|  |  |
| --- | --- |
| **Course** | CS504070 - Service-Oriented Architecture  (HK2/2023-2024) |
| **Fullname** | Nguyễn Đình Việt Hoàng |
| **Student ID** | 522H0120 |
| **Homework** | Homework 1 |

Question 1:

1. Domain-Driven Design (DDD) is an approach to software development that focuses on understanding and modeling the business domain. It was introduced by Eric Evans in his book "Domain-Driven Design: Tackling Complexity in the Heart of Software," published in 2003. Domain-Driven Design is particularly useful in complex and large-scale software projects where understanding and modeling the business domain is crucial for success. It helps align the development team with the business goals and creates a shared understanding of the problem domain, leading to more effective and maintainable software solutions.
2. Key components and principles of Domain-Driven Design (DDD):

+ Ubiquitous Language:

* A common, shared language between developers and domain experts.
* Ensures a consistent understanding of domain concepts.
* Helps bridge the communication gap.

+ Bounded Context:

* Acknowledges the need for different models in different parts of the system.
* Defines clear boundaries for models and languages.

+ Aggregates:

* Ensures consistency and transactional integrity within a boundary.
* Defines a cluster of entities and value objects treated as a single unit.

+ Entities and Value Objects:

* Entities have distinct identity and represent things with a lifecycle.
* Value objects have no distinct identity and are immutable.

+ Repositories:

* Provide a collection-like interface to access and persist aggregates.
* Shield the domain model from data storage details.

+ Services:

* Stateless components focused on behavior.
* Used for domain logic that doesn't naturally fit into entities or value objects.

+ Domain Events:

* Used to communicate changes in the state of the system.
* Decouples components and allows for more flexibility in reacting to changes.

**Example: E-commerce Order Processing**

In an e-commerce system, managing orders is a critical aspect. Applying DDD principles, we can design the "Order" domain as follows:

1. **Entity:**
   * The "Order" is an entity because it has a distinct identity and lifecycle. Each order placed by a customer is unique and can be tracked independently.
2. **Value Objects:**
   * The "Order" may contain value objects such as "OrderLine," which represents individual items or products within the order. Each order line has attributes like product, quantity, and price.
3. **Aggregate:**
   * The "Order" can be treated as an aggregate, grouping together the main entity ("Order") and its associated value objects ("OrderLines"). The aggregate ensures transactional consistency; all changes to the order, including order lines, are made within the context of the order itself.
4. **Repositories:**
   * A "OrderRepository" is responsible for storing and retrieving orders. It provides a way to access and persist entire order aggregates, maintaining the integrity of the order data.
5. **Services:**
   * A "OrderService" might be responsible for handling complex order-related logic, such as calculating total order amounts, applying discounts, or handling special cases during order processing.
6. **Domain Events:**
   * A "OrderPlaced" domain event might be triggered when a new order is successfully placed. This event could be used to notify other parts of the system, such as inventory management, that new products need to be shipped.

**Explanation:**

* **Entity ("Order"):** Represents a single order with a unique identity and lifecycle. It encapsulates information like order number, customer details, and order status.
* **Value Object ("OrderLine"):** Represents individual items within the order. It encapsulates information like the product, quantity, and price. Since each order line is dependent on the order, it makes sense to treat it as a value object.
* **Aggregate ("Order"):** Groups the main entity ("Order") and its associated value objects ("OrderLines"). This ensures that changes to the order and its components are consistent and transactional.
* **Repository ("OrderRepository"):** Manages the persistence of entire order aggregates. It abstracts away the details of how orders are stored and retrieved, providing a clean interface to the application.
* **Service ("OrderService"):** Handles complex order-related logic that doesn't naturally fit within the order entity or value objects. For example, calculating the total order amount, applying discounts, or checking eligibility for special promotions.
* **Domain Event ("OrderPlaced"):** Represents a meaningful change in the state of the system. In this case, the "OrderPlaced" event can be used to trigger actions in other parts of the system, such as updating inventory levels or notifying the shipping department.

Question 2:

* + **Coupling:**

Coupling refers to the degree of interdependence between modules or components within a system. It measures how much one module knows about the internal workings or details of another module. In other words, coupling describes the connections and relationships between different parts of a system.

There are two main types of coupling:

1. **Loose Coupling:**
   * In loose coupling, modules are designed to interact with each other through well-defined interfaces.
   * Changes in one module are less likely to affect others, promoting flexibility and maintainability.
   * Loose coupling is generally considered a good design principle as it reduces dependencies and allows for easier modification.
2. **Tight Coupling:**
   * Tight coupling occurs when modules are highly dependent on each other, and changes in one module may have a significant impact on others.
   * It makes the system less flexible and more difficult to maintain because modifications in one part of the system can cause unintended consequences in other parts.

Reducing coupling is often a goal in software design as it promotes modularity and facilitates changes without affecting the entire system.

* + **Cohesion:**

Cohesion, on the other hand, refers to the degree to which the components within a module or class are related to each other. It measures how closely the members of a module are associated in terms of functionality and responsibility. High cohesion indicates that the elements within a module are closely related and work together to achieve a specific task.

There are different types of cohesion:

1. **Functional Cohesion:**
   * Elements within a module perform a single, well-defined task or function.
   * This is the strongest type of cohesion and is generally considered desirable.
2. **Sequential Cohesion:**
   * Elements within a module are related in a sequential manner, with the output of one part becoming the input for the next.
   * Not as strong as functional cohesion, but still acceptable in some cases.
3. **Communicational Cohesion:**
   * Elements within a module operate on the same set of data.
   * This type of cohesion is often necessary but should be minimized to avoid excessive dependencies.
4. **Procedural Cohesion:**
   * Elements within a module are grouped together because they are processed in a similar way.
   * Like communicational cohesion, procedural cohesion should be minimized for better maintainability.
5. **Temporal Cohesion:**
   * Elements within a module are related by the timing of their execution.
   * This is usually the weakest form of cohesion and is often an indication of poor design.

High cohesion is generally desirable because it leads to more maintainable and modular code. Modules with high cohesion are easier to understand, test, and modify without affecting other parts of the system.

Question 3:

* + The key components of microservices:

+ Microservices: These are small, independent, and loosely coupled services. Each service is a separate codebase, which can be managed by a small development team.

+ Containers: Containers are lightweight, stand-alone, executable packages that include everything needed to run a piece of software.

+ Service Mesh: This is an infrastructure layer for handling service-to-service communication. It’s responsible for the reliable delivery of requests through the complex topology of services that comprise a modern, cloud-native application.

+ Service Discovery: This involves three key components: the service provider, registry, and consumer. The provider registers its presence, the registry maintains the list of available services, and the consumer queries the registry to locate services.

+ API Gateway: The API gateway is the entry point for clients. Instead of calling services directly, clients call the API gateway, which forwards the call to the appropriate services on the back end.

* + The importance of each component can vary depending on the specific needs of a project. For instance, if my project involves a large number of services that need to communicate with each other, the Service Mesh might be particularly important. On the other hand, if my project needs to expose a unified interface to its clients, the API Gateway could be the most crucial component. It’s essential to consider the specific requirements and constraints of my project when determining the importance of each component.

Question 4:

Here’s a comparison of the main attributes of these licenses:

* + Apache License 2.0:

+ It’s a permissive license that allows users to use, modify, and distribute the software with minimal restrictions.

+ It provides an express grant of patent rights from contributors to users.

+ It requires preservation of the copyright notice and disclaimer.

* + GNU General Public License v3.0:

+ It’s a strong copyleft license that requires derivatives to be distributed under the same license terms.

+ It provides an express grant of patent rights from contributors to users.

+ It requires preservation of the copyright notice and disclaimer.

* + MIT License:

+ It’s a short, simple, and permissive license that allows users to do virtually anything they want with the software, including modification, distribution, and private use.

+ It requires that all copies of the software or its substantial portions include a copy of the terms of the MIT License and also a copyright notice.

* + Mozilla Public License 2.0:

+ It’s a weak copyleft license that allows the software to be modified or redistributed without any restrictions or compulsory payments for the licensed code.

+ It provides an express grant of patent rights from contributors to users, with a termination clause triggered if a user files a patent infringement lawsuit.

* + Open Software License 3.0:

+ It’s a copyleft license that does not require reciprocal licensing on linked works.

+ It provides an express grant of patent rights from contributors to users, with a termination clause triggered if a user files a patent infringement lawsuit.