OOP Project Sprint 1 & 2 Outline and Contribution Report

PlayOps - A retro videogame & gaming hardware retail logistics solution
BSc in Artificial Intelligence

Assigning / Accepting Professor: Prof. Evis Plaku
December 15, 2025

Contents

1 Project Overview

PlayOps is a logistics solution designed for retailers focused on the selling and renting of video games, as well as the sale of gaming hardware. It allows operators to add, remove, search, display, and manage customers, transactions, and games.

PlayOps strives to align with OOP principles, taking pride in its development process and maintaining crucial awareness to principles, such as those of:

- **Sprint 1**: Fundamental object oriented design, management through arrays/lists encapsulation, modular design and data abstraction.

- **Sprint 2**: Inheritance based - interface compliant design, polymorphic behavior, encapsulation with access modifiers, pre-implemented and custom exception handling, file persistence.

2 UML Class Diagram

The UML diagram explains the structure and relationships of the components within this project.
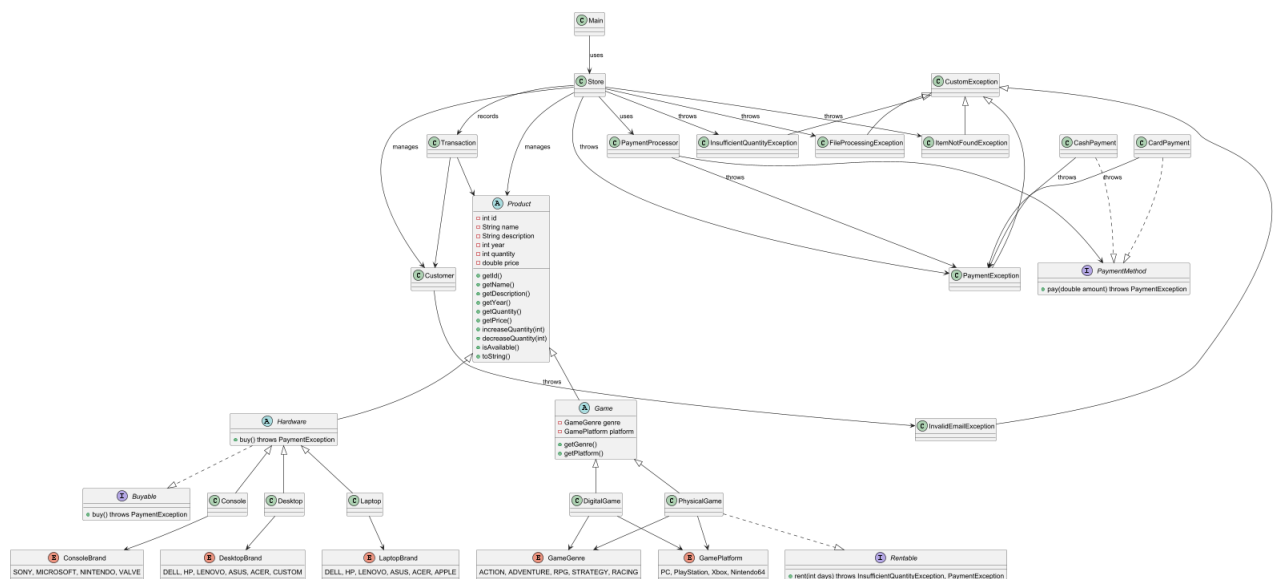
3 Sprint 1

3.1 Project Structure and Design

The project follows a modular object-oriented structure:

- **com.playops.model.Customer.java** — Manages customer data and an instance of validation in the email entries.

- **com.playops.model.Game.java** — Represents the game properties and their attributes.
- **com.playops.store.Store.java** — Manages lists of customers and games.
- **com.playops.app.Main.java** — Entry point that handles user interaction.

Each class was designed with primary OOP philosophy of dividing work into classes with reusability and user convenience.

4 Sprint 2

4.1 Improved Project Structure and Design

Project design has relied on collaborative work heavily utilizing git.

The updated structure altered the foundations in a way that honors the simplistic approach with few classes that have now expanded to a plethora of specialized single-purpose classes.

The project utilizes the Product, Customer, and Transaction classes, all intertwined in the connecting Store class to do most of the heavy lifting, assisted by the interfaces and exceptions that make possible the seamless integration of methods catering to necessary functionality. Below is an ASCII representation of the logical implementation.

```
com.playops
|
+-- app
|   +-- Main
|       - Entry point of the application
|       - Uses Store, Customers, Products, PaymentMethods, Transactions
|
+-- exceptions
|   +-- CustomException
|   +-- FileProcessingException
|   +-- InsufficientQuantityException
|   +-- InvalidEmailException
|   +-- ItemNotFoundException
|   +-- PaymentException
|       - Used by PaymentProcessor, Store, and PaymentMethod implementations
|
+-- interfaces
|   +-- Buyable
|   |   - Implemented by Product/Game classes
|   +-- PaymentMethod
|   |   - Implemented by CardPayment, CashPayment
|   +-- Rentable
|       - Implemented by PhysicalGame
|
+-- model
```

```
|   +-- Product
|   |   - Superclass of Hardware and Game
|   |   - Implements Buyable
|   |
|   +-- Hardware
|   |   +-- Console
|   |   +-- Desktop
|   |   +-- Laptop
|   |
|   +-- Game
|   |   +-- DigitalGame
|   |   +-- PhysicalGame (also implements Rentable)
|   |
|   +-- Customer
|   |   - Used by Store and Transaction
|   |
|   +-- Transaction
|       - References Product and Customer
|       - Stored in Store
|
+-- payment
|   +-- PaymentProcessor
|   |   - Processes PaymentMethod (CashPayment, CardPayment)
|   +-- CardPayment
|   +-- CashPayment
|
+-- store
    +-- Store
        - Holds ArrayLists of Products, Customers, Transactions
        - Performs buyProduct, rentGame, add/remove/display operations
        - Uses PaymentProcessor to handle payments
```

4.2 Inheritance, Polymorphism, Interfaces

The project architecture heavily relies on inheritance. The most evident example is that of the PhysicalGame, DigitalGame, Laptop, Desktop, Console classes. All these classes stem from a common abstract class, Product. Product provides the ambiguous definitions that all products have, and extends to a second layer of abstraction where the abstract classes of Game and Hardware lay the foundations for the former pair of atomic classes.

Polymorphism and Interfaces work hand-in-hand, where polymorphism takes shape in the overriding of methods in classes implementing interfaces, populating fields demanded by the latter.

4.3 Exception Handling

Exceptions exist, and the way we handle them matters more and more. It is our weaknesses that make us stronger, and our response to stress, anxiety, panic and turmoil are what dictates if we stand strong in the real world. Similarly, by utilizing the exceptions

class that enables the seamless interruption of an action that violates a constraint, we are able to have a logic smarter than checking values with if conditions. To access this powerful utilization we use try-catch blocks. By providing our expected runtime scenario in the try section, and the exceptions that may occur in the catch sections, we create an environment where all conditions that would violate a logical semantic operation are caught and tended to.

A few examples of situations where the program catches exceptions and recovers can be found below. If these exceptions were to not be caught, in most cases, the default response from the JVM is to terminate.

- **Missing Files:** If say, transactions.txt doesn't exist, the message "No transactions file found. Starting fresh." is printed and the program proceeds to run.

- **Invalid Input:** When a user tries to add a customer with an invalid email, the app prints the error but does not terminate.

- **Payment Failures:** If a payment fails (usually insufficient payment), the app prints a failure message, and no transaction is recorded.

- **Out of Stock:** If a product is unavailable, the purchase/rental is blocked, and the user is informed.

5 Team Roles and Contributions

Name	Responsibilities and Contributions
Blevis Allushi	Co-author. Designed overall class architecture and populated most helper classes such as Customer, Transaction and Product as well as critical behavior in the (Store) class. Ensured key methods were present and validated. Implemented restrictions and exception handling. UML modelling and documentation.
Kristjan Seraj	Co-author. Designed file I/O architecture and implemented proper logic and restrictions regarding the successful behavior. Populated and designed Main. Debugging and refinement. Ensured no errors propagated beyond main. Polishing work in UML and documentation.

6 Reflection

The second sprint possessed a much more interesting appeal. It encompassed more advanced OOP principles that kept the design process interesting and creative. The foundations laid in Sprint 1 paid off and were a testament to how Object Oriented Programming as a whole works and builds upon itself, as evidenced by Sprint 2.