Matrikel-Nr. 12325653, Student - Maksym Rohachov

# Advanced Software Engineering
# Work Sheet: Domain-Driven Design

Please solve all tasks of this worksheet on Moodle till the deadline that is published on Moodle. Up to 0.142 points can be achieved for each task, totaling 1 point. No deadline extensions will be granted.

If you would like to receive bonus points for active participation in the lecture unit of this worksheet, please prepare yourself. For multiple-choice questions, bringing notes is usually sufficient; for open-ended or design/code tasks, a solution on a laptop or *USB* stick is usually better suited. For instance, you can prepare short presentation slides per task or be ready to show your solution in an IDE or code editor.

## 1. Task: Domain-driven design basics

Answer the following questions about domain-driven design.

a) What is the role of the model in Domain-Driven Design (DDD)?

☑ ~~Models in domain-driven design are not just design artifacts but the backbone of the project, used in various phases and activities.~~

This statement is correct. In DDD, models are central to the entire development process, influencing design, implementation, and maintenance. They are fundamental to understanding and solving the domain's problems.

☐ Models act solely as a documentation artifact.

This statement is incorrect. While models do serve a documentation purpose by capturing domain knowledge, their role extends far beyond documentation. They are actively used in designing, developing, and maintaining the system.

☑ ~~Models facilitate communication between developers, designers, and domain experts.~~

This statement is also correct. One of the primary purposes of the model is to provide a common language and understanding among all stakeholders, ensuring that everyone is aligned and can communicate effectively about the domain.

- [x] ~~The development of a ubiquitous language and a single model facilitates communication between developers, designers, and domain experts. (Note that 'single' is not to be taken too literally here: it can be split into many submodels and views.)~~

  This statement is correct as well. The creation of a ubiquitous language and a cohesive model (or set of submodels) is crucial for clear communication and consistency within the team and across the project.

b) A common view in agile methods is summarized in this quote: "Truth can only be found in one place: the code." (Robert C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship).

   How does this relate to domain-driven design where models are used as another source of "truth"? Is there a contradiction between the two views?

- [ ] Domain-driven design rejects the focus on code altogether, emphasizing only the development of models.

  This statement is incorrect. DDD does not reject the focus on code; rather, it emphasizes that the code and the model should be closely aligned. The model should inform the code, and the code should accurately reflect the model, ensuring consistency between the two.

- [ ] The ubiquitous language is the primary carrier of the aspects of design that do not appear in code.

  This statement is partially correct. The ubiquitous language is indeed a crucial part of DDD and serves as a bridge between domain experts and developers, capturing domain knowledge and ensuring everyone speaks the same language. However, aspects of design should ideally appear in both the model and the code to maintain alignment.

- [x] ~~The vocabulary of the ubiquitous language includes names of classes and prominent operations.~~

  This statement is correct. In DDD, the ubiquitous language directly influences the code by dictating the names of classes, methods, and other code constructs. This ensures that the code reflects the domain accurately and remains understandable to both developers and domain experts.

- [x] ~~DDD advises a very literal mapping of models to code.~~

  This statement is correct. DDD promotes a close and literal mapping between the model and the code. The model should be implemented directly in the code, ensuring that changes in the domain model are reflected in the codebase and vice versa.

c) Why and how can the ubiquitous language concept and DDD improve the software quality produced in a SW development project?

- ☐ The ubiquitous language concept in DDD improves software quality by isolating developers from domain experts.

  This statement is incorrect. The purpose of the ubiquitous language is to enhance communication and collaboration between developers and domain experts, not to isolate them. By creating a shared language, DDD aims to bring developers and domain experts closer together to ensure a common understanding of the domain.

- ☑ ~~Continuous learning and communication facilitated by the ubiquitous language contribute to better software quality in a domain-driven design project.~~

  This statement is correct. The ubiquitous language promotes ongoing learning and communication among all stakeholders. This continuous interaction helps in uncovering domain knowledge, refining the model, and ensuring that the software accurately represents the domain, thereby improving software quality.

- ☑ ~~The ubiquitous language contains all concepts and relations in one place that is understandable to domain experts, designers, and developers alike.~~

  This statement is correct. The ubiquitous language is designed to be a common language that encapsulates all relevant domain concepts and relationships. This shared understanding ensures that everyone involved in the project, from domain experts to developers, has a clear and consistent view of the domain, leading to better alignment and higher-quality software.

- ☑ ~~Persistent use of the ubiquitous language will force the model's weaknesses into the open.~~

  This statement is correct. By consistently using the ubiquitous language, any ambiguities, inconsistencies, or gaps in the model are more likely to be identified and addressed. This process of continuous refinement and improvement helps to strengthen the model and, consequently, the quality of the software.

d) How long must a DDD model be maintained and evolved?

- ☐ A DDD model should only be maintained during the initial project phase.

  This statement is incorrect. DDD emphasizes continuous evolution and maintenance of the model throughout the entire project lifecycle, not just during the initial phase.

- ☐ Models should be evolved only when significant issues arise, not continuously.

  This statement is incorrect. DDD encourages continuous iteration and evolution of the model to reflect ongoing learning and changes in the domain, not just when significant issues arise.

- ☑ ~~A DDD model should be iterated permanently through refactoring and discussion.~~

  This statement is correct. DDD promotes the permanent iteration of the model through regular refactoring and discussions among stakeholders. This helps to ensure that the model remains accurate and relevant as the project evolves.

- ☑ ~~You should evolve the model throughout the lifetime of the project.~~

  This statement is correct. In DDD, the model should be continuously evolved and refined throughout the entire lifetime of the project. This ongoing evolution ensures that the model and the software remain aligned with the current understanding of the domain.

e) Which / how many developers in a DDD-based software development project need to learn about modeling and DDD concepts?

- ☑ ~~Anyone responsible for changing code must learn to express a model through the code.~~

  Correct. Any developer who is going to modify the code needs to understand how to express the domain model in the code. This ensures that the code remains consistent with the domain model and that changes do not introduce inconsistencies.

- ☑ ~~Every developer must be involved in some level of discussion about the model and have contact with domain experts.~~

  Correct. While not every developer may be deeply involved in modeling, it is essential that all developers have a basic understanding of the model and participate in discussions. Interaction with domain experts helps ensure that everyone has a shared understanding of the domain and the ubiquitous language.

- ☐ Only developers with a background in domain expertise need to be involved in modeling and DDD concepts.

  Incorrect. DDD emphasizes collaboration between developers and domain experts. It is not limited to those with pre-existing domain expertise. All developers need to understand the basics of the domain to effectively contribute to the project.

- ☐ Only a select few developers responsible for design need to learn about modeling and DDD concepts.

  Incorrect. While having a core team responsible for the design and deep modeling can be beneficial, it is important for the broader development team to understand the principles of DDD and how the model is applied. This promotes better code quality and ensures that the team can collectively evolve the model as needed.

## 2. Task: Relation to the development process

Here is a discussion of the relations of domain-driven design to the development process. Specifically, it focuses on how domain-driven design relates to processes like RUP or agile methods like SCRUM. Insert the missing parts in the correct places in the text below (at the places marked with _________________). Note that some answers can be used multiple times or not at all.

1) Design
2) Rational Unified Process (RUP)
3) Refine and refactor the domain model
4) SCRUM
5) Model the domain
6) Unit and Integration Testing
7) Development
8) aligns well
9) does not align well
10) the waterfall process model

In the domain-driven design (DDD) context, the implementation project follows key phases similar to other software development approaches. These phases include:

- <u>Model the domain</u>: Define and create a conceptual representation of the problem domain.
- <u>Design</u>: Develop a comprehensive design based on the modeled domain, outlining the structure and architecture of the software system.
- <u>Development</u>: Actively implement the designed system based on the established models.
- <u>Unit and Integration Testing</u>: Conduct thorough testing at both unit and integration levels to ensure the correctness and functionality of the developed components.
- <u>Refine and refactor the domain model</u>: Continuously improve and optimize the domain model based on insights gained during the design and development phases.

From a project management perspective, an agile software development methodology aligns well with DDD principles. Agile methods, such as SCRUM, share a common focus on delivering business value, a key objective of DDD in aligning the software system with the business model. The iterative nature of DDD harmonizes with the agile approach, making frameworks like SCRUM suitable for managing DDD implementation projects. The outlined phases of DDD are well reflected in the Rational Unified Process (RUP) as well. Rational Unified Process (RUP), like DDD, emphasizes iterative development and continuous refinement of models, making it a compatible framework for managing projects that incorporate DDD principles. The synergy between DDD, agile methodologies like SCRUM, and frameworks like RUP underscores the flexibility and adaptability of DDD in various software development contexts.

## 3. Task: Domain-driven design layers

In the context of Domain-Driven Design (DDD), a standard architecture for enterprise applications encompasses four fundamental layers:

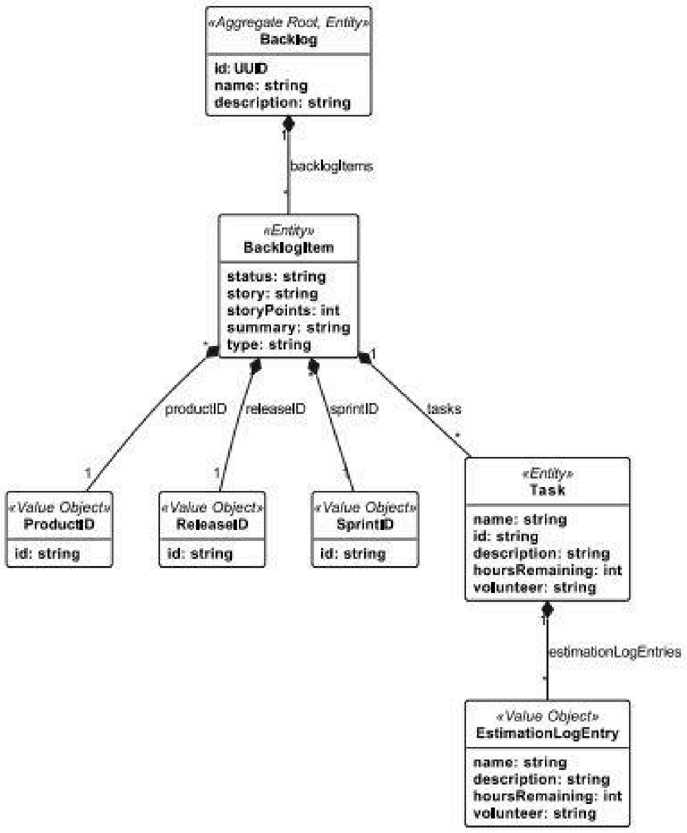
• **Presentation Layer (User Interface):** Its primary function is to present information to users and interpret their commands.
• **Application Layer:** Responsible for coordinating the overall application activity without any business logic. It abstains from holding the state of business objects but may retain the progress of application tasks.
• **Domain Layer:** This layer houses information on the business domain, encompassing the state of business objects. The persistence of these objects, along with their potential states, is entrusted to the infrastructure layer.
• **Infrastructure Layer:** Providing a supportive function for all other layers, it facilitates communication between layers, implements persistence mechanisms for business objects, and includes auxiliary libraries, among other functionalities.

Consider an application contains the following classes and map each of them to the best fitting layer:
• *UserAuthenticationService:*
**Description:** Manages user authentication and authorization.
**Tasks:** Validate user credentials. Authorize user actions. Manage user sessions.
• *DatabaseRepository:*
**Description:** Manages the persistence of domain objects in a relational database.
**Tasks:** Save and retrieve product, order, and user data from the database. Translate domain objects to database records.
• *Product:*
**Description:** Represents a product in the e-commerce domain.
**Tasks:** Store product details (name, price, description). Enforce business rules related to product attributes.
• *ProductViewController:*
**Description:** Handles user interactions related to product browsing and viewing.
**Tasks:** Display a list of products. Handle user clicks on a product for a detailed view. Manage user input for product search.
• *Order:*

**Description:** Represents a customer's order in the domain.
**Tasks:** Maintain order details (items, quantity, total). Enforce business rules for order processing.
• *PaymentGateway:*
**Description:** Integrates with external payment services for processing payments.
**Tasks:** Communicate with third-party payment APIs. Handle responses and errors from payment services.
• *ShoppingCartViewController:*
**Description:** Manages the shopping cart and user's interaction with it.
**Tasks:** Display the contents of the shopping cart. Allow users to add/remove items from the cart. Handle the checkout process.
• *OrderProcessingService:*
**Description:** Coordinates the processing of orders and interactions with the domain layer.
**Tasks:** Validate order details before forwarding to the domain layer. Invoke payment processing services. Communicate with the domain layer for order fulfillment.
• *User:*
**Description:** Represents a registered user in the e-commerce system.
**Tasks:** Store and handle user information.

## 4. Task: Domain-driven design example

Study the following design.1 Answer the following questions about this domain-driven design.



1) If we delete a Backlog instance, what are the consequences for the instances of the other classes in the model?

- ☐ All instances of the other six classes in the aggregate must also be deleted.
- ☐ The three ID classes must be deleted; the tasks might be shared with other backlogs.
- ☐ All value objects must be deleted.
- ☑ ~~All entities must be deleted.~~

This is because `BacklogItem` and `Task` are entities that are part of the `Backlog` aggregate, and their deletion must cascade when the `Backlog` instance is deleted. Value objects will also be deleted as part of their owning entities, but since the question specifies consequences for other classes, the focus is on entities.

2) What are the consequences for Products, Releases, and Sprints when deleting a BacklogItem instance? What is the relation of those concepts to the classes in the model? If we model Products, Releases, and Sprints, would they be entities or value objects?

☑ ~~The IDs are shareable objects; only the ID references get deleted when the *BacklogItem* gets deleted.~~

Correct. In the given design, deleting a `BacklogItem` instance results in the deletion of the ID references (`ProductID`, `ReleaseID`, `SprintID`) within that instance, but not the actual `Products`, `Releases`, and `Sprints` themselves. The IDs are value objects and only their references within `BacklogItem` are removed.

☐ Related Products, Releases, and Sprints get deleted when the *BacklogItem* gets deleted.

Incorrect. There is no indication in the diagram that deleting a `BacklogItem` results in the deletion of the actual `Product`, `Release`, or `Sprint` entities. Only the ID references are removed.

☑ ~~The *IDs* must be used to look up *Products*, *Releases*, and *Sprints* in the given design.~~

Correct. The `ProductID`, `ReleaseID`, and `SprintID` value objects are used to reference and look up the corresponding `Product`, `Release`, and `Sprint` entities. This lookup ensures that the `BacklogItem` is associated with the correct instances of these entities.

☑ ~~*Products*, *Releases*, and *Sprints* would be entities as they are likely not immutable.~~

Correct. In a typical DDD context:

- **Entities** have a distinct identity that runs through time and different states. They are mutable and their lifecycle is tracked.
- **Value Objects** are immutable and defined by their attributes rather than identity.

  Given that `Products`, `Releases`, and `Sprints` are likely to have a lifecycle, change over time, and be uniquely identifiable, they would be modeled as entities.

3) Could we link *Products*, *Releases*, and *Sprints* similarly to *BacklogItem* as *ProductID*, *ReleaseID*, and *SprintID* are linked?

☐ Yes, because they should be modeled as *Value Objects*.

Incorrect. Products, Releases, and Sprints are likely entities due to their unique identities and mutable nature over their lifetimes. Modeling them as value objects would not capture their uniqueness and lifecycle management correctly.

☐ Yes, because they should be modeled as *Aggregate Roots*.

Partially correct but contextually incomplete. While Products, Releases, and Sprints can be aggregate roots (each managing their own lifecycle and consistency rules), linking them using IDs (value objects) is a common practice to reference other aggregates without embedding them directly. This ensures that the aggregate boundaries are respected and maintains the integrity of each aggregate.

☐ No, if they are entities, that would not be good, as they get deleted when the aggregate gets deleted.

Incorrect. Deleting an aggregate (like Backlog) does not necessarily delete the entities (like Products, Releases, and Sprints) referenced by their IDs. Only the references (IDs) within the aggregate are removed. The actual entities remain in their respective aggregates. This misunderstanding conflates the deletion of references with the deletion of the entities themselves.

☑ ~~No, because they would only be locally unique in the Backlog scope, but Products, Releases, and Sprints are global concepts in this application that span across various contexts.~~

Correct. Since Products, Releases, and Sprints are global concepts and span across various contexts in the application, their uniqueness should be maintained globally. Using IDs (value objects) to reference these entities ensures that they are not confined to the local scope of a single aggregate like Backlog but are accessible and unique across the entire application.

## 5. Task: Domain-driven design implementation

Consider the classes used in the following *main()* method. Identify their appropriate role in a DDD-based design (*Entities*, *Value Objects*, *Aggregate Roots*, and *Services*). Realize an implementation of these classes according to these roles in *Java*.

```
public class Main {
    public static void main(String[] args) {

        Product laptop = new Product("Laptop", 999.99);
        Product phone = new Product("Smartphone", 499.99);

        ShoppingCart shoppingCart = new ShoppingCart();

        shoppingCart.addItem(laptop, new ProductQuantity(2));
        shoppingCart.addItem(phone, new ProductQuantity(3));

        System.out.println("Shopping Cart Items:");
        for (CartItem cartItem : shoppingCart.getCartItems()) {

            Product product = cartItem.getProduct();
            ProductQuantity quantity = cartItem.getQuantity();
            System.out.println(product.getName() +
                    " - Quantity: " + quantity.getQuantity() +
                    " - Price per unit: €" + product.getPrice() +
                    " - Subtotal: €" + (quantity.getQuantity() *
                    product.getPrice()));
        }
    }
}
```

## 6. Task: Domain-driven design example

Model the following situation as a *UML* class diagram following *DDD* guidelines: “A flight has a unique flight number, a departure time, and an arrival date. Airports have a unique code like *“VIE”* for Vienna. Each flight has a *departure* and *arrival airport*. One plane operates several flights. A plane has a unique *ID*, *type*, and several *seats*, each with a *seat number*. A passenger on a flight has a *first* and a *last name*. Several passengers are booked on a flight. Per booked flight, passengers book a *seat* and provide a *meal preference* (a simple text). A passenger needs one booking for a flight containing an *outbound flight* and a *return flight*. A ticket has a *price* and *status*, either *pending* or *purchased*. *Flights* can be booked, and *flight bookings* can be canceled. For this, flights can be found by number or by airport.”

Use UML stereotypes on the classes to denote *Entities*, *Value Objects*, *Aggregate Roots*, and *Services* in your *DDD* design.

## 7. Task: DDD service design

Which statements about good service design in domain-driven design are correct?

- ☑ ~~The service operation relates to a domain concept that is not a natural part of an entity or value object.~~

  Correct. Services in DDD are used when operations don't fit naturally within entities or value objects. They encapsulate domain logic that doesn't belong to any particular entity or value object.

- ☐ The service operations relate to a domain concept and must be a natural part of an entity or value object.

  Incorrect. If an operation is a natural part of an entity or value object, it should be placed within that entity or value object rather than in a service.

- ☑ ~~The service interface is defined in terms of other domain model elements.~~

  Correct. A good service interface in DDD should interact with other elements of the domain model, such as entities, value objects, and other services. This ensures the service remains within the domain context and follows the ubiquitous language.

- ☐ The service interface contains no other elements of the domain model.

  Incorrect. Services should interact with and use domain model elements to perform their operations. Isolating services from the domain model would defeat the purpose of aligning services with the domain.

- ☐ The service operation maintains an internal state that affects its behavior (i.e., it is *stateful*).

  Incorrect. Services in DDD are typically stateless. They perform operations and return results without maintaining an internal state that influences their behavior. This promotes better scalability and testability.

- ☑ ~~The service operation does not maintain an internal state that affects its behavior (i.e., it is *stateless*).~~

  Correct. Stateless services are preferred in DDD as they are simpler to manage and scale. They do not retain state between calls, making them easier to test and understand.

- [ ] The service operation can either be stateful or stateless.

  Partially correct. While it is technically possible for a service to be stateful, in good service design within DDD, stateless services are recommended for the reasons mentioned above. However, certain contexts might require stateful services, though these are exceptions rather than the norm.