Rich Domain Model Design and Persistence Blueprint for Applications Using a Relation Database in .NET 9

Bachelor's thesis draft

Author: Jonathan Sillak

Supervisor:

Topic explanation

Domain model design is one of the underlying tasks for creating an efficient software application. Whether building a simple webpage or a worldwide software application, there is no way around domain models when persisting information in the database.

Overall, there are two different approaches to domain model design: Anemic and Rich Domain Model design. Anemic Domain Model design keeps the database entities simple by separating the business logic into services. The classification of an Anemic Domain Model as an anti-pattern is a subject of debate among software architects. Anemic Domain Model design suits smaller projects, where the business complexity is relatively low, and the overhead of implementing a fully-fledged domain model may outweigh its benefits [1].

On the other hand, Rich Domain Model design encapsulates the validation logic inside the domain class – making the domain models harder to persist in the database [3] but providing stronger guarantees of data integrity and business rule enforcement. While an Anemic Domain Model approach can suffice for smaller projects, allowing for quicker development and easier maintenance, but as the project scales and business rules become more intricate, the limitations of this pattern become apparent, potentially leading to increased technical debt and decreased code maintainability in the long run [2].

While many developers are aware of these two domain model designs, they tend to favor the Anemic design for the ease of implementation [3] and lack of available material on the alternative. Another problem is the complexity of creating a scalable, Rich Domain Model design, which is easy to extend when the business requirements change or evolve. Implementing the Rich Domain Model is a complex task because it requires foresight and deep analysis of design patterns.

Overview of scientific publications. Problem explanation

The issue of designing a Rich Domain Model and a thin service layer dates back over 30 years when Eric Evans writes in 2003: “I have spent the past decade focused on developing complex systems in several business and technical domains. [...] A feature common to the successes was a rich domain model that evolved through iterations of design and became part of the fabric of the project.” [4]. Another article from the same year by Martin Fowler emphasizes the relevant problem of Anemic Domain Model design as anti-pattern having a spurt in the software development industry [3].

The main obstacle of implementing Rich Domain Model design is the complexity of persisting these entities in the database. Rich Domain Model focuses on the use of value objects instead of primitive types, this works perfectly until the developer starts saving these entities in the database [3]. Why? Well, we don’t want to create a database entity for every value object in our domain model. On the other hand we can’t directly save objects into the database without creating a new entity.

There are techniques to get around this problem, but the solutions don’t have a clear set of principles. One technique is the separation of domain and persistence models, where all the validation logic is handled in the Rich Domain Model and the actual saving to database is done through another domain entity [5].

Another solution is creating a piece of middleware code, which translates the object into primitive types (like strings) when saving to database and reconstructs them into objects when loading from the database [6].

For the technical details, the author of the thesis will rely on available material on the topic. One author in this field is Zoran Horvat, who constantly publishes up to date content on Domain Driven Design using the latest techniques in the .NET world. Another great source of knowledge is Milan Jovanović, who provides content using the .NET framework.

This thesis addresses a critical gap in modern software development practices, particularly in the ASP.NET ecosystem. While Rich Domain Model design has been recognized as a powerful approach for complex systems, there is a notable lack of comprehensive, up-to-date guidance on implementing this pattern in ASP.NET applications. This work aims to bridge that gap by providing a practical, scalable blueprint that combines theoretical foundations with cutting-edge implementation techniques. It serves as both a foundation for more complex projects and a philosophical guide for sound architectural practices, rather than a rigid rulebook for every scenario.

Methodology

In the course of the thesis, the author will constantly read, old and new, relevant material to expand his knowledge on the topic and be up to date. Since software development and ASP.NET framework develops progressively and as a result previous knowledge depreciates quickly, the author will be using the .NET 9 version.

This thesis combines the ideas of multiple authors and the practical expirience of the bachelor's thesis author to find the latest and efficient solution for encapsulating the business logic inside the domain model while simplifying the persistence to the database.

Work structure

Implement the base properties, like id, creation and update information, as a separate, inheritable class, reducing code duplication on every domain model.

Create sample domain models to demonstrate the common problematic use cases of the Anemic Domain Model and refactor them following the principles of Domain Driven Design, thus demonstrating the benefits of the Rich Domain Model design.

Implement the persistence logic for these domain models, analysing the different possibilities and choosing the suitable method according to the use case. This involves converting the value objects into primitive types and vice versa whenever possible.

Validation

For the validation of this thesis, the author will be comparing the Anermic and Rich Domain Model design implementations and analyse their performance in a growingly complex hypothetical scenario. This analysis will examine scalability, readability, mainteance and testability.

References

[1] Milan Jovanović (2023). Refactoring From an Anemic Domain Model To a Rich Domain Model. https://www.milanjovanovic.tech/blog/refactoring-from-an-anemic-domain-model-to-a-rich-domain-model

[2] Khalil Stemmler (2019). “Anemic Domain Model”. https://khalilstemmler.com/wiki/anemic-domain-model/

[3] Martin Fowler (2003). Anemic Domain Model. https://martinfowler.com/bliki/AnemicDomainModel.html

[4] Eric Evans (2003). Domain-Driven Design: Tackling Complexity in the Heart of Software. pp. 7

[5] Vladimir Khorikov (2016. Having the domain model separated from the persistence model. https://enterprisecraftsmanship.com/posts/having-the-domain-model-separate-from-the-persistence-model/

[6] Zoran Horvat (2024). Here Is the Most Powerful EF Core Configuration Technique. https://www.patreon.com/posts/source-for-here-111955171