Architectural Decision Record (ADR) for Project "Shop"

Title: Architectural Decision for Monolithic Architecture with Clean Architecture and DDD

Date: October 8, 2024

Status: Accepted

---

1. Context

The "Shop" project is an e-commerce application that requires a robust and scalable architecture due to the complexity of the business domain. The project involves several domain-specific rules and invariants that must be managed properly. Given the criticality of maintainability, performance, and ease of testing, an architectural decision needs to be made that supports these requirements.

The key **quality attributes** for this project include:

- **Evolvability**: Ability to extend and evolve the system without significant refactoring.

  For example, in the future, we want to transition the project to a distributed architecture with minimal side effects.

- **Testability**: The system must be easily testable, especially in the domain layer.
- **Readability**: Code should be clean, well-organized, and easily understood by developers.
- **Maintainability**: The architecture must support long-term maintenance and low technical debt.
- **Performance**: The system must be performant under high load, especially in handling business logic.

---

2. Decision

1. **Monolithic Architecture**: We have decided to implement a **monolithic** architecture for the initial development of the "Shop" project. Although microservices are a popular approach, the current scope and size of the project suggest that a monolith will be easier to manage and deploy at this stage. This will also allow us to delay the complexity associated with managing distributed systems until it's absolutely necessary.
2. **Use of Clean Architecture**: We will implement the **Clean Architecture pattern** to achieve separation of concerns and ensure that business logic is not tightly coupled to external frameworks or databases. This layered approach will also promote testability and maintainability. Layers will include:
    - **Domain Layer**: Core business rules (entities, value objects, services, repositories).
    - **Application Layer**: Use cases that orchestrate business operations.

- **Infrastructure Layer**: Communication with external systems (databases, APIs, messaging).
- **Interface Layer**: User interfaces or API endpoints.
- **Service Host Layer**: It is responsible for hosting various interfaces such as REST and gRPC.

3. **Domain-Driven Design (DDD)**: Due to the **complexity and rich business rules** within the e-commerce domain, we will leverage **Domain-Driven Design (DDD)** principles. This includes:
   - Defining **aggregates** and **entities** to encapsulate business logic.
   - Using **value objects** for immutable properties and rules.
   - **Domain events** will be used to decouple certain processes within the domain.
   - **Repositories** to abstract data access, following DDD best practices.
4. **State Pattern Instead of Enum**: To handle different states of entities like **Order** or **Payment**, we will replace the commonly used enum for state management with the **State Pattern**. This pattern will make the system more **extensible** as the number of states grows and help avoid large switch or if-else blocks. It will also encapsulate state-specific logic within each state class, improving **readability** and **maintainability**.
5. In our project, we utilized the **Service Host pattern** due to the diverse interfaces we have, such as REST and gRPC. This choice allows us to efficiently manage these interfaces in a cohesive manner, ensuring that they can interact smoothly and that the system remains scalable and maintainable.

---

3. Consequences

- **Monolithic Architecture**:
  - **Positive**: Simpler to deploy and manage in the early stages of the project.
  - **Negative**: May require refactoring into a distributed architecture in the future as the system grows.
- **Clean Architecture**:
  - **Positive**: Encourages separation of concerns, which improves **testability** and **evolvability**. Core business logic is isolated from frameworks, making it more flexible to changes.
  - **Negative**: Adds some initial complexity, especially for developers who are unfamiliar with this pattern.
- **Domain-Driven Design (DDD)**:

    - **Positive**: DDD is well-suited to manage the domain's complexity and enforce business invariants, improving **maintainability**. It also supports **evolvability** as the domain model can grow and evolve without affecting other layers.
    - **Negative**: DDD introduces a learning curve and requires developers to adhere strictly to DDD concepts, which may slow down initial development.
- **State Pattern**:
    - **Positive**: Using the **State Pattern** provides better **readability**, **maintainability**, and **testability** by encapsulating state-specific behavior and avoiding large conditional blocks.
    - **Negative**: Introduces additional classes and structure, which may seem like overhead for smaller use cases, but pays off as the system grows.
- **Service Host Pattern:**
    - **Positive**: The Service Host Pattern enables seamless communication between different service interfaces, promoting better organization, scalability, and maintainability as the system evolves.
    - **Negative**: It may introduce additional complexity in managing various interfaces, which can require extra overhead in configuration and management, especially in smaller projects.

---

4. Rationale

The decision to use a **monolithic architecture** is based on the need for simplicity in deployment and development in the early stages. The current project size does not justify the complexity that microservices would introduce.

The use of **Clean Architecture** ensures that business logic is isolated from external dependencies, promoting testability and long-term maintainability.

DDD is essential for managing the complex and rich domain of the e-commerce system. It allows us to model the domain accurately and enforce business rules through aggregates, entities, and domain events.

Finally, the **State Pattern** is chosen over traditional enum usage to handle states more effectively, especially given the evolving nature of the order and payment processing flows. This pattern improves flexibility as new states may be added in the future without breaking existing logic.

---

5. Alternatives Considered

1. **Using TDD**:
    - Although adopting TDD could have provided us with significant benefits, we decided not to implement it due to the team's lack of knowledge regarding the approach and the insufficient time available. Instead, we opted to write the tests after the code was developed.
2. **Microservices Architecture**:
    - We considered breaking the system into microservices but decided against it due to the additional complexity in terms of infrastructure and deployment overhead. A monolithic architecture better fits the current scope.
3. **Enum for State Management**:
    - Using an enum for state management was an option, but it would have led to large and complex conditional structures (if-else or switch statements), making the code less readable and harder to maintain as the number of states grows.
4. **Not using the Service Host pattern**:
    - If the interface layer is not utilized, the service host could be merged, which might be a good point in terms of simplicity, but it would lead to clutter and reduced code readability.
5. **Not using DDD**:
    - We could choose not to use DDD. While this would improve simplicity, we would lose evolvability, readability, and other important qualities in that case.

Authors: Ali Asgari
Version: 1.0