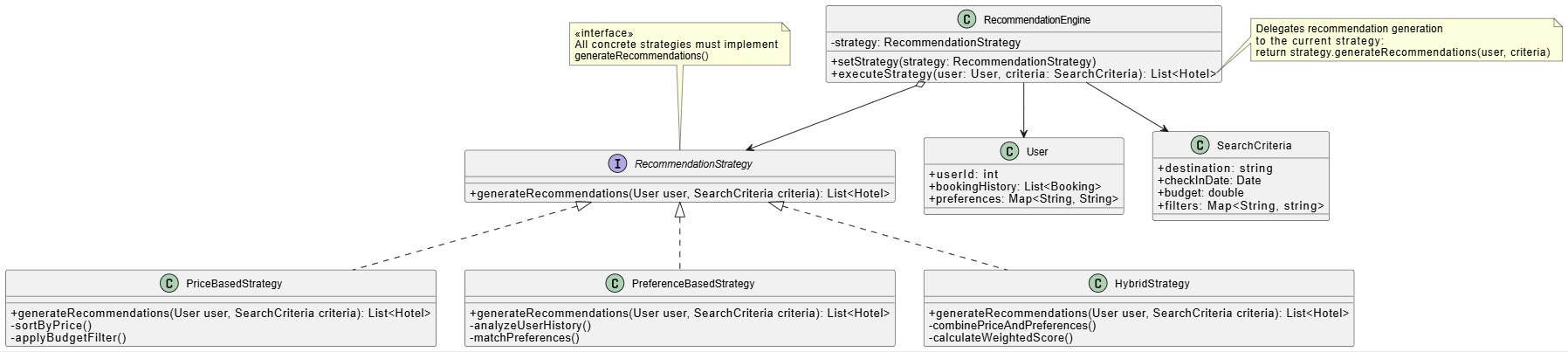
**1. Project Overview**

This document outlines the AI-driven hotel reservation system’s architecture, emphasizing **Strategy** and **Adapter** design patterns. The **Strategy Pattern** enables dynamic AI recommendation algorithms (price/preference-based), while the **Adapter Pattern** standardizes third-party API integrations (e.g., docs.hotelapi). Together, they ensure scalable personalization, real-time data compatibility, and maintainable workflows. The approach aligns with iterative Scrum delivery, prioritizing user-centric features and robust technical foundations.

**2. Strategy Pattern**

The Strategy Pattern defines a family of interchangeable algorithms, encapsulating each one and allowing dynamic selection at runtime. In this project, it is applied to the AI recommendation engine, enabling the system to switch between different recommendation strategies (e.g., *price-based*, *preference-based*, or *hybrid*) based on user behavior or context.

**Why We Chose It**:

1. **Flexibility**: Easily add/update recommendation algorithms (e.g., seasonal trends) without altering core logic.
2. **Scalability:** Supports future AI/ML enhancements (e.g., neural networks) as new strategies.
3. **User-Centric:** Tailors results dynamically (e.g., prioritize budget vs. luxury) for personalized experiences.
4. **Decoupled Design:** Keeps AI logic isolated, simplifying testing and maintenance.

**Figure 1:** Strategy Pattern Design in the AI Hotel Reservation System

// Initialize strategy based on user behavior

$strategy = new HybridStrategy();

$engine = new RecommendationEngine($strategy);

// Execute recommendations

$criteria = new SearchCriteria("Paris", "2023-12-01", 200.00, ["rating" => "4+"]);

$recommendations = $engine->executeStrategy($currentUser, $criteria);

// 3. Switch strategies dynamically

$engine->setStrategy(new PriceBasedStrategy());

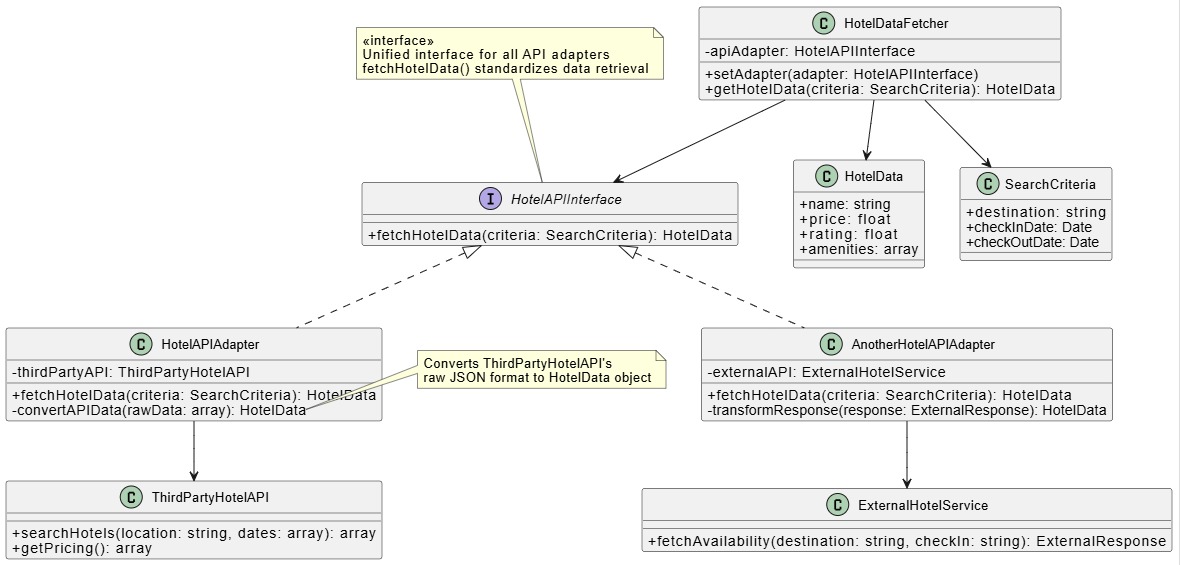
**Figure 2:** Code Implementation for Strategy Pattern Recommendation Engine

**3. Adapter Pattern**

The Adapter Pattern acts as a bridge between incompatible interfaces, allowing seamless communication between the system and external services. In this project, it standardizes data from third-party hotel APIs (e.g., docs.hotelapi, Amadeus) into a unified format, ensuring consistent processing of real-time availability, pricing, and hotel details.

**Why We Chose It**:

1. **API Agnosticism**: Integrate diverse APIs (JSON/XML/REST) without rewriting core logic.
2. Simplified Maintenance: Centralize data conversion (e.g., mapping "base\_price" to "price") in adapters.
3. Scalability: Add new partners (e.g., Expedia, Booking.com) by creating new adapters, avoiding system-wide changes.
4. Error Handling: Isolate API-specific issues (e.g., rate limits, retries) within adapters for robustness.



**Figure 3:** Adapter Pattern Design in the AI Hotel Reservation System

interface HotelAPIInterface { // 1. Target Interface

public function fetchHotelData(SearchCriteria $criteria): HotelData;

}

class ThirdPartyHotelAPI { // 2. Adaptee: Existing Third-Party API

public function searchHotels(string $location, array $dates): array {

return $this->callAPI("https://api.hotelprovider1.com/search");

}

}

class HotelAPIAdapter implements HotelAPIInterface { // 3. Adapter Implementation

private $thirdPartyAPI;

public function \_\_construct(ThirdPartyHotelAPI $api) {

$this->thirdPartyAPI = $api;

}

public function fetchHotelData(SearchCriteria $criteria): HotelData {

$rawData = $this->thirdPartyAPI->searchHotels( // Call the legacy API with its native format

$criteria->destination,

[$criteria->checkInDate, $criteria->checkOutDate]

);

// Convert to unified HotelData format

return new HotelData(

$rawData['hotel\_name'],

$rawData['base\_price'],

$rawData['rating'],

$rawData['amenities']

);

}

}

// 4. Client Code Usage (Laravel Example)

class HotelDataFetcher {

private $adapter;

public function setAdapter(HotelAPIInterface $adapter) {

$this->adapter = $adapter;

}

public function search(SearchCriteria $criteria): HotelData {

return $this->adapter->fetchHotelData($criteria);

}

}

// Usage

$criteria = new SearchCriteria("Paris", "2023-12-01", "2023-12-05");

$adapter = new HotelAPIAdapter(new ThirdPartyHotelAPI());

$fetcher = new HotelDataFetcher();

$fetcher->setAdapter($adapter);

$hotelData = $fetcher->search($criteria);

**Figure 4:** Code Implementation for Adapter Pattern Fetching Api Logic