Reverse Engineering Analysis

Legacy SG Technologies POS System

Senior Reengineering Team

2025–12–04

Comprehensive analysis of architecture, design patterns, code/data smells, and reengineering recommendations for the legacy Java Swing + flat-file POS system.

Executive Summary

This document presents a complete reverse engineering of the legacy SG Technologies Point of Sale system. The analysis covers architectural structure, design patterns, data storage, code and data smells, workflows, security issues, and actionable reengineering recommendations.

1 System Overview

The system is a **desktop Java Swing application** with **file-based persistence** supporting:

- Direct item sales with tax and coupon support
- Item rentals tracked by customer phone number
- Return processing for rented items
- Employee management (Admin/Cashier roles)
- Real-time inventory tracking

2 High-Level Architecture Diagram

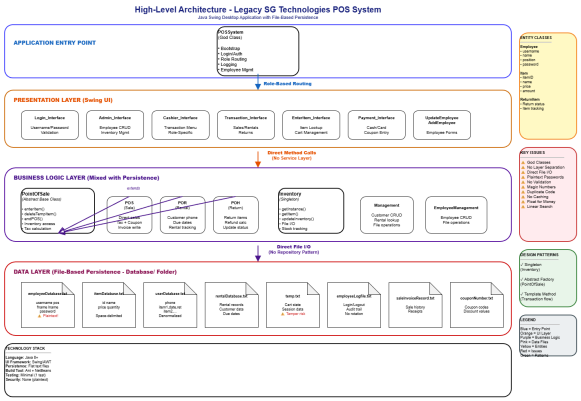

Figure 1: **High-Level Architecture of Legacy POS System**
Showing POSSystem as entry point, role-based routing, inheritance hierarchy, and Singleton inventory access.

Note: Replace `architecture-diagram.png` with your actual diagram file (PNG, PDF, or JPG). Recommended tools: draw.io, Lucidchart, PlantUML, or Visio.

3 Class Hierarchy & Key Components

Core Domain Classes

- `Employee`: username, name, position, **plaintext password**
- `Item`: itemID, name, price, quantity
- `ReturnItem`: tracks return status

Transaction Hierarchy

`PointOfSale` (abstract base)

- `POS` → Direct sales
- `POR` → Rentals (with phone number)
- `POH` → Returns

Key Classes

- `POSSystem`: God class — login, routing, logging
- `Inventory`: Singleton — file-based access
- `Management`: Customer/rental operations
- `EmployeeManagement`: Employee CRUD

4 Design Patterns Identified

Pattern	Implementation
Singleton	`Inventory.getInstance()`
Abstract Factory	`PointOfSale` abstract with concrete POS/POR/POH
Template Method	`PointOfSale` defines transaction skeleton (`enterItem`, `endPOS`)

5 Data Storage – File-Based (Database/ Folder)

File	Format	Issues
`employeeDatabase.txt`	`username pos fname lname password`	Plaintext passwords
`itemDatabase.txt`	`id name price quantity`	Space-delimited
`userDatabase.txt`	`phone item1,date,ret item2,...`	Denormalized
`temp.txt`	Temporary cart state	Tamper-prone
`employeeLogfile.txt`	Login/logout audit trail	No rotation

6 ⚠ Code Smells Detected (Top 10)

⚠ God Class (`POSSystem`, `PointOfSale`, `Management`)
⚠ Long Method (>50 lines in critical paths)
⚠ Duplicate Code (`deleteTempItem()` ×3)
⚠ Magic Numbers (tax=1.06, discount=0.90f)
⚠ Primitive Obsession (phone as `long`, money as `float`)
⚠ Feature Envy (UI does file I/O)
⚠ Data Clumps (file paths, OS checks)
⚠ Inappropriate Intimacy (direct Inventory access)
⚠ Swallowed Exceptions
⚠ Comments admitting bad code

7 ⚠ Data Smells Detected

- ⚠ No normalization — rental history embedded in user line
- ⚠ No validation — negative stock possible?
- ⚠ Inconsistent formats across files
- ⚠ Duplicate data (`rentalDatabase.txt` mirrors inventory)
- ⚠ No referential integrity

8 Security Vulnerabilities

- Plaintext passwords in `employeeDatabase.txt`
- No input validation — potential file path tampering
- Temp files in predictable location — session hijacking risk
- No session timeout or proper logout
- All-or-nothing file system permissions

9 Performance Issues

- Full file read on every operation
- Linear search through lists
- No caching — repeated disk access
- Synchronous blocking I/O

10 Testing Status

Current Test Coverage

- Only 1 test: `EmployeeTest.getUsername()`
- Estimated coverage: $< 5\%$
- No tests for transactions, inventory, or file I/O

11 Recommendations for Reengineering

11.1 Immediate Refactoring (Safe, No Behavior Change)

1. Extract all magic numbers → `Constants` class
2. Extract OS detection → `SystemUtils`
3. Consolidate duplicate `deleteTempItem()` methods
4. Extract file paths into constants
5. Add characterization tests before major changes

11.2 Medium-Term Architectural Goals

1. Introduce layered architecture (UI → Service → Repository)
2. Migrate to PostgreSQL with normalized schema
3. Replace `float` → `BigDecimal` for money
4. Implement proper authentication (BCrypt + JWT)
5. Add comprehensive unit + integration tests

11.3 Target Modern Stack

- **Backend**: Spring Boot + JPA
- **Frontend**: React + TypeScript
- **Database**: PostgreSQL
- **Security**: Spring Security + BCrypt
- **Testing**: JUnit 5 + React Testing Library

Document Version	1.0
Date	2025–11–28
Analyst	Senior Reengineering Team
Status	Complete – Ready for Refactoring Phase