Design Pattern Selection

October 2025

Contents

1 Introduction

In the implementation of the **Intelligent Parking Management System**, software architecture determines how well the system scales, evolves, and integrates with external modules. The system interfaces with multiple components such as OCR-based license plate recognition, payment gateways, and IoT devices (Raspberry Pi or Arduino), all of which require a consistent, flexible backend design.

This section identifies candidate design patterns, evaluates them based on objective criteria, and justifies the final selection. The chosen pattern must:

- Be lightweight and understandable for an MVP.
- Allow future extensions for hardware and APIs.
- Align with the layered FastAPI architecture (API $\rightarrow$ Service $\rightarrow$ Repository).
- Be simple enough for new contributors to adopt quickly.

2 Candidate Design Patterns

Several classical software design patterns were assessed for their suitability in modularizing, extending, and controlling object creation and system behavior.

2.1 Factory Pattern

The Factory pattern creates objects without explicitly defining their concrete class. It is ideal for instantiating OCR engines, payment modules, or barrier control adapters based on configuration.

2.2 Strategy Pattern

The Strategy pattern encapsulates interchangeable algorithms, allowing runtime selection—such as switching between EasyOCR and OpenALPR for license plate recognition under different environmental conditions.

2.3 Adapter Pattern

The Adapter pattern enables interoperability between incompatible interfaces. It is useful for integrating third-party SDKs or hardware APIs into a standardized project interface.

2.4 Singleton Pattern

The Singleton pattern restricts a class to one instance throughout the application lifecycle, suitable for configuration, database sessions, or shared hardware controllers.

2.5 State Pattern

The State pattern modifies an object's behavior based on its internal state—applicable to parking session transitions such as `ENTERED`, `PENDING_PAYMENT`, `PAID`, and `EXITED`.

3 Decision-Based Criteria Evaluation

Purpose of the Table? It presents a structured comparison of the five design patterns based on eight decision criteria: implementation ease, comprehensibility, architectural alignment, scalability, runtime flexibility, reusability, MVP suitability, and testability.

How to interpret? A checkmark (✓) indicates strong suitability, while a star (★) represents partial or acceptable alignment for the MVP context.

Criterion	Factory	Strategy	Adapter	Singleton	State
Ease of Implementation	✓ Simple and minimal	★ Requires multiple strategies	★ Wrapper setup	✓ Very simple	★ State logic required
Comprehensibility	✓ Easy for beginners	★ Slightly abstract	★ Needs interface mapping	✓ Straight-forward	★ Moderate complexity
Alignment with Architecture	✓ Ideal for layered backend	✓ Supports runtime choice	✓ Fits 3rd-party APIs	★ Limited (global configs)	✓ Useful for session flow
Extensibility / Scalability	✓ Easily add OCR/payment modules	✓ Independent strategies	✓ Add adapters easily	★ Limited	★ Add new states
Runtime Flexibility	✓ Config-driven selection	✓ Dynamic algorithm swap	★ Mostly static	★ Static global scope	★ Entity-based
Reusability	✓ High	✓ High	✓ High	★ Low	★ Low
MVP / IoT Suitability	✓ Excellent, low complexity	★ Medium, extra setup	★ Moderate	★ Risk of misuse	★ Medium
Testing / Mocking Ease	✓ Excellent, easy mocks	✓ Excellent	✓ Excellent	★ Medium	★ Needs transitions

Table 1: Comparative evaluation of candidate design patterns for the Intelligent Parking Management System.

4 Decision and Justification

After systematic evaluation, the **Simple Factory Pattern** was identified as the optimal choice for the MVP stage of the project.

Justification:

1. **Simplicity and clarity:** Provides a clean, minimal mechanism for creating system components through configuration-based instantiation.

2. **Architectural alignment:** Fits perfectly between the Service and Repository layers, acting as a central point for dependency creation.

3. **Extensibility:** Enables easy registration of new providers (OCR, payment, or IoT) without modifying existing logic.

4. **Configurability:** Supports environment-based switching between production and test implementations.

5. **Lightweight singleton behavior:** Cached instances ensure efficient resource use.

6. **Ease of testing:** Mocked providers can be seamlessly registered for unit or integration testing.

5 Example Integration Overview

At runtime, the Factory loads components based on configuration:

- `OCR_IMPL=easyocr`
- `PAYMENT_IMPL=stripe`
- `BARRIER_IMPL=rpi`

These map respectively to:

- `EasyOCROCR` for license plate recognition.
- `StripePayment` for payment handling.
- `RPiBarrier` for hardware control.

For local testing:

- `OCR_IMPL=mock`
- `PAYMENT_IMPL=mock`
- `BARRIER_IMPL=mock`

This structure allows effortless switching between real and simulated components, enabling safer testing and faster development cycles.

6 Conclusion

The **Simple Factory Pattern** provides the best trade-off between clarity, flexibility, and maintainability for the Intelligent Parking Management System. It ensures modular instantiation of critical subsystems while staying lightweight and compatible with the project's layered FastAPI architecture. Future expansions—such as adaptive OCR (Strategy), advanced API integrations (Adapter), or lifecycle management (State)—can build naturally upon this foundation.

7 Summary Table

Selected Pattern	**Implementation Effort**	**Complexity**	**Extensibility**	**Testability**
Simple Factory Pattern	Low	Low	High	High
Justification: Configuration-driven creation of OCR, payment, and IoT adapters; aligns with the layered architecture; easy to extend and test.				

Table 2: Summary of the final selected design pattern.