SYOS-POS System CCCP1 & CCCP2 Blueprint

Comprehensive MVC Architecture with SOLID Principles, Design Patterns, Separate Normal and Online Customers, LKR Currency, and Thread-Safe Singleton Connection Pooling

---

1. Assignment Context & Objectives

1.1 Overview

- Purpose: Automate billing, stock management, and reporting for Synex Outlet Store (SYOS).
- Phases:

1. CCCP1: Console-based Java application with MySQL backend.
2. CCCP2: Multi-user, multi-tier client-server system with GUI, concurrency, and multiple payment methods.

- Currency: Sri Lankan Rupees (LKR).
- Customer Types: Separate entities and database tables for normal (in-store) and online customers.
- Quality: Follow clean code, SOLID principles, design patterns, and prepare for CCCP2.

1.2 Deliverables

- Fully functional console application.
- Comprehensive unit and integration tests.
- Detailed design and implementation report.
- Clean, maintainable, extensible codebase ready for CCCP2.

---

2. Architectural Design: MVC with Clean Architecture and CCCP2 Preparation

2.1 MVC Pattern

- Model: Business logic, domain entities, and persistence abstractions.
- View: Console UI for CCCP1; designed for easy replacement by GUI/web UI.
- Controller: Mediates input/output and invokes application services.

2.2 Clean Architecture Layers

- Domain Layer: Core entities, value objects, enums, domain services encapsulating business rules.
- Application Layer: Use case services (facades), repository and service interfaces defining contracts.
- Infrastructure Layer: JDBC DAOs, utilities, configuration, transaction management, and connection pooling.

2.3 Dependency Rules

- Dependencies flow inward: Infrastructure → Application → Domain.
- Model independent of View and Controller.
- Interfaces invert dependencies for flexibility and testability.

---

3. Database Design: Separate Tables for

Table Name	Purpose	Key Fields Example
customer	Normal (in-store) customers	customer_id, name, phone
online_customer	Online customers for website sales	online_customer_id, name, email, address, password_hash

Normal and
4. Online Customers

- Bills reference either table based on transaction type (COUNTER or ONLINE).
- Justification:
- Different data requirements and security needs.
- Performance optimization.
- Clear domain separation.
- Facilitates CCCP2 web integration.

---

4. Domain Model: Entities and Value Objects

4.1 Entities

-

Item:
Fields: itemId, name, code, unitPrice (Money), discount, reorderLevel.

•
•

Customer:
Normal customers with customerId, name, phone.

•
•

OnlineCustomer:
Online customers with onlineCustomerId, name, email, address, passwordHash.

•
•

User :
System users with roles (Cashier, Manager, Admin, OnlineCustomer).

•
•

Bill:
Fields: billId, serialNumber, date, customer reference, transaction type, list of BillItems, totals, payment method.

•

BillItem:
Line items in a bill, linking to Item and quantity.

•
•

StockBatch, ShelfStock, WebsiteInventory:
Represent stock management entities.

•

4.2 Value Objects

•

Money:
Immutable wrapper around BigDecimal representing amounts in Sri Lankan Rupees (LKR). Provides precise arithmetic and comparison operations.

•
•

Quantity:
Immutable wrapper for item quantities.

•

Password:
Encapsulates hashed password strings.

ItemCode:
Immutable item code.

4.3 Enums

- User Role (CASHIER, MANAGER, ADMIN, ONLINE_CUSTOMER)
- TransactionType (COUNTER, ONLINE)
- PaymentMethod (CASH, CREDIT_CARD, PAYPAL)

---

5. Application Layer: Use Case Services and Interfaces

5.1 Repository Interfaces

- User Repository
- ItemRepository
- CustomerRepository (normal customers)
- OnlineCustomerRepository (online customers)
- BillRepository
- StockBatchRepository
- ShelfStockRepository
- WebsiteInventoryRepository

5.2 External Service Interfaces

- BillPrinter
- Logger
- SerialNumberGenerator
- PaymentGateway

5.3 Use Case Services (Facades)

- AuthenticationAppService: Handles login and user management.

- BillingAppService: Processes sales, generates bills, updates stock, handles payments.
- ItemAppService: Manages item CRUD operations.
- StockAppService: Manages stock receiving, shelving, reorder alerts.
- ReportAppService: Generates sales and stock reports.
- CustomerAppService: Manages normal customer registration and retrieval.
- OnlineCustomerAppService: Manages online customer registration, authentication, and profile management.
- WebsiteSalesAppService: Handles online order processing and payment.

5.4 Design Notes

- Services are stateless and thread-safe.
- Use constructor injection for dependencies.
- Wrap multi-step DB operations in TransactionManager.
- Use DTOs for data transfer and future client-server communication.

---

6. Infrastructure Layer: Persistence, Utilities, and Database Connection Management

6.1 Persistence Implementations

- JDBC DAO classes implementing repository interfaces.
- Separate DAOs for Customer and OnlineCustomer.
- Proper resource management and exception handling using try-with-resources.

6.2 Utilities

- PasswordHasher for secure password hashing and verification.
- ValidationUtil for input validation.

6.3 Configuration Management

- Implement a ConfigManager class responsible for loading database connection parameters from an external application.properties file.
- Parameters include:
- db.url (e.g., jdbc:mysql://localhost:3306/syos_pos)
- db.username
- db.password
- db.driver (e.g., com.mysql.cj.jdbc.Driver)
- This externalization allows easy environment changes without code modification.

6.4 Server-Side Database Connection Management with Thread-Safe Singleton Connection Pool

6.4.1 Overview

- To efficiently handle multiple simultaneous servlet requests (e.g., in Apache Tomcat 10.1), the system uses a Singleton DBConnection class managing a fixed-size connection pool.
- The pool uses a thread-safe LinkedBlockingQueue to hold reusable MySQL connections.
- This design supports concurrent transactions by multiple threads, minimizing connection creation overhead and preventing resource exhaustion.
- Connections are acquired via getConnection() and released via releaseConnection().
- Adapters (e.g., BillManagementSQLAdapter) acquire connections from the pool and release them after use to prevent leaks.
- Tomcat's thread pool manages servlet request threads, enabling parallel processing.

6.4.2 Design Details

Singleton Pattern:
Ensures a single instance of DBConnection manages all database connections, centralizing control and avoiding conflicts.

Connection Pool:

- Fixed size (e.g., 5 connections) configurable.
- Uses LinkedBlockingQueue<Connection> for thread-safe, blocking access.
- Connections are created once during initialization.
- getConnection() blocks if no connections are available until one is released.
- releaseConnection(Connection) returns connection to the pool and notifies waiting threads.

Thread Safety:

- LinkedBlockingQueue provides built-in thread safety.
- getInstance() method uses double-checked locking to ensure singleton.
- Adapters use JDBC transactions to ensure atomicity.
- HttpSession isolates user-specific data, preventing cross-session conflicts.

6.4.3 Example Skeleton Code

java50 lines
Copy codeDownload code
Click to expand

```
public class DBConnection {
private static volatile DBConnection instance;
...
```

6.4.4 Usage in Adapters and Services

- Adapters call DBConnection.getInstance().getConnection() to acquire a connection.
- After completing DB operations and committing/rolling back transactions, adapters call releaseConnection() to return the connection.
- Use try-finally blocks to guarantee connection release even on exceptions.
- Example:

java11 lines
Copy codeDownload code
Click to expand

```
Connection conn = null;
try {
...
```

6.5 Transaction Management

- TransactionManager class wraps transaction control on connections.
- Application services:

1. Obtain connection from DBConnection pool.
2. Begin transaction (conn.setAutoCommit(false)).
3. Perform DAO operations.
4. Commit if successful.
5. Rollback on failure.
6. Release connection back to pool.

6.6 External Services

- ConsoleBillPrinter
- SimpleConsoleLogger
- UUIDSerialNumberGenerator
- MockPaymentGateway

7. View Layer: Console UI

- Separate views per user role.
- Minimal logic; input/output handling.
- Observer pattern for model updates.
- Designed for easy GUI/web UI replacement.

8. Controller Layer

- Controllers per user role.
- Handle input, invoke services, update views.
- Designed for easy refactoring into Servlet controllers.
- Optional Command pattern for UI actions and concurrency.

9. SOLID Principles Applied

9.1. Single Responsibility Principle (SRP)

- Each class has one responsibility.
- Example: Money only handles monetary values and operations.

9.2. Open/Closed Principle (OCP)

- Classes open for extension, closed for modification.
- Example: New payment methods added by implementing PaymentGateway interface.

9.3. Liskov Substitution Principle (LSP)

- Subtypes replace base types without altering correctness.
- Example: OnlineCustomerRepository and CustomerRepository implement common interfaces.

9.4. Interface Segregation Principle (ISP)

- Clients depend only on interfaces they use.
- Separate repository interfaces for different entities.

9.5. Dependency Inversion Principle (DIP)

- High-level modules depend on abstractions.
- Application services depend on repository interfaces, not concrete implementations.
- Constructor injection supports this.

---

10. Design Patterns Used

- MVC: Separation of concerns.
- Facade: Application services simplify workflows.
- Repository: Abstract data access.
- Singleton: For config and DB connection management.
- Strategy: For discount and shelving algorithms.
- Builder: For complex bill construction.
- Observer: Views observe model changes.
- Command: Encapsulate UI actions for concurrency.
- Value Object: Domain primitives like Money.
- Dependency Injection: Constructor injection for flexibility.

---

11. Thread-Safety Measures

- DBConnection singleton with synchronized double-checked locking.
- LinkedBlockingQueue ensures thread-safe connection pooling.
- JDBC transactions in adapters ensure atomic updates.
- HttpSession isolates user-specific data.
- Adapters close/release connections properly to prevent leaks.
- Tomcat thread pool manages concurrent servlet requests efficiently.

---

12. Transaction Management & Atomicity

- Centralized TransactionManager manages JDBC transactions.
- Application services wrap multi-step DB operations.

- Ensures data consistency and concurrency readiness.
- Proper connection and resource management to avoid leaks.

---

13. Testing Strategy

- Unit tests with JUnit 5 and Mockito.
- Integration tests with H2 and MySQL.
- Manual acceptance tests.
- Tests follow F.I.R.S.T principles.

---

14. Project Setup & Build

- Maven project.
- Dependencies: JUnit 5, Mockito, MySQL Connector/J, H2, SLF4J, Logback.
- schema.sql with separate customer tables.
- application.properties example:

```
RunCopy code
db.url=jdbc:mysql://localhost:3306/syos_posdb.username=your_usernamedb.password=your_passw
orddb.driver=com.mysql.cj.jdbc.Driver
```

- logback.xml for logging configuration.

---

15. Implementation Steps

1. Setup Maven and package structure.
2. Create MySQL schema with separate customer tables.
3. Implement ConfigManager to load DB config.
4. Implement thread-safe singleton DBConnection with LinkedBlockingQueue connection pool.
5. Implement TransactionManager.
6. Implement Domain Layer entities and value objects.
7. Define Application Layer interfaces and use case services.
8. Implement Infrastructure Layer with JDBC DAOs using pooled connections and try-with-resources.
9. Implement Views and Controllers.

10. Write unit and integration tests.
11. Document design decisions and testing.

---

16. Value Object: Money (Sri Lankan Rupees – LKR)

- Immutable wrapper around BigDecimal.
- Represents monetary amounts in LKR only.
- Provides precise arithmetic and comparison methods.
- Used consistently for prices, totals, payments, and change.

---

17. Summary

This blueprint ensures:

- Robust, efficient, and thread-safe database connection management using a Singleton DBConnection class with a fixed-size LinkedBlockingQueue connection pool.
- Proper resource handling with try-with-resources and guaranteed connection release.
- Clear separation of normal and online customers with dedicated entities and tables.
- Strict adherence to SOLID principles ensuring maintainability and extensibility.
- Comprehensive use of design patterns for clean, modular, and testable code.
- Full MVC architecture with clean layering and dependency inversion.
- Extensive testing strategy for quality assurance.
- Preparation for CCCP2 multi-user, multi-tier, GUI, concurrency, and multiple payment methods.
- A maintainable, extensible, and testable codebase ready for future enhancements.