrange, Act, Assert) pattern.	None
	Are individual test cases independent?	Yes	Tests are independent.	None
	Are descriptive names used for test methods?	Yes	Test method names are descriptive.	None
Assertions	Are assertions used to verify expected results?	Yes	Assertions are used in all test cases.	None
Boundary and Edge Cases	Are edge cases and boundary conditions tested?	No	Boundary conditions not tested (e.g., very long names, empty strings).	Add tests for boundary and edge cases.
	Are invalid inputs covered by tests?	Yes	Invalid input tests (e.g., wrong username) are covered.	None
Mocking and Stubbing	Are mocks or stubs used to isolate the unit under test?	No	Not applicable, as there is no external system interaction.	None
Test Maintainability	Are test methods organized and modular?	Yes	Test methods are organized.	None

Housekeeping

Category	Checklist Item	Yes/No	Issue	Fix
Code Smells	Are there any code smells not covered by the checklist?	No	None	None

Coding Standards	Are there any coding standard violations not covered by the checklist?	No	None	None
Performance Inefficiencies	Are there any performance inefficiencies not covered by the checklist?	No	None	None

Refactoring Task 1: Add Input Validation for Employee Attributes

- **Issue**: The `Employee` class lacks input validation for the attributes, particularly the username, password, and position fields.
- **Refactoring Plan**:
    1. Implement input validation for the `username`, `password`, and `position` fields to ensure that they meet specific criteria (e.g., non-empty, valid length, correct format).
    2. Update the constructor and setter methods to include validation checks before assigning the values to the fields.
    3. Throw custom exceptions (e.g., `InvalidUsernameException`, `InvalidPasswordException`) if the input doesn't meet the required conditions.
- **Benefit**: Adds data integrity and security to the application by preventing invalid or malicious input. It improves the robustness of the class and ensures that only valid data is accepted.

Refactoring Task 2: Consolidate Setter Methods

- **Issue**: The `Employee` class has separate setter methods for `name`, `position`, and `password`. While this is not inherently wrong, it could be simplified to make the code cleaner and more flexible.
- **Refactoring Plan**:

    1. Create a single setter method, `updateEmployeeInfo(String name, String position, String password)`, which updates the `name`, `position`, and `password` fields all at once.
    2. Replace the individual setter calls with this new consolidated method in places where multiple attributes need to be updated at once.

- **Benefit**: Simplifies the code and reduces the number of method calls, especially in scenarios where multiple employee attributes need to be updated simultaneously. It also improves maintainability, making it easier to update or expand in the future.

Refactoring Task 3: Extract Password Handling Logic

- **Issue**: The handling of the `password` attribute is done directly in the `Employee` class without any abstraction. This may cause issues if more complex password-related logic (e.g., encryption, validation) is introduced later.
- **Refactoring Plan**:

    1. Extract the password handling logic into a separate class, e.g., `PasswordManager`, which handles password validation and hashing.
    2. Modify the `Employee` class to delegate password-related logic to the `PasswordManager` class, such as validating or setting the password.
    3. Implement the `PasswordManager` to include methods like `isValidPassword(String password)` and `hashPassword(String password)`.

- **Benefit**: Improves maintainability and scalability by isolating password-related logic in a dedicated class. It also prepares the code for future enhancements (e.g., adding password encryption) without bloating the `Employee` class with concerns outside of its core responsibility.

Appendix:

Code Smells

Code Smell	Description	Example	Suggested Solution
Long Methods	Methods that perform multiple tasks, making them hard to understand and maintain.	The updateInventory method combines inventory updating and file I/O logic.	Split updateInventory into smaller methods: one for inventory updating and another for file operations.
Nested Loops	Deeply nested loops reduce readability and increase complexity.	Nested looping over transactionItem and databaseItem.	Use a HashMap to cache databaseItem by itemID to minimize nested loops.
Duplicate Code	Repeated code for file handling and	File reading and writing logic	Extract file handling logic into reusable helper methods.

	reading/writing inventory data.	appears in multiple places.	
Magic Numbers/Strings	Hardcoded values or strings used without explanation.	Hardcoded file paths like "inventory.txt" and string keys like "ItemID".	Define constants for file paths and string keys (e.g., FILE_PATH_INVENTORY, KEY_ITEM_ID).
Primitive Obsession	Overuse of primitive types instead of meaningful abstractions.	Using strings to represent itemID or transaction status instead of enums or classes.	Use an enum for transaction types and consider using a class for inventory items.

Performance Inefficiencies

Issue	Description	Example	Suggested Solution
Repeated Object Lookups	Scanning the entire databaseItem list for every transactionItem.	Iterating through the entire list for each transaction to find the matching item.	Use a HashMap<Integer, Item> to store databaseItem for constant-time lookups.
Redundant Calculations	Performing the same calculation repeatedly in a loop.	Repeated calls to getItemID() inside nested loops.	Store itemID in a local variable before the loop to minimize method calls.
Inefficient String Ops	Using string concatenation inside loops.	Using result += "Item: " + itemName in a loop.	Use StringBuilder for concatenating strings inside loops to improve performance.
Excessive Logging	Printing to the console excessively instead of structured logging.	Using System.out.println for error reporting in catch blocks.	Replace System.out.println with a proper logging framework like java.util.logging or Log4j.

Housekeeping

Category	Issue	Fix
Code Smells	Duplicate file handling code.	Extract file reading/writing logic into dedicated helper methods.
	Combined logic in updateInventory that violates single responsibility principle.	Separate updateInventory into modular methods for inventory processing and file updates.
Coding Standards	Hardcoded strings like file paths and key names.	Define constants for all commonly used strings.
	Inconsistent error handling (direct printing in catch blocks).	Replace direct System.out.println calls with proper logging using a logging framework.
Performance Inefficiencies	Repeated list traversal for inventory lookup.	Cache databaseItem in a HashMap for efficient lookups.
	Direct string concatenation inside loops.	Use StringBuilder for string manipulations inside loops.