**Bouncy Sports Equipment’s Inc**

**Object Design Document**

Dmytro Poliak

Student Number: 100443432

Date: 13.11.2024

1. **Introduction**

This document provides a detailed object-level design for the system that Bouncy Sports Equipment Inc. will use to monitor and control the number of customers in their stores. The system aims to meet with customer count limitations by using simulated sensor inputs to detect entries and exits, adjusting barrier controls accordingly, and allowing management to oversee and modify the store’s maximum occupancy. This design supports the functional requirements outlined in the Requirements Analysis Document (RAD).

* 1. **Object Design Trade-offs**

Here are several trade-offs in the design process to balance functionality, efficiency, and simplicity:

1. Simulation vs. Real Sensors: Given the console-based nature of this project, actual hardware sensors are replaced by simulated input methods. This approach allows the program to function independently of physical hardware and makes testing accessible in a non-store environment. Entry and exit events are triggered by user input, simplifying the development and demonstration of system functionality.
2. Command-Line Interface (CLI): To keep the design lightweight and efficient, a command-line interface was chosen instead of a graphical interface. This decision aligns with the requirements for simplicity and ease of testing, as well as memory efficiency. Additionally, it allows a clear focus on the underlying logic without the complexity of graphical elements.
3. Modular Design for Maintainability: The system is structured with distinct modules for each main component (e.g., PeopleCounter, Sensor, Barrier, SimulationManager). This modular approach enables each component to operate independently yet integrate seamlessly, making future updates or component replacements easier if Bouncy Sports Equipment Inc. decides to expand the system.
4. Observer and Strategy Patterns for Flexibility: The Observer and Strategy design patterns were selected to manage system complexity and enhance flexibility. The Observer pattern enables real-time updates between components, such as the PeopleCounter and Barrier, allowing the barrier status to react dynamically to changes in customer count. The Strategy pattern supports flexible authentication mechanisms, enabling StorePersonnel to bypass barriers through a card-based authentication strategy.
   1. **Interface Documentation Guidelines**

To ensure clarity and consistency in the design, the following conventions are adopted for class interfaces:

* Naming Conventions: Classes are named with singular nouns, aligning with their function (e.g., PeopleCounter, Barrier, Sensor). Methods are named with verb phrases that describe their actions, such as incrementCount() for adding to the customer count and open() for changing the barrier status.
* Error Handling: Exceptions are used to handle errors and manage control flow, especially for boundary conditions (e.g., attempting to increment customer count beyond capacity). This approach simplifies error management and makes debugging easier in a console environment.
* Method Standardization: For collections, methods like getElements() return an iterator type to handle multiple entries (like personnel or customers) effectively. This allows safe and smooth iteration, even if the collection changes during use.
  1. **Definitions, Acronyms, and Abbreviations**
* CLI: Command-Line Interface
* RAD: Requirements Analysis Document
* ODD: Object Design Document
* PeopleCounter: The component responsible for tracking and controlling customer count in the store.
* Barrier: The physical or simulated mechanism that controls entry based on current capacity.
* Sensor: The component detecting customer entry or exit, simulated through user input.
  1. **References**
* Textbook Reference: Bruegge, B., & Dutoit, A. H. (2010). Object-Oriented Software Engineering Using UML, Patterns, and Java. Prentice Hall.
* <https://www.cs.fsu.edu/~lacher/courses/COP3331/rad.html>
* <https://personal.utdallas.edu/~chung/SP/ObjectDesignDocumentTemplate.htm>
* <https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://greg4cr.github.io/courses/fall15csce740/Documents/DesignTemplate.doc&ved=2ahUKEwis0KT349WJAxXYJDQIHQz_EdcQFnoECBUQAQ&usg=AOvVaw2Ztjs5NzXTVLNHJ6b6zMk1>

1. **Packages**

Each package groups related classes based on their functionality and responsibilities.

* 1. **Package Overview**

1. core

* Purpose: This package includes all the primary classes that manage core functionality, such as customer counting, barriers, and simulation. By grouping key functionality together, we keep the package structure manageable while maintaining logical separation of components.
* Classes:
  + PeopleCounter: Tracks customer count and manages store capacity.
  + Barrier: Controls customer access based on the count.
  + SimulationManager: Coordinates entry and exit simulations.
  + Sensor: Detects entry and exit events (simulated).

1. management

* Purpose: Contains classes that allow store managers to monitor and adjust customer limits. This package is focused on providing a higher-level interface for viewing and setting store capacity.
* Classes:
  + StoreManager: Allows management to view and adjust store capacity limits.

1. authentication

* Purpose: Manages personnel authentication using the Strategy pattern, allowing store personnel to bypass barriers when authenticated. This package is small but maintains clear boundaries for authentication functionality.
* Classes:
  + StorePersonnel: Represents employees who need special access.
  + AuthenticationStrategy: Interface for different authentication methods.
  + CardAuthentication: Implements card-based authentication.
  1. **Package Dependencies and Usage**

1. core

* Description: This package contains all essential classes that manage the system’s primary functions, including customer counting, barrier control, and entry/exit simulation.
* Internal Classes and Usage:
  + PeopleCounter: Manages the count of customers in the store and checks against the maximum capacity.
  + Barrier: Controls access to the store based on the customer count from PeopleCounter.
  + SimulationManager: Coordinates entry and exit simulation and interacts with Sensor to simulate customer actions.
  + Sensor: Detects entry and exit events (through simulation).
* External Dependencies:
  + None. This package is self-contained, handling the core customer management functionality internally.

1. management

* Description: The management package provides store management functionality, allowing the store manager to view the current customer count and set a maximum capacity limit.
* Internal Classes and Usage:
  + StoreManager: Enables management to monitor customer count and adjust the capacity limit.
* Dependencies:
  + Depends on core for retrieving the customer count and updating the capacity limit via the PeopleCounter class.

1. authentication

* Description: This package manages personnel authentication using the Strategy pattern, allowing authorized personnel to bypass barriers.
* Internal Classes and Usage:
  + StorePersonnel: Represents employees who need store access.
  + AuthenticationStrategy: Interface for various authentication methods.
  + CardAuthentication: A specific implementation for card-based authentication.
* Dependencies:
  + Uses Barrier from core to allow personnel access once authenticated.

**Summary of Package Interactions**

* core: Contains primary classes and doesn’t depend on other packages.
* management: Uses core to access and modify the customer count.
* authentication: Uses core for controlling personnel access based on authentication status.
  1. **Package Diagram**

A screenshot of a computer

Description automatically generated

1. **Class Interfaces**

**3.1 Class Descriptions and Interfaces**

Each class description includes:

* Public Attributes and Operations: Methods exposed to other classes or packages.
* Dependencies: How classes interact with each other.
* Constraints and Contracts: Preconditions, postconditions, and invariants to maintain consistency.

Core Classes

1. PeopleCounter

* Attributes:
  + currentCount: int – Tracks the number of customers in the store.
  + maxCapacity: int – Defines the store’s capacity limit.
* Operations:
  + incrementCount(): Adds to currentCount if under maxCapacity.
    - Precondition: currentCount < maxCapacity
    - Postcondition: currentCount increased by 1.
  + decrementCount(): Reduces currentCount.
    - Precondition: currentCount > 0
    - Postcondition: currentCount decreased by 1.
* Dependencies: Observes changes to Barrier to control store access.
* Exceptions: Throws CapacityExceededException if maxCapacity is exceeded.

1. Barrier (Observer)

* Attributes:
  + status: BarrierStatus – Tracks if barriers are Open or Closed.
* Operations:
  + updateStatus(): void – Updates status based on PeopleCounter data.
* Dependencies: Observes PeopleCounter to respond to currentCount changes.

1. Sensor (Strategy)

* Attributes:
  + sensorType: SensorType – Specifies if sensor is entry or exit.
* Operations:
  + detectPerson(): Simulates person detection, calling incrementCount or decrementCount.
* Dependencies: Uses PeopleCounter to adjust customer count based on detection.

1. AuthenticationStrategy (Strategy)

* Attributes:
  + Abstracts different types of authentication (e.g., CardAuthentication).
* Operations:
  + authenticatePersonnel(id: String): boolean – Verifies personnel ID.
* Dependencies: Used by StorePersonnel.

1. StoreManager

* Attributes:
  + currentLimit: int – Sets and retrieves in-store customer limit.
* Operations:
  + setCapacity(limit: int): void – Updates maxCapacity in PeopleCounter.
    - Precondition: limit > 0
    - Postcondition: maxCapacity set to limit.
* Dependencies: Interacts with PeopleCounter to adjust maxCapacity.