Justification for EasyParkPlus Code Review

Software Design and Architecture Project

October, 2025

Project Overview

The project was presented with a preliminary prototype, and a pattern to improve it was made by identifying the anti-patterns and removing them, and designing a software architecture.

Project description

The project involved analysing, updating, and extending an existing codebase to improve its structure, maintainability, and overall quality. This project reflects a common industry scenario where engineers work primarily with legacy or prototype systems rather than building applications from scratch. The ability to understand and evolve such systems is essential for long-term software sustainability.

To address recurring design challenges, targeted design patterns that align with the system's needs were implemented:

- **Structural patterns** were applied to improve modularity and reusability.
- **Architectural patterns** helped eliminate duplication and streamline component interaction.
- **Behavioral patterns** simplified object responsibilities and enhanced clarity in control flow.

Each pattern was selected based on its relevance to the problem domain and justified through its impact on code clarity, extensibility, and performance.

The refactored codebase now demonstrates improved cohesion, reduced complexity, and better alignment with object-oriented principles. The changes are not merely cosmetic but represent meaningful architectural enhancements. By applying design patterns judiciously and removing anti-patterns, the system is now more robust and easier to maintain.

This project underscores the importance of thoughtful design in software engineering and the value of continuous improvement when working with evolving codebases.

Justification for Change

Steps Taken

- Code Review & Anti-pattern Identification
  - Analyzed the original codebase for maintainability, scalability, and adherence to OOP principles.
  - Identified key anti-patterns (see below).
- MVC Refactor
  - Separated concerns into Models (models/), Controllers (controllers/), and Views (views/).
  - Moved business logic from UI and data classes into controllers.
  - Created clear interfaces between layers.
- Design Pattern Application
  - Applied appropriate OOP and architectural patterns to improve code quality and extensibility (see below).
- Documentation & Justification
  - Documented the rationale for changes, including design decisions and pattern usage.
  - Provided markdown summaries for pull requests and design justifications.
- Error Resolution & Import Fixes
  - Addressed module import errors by adjusting package structure and relative imports.
- Behavioral Documentation
  - Generated PlantUML sequence diagrams to visualise system behaviour.

Design Patterns Introduced

- Model-View-Controller (MVC):
  - Where: Entire codebase (models, controllers, views folders)
  - Why: Decouples UI, business logic, and data, improving maintainability and testability.
- Factory Pattern:
  - Where: Vehicle and ParkingSpace creation (e.g., models/vehicle.py, models/space.py)
  - Why: Simplifies instantiation of different vehicle/space types.
- Strategy Pattern:
  - Where: Parking allocation logic (e.g., choosing space for vehicle type)
  - Why: Allows flexible parking strategies for different vehicle types.
- Template Method Pattern:
  - Where: Vehicle and ParkingSpace class hierarchies
  - Why: Defines skeleton of operations, allowing subclasses to override specific steps.
- Controller as Facade:
  - Where: controllers/parking_controller.py
  - Why: Provides a unified interface for the view to interact with the system, hiding complexity.

Anti-patterns Identified and Resolved

- God Class:
    - Original Issue: Parking logic, UI, and data were all in a single class/file.
    - Resolution: Split into MVC layers.
- Tight Coupling:
    - Original Issue: UI directly manipulated data and business logic.
    - Resolution: Introduced controllers as intermediaries.
- Code Duplication:
    - Original Issue: Repeated logic for different vehicle/space types.
    - Resolution: Used inheritance and the factory pattern.
- Lack of Separation of Concerns:
    - Original Issue: No clear boundaries between data, logic, and UI.
    - Resolution: Enforced MVC structure.
- Poor Extensibility:
    - Original Issue: Difficult to add new vehicle/space types or features.
    - Resolution: Used OOP patterns and modular design.