UNIVERSIDADE FEDERAL DO CEARÁ

CURSO DE GRADUAÇÃO EM EGENHARIA DE SOFTWARE

CARLOS ALEXANDRE LIMA DA SILVA

EDUCAVERSO: ARQUITETANDO UMA PLATAFORMA DE AGREGAÇÃO DE CONTEÚDOS EDUCACIONAIS UTILIZANDO *DOMAIN-DRIVEN-DESIGN*

RUSSAS

2023

CARLOS ALEXANDRE LIMA DA SILVA

EDUCAVERSO: ARQUITETANDO UMA PLATAFORMA DE AGREGAÇÃO DE CONTEÚDOS EDUCACIONAIS UTILIZANDO *DOMAIN-DRIVEN-DESIGN*

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Egenharia de Software do da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Egenharia de Software.

Orientador: Prof. Dr. Reuber Régis de Melo

Coorientador: Prof. Esp. Thiago Felippe de Lima Bandeira

RUSSAS

2023

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

S579e Silva, Carlos Alexandre Lima da.
Educaverso : arquitetando uma plataforma de agregação de conteúdos educacionais utilizando Domain-Driven-Design / Carlos Alexandre Lima da Silva. – 2023.
89 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Russas, Curso de Engenharia de Software, Russas, 2023.
Orientação: Prof. Dr. Reuber Regis de Melo.
Coorientação: Prof. Esp. Thiago Felippe de Lima Bandeira.

1. Domain-Driven Design. 2. Arquitetura de Software. 3. Plataforma Educacional. I. Título.
CDD 005.1

CARLOS ALEXANDRE LIMA DA SILVA

EDUCAVERSO: ARQUITETANDO UMA PLATAFORMA DE AGREGAÇÃO DE CONTEÚDOS EDUCACIONAIS UTILIZANDO *DOMAIN-DRIVEN-DESIGN*

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Egenharia de Software do da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Egenharia de Software.

Aprovada em: 11/12/2023

BANCA EXAMINADORA

Prof. Dr. Reuber Régis de Melo (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Esp. Thiago Felippe de Lima Bandeira (Coorientador)
Instituto Federal de Educação, Ciência e Tecnologia do Ceará (IFCE)

Prof. Dr. Pablo Luiz Braga Soares
Universidade Federal do Ceará (UFC)

Prof. Me. Manoel Marisergio Alves de Oliveira
Instituto Federal de Educação, Ciência e Tecnologia
do Ceará (IFCE)

À minha família, por sua capacidade de acreditar em mim e investir em mim. Mãe, seu cuidado e dedicação foi que deram, em alguns momentos, a esperança para seguir. Pai, sua presença significou segurança e certeza de que não estou sozinho nessa caminhada.

AGRADECIMENTOS

Agradeço por Deus nunca faltar em minha vida. Agradeço a minha família, em especial à minha mãe, que sempre esteve presente e me impulsiona a cada dia a realizar meus sonhos. Agradeço aos meus amigos, que estão sempre ao meu lado em momentos difíceis e de felicidade. Agradeço à minha namorada Liana Chagas por ter me acompanhado nessa etapa, me ajudando na elaboração desse trabalho. Agradeço ao meu amigo, professor e coorientador, Thiago Felippe, por ter me ajudado no desenvolvimento e elaboração desse trabalho. Agradeço ao meu orientador, Reuber Regis, por disponibilizar seu tempo e se dedicar em me orientar nesse trabalho.

Por fim, quero expressar minha gratidão a todos que fizeram uma diferença positiva em minha vida.

“A persistência é o caminho do êxito.”

(Charles Chaplin)

RESUMO

O desenvolvimento de software é uma atividade essencial no atual contexto da tecnologia. Portanto, é importante adotar abordagens que não se limitem apenas à implementação de código para melhorar a qualidade do software. Nesse sentido, o *Domain Driven Design* (DDD) é uma abordagem de modelagem/desenvolvimento que se concentra na compreensão profunda do domínio de um problema. O design e a estrutura do código são moldados pela linguagem e conceitos do domínio, facilitando a comunicação entre desenvolvedores e especialistas do assunto. Este trabalho apresenta a aplicação do DDD em uma plataforma de agregação de conteúdos educacionais, chamada de Educaverso. O Educaverso tem como objetivo centralizar e organizar recursos de forma eficiente, proporcionando uma experiência integrada para estudantes. A aplicação do DDD no Educaverso resultou em uma arquitetura capaz de atender aos requisitos técnicos do projeto, sendo escalável, fácil de manter e de qualidade. Além disso, a aplicação do DDD permitiu uma boa organização de código, com boa cobertura nos testes e baixo acoplamento entre módulos do sistema.

Palavras-chave: *Domain-Driven Design*; arquitetura de software; plataforma educacional

ABSTRACT

Software development is a critical activity in the current technological context. Therefore, it is important to adopt approaches that go beyond code implementation to improve software quality. In this context, DDD is a modeling/development approach that focuses on deep understanding of the domain of a problem. The design and structure of the code are shaped by the language and concepts of the domain, facilitating communication between developers and domain experts. This work presents the application of DDD in an educational content aggregation platform, called Educaverso. Educaverso aims to centralize and organize resources efficiently, providing a unified experience for students. The application of DDD in Educaverso resulted in an architecture that met the technical requirements of the project, being scalable, maintainable, and of high quality. Additionally, the application of DDD allowed for good code organization, with good test coverage and low coupling between system modules.

Keywords: Domain-Driven Design; software architecture; educational platform.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
CAPES	Coordenação de Aperfeiçoamento de Pessoal de Nível Superior
DDD	*Domain Driven Design*
IEEE	*Institute of Electrical and Electronic Engineers*
JSON	*JavaScript Object Notation*
NoSQL	*Not Only SQL*
TDD	*Test Driven Development*
UI	*User Interface Design*

SUMÁRIO

1 INTRODUÇÃO

Quando se fala em desenvolvimento de *software*, é importante conhecer as técnicas capazes de melhorar a qualidade dos processos, o que exige abordagens que vão além da simples implementação do código. Nesse contexto, DDD surge como uma metodologia que visa não apenas construir sistemas eficientes, mas também alinhar o desenvolvimento de *software* com as complexas necessidades dos domínios de problemas (EVANS, 2004). Dessa forma, DDD coloca em ênfase a modelagem do domínio do problema, atacando a complexidade no coração do *software* (EVANS, 2004). Essa modelagem não é apenas um exercício técnico, ela cria uma ponte entre as preocupações técnicas e os objetivos do negócio (EVANS, 2004).

Ao entender melhor o contexto em que o *software* é utilizado, as decisões podem ser tomadas com foco na entrega de valor real aos usuários finais, contribuindo para o sucesso do projeto. Essa abordagem proporciona flexibilidade e manutenção no código, definindo alguns conceitos como agregados, entidades, objetos de valor, serviços e eventos de domínio, etc. Essa abordagem modular, resulta em um código mais flexível e fácil de manter ao longo do tempo (EVANS, 2004). Além disso, o DDD promove uma linguagem comum que pode ser compartilhada entre as equipes técnicas e não técnicas. Isso facilita a colaboração entre as equipes, promovendo uma comunicação eficaz entre as partes envolvidas (EVANS, 2004). Arquitetar um *software* não é uma tarefa fácil, especialmente quando o problema é organizar e produzir bons *softwares*.

Nesse sentido, ter uma arquitetura de *software* bem estruturada é fundamental para que o projeto não apresente problemas durante a sua execução (ROBERT, 2019). De acordo com a norma ISO/IEC 9126-1 (INTERNATIONAL ORGANIZATION FOR STANDARDIZATION, 2003), um *software* de boa qualidade deve fornecer algumas características de qualidade de *software*, que são: funcionalidade, confiabilidade, usabilidade, eficiência, manutenibilidade e portabilidade. A funcionalidade diz respeito “à capacidade do produto de *software* de prover funções que atendam às necessidades explícitas e implícitas , quando o *software* estiver sendo utilizado sob condições específicadas”. A confiabilidade é “a capacidade do produto de *software* de manter um nível de desempenho especificado, quando usado em condições especificadas”. Usabilidade é “a capacidade de do produto de *software* de ser compreendido, aprendido, operado e atraente ao usuário quando usado sob condições especificadas”. Eficiência é “a capacidade do produto de *software* de apresentar desempenho apropriado, relativo à quantidade de recursos usados, sob condições especificadas”. Manutenibilidade diz respeito “à capacidade do produto de

software de ser modificado. As modificações podem incluir correções, melhorias ou adaptações do *software* devido a mudanças no ambiente e nos seus requisitos ou especificações funcionais“. Portabilidade é “a capacidade do produto de *software* de ser transferido de um ambiente para outro”.

Neste contexto, este trabalho explora a aplicação do DDD para o desenvolvimento da arquitetura de uma plataforma de agregação de conteúdos educacionais, chamada de educaverso. O educaverso, visa ser uma plataforma educacional com vistas a transformar os alunos em protagonistas do próprio aprendizado, através da organização de conteúdos educacionais disponibilizados pela plataforma Youtube. O Youtube conta com uma enorme quantidade de vídeos educacionais produzidos por diversos autores, de diferentes níveis de conhecimento e áreas de atuação. Isso torna possível a criação do educaverso, e portanto a sua criação é uma iniciativa para o desenvolvimento da educação à distância no Brasil, pois o educaverso pode oferecer foco em conteúdos educacionais, facilidade de acesso, inovação e desenvolvimento profissional. Além disso, uma plataforma específica pode oferecer recursos para organizar os vídeos de forma mais eficiente, avaliar o aprendizado dos alunos ou para interagir com os organizadores de conteúdo e outros alunos.

1.1 Objetivo Geral

Arquitetar uma plataforma de agregação de conteúdos educacionais, chamada de Educaverso, utilizando os princípios e práticas do DDD.

1.2 Objetivo Específicos

- Definir requisitos e características-chave do Educaverso, considerando a eficácia na organização e busca de conteúdos educacionais, utilizando os princípios e práticas do DDD;
- Projetar a arquitetura da plataforma utilizando DDD, identificando contextos, entidades, agregados, objetos de valor, serviços de domínio etc;
- Implementar um protótipo funcional do back-end da plataforma para demonstrar uma aplicação prática, desenvolvida nos princípios do DDD;
- Avaliar a eficácia do DDD na criação da plataforma, considerando critérios como manutenibilidade, escalabilidade e adaptabilidade, com base em um conjunto de testes e análises.

1.3 Estrutura do trabalho

A estrutura do trabalho está organizada em seções, delimitada da seguinte forma:

Seção 1 - Introdução: A introdução contextualiza o tema e apresenta o objetivo geral e os objetivos específicos. A partir daí, o trabalho será desenvolvido em torno desses objetivos;

Seção 2 - Fundamentação teórica: A fundamentação teórica explica os conceitos relacionados ao trabalho com base em pesquisas anteriores. Essa fundamentação fornece o suporte teórico necessário para o desenvolvimento do trabalho;

Seção 3 - Metodologia: A metodologia caracteriza os métodos, técnicas e instrumentos utilizados para satisfazer os objetivos do trabalho. Essa caracterização permite compreender como o trabalho foi desenvolvido;

Secão 4 - Resultados: Nessa seção discorreu os resultados obtidos durante o desenvolvimento da plataforma utilizando o DDD;

Seção 5 - Conclusão: Nessa seção discorreu o que foi feito, vantagens, desvantagens e trabalhos futuros;

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Arquitetura de *Software*

Com o aumento da complexidade dos sistemas de *software*, a arquitetura de *software* tornou-se essencial para o desenvolvimento de sistemas de alta qualidade. Uma arquitetura de *software* pertence à essência da aplicação, sendo retratada por diversos artefatos como diagramas, modelos e documentos, além de englobar os princípios e padrões de *design* pelos quais uma aplicação é estruturada, destacando as escolhas fundamentais para aquela arquitetura específica (BERTHOLINO *et al.*, 2023). Além disso, ela compreende a maneira como aplicativos ou projetos são construídos, tanto em seus aspectos físicos quanto lógicos (ROMERO *et al.*, 2021).

Ainda, na engenharia de *software* uma arquitetura visa simplificar o desenvolvimento de sistemas complexos, ensinando como estruturar e organizar sistemas para alcançar melhores qualidades de desenvolvimento, além de fornecer uma estrutura sólida para uma determinada solução (BARBOSA, 2021). Contudo, tal estrutura requer conhecimento técnico significativo e desempenha um papel crítico na determinação ou fracasso de uma solução de *software* (BARBOSA, 2021).

Além do mais, uma arquitetura de *software* deve dar suporte ao ciclo de vida do sistema para minimizar o seu custo de vida útil e maximizar a produtividade do programador, tonarndo a aplicação fácil de entender, fácil de desenvolver, fácil de manter e fácil de implmentar (ROBERT, 2019). Portando, a escolha da arquitetura de *software* é fundamental para o sucesso de um projeto, pois ela define a estrutura do sistema e facilita que atenda às necessidades do negócio.

2.2 *Clean Architecture*

Clean Architecture é um modelo que separa as preocupações do *software* em camadas, utilizando uma abordagem de composição de sistemas que visa organizar códigos de forma a facilitar o desenvolvimento, implantação, manutenção e integração de regras de negócios (FERREIRA *et al.*, 2022).

O modelo foi proposto por (ROBERT, 2019) e é baseado na ideia de dividir o sistema de camadas bem definidas, onde cada camada é responsável por um conjunto específico de tarefas. A camada mais interna é responsável pela lógica do domínio, que representa as regras

de negócios do sistema. As camadas externas são responsáveis pela interação com o usuário ou com outros sistemas (FERREIRA *et al.*, 2022).

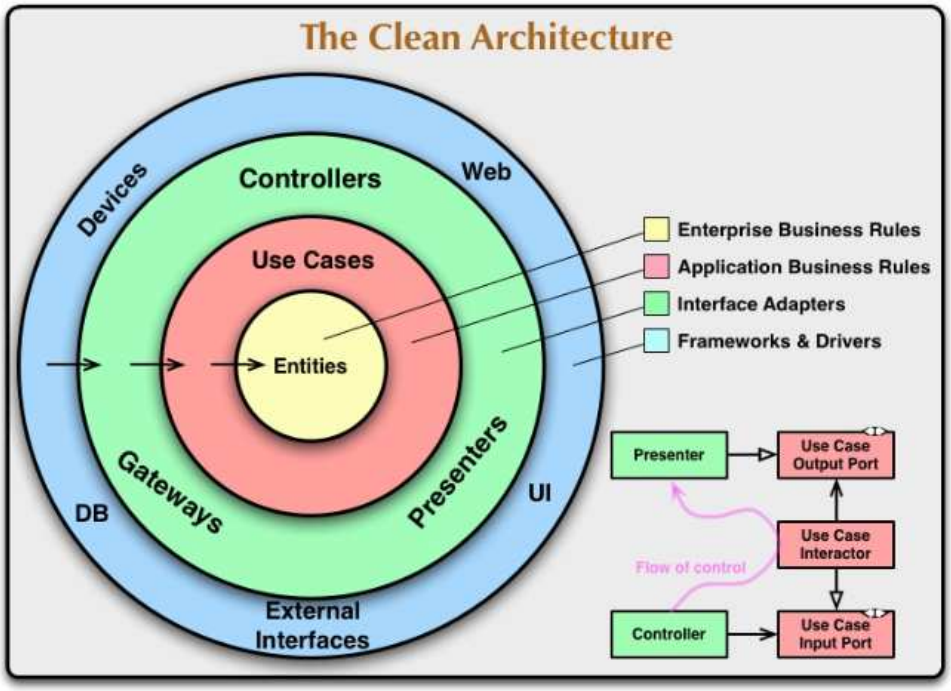


Figura 1 – Clean Architecture
Fonte: (BOB, 2019)

Conforme a Figura 1, os círculos representam diversas áreas de *software*, como *User Interface Design* (UI), *Web*, banco de dados entre outros. A fundamentação dessa arquitetura é baseada na regra de dependência, a qual estabelece que as regras de dependência do código fonte devem apontar de fora para dentro. Isso implica que nenhum elemento declarado em uma camada externa pode ser acessado ou observado diretamente pelas camadas internas (FERREIRA *et al.*, 2022).

Os círculos mais internos são chamados de camada de domínio, resposáveis por encapsular as regras de negócio do domínio. Essas camadas são menos propensas a mudar quando alguma regra externa da aplicação se altera. As camadas de domínio são o coração de uma aplicação. As entidades encapsulam regras de negócio de toda a aplicação (FERREIRA *et al.*, 2022).

Além disso, existem os casos de uso que incorporam as regras existentes da aplicação. Nessa camada, todos os casos de uso são encapsulados e implementados. Qualquer modificação realizada nesta camada deve impactar as entidades, porém, não deve ser influenciada por mudanças externas. Ela deve permanecer como uma camada totalmente independente, utilizada pelas

camadas externas (FERREIRA *et al.*, 2022).

Além do mais, temos a camada de adptadores de interface que consiste em um conjunto de adptadores responsáveis por converter dados no formato mais conveniente para os casos de uso e entidades para o formato mais conveniente para componentes externos, como banco de dados ou a *web* (BOB, 2019).

2.3 *Domain-Driven Design*

O DDD é uma abordagem de desenvolvimento de *software* que foca no domínio do problema (EVANS, 2004). Essa abordagem defende que os desenvolvedores devem trabalhar em conjunto com especialistas do domínio para construir um modelo do domínio que reflita com precisão e eficiência os conceitos e processos presentes no domínio (EVANS, 2004). O DDD pode ajudar a criar softwares mais robustos, fáceis de manter e de adaptar às mudanças nas necessidades do negócio (VIEIRA, 2023). Para isso, o DDD oferece um conjunto de princípios e práticas que ajudam os desenvolvedores a entender o domínio, modelá-lo e implementá-lo de forma eficaz (VIEIRA, 2023).

2.3.1 Modelagem Estratégica

A modelagem estratégica no desenvolvimento de *software* envolve categorizar áreas problemáticas, estabelecer limites conceituais para aplicar modelos específicos e integrar esses limites de maneira coesa aos objetivos da organização. Essa abordagem facilita a visualização nítida de metas específicas para cada limite, criando contextos limitados que representam soluções de *software* para conjuntos particulares de desafios organizacionais. A modelagem estratégica permite uma compreensão mais clara dos objetivos específicos associados a cada limite (FARIAS *et al.*, 2021).

2.3.2 Modelagem Tática

A modelagem tática opera dentro de contextos delimitados. Ao aplicar DDD no desenvolvimento de uma solução organizacional em *software*, é necessário mapear o modelo lógico utilizado pela organização para descrever os processos da solução, transformando-o em um modelo em *software*. Essa transição é realizada pela modelagem tática, que utiliza blocos construtivos padronizados, incorporando fundamentos e boas práticas do DDD para criar o

modelo em *software* correspondente (FARIAS *et al.*, 2021)

2.3.3 Subdomains

No DDD, um subdomínio é uma área distinta de conhecimento, responsabilidade ou especialização dentro do domínio geral. Subdominios representam uma forma de dividir um domínio complexo em partes menores e mais gerenciáveis, tornando mais fácil compreender, projetar e implementar soluções de *software* (GOMES, 2020).Em síntese, temos três tipos de subdomínios:

- **Domínio Principal (*Core Domain*)**: O domínio principal representa o diferencial competitivo da empresa (GOMES, 2020).
- **Subdomínios de Suporte**: Oferecem funções ou serviços auxiliares que apoiam o domínio principal (GOMES, 2020).
- **Subdomínios Genéricos**: Representam componentes ou serviços reutilizáveis que podem ser compartilhados entre vários domínios, como *Keycloak*, que é um produto de software de código aberto que permite autenticação e autorização através de gerenciamento de identidade (GOMES, 2020).

2.3.4 Ubiquitous Language

A *Ubiquitous Language* é um conjunto de termos e conceitos que são usados para descrever o domínio de negócios de um sistema de software (VERNON, 2013). Ela serve como um meio de comunicação entre especialistas do domínio e desenvolvedores (VERNON, 2013). Além disso, ajuda a garantir que todos os envolvidos no desenvolvimento do sistema tenham um entendimento comum do domínio de negócios (VERNON, 2013)

O uso da Linguagem Ubíqua proporciona vários benefícios, que podem ser resumidos em três principais:

- A criação de um modelo de domínio mais elaborado, pois a Linguagem Ubíqua promove uma compreensão comum e precisa do domínio de negócios, o que leva à criação de um modelo de domínio que é preciso e abrangente. Isso, por sua vez, reduz a necessidade de análises extensas e modelos de projeto complexos, economizando tempo e esforço (BARBOSA, 2021).
- Um desenvolvimento mais eficiente, pois a Linguagem Ubíqua ajuda a evitar mal-entendidos e conflitos de interpretação, o que permite que a equipe de projeto se concentre nas tarefas

essenciais do desenvolvimento, sem perder tempo resolvendo problemas de comunicação (BARBOSA, 2021).

- Um software que atende às necessidades do domínio, pois a Linguagem Ubíqua ajuda a garantir que o software seja projetado de acordo com as necessidades dos usuários. Isso ocorre porque a Linguagem Ubíqua é baseada na terminologia e na compreensão do domínio de negócios (BARBOSA, 2021).

2.3.5 Bounded Contexts

Outro conceito fundamental no contexto do DDD é o *Bounded Context*, que se refere a limites bem definidos ou agrupados de domínios dentro de um determinado negócio (BARBOSA, 2021). Essencialmente, ele estabelece que as entidades criadas dentro do contexto devem ser cuidadosamente concebidas para refletir as intenções específicas daquele contexto ao qual pertencem. Isso significa que as entidades não devem ser projetadas isoladamente, mas sim como partes integradas de um sistema maior que compartilha uma compreensão comum do domínio de negócio em questão. Esse enfoque garante que as entidades sejam coesas e alinhadas com os requisitos e objetivos do *Bounded Context* em que operam, contribuindo assim para a clareza e eficácia da modelagem do domínio (BARBOSA, 2021).

Essa fronteira afeta a organização das equipes, bem como as manifestações físicas, como bases de código e esquemas de banco de dados. O design interno de um *Bounded Context* é especificado por padrões táticos do DDD, incluindo o padrão agregado. Neste contexto, um agregado é um agrupamento de objetos de domínio, como entidades que possuem identificadores e ciclos de vida, objetos de valor que são imutáveis e sem estado, e serviços que são mantidos consistentes com relação a invariáveis específicas. Um agregado também representa uma unidade de trabalho em relação a transações do sistema (KAPFERER; ZIMMERMANN, 2020).

2.3.6 Contexts Maps

Um *Context Map* é um diagrama que representa as relações entre vários *Bounded Context*. Conforme descrito por (EVANS, 2004), é uma representação visual dessas relações e consiste em três elementos, que são: O sistema em desenvolvimento, os atores (sejam pessoas ou sistemas) que interagem com ele, e os eventos que desencadeiam tais interações.

Ao aplicar *Context Map*, a comunicação entre desenvolvedores e especialistas de domínio é significativamente aprimorada. A visualização clara das necessidades dos usuários e

requisitos de negócios fortalece a colaboração, criando uma base sólida para o desenvolvimento (EVANS, 2004).

O *Context Map* desempenha um papel crucial na garantia de que o sistema atenda às necessidades dos usuários. Ao definir claramente os requisitos, os desenvolvedores podem direcionar seus esforços de maneira mais eficaz, garantindo um produto final alinhado com as expectativas (EVANS, 2004).

2.3.7 Value Objects

Value Objects são objetos imutáveis que representam um único conceito no modelo de domínio.Eles não têm identidade própria e seu valor é determinado pela composição de suas propriedades (BARBOSA, 2021).

Por possuir características de imutabilidade, seus atributos não podem ser alterados após sua criação, garantindo que o valor do objeto permaneça consistente ao longo do tempo. Além disso, os objetos de valores tornam o código mais claro, pois eles fornecem um foco claro no valor que está sendo representado (BARBOSA, 2021).Ainda, os objetos de valor reduzem a complexidade do código, pois eles podem ser usados para representar aspectos quantificáveis do domínio de forma concisa.

2.3.8 Aggregates

Segundo (EVANS, 2004), *Aggregates* é uma coleção de entidades e objetos de valores que estão logicamente relacionados e que são gerenciados como uma entidade. Uma agregação é gerenciada como uma unidade. Isso significa que todas as operações que afetam a agregação devem ser executadas em conjunto (EVANS, 2004).

Além disso, um *Aggregate* tem sua própria identidade, isso significa que ela pode ser diferenciada de outras agregações. As agregações tornam o código mais claro, pois agrupam entidades e objetos de valor relacionados, além de ajudar a reduzir a complexidade do código ao representar entidades e objetos de valores complexos de forma concisa (EVANS, 2004).

2.3.9 Repositories

Em seu livro, *Vaughn Vernon* (VERNON, 2013) define *repositories* como uma abstração para o armazenamento de dados que fornece operações para acessar e gerenciar dados.

Isso simplifica o acesso e manipulação de dados, permitindo que o código que os utiliza não precise conhecer os detalhes de como os dados são armazenados. Além disso, eles desempenham um papel importante na garantia da consistência dos dados, assegurando que os dados no repositório estejam sempre alinhados com o estado atual do domínio (VERNON, 2013).

Os repositórios proporcionam flexibilidade, tornando o código capaz de se adaptar a mudanças no acesso aos dados sem impactar o código que os utiliza. Eles também aprimoram a testabilidade, permitindo que os testes sejam conduzidos por meio de uma interface abstrata. Além disso, contribuem para a redução da complexidade, representando de forma mais concisa o acesso aos dados (VERNON, 2013).

2.3.10 Services

Em seu livro (EVANS, 2004) Eric Evans define *Services* com uma classe ou função que implementa uma operação complexa que não pode ser facilmente representada por uma entidade ou objeto de valor.

Portando os *services* têm a vantagem de melhorar a clareza do código, uma vez que reúnem operações complexas em um único local, o que facilita a compreensão. Além disso, contribuem para a redução da complexidade ao representar operações complexas de forma concisa. Também melhoram a testabilidade, permitindo testes independentes de entidades e objetos de valor (EVANS, 2004).

2.3.11 Eventstorming

O *Event Storming* é um *workshop* criado por Alberto Brandolini, que visa facilitar a visualização de subdomínios e *bounded contexts*, além de auxiliar no processo de estabelecimento da linguagem ubíqua (BRANDOLINI, 2018). *Event Storming* é uma abordagem que reúne equipes de diferentes áreas para criar uma representação visual dos processos e eventos relacionados a um projeto.

A dinâmica é realizada de maneira interativa com canetas e post-its em um quadro, isso acelera o processo de aprendizado em grupo, permitindo o entendimento unificado do domínio. Cada “post-it” possui uma cor específica e deve passar a ser colocado dentro do quadro em uma ordem específica, onde cada cor representa uma estrutura dentro do *Event Storming* (BRANDOLINI, 2018). conforme pode ser observado na Figura 2.

- ***Domain Events***: Um *domain event* é algo que acontece com os componentes de negócio e

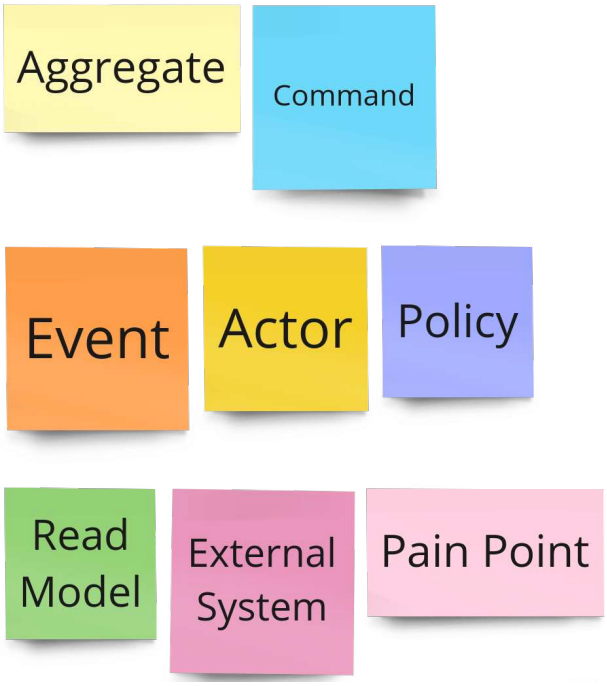


Figura 2 – Post-its representando os vários elementos do processo Event Storming
Fonte: Produzido pelo autor

que tem significado de negócio (FARIAS *et al.*, 2021).

- ***Commands***: Um comando representa uma ação capaz de causar eventos ou fluxos de eventos de domínio. Os comandos são representados por *post-its* azuis durante as sessões de *Event Storming* (FARIAS *et al.*, 2021).
- ***Policies***: Uma política é uma regra ou restrição que define como o sistema deve reagir a determinados eventos. É essencialmente uma reação a um evento que desencadeia uma ação ou comportamento específico. As políticas são representadas por *post-its* grandes roxos durante as sessões de *Event Storming* (FARIAS *et al.*, 2021).
- ***Actors***: Um ator representa um grupo de pessoas, um departamento, uma equipe ou uma pessoa específica envolvida em torno de um (grupo de) Eventos de Domínio. É essencialmente uma entidade que interage com o sistema e desempenha um papel na ocorrência de eventos ou na execução de comandos. Os atores são representados por pequenos *post-its* amarelos durante as sessões de *Event Storming* (FARIAS *et al.*, 2021).
- ***Aggregates***: Um aggregate representa um conjunto de objetos que são gerenciados como uma unidade lógica. Ele representa um conceito do domínio de negócios que é intrinsecamente relacionado e que deve ser tratado como uma unidade. *Aggregates* são representados por *post-its* amarelos claros durante as sessões de *Event Storming* (FARIAS *et al.*, 2021).
- ***Read Models***: um read model representa uma estrutura de dados que de uma visão

específica dos dados no domínio. Ele é projetado para ser usado por aplicativos externos para acessar dados do domínio. *Read Models* são representados por *post-its* verdes (FARIAS *et al.*, 2021).

- ***External Systems***: um sistema externo representa softwares que interagem com o sistema em questão, trocando dados ou serviços. Ele pode representar um sistema legado, um sistema de terceiros ou um sistema futuro. *External Systems* são representados por post-its rosa durante as sessões de *Event Storming* (FARIAS *et al.*, 2021).
- ***Pain points***: um *Pain Point* representa um ponto de dor identificado durante a dinâmica que indica áreas problemáticas no domínio. Eles podem destacar desafios, incertezas ou questões que merecem atenção especial. *Pain points* são representados por *post-its* rosa claro durante as sessões de *Event Storming* (FARIAS *et al.*, 2021).

2.4 *Test Driven Development*

Test Driven Development (TDD) é uma abordagem no desenvolvimento de *software* que enfatiza a criação incremental de testes unitários automatizados antes da implementação efetiva do código de produção. Nessa prática, os testes são concebidos e implementados de maneira progressiva, fornecendo uma estrutura sólida para orientar o desenvolvimento do *software* (BENATO; VILELA, 2021).

O TDD opera em ciclos iterativos e incrementais. Estes ciclos são frequentemente denominados como "*Red-Green-Refactor*"(Vermelho-Verde-Refatorar), e o processo segue as etapas a seguir:

- *Red* (Vermelho): Inicialmente, um teste é escrito antes da implementação do código. Este teste deve falhar, representando a condição inicial "vermelha"porque a funcionalidade ainda não foi implementada (BECK, 2022).
- *Green* (Verde): O código de produção é então desenvolvido para passar no teste recém-criado. A meta é transformar o estado "vermelho"em "verde", indicando que a funcionalidade agora está corretamente implementada (BECK, 2022).
- *Refactor* (Refatorar): Após a aprovação do teste, o código é aprimorado sem alterar seu comportamento externo. Este é o momento de refatorar para melhorar a estrutura, clareza e eficiência do código (BECK, 2022).

Este ciclo se repete continuamente, com novos testes sendo adicionados para novas funcionalidades ou alterações no código existente. O processo de TDD proporciona uma

abordagem estruturada e disciplinada para o desenvolvimento de software, garantindo testes abrangentes e promovendo a evolução constante do código (BECK, 2022).

2.5 Trabalhos relacionados

2.5.1 Desenvolvimento dirigido a domínio: um estudo de caso

O propósito do (BARBOSA, 2021) consistiu em examinar os diferentes padrões de aplicação do DDD e destacar suas principais vantagens em comparação com outras abordagens existentes. Foi desenvolvido um protótipo com o intuito de aprimorar a organização de uma empresa especializada em decoração de festas, concentrando-se exclusivamente no desenvolvimento do back-end da aplicação e aplicando os princípios do DDD (BARBOSA, 2021). Os resultados demonstraram que o protótipo conseguiu com sucesso implementar os conceitos do DDD, incluindo testes de unidade, desenvolvimento iterativo, separação em camadas e modelagem de domínio (BARBOSA, 2021).

2.5.2 Roteiro para a Utilização da Arquitetura Hexagonal no Projeto Dirigido pelo Domínio

(CESAR, 2016) aborda a complexidade no desenvolvimento de software, destacando a importância do DDD para criar modelos de domínio sólidos. Ele também enfatiza a necessidade de uma arquitetura adequada, como a Arquitetura Hexagonal, para separar o domínio da aplicação das complexidades de implementação.

O principal objetivo é fornecer um guia prático para desenvolvedores iniciantes, explicando como usar a Arquitetura Hexagonal de maneira coesa com o DDD em projetos de *software* complexos. Esse guia visa simplificar a integração entre domínios e oferecer flexibilidade para lidar com dispositivos de comunicação diversos. Em resumo, o trabalho busca ajudar os desenvolvedores a gerenciar a complexidade do desenvolvimento de *software*, garantindo a aderência ao DDD e a aplicação eficiente da Arquitetura Hexagonal para lidar com desafios de integração e comunicação (CESAR, 2016)

2.5.3 Desenvolvimento de Aplicações Utilizando DDD e Cloud Computing

(VIEIRA, 2023) aborda a aplicação dos conceitos de DDD no desenvolvimento de *software* e a criação de uma infraestrutura em nuvem. Para isso, foram propostos requisitos

no formato de casos de uso de um sistema. Esses requisitos foram utilizados para demonstrar que, através dos pilares do DDD, é possível separar domínios, utilizar boas práticas de desenvolvimento, definir uma linguagem comum a todos que fazem parte do sistema e construir um design orientado por modelos. Com a arquitetura proposta, foram criados recursos necessários de infraestrutura, através de código, para serem aplicados na nuvem. Esses recursos possibilitaram realizar a entrega das aplicações utilizando imagens Dockers. Além disso, toda a configuração de integração, permissão e rede de acesso, foram provisionadas através de códigos (VIEIRA, 2023).

3 METODOLOGIA

A presente pesquisa analisa a aplicação das práticas do DDD para a modelagem do domínio do Educaverso, uma plataforma de agregação de conteúdos educacionais. Para isso, foi realizada uma revisão de literatura que forneceu uma base sólida para o desenvolvimento da metodologia de pesquisa. A revisão abordou os principais conceitos do DDD, uma abordagem de desenvolvimento de software que foca na modelagem do domínio. Além dos principais livros sobre o tema, foram consultadas dissertações e artigos científicos em plataformas de pesquisa como Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), *Scielo*, *Institute of Electrical and Electronic Engineers* (IEEE) e *Google Scholar*. Em seguida, foi desenvolvido um protótipo de aplicação *back-end* para o *bounded context* de cursos, que é responsável por gerenciar as interações entre os principais usuários da plataforma (organizadores de conteúdos e estudantes) e os conteúdos da plataforma.

3.1 Desenvolvimento

Os conceitos do DDD foram aplicados para definir a metodologia deste trabalho, que consiste nos seguintes passos: levantamento de requisitos; aplicação do TDD; aplicação do DDD.

3.1.1 Levantamento dos requisitos

O levantamento de requisitos consiste em uma etapa fundamental no desenvolvimento de qualquer projeto. É neste momento que são identificados os objetivos do projeto, as funcionalidades que serão implementadas e as restrições que devem ser atendidas. Os requisitos podem ser divididos em dois tipos principais, que são: requisitos funcionais, que definem as funcionalidades que o sistema deve oferecer; e requisitos não funcionais, que definem as restrições e qualidades que o sistema deve atender.

3.1.1.1 Requisitos Funcionais

Os requisitos funcionais do Educaverso foram categorizados em três seções principais: (i) Core; (ii) Suporte; e (iii) Genérico:

- Os **requisitos core** representam o núcleo do domínio do Educaverso, ou seja, os conceitos

e processos essenciais para o funcionamento do sistema: Criar novo curso; Atualizar curso; Publicar curso; Despublicar curso; Criar módulo de curso; Atualizar módulo de curso; Ativar módulo de curso; Desativar módulo de curso; Criar Vídeo; Atualizar Video; Visualizar Vídeo; Completar Vídeo; Realizar Matrícula; Cancelar Matrícula; Visualizar Curso; Concluir Curso; Cadastrar Pergunta; Atualizar Pergunta; Resolver Desafio; Gerar Certificado.

- Os **requisitos de suporte** fornecem ao núcleo do domínio, ou seja, os conceitos e processos que são necessários para o funcionamento do sistema, mas que não são essenciais para o seu núcleo: Gerenciar Catálogo de Curso; Realizar Pesquisa por Cursos.
- Os **requisitos genéricos** não se encaixam nas categorias anteriores, ou seja, os conceitos e processos que são comuns a vários domínios: Autenticação e Autorização.

3.1.1.2 Requisitos não Funcionais

Os requisitos não funcionais do Educaverso são: (i) baixa latência; (ii) escalabilidade; (iii) disponibilidade; e (iv) consistência de dados:

- **Baixa Latência**: o sistema deve garantir tempos de resposta rápidos para as interações dos usuários, minimizando a latência das operações.
- **Escalabilidade**: o sistema deve ser projetado de forma a ser facilmente escalável, permitindo a adição de recursos e capacidade de processamento à medida que o número de usuários ou a carga de trabalho aumenta.
- **Disponibilidade**: o sistema deve manter um alto nível de disponibilidade, minimizando o tempo de inatividade planejado ou não planejado, de modo que os usuários possam acessá-lo de forma consistente.
- **Consistência de dados**: o sistema deve manter a integridade dos dados, garantindo que as informações sejam consistentes e atualizadas em todas as partes do sistema, independentemente da carga ou número de usuários.

3.1.2 Aplicação do TDD

A codificação das funcionalidades do Educaverso foi iniciada com testes unitários, seguindo o ciclo ***Red***, ***Green*** e ***Refactor***. Primeiro, foi escrito um teste que deveria falhar (*Red*), pois as entidades, regras de negócio etc. ainda não estavam implementadas. Em seguida, foi desenvolvido o mínimo de código necessário para que o teste passe (*Green*). Por fim, o código

foi refatorado (*Refactor*), aplicando boas práticas de programação.

3.1.3 Aplicação do DDD

O protótipo desenvolvido para demonstrar a arquitetura do educaverso utilizou o DDD em conjunto com *Clean Architecture* para organizar o código. Durante o desenvolvimento, foi realizada a modelagem dos agregados, entidades, valores de objetos, eventos de domínio, repositórios e serviços. Testes unitários e análise de cobertura de código também foram utilizados durante o desenvolvimento.

3.1.4 Design da plataforma

A Figura 3 mostra a organização estrutural do *bounded context* do serviço de cursos. A arquitetura da plataforma do Educaverso, inspirada na *clean architecture*, é composta por quatro camadas principais:

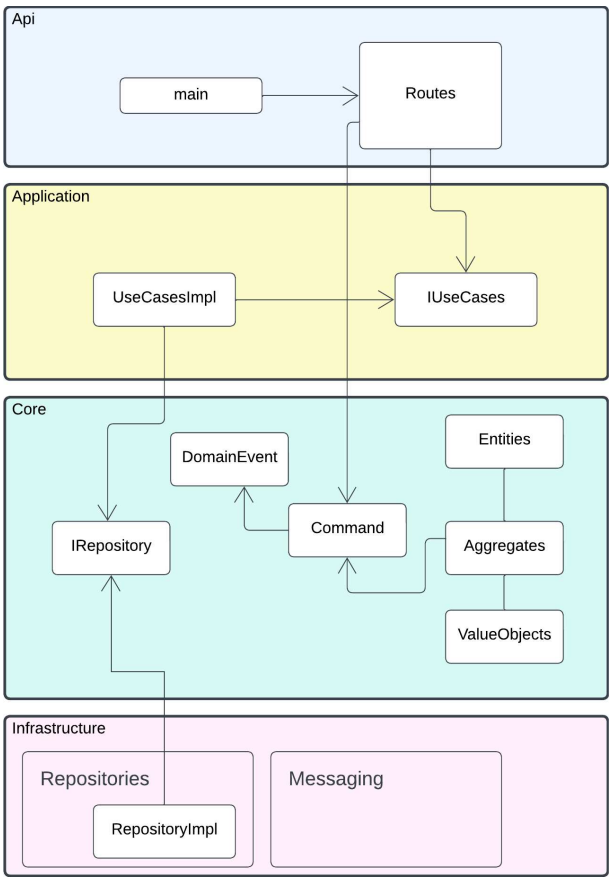


Figura 3 – Organização estrutural de bounded context do serviço de cursos
Fonte: Desenvolvido pelo autor

- **Camada de *Application Programming Interface* (API)**: fornece uma interface para que outros sistemas e aplicações possam interagir com o Educaverso;
- **Camada Aplicação**: implementa as funcionalidades do Educaverso, como gerenciamento de cursos, módulos, perguntas etc;
- **Camada de Domínio**: representa o domínio do negócio do Educaverso, como os conceitos e relacionamentos entre os objetos do mundo real;
- **Camada de Infraestrutura**: fornece os recursos físicos e lógicos necessários para o funcionamento do Educaverso, como servidores, banco de dados e redes;

3.1.5 Tecnologias utilizadas

- O ***Kotlin*** [1] é uma linguagem de programação moderna que é compilada para Java *bytecode*. Além disso, é uma linguagem segura, eficiente e expressiva que é uma ótima opção para desenvolvimento de aplicações Java. O *Kotlin* oferece uma variedade de recursos que o tornam uma linguagem atraente para o desenvolvimento de aplicações *web*, como suporte a programação funcional, injeção de dependência e testes de unidade.
- O ***Ktor*** [2] é um *framework web* para desenvolvimento de aplicações *Kotlin*. O *Ktor* é uma ferramenta poderosa e flexível que permite aos desenvolvedores criar aplicações *web* robustas e escaláveis. O *Ktor* oferece uma variedade de recursos que o tornam uma ótima opção para desenvolvimento de aplicações *web*, como suporte a *HTTP/2*, *WebSockets* e gerenciamento de sessões.
- O ***MongoDB*** [3] é um banco de dados *Not Only SQL* (NoSQL) que é escalável e eficiente. Dessa forma, é uma ótima opção para armazenar dados de aplicações *web*, pois oferece uma variedade de recursos que o tornam uma ótima opção para armazenamento de dados de aplicações *web*, como suporte a documentos *JavaScript Object Notation* (JSON), replicação e escalabilidade horizontal.
- O ***ElasticSearch*** [4] é um mecanismo de busca e análise de dados distribuído que pode ser usado para pesquisar, analisar e visualizar dados estruturados ou não estruturados.
- O ***IntelliJ*** [5] é uma ferramenta essencial para o desenvolvimento de aplicações *Kotlin*. Ele oferece suporte a recursos avançados de *Kotlin*, como inspeções de código, autocompleta-

[1] <https://kotlinlang.org/>
[2] <https://ktor.io/docs/welcome.html>
[3] <https://www.mongodb.com/docs/>
[4] <https://www.elastic.co/guide/index.html>
[5] <https://www.jetbrains.com/pt-br/idea/>

mento e refatoração. O *IntelliJ* também oferece uma variedade de plugins que podem ser usados para estender suas funcionalidades.

- O ***Docker*** [6] é uma tecnologia fundamental para a implantação e o gerenciamento de aplicativos em ambientes de nuvem. O *Docke*r permite que aplicativos sejam empacotados em unidades chamadas de containers. Os containers são leves e portáteis, o que os torna ideais para implantação em ambientes de nuvem.
- O ***GitHub***[7] é uma ferramenta essencial para o desenvolvimento colaborativo. Ele permite que equipes trabalhem juntas em projetos de *software*, mesmo que estejam localizadas em diferentes partes do mundo. O *GitHub* oferece uma variedade de recursos que facilitam a colaboração, como repositórios, *issues* e *pull requests*.
- O ***Postman*** [8] é uma ferramenta de API *testing* que permite aos desenvolvedores testar *APIs* de forma fácil e rápida. Alem disso, oferece uma variedade de recursos que o tornam uma ferramenta essencial para o desenvolvimento e a manutenção de *APIs*, como suporte a autenticação, geração de documentação e testes automatizados.
- O ***Jacoco*** [9] é um ferramenta de código aberto para análise de cobertura de código, que é uma métrica que mede a quantidade de código que é executado por um conjunto de testes. O *Jacoco* pode ser usado para código Java, *Kotlin* e *Scala*.
- ***Kafka*** [10] é um sistema de mensageria de publicação/assinatura que permite que os produtores publiquem dados em tópicos e os consumidores se inscrevam em tópicos para receber dados. Além disso, o *Kafka* trabalha em tempo real, o que significa que os dados podem ser publicados e recebidos em tempo real.
- ***Kafka Connect*** [11] é uma estrutura utilizada para importar e exportar serviços. Seu objetivo é facilitar a conexão entre serviços e aplicações de forma simples, rápida e eficiente utilizando o próprio *Kafka*.

Essas tecnologias são essenciais para o desenvolvimento do Educaverso, pois permitem que a plataforma seja desenvolvida de forma eficiente e segura, e que seja implantado em ambientes de nuvem.

---

6 <https://docs.docker.com/>
7 <https://github.com/>
8 <https://www.postman.com/>
9 <https://www.eclemma.org/jacoco/index.html>
10 <https://kafka.apache.org/documentation/>
11 <https://docs.confluent.io/platform/current/connect/index.html>

4 RESULTADOS

Nesta seção, são apresentados os resultados da aplicação de técnicas de arquitetura de *software* para o desenvolvimento do *bounded context* de serviço de cursos, responsável pelo gerenciamento de cursos na plataforma Educaverso.

4.1 *Bounded Contexts*

Os dois principais contextos delimitados da aplicação são os serviços de administração de cursos (*course service*) e o serviço de catálogo de cursos. A comunicação entre os *bounded contexts* de serviço de administração de cursos (*course service*) e serviço de catálogo de cursos (*course catalog service*) é assíncrona, através do pub/sub no Kafka. Isso permite a comunicação desacoplada e escalável. A Figura 4 ilustra o "*Context Map*" dos contextos identificados.

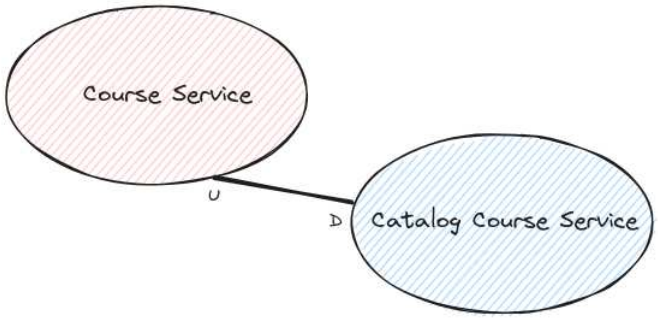


Figura 4 – *Context map* do serviço de cursos
Fonte: Produzido pelo autor, 2023

4.2 Modelagem das entidades, agregados e objetos de valores

Para modelar os elementos do serviço de curso de forma tática, foram identificadas entidades importantes para a aplicação das regras de negócio desse contexto, conforme a figura 5, utilizando a linguagem ubíqua dentro do respectivo *bounded context*. Por exemplo, o *aggregate root* chamado *Course*, representa a entidade de Cursos da plataforma. Nesta entidade, há uma regra de negócio em que o nome de um curso não pode ser vazio ou maior do que 255 caracteres. Para isso, essa regra foi facilmente implementada através do objeto de valor chamado *CourseName*, conforme observado no código fonte 1. Dessa forma, as regras de negócio ficam

desacopladas, facilitando a manutenção do código. O código completo do serviço de catálogo de cursos está disponível no apêndice A.

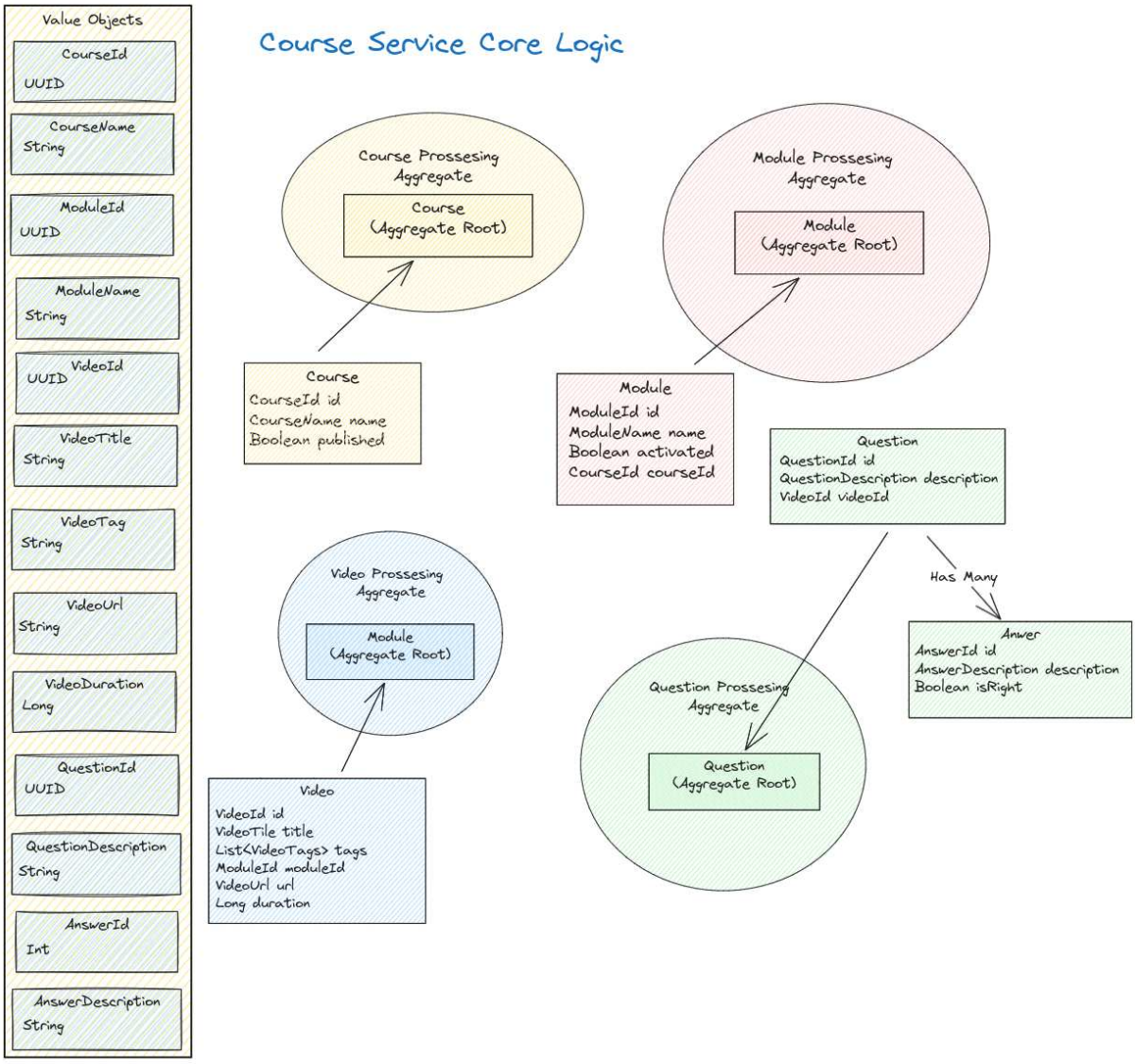


Figura 5 – Modelagem das entidades, agregados e objetos de valores para o serviço de cursos
Fonte: Produzido pelo autor, 2023, utilizando a ferramenta excalidraw

Código-fonte 1 – Objeto de valor *CourseName*

```
1 package app.educaverso.course.service.core.courses.fields
2
3 class CourseName(val value: String) {
4     init {
5         require(value.isNotEmpty()) { "Course name cannot
```

```
            be empty" }
        require(value.length <= 255) { "Course name cannot
            be longer than 255 characters" }
    }
}
```

4.3 Modelagem de contexto

Para ter uma visão geral da arquitetura do projeto, optou-se por criar uma diagrama de contexto fornecendo uma visão geral de alto nível do educaverso. A Figura 6 representa a aplicação, onde as partes interessadas são estudantes e organizadores de cursos. Os estudantes interagem com a aplicação acessando o serviço de catálogo de cursos, podendo realizar matrículas, ver o conteúdo de um curso, resolver desafios e gerar certificados. Os organizadores interagem com a aplicação para criar e gerenciar cursos.

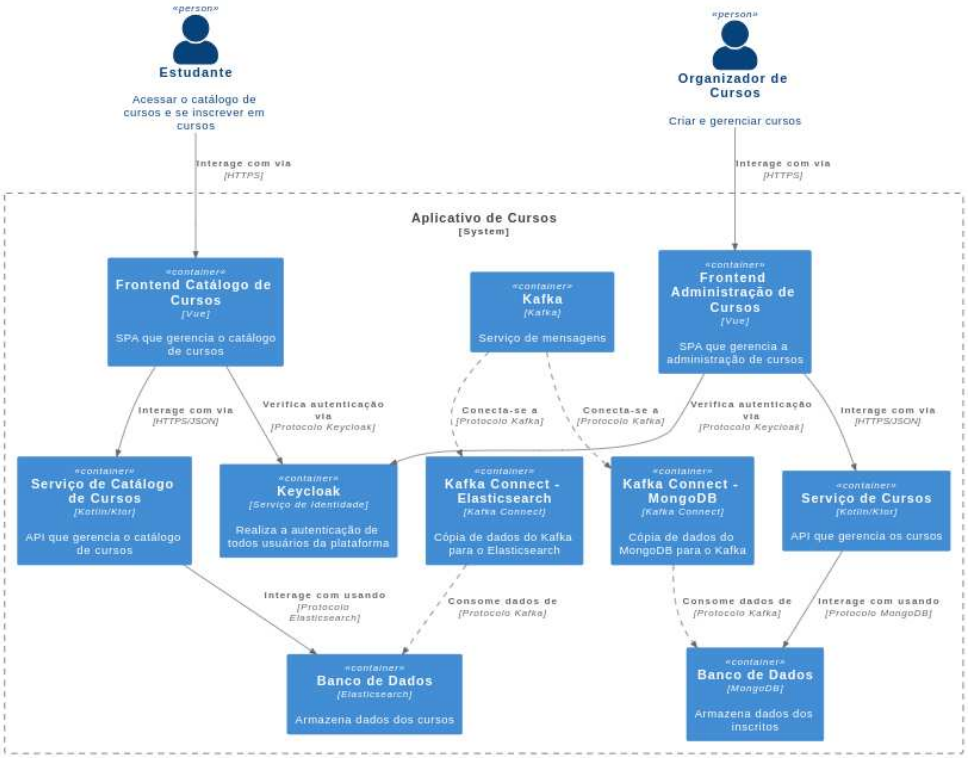


Figura 6 – Modelagem de Contexto
Fonte: Produzido pelo autor, 2023, utilizando o PlantUML

A plataforma é composta por quatro *containers* principais:

- ***Front-end* Catálogo de Cursos**: consiste em um aplicativo web que exibe o catálogo de

cursos e permite que os estudantes possam realizar matrículas etc.

- ***Front-end* Administração de Cursos**: consiste em um aplicativo web que permite que os organizadores possam criar e gerenciar cursos.
- **Serviço de Catálogo de Curso**: consiste em uma API que fornece dados sobre os cursos para o *front-end* de catálogo de cursos.
- **Serviço de Cursos**: consiste em uma API que fornece recursos de gerenciamento de cursos para o *front-end* de administração de cursos.

Os contêineres que representam o *Keycloak*, *Kafka*, *Kafka Connect*, banco de dados *MongoDB* e Banco de dados *Elasticsearch* são serviços auxiliares no subdomínio genérico, conforme os conceitos do DDD.

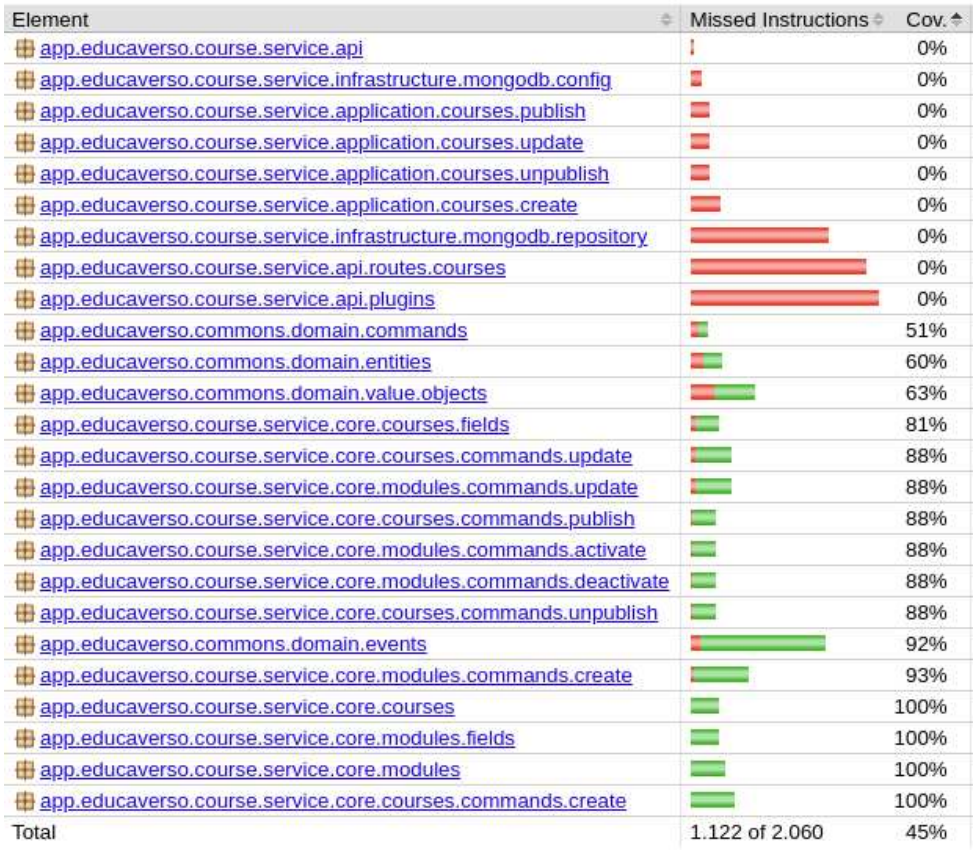
4.4 Cobertura de Testes

A cobertura de testes é uma medida da quantidade de código que é executado pelos testes de unidade. Ela é uma métrica importante para avaliar a qualidade dos testes e garantir que o código esteja funcionando corretamente. A figura 7, obtida a partir do relatório aplicado ao Educaverso através da ferramenta chamada Jacoco, mostra o resultado inicial da cobertura de testes. Ele funciona adicionando código instrumentador ao código fonte, o que permite que ele rastreie quais linhas de código são executadas pelos testes. A imagem indica que 1.122 instruções foram ignoradas pelos testes e que a cobertura de testes é de 45

Após analisar o relatório fornecido pelo Jacoco, testes adicionais foram realizados para cobrir as instruções ignoradas pelos testes atuais. Como exemplo, as figuras 8 e 9 mostram os principais problemas relacionados ao objeto de valor “CourseName”.

Após a aplicação de testes adicionais no objeto de valor “CourseName”, houve uma redução de 9 instruções ignoradas pelos testes, significando que 9 instruções que antes não eram executadas pelos testes agora são executadas. Isso é um progresso positivo, pois significa que o código está sendo testado com mais rigor. As figuras 10, 11 e 12 mostram o progresso positivo, mas que ainda é necessário para melhorar a cobertura de testes.

Após a escrita de sucessivos testes adicionais guiados pelo relatório da ferramenta Jacoco, obteve-se uma redução de 1.122 instruções ignoradas por testes para 810 instruções, representando uma redução de 29%. Isso significa que 312 instruções que antes não eram executadas pelos testes agora são executadas. A figura 13 mostra o relatório final apresentado pela ferramenta Jacoco.



Element	Missed Instructions	Cov.
app.educaverso.course.service.api		0%
app.educaverso.course.service.infrastructure.mongodb.config		0%
app.educaverso.course.service.application.courses.publish		0%
app.educaverso.course.service.application.courses.update		0%
app.educaverso.course.service.application.courses.unpublish		0%
app.educaverso.course.service.application.courses.create		0%
app.educaverso.course.service.infrastructure.mongodb.repository		0%
app.educaverso.course.service.api.routes.courses		0%
app.educaverso.course.service.api.plugins		0%
app.educaverso.commons.domain.commands		51%
app.educaverso.commons.domain.entities		60%
app.educaverso.commons.domain.value.objects		63%
app.educaverso.course.service.core.courses.fields		81%
app.educaverso.course.service.core.courses.commands.update		88%
app.educaverso.course.service.core.modules.commands.update		88%
app.educaverso.course.service.core.courses.commands.publish		88%
app.educaverso.course.service.core.modules.commands.activate		88%
app.educaverso.course.service.core.modules.commands.deactivate		88%
app.educaverso.course.service.core.courses.commands.unpublish		88%
app.educaverso.commons.domain.events		92%
app.educaverso.course.service.core.modules.commands.create		93%
app.educaverso.course.service.core.courses		100%
app.educaverso.course.service.core.modules.fields		100%
app.educaverso.course.service.core.modules		100%
app.educaverso.course.service.core.courses.commands.create		100%
Total	1.122 of 2.060	45%

Figura 7 – Relatório inicial do Jacoco

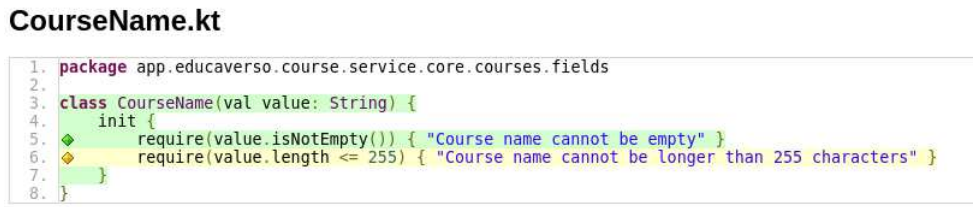
Fonte: Produzido pelo autor, 2023, utilizando a ferramenta jacoco

app.educaverso.course.service.core.courses.fields

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
CourseName		81%		75%	2	6	0	5	0	2	0	1
Total	9 of 48	81%	2 of 8	75%	2	6	0	5	0	2	0	1

Figura 8 – Relatório do objeto de valor CourseName - Cobertura de 75%

Fonte: Produzido pelo autor, 2023, utilizando a ferramenta jacoco

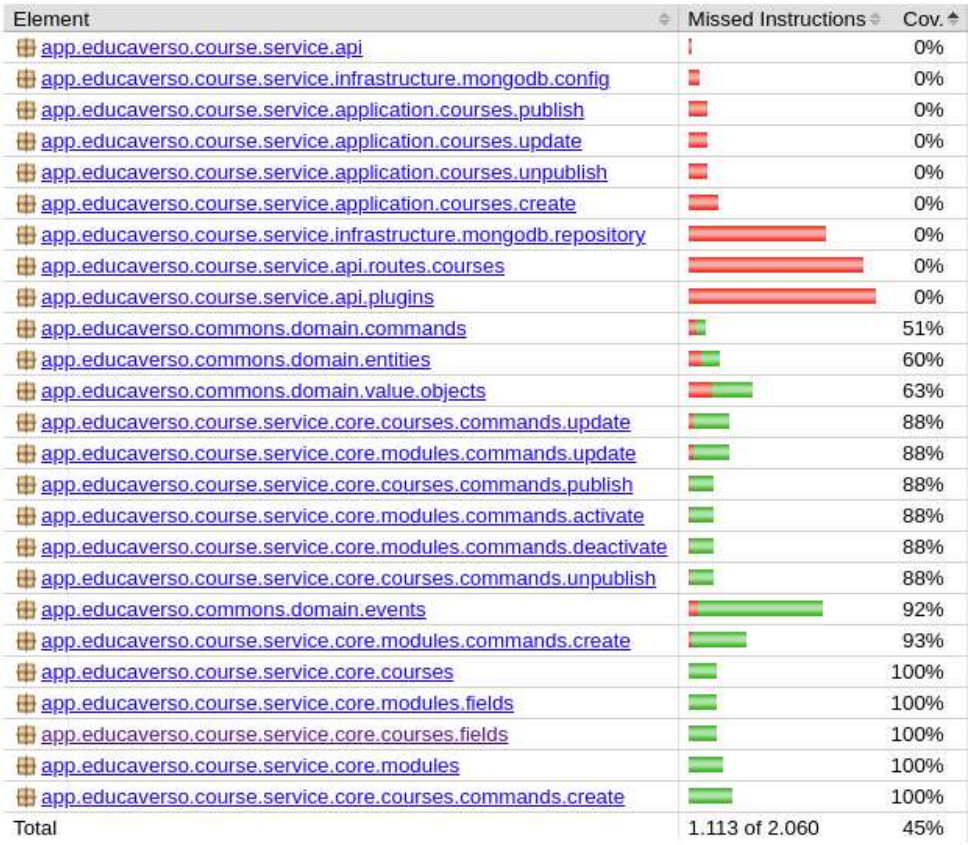


CourseName.kt

```
1. package app.educaverso.course.service.core.courses.fields
2.
3. class CourseName(val value: String) {
4.     init {
5.         require(value.isNotEmpty()) { "Course name cannot be empty" }
6.         require(value.length <= 255) { "Course name cannot be longer than 255 characters" }
7.     }
8. }
```

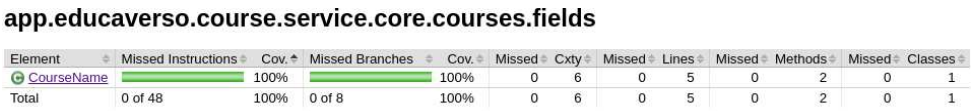
Figura 9 – Relatório de cobertura para objeto de valor CourseName

Fonte: Produzido pelo autor, 2023, utilizando a ferramenta jacoco



Element	Missed Instructions	Cov.
app.educaverso.course.service.api		0%
app.educaverso.course.service.infrastructure.mongodb.config		0%
app.educaverso.course.service.application.courses.publish		0%
app.educaverso.course.service.application.courses.update		0%
app.educaverso.course.service.application.courses.unpublish		0%
app.educaverso.course.service.application.courses.create		0%
app.educaverso.course.service.infrastructure.mongodb.repository		0%
app.educaverso.course.service.api.routes.courses		0%
app.educaverso.course.service.api.plugins		0%
app.educaverso.commons.domain.commands		51%
app.educaverso.commons.domain.entities		60%
app.educaverso.commons.domain.value.objects		63%
app.educaverso.course.service.core.courses.commands.update		88%
app.educaverso.course.service.core.modules.commands.update		88%
app.educaverso.course.service.core.courses.commands.publish		88%
app.educaverso.course.service.core.modules.commands.activate		88%
app.educaverso.course.service.core.modules.commands.deactivate		88%
app.educaverso.course.service.core.courses.commands.unpublish		88%
app.educaverso.commons.domain.events		92%
app.educaverso.course.service.core.modules.commands.create		93%
app.educaverso.course.service.core.courses		100%
app.educaverso.course.service.core.modules.fields		100%
app.educaverso.course.service.core.courses.fields		100%
app.educaverso.course.service.core.modules		100%
app.educaverso.course.service.core.courses.commands.create		100%
Total	1.113 of 2.060	45%

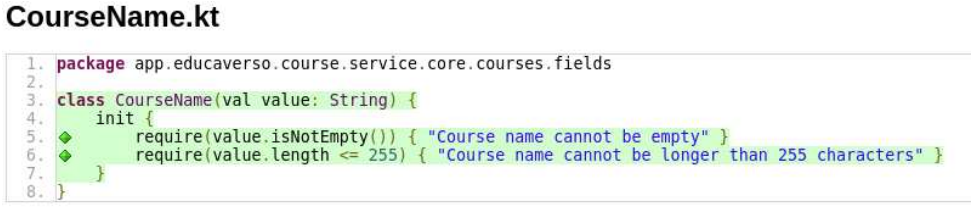
Figura 10 – Relatório de cobertura atualizado do objeto de valor CourseName
Fonte: Produzido pelo autor, 2023, utilizando a ferramenta jacoco



app.educaverso.course.service.core.courses.fields

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
CourseName		100%		100%	0	6	0	5	0	2	0	1
Total	0 of 48	100%	0 of 8	100%	0	6	0	5	0	2	0	1

Figura 11 – Relatório do objeto de valor CourseName
Fonte: Produzido pelo autor, 2023, utilizando a ferramenta jacoco



CourseName.kt

```
1. package app.educaverso.course.service.core.courses.fields
2.
3. class CourseName(val value: String) {
4.     init {
5.         require(value.isNotEmpty()) { "Course name cannot be empty" }
6.         require(value.length <= 255) { "Course name cannot be longer than 255 characters" }
7.     }
8. }
```

Figura 12 – Relatório atualizado do objeto de valor CourseName
Fonte: Produzido pelo autor, 2023, utilizando a ferramenta jacoco

Element	Missed Instructions	Cov.
app.educaverso.course.service.api		0%
app.educaverso.course.service.api.routes.courses		0%
app.educaverso.course.service.api.plugins		0%
app.educaverso.course.service.infrastructure.mongodb.repository		18%
app.educaverso.commons.domain.value.objects		87%
app.educaverso.commons.domain.commands		93%
app.educaverso.commons.domain.events		93%
app.educaverso.commons.domain.entities		96%
app.educaverso.course.service.infrastructure.mongodb.config		100%
app.educaverso.course.service.application.courses.publish		100%
app.educaverso.course.service.application.courses.update		100%
app.educaverso.course.service.application.courses.unpublish		100%
app.educaverso.course.service.core.courses.commands.publish		100%
app.educaverso.course.service.core.modules.commands.activate		100%
app.educaverso.course.service.core.modules.commands.deactivate		100%
app.educaverso.course.service.core.courses.commands.unpublish		100%
app.educaverso.course.service.core.courses		100%
app.educaverso.course.service.core.modules.fields		100%
app.educaverso.course.service.core.courses.fields		100%
app.educaverso.course.service.application.courses.create		100%
app.educaverso.course.service.core.modules		100%
app.educaverso.course.service.core.courses.commands.update		100%
app.educaverso.course.service.core.modules.commands.update		100%
app.educaverso.course.service.core.courses.commands.create		100%
app.educaverso.course.service.core.modules.commands.create		100%
Total	810 of 2.078	61%

Figura 13 – Relatório geral após a escrita de sucessivos testes adicionais

Fonte: Produzido pelo autor, 2023, utilizando a ferramenta jacoco

5 CONCLUSÃO

O Educaverso tem como objetivo centralizar e organizar recursos de forma eficiente, proporcionando uma experiência integrada para estudantes. Portanto, os requisitos e características-chave da plataforma foram definidas com sucesso, considerando a eficácia na organização e busca de conteúdos educacionais, utilizando o *Event Storming* para identificar tais elementos. Além disso, a aplicação do DDD permitiu que a arquitetura do Educaverso tenha uma boa organização de código, com boa cobertura nos testes e baixo acoplamento entre módulos do sistema. Isso foi feito identificando subdomínios, como o subdomínio de serviços de cursos e o subdomínio de catálogo de cursos. Cada subdomínio foi então delimitado por um contexto específico, o que ajudou a garantir que o código fosse desenvolvido de forma consistente e coesa. Além disso, foi implementado um protótipo funcional que resultou em código capaz de atender aos requisitos técnicos do projeto, sendo escalável e fácil de manter. Portanto, o código desenvolvido possui critérios de qualidade por ser de fácil manutenção, escalável e adaptável.

Além de tudo isso, o DDD é capaz de tornar a comunicação entre especialistas de negócio e especialistas técnicos eficiente, melhorando a interação entre diversos papéis das pessoas que integram o processo de desenvolvimento de software. Isso permite que o código desenvolvido reflita o máximo possível o domínio. O DDD ajuda a manter o código alinhado com o modelo de negócios, proporcionando uma base sólida para o desenvolvimento. Ele também facilita a criação de bounded contexts, permitindo diferentes modelos em contextos distintos. Isso simplifica a complexidade global do sistema.

Contudo, o uso do DDD pode não ser a opção mais apropriada para projetos de menor complexidade ou que envolvam domínios mais simples. Isso ocorre porque o DDD introduz uma complexidade adicional ao código, sendo mais adequado para projetos que possuam um domínio robusto e desafios de negócios complexos. Além disso, a implementação eficaz do DDD demanda estudo e compreensão aprofundados, devido à sua natureza não trivial. Isso resulta em uma curva de aprendizado significativa para o time de desenvolvimento.

Por fim, pretende-se concluir o desenvolvimento da aplicação conforme descrito na Figura 6, desenvolvendo outros contextos da aplicação, com foco nos front-ends e nos contextos do catálogo de cursos, focando em requisitos de comunicação e segurança.

REFERÊNCIAS

BARBOSA, E. P. Desenvolvimento dirigido a domínio: um estudo de caso. Universidade Federal de Uberlândia, 2021.

BECK, K. **Test driven development: By example**. [S.l.]: Addison-Wesley Professional, 2022.

BENATO, G. B.; VILELA, P. R. S. Test-driven development: uma revisão sistemática. **Revista Brasileira de Computação Aplicada**, v. 13, n. 1, p. 75–87, 2021.

BERTHOLINO, F. B. d. A.; SOUZA, G. V.; PERISSINOTTI, T. Desenvolvimento de uma arquitetura de software para um mecanismo de centralização e compartilhamento de dados médicos sensíveis. Universidade Presbiteriana Mackenzie, 2023.

BOB, U. The clean coder blog-the clean architecture. **Accessedon July**, v. 16, 2019. Disponível em: <https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>.

BRANDOLINI, A. Introducing eventstorming: An act of deliberate collective learning. **Leanpub: Victoria, BC, Canada**, 2018.

CESAR, L. J. Roteiro para a utilização da arquitetura hexagonal no projeto dirigido pelo domínio. 2016.

EVANS, E. **Domain-driven design: tackling complexity in the heart of software**. [S.l.]: Addison-Wesley Professional, 2004.

FARIAS, M. D. F.; KISHIMOTO, R. M. *et al.* Implementação da modelagem domain driven design em sistema de agendamento web: um relato de experiência. Centro Universitário do Estado do Pará, 2021.

FERREIRA, V. B. S.; FERREIRA, C. A.; GRANDE, E. T. G. Estado da arte da pesquisa em: Clean architecture e princípios de solid. **Research, Society and Development**, v. 11, n. 16, p. e335111637198–e335111637198, 2022.

GOMES, E. L. **BPM2DDD: identicando domínios a partir de processos de negócio**. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Norte, 2020.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. **ISO/IEC 9126-1:2003 - Engenharia de software – Qualidade de produto – Parte 1: Modelo de qualidade**. 2003. Disponível em: <https://jkolb.com.br/wp-content/uploads/2014/02/NBR-ISO_IEC-9126-1.pdf>.

KAPFERER, S.; ZIMMERMANN, O. Domain-specific language and tools for strategic domain-driven design, context mapping and bounded context modeling. In: **MODELSWARD**. [S.l.: s.n.], 2020. p. 299–306.

ROBERT, C. **Clean Architecture: A Craftsman's Guide to Software Structure and Design (Robert C. Martin Series)**. [S.l.]: Prentice Hall, 2019.

ROMERO, C. A. T.; ORTIZ, J. H.; KHALAF, O. I.; ORTEGA, W. M. Software architecture for planning educational scenarios by applying an agile methodology. **International Journal of Emerging Technologies in Learning (Online)**, International Association of Online Engineering (IAOE), v. 16, n. 8, p. 132, 2021.

VERNON, V. **Implementing domain-driven design**. [S.l.]: Addison-Wesley, 2013.

VIEIRA, M. D. S. Desenvolvimento de aplicações utilizando ddd e cloud computing. Universidade Federal de Uberlândia, 2023.

APÊNDICE A – CÓDIGO FONTE

```
1  package app.educaverso.commons.domain.entities
2
3  import app.educaverso.commons.domain.commands.ActionCommand
4  import app.educaverso.commons.domain.events.Event
5  import app.educaverso.commons.domain.value.objects.BaseId
6
7  abstract class BaseEntity<Id : BaseId<*>>(val id: Id) {
8
9      open fun <T : BaseEntity<Id>, E : Event> execute(
           actionCommand: ActionCommand<T, E>) {
10         val command = this as T
11         actionCommand.execute(command)
12     }
13
14     override fun hashCode(): Int {
15         return id.hashCode()
16     }
17
18     override fun equals(other: Any?): Boolean {
19         if (this === other) return true
20         if (other !is BaseEntity<*>) return false
21         return id == other.id
22     }
23
24 }
```

```
1  package app.educaverso.commons.domain.entities
2
3  import app.educaverso.commons.domain.value.objects.BaseId
4  import junit.framework.TestCase.assertNotNull
```

```
5  import junit.framework.TestCase.assertTrue
6  import org.junit.Test
7  import org.mockito.kotlin.mock
8  import kotlin.test.assertFalse
9
10 class BaseEntityTest {
11
12     @Test
13     fun `should has hashCode and equals`() {
14         // Given
15         val id = mock<BaseId<Int>> {}
16
17         // When
18         val baseEntity = object : BaseEntity<BaseId<Int>>(
               id) {}
19
20         // Then
21         assertNotNull(baseEntity.hashCode())
22         assertTrue((baseEntity == baseEntity))
23     }
24
25     @Test
26     fun `should be equals to other BaseEntity with same id
           `() {
27         // Given
28         val id = object : BaseId<Int>(1) {}
29         val otherId = object : BaseId<Int>(1) {}
30
31         // When
32         val baseEntity = object : BaseEntity<BaseId<Int>>(
               id) {}
33         val otherBaseEntity = object : BaseEntity<BaseId<
```

```
            Int>>(otherId) {}

        // Then
        assertTrue(baseEntity.equals(otherBaseEntity))

    }

    @Test
    fun `should be equals to other BaseEntity with same id
       and different type`() {
        // Given
        val id = object : BaseId<Int>(1) {}
        val otherId = object : BaseId<Int>(2) {}

        // When
        val baseEntity = object : BaseEntity<BaseId<Int>>(
           id) {}
        val otherBaseEntity = object : BaseEntity<BaseId<
           Int>>(otherId) {}

        // Then
        assertFalse(baseEntity.equals(otherBaseEntity))

    }

}
```

```
package app.educaverso.commons.domain.entities

import app.educaverso.commons.domain.value.objects.BaseId

abstract class AggregateRoot<Id : BaseId<*>>(id: Id) :
```

```
    BaseEntity<Id>(id)
```

```
package app.educaverso.commons.domain.value.objects

abstract class BaseId<T>(val value: T) {

    override fun hashCode(): Int {
        return value.hashCode()
    }

    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (other !is BaseId<*>) return false
        return value == other.value
    }
}
```

```
package app.educaverso.commons.domain.value.objects

import java.util.*

class CourseId(value: UUID) : BaseId<UUID>(value) {
    companion object {
        fun generate(): CourseId {
            return CourseId(UUID.randomUUID())
        }

        fun fromString(value: String): CourseId {
            return CourseId(UUID.fromString(value))
        }

    }
```

```
}
```

```
package app.educaverso.commons.domain.value.objects

import java.util.*

class EventId(value: UUID) : BaseId<UUID>(value) {
    companion object {
        fun generate(): EventId {
            return EventId(UUID.randomUUID())
        }
    }
}
```

```
package app.educaverso.commons.domain.value.objects

import java.util.*

class ModuleId(value: UUID) : BaseId<UUID>(value) {
    companion object {
        fun generate(): ModuleId {
            return ModuleId(UUID.randomUUID())
        }

        fun fromString(value: String): ModuleId {
            return ModuleId(UUID.fromString(value))
        }

    }
}
```

```
package app.educaverso.commons.domain.events
```

```
2
3  import kotlinx.coroutines.launch
4  import kotlinx.coroutines.runBlocking
5
6  object DomainEventDispatcher : EventDispatcher<Event> {
7
8      private val handlers: MutableMap<String, MutableList<
           EventHandler<Event>>> = mutableMapOf()
9
10     override fun register(handler: EventHandler<Event>) {
11         require(!has(handler)) { return }
12
13         this.handlers.computeIfAbsent(handler.eventName())
               { mutableListOf() }.add(handler)
14     }
15
16     override fun remove(handler: EventHandler<Event>) {
17
18         if (has(handler)) {
19             this.handlers[handler.eventName()]?.clear()
20
21             if (this.handlers[handler.eventName()].
                   isNullOrEmpty())
22                 this.handlers.remove(handler.eventName())
23
24         }
25
26     }
27
28     override fun dispatch(event: Event) {
29         runBlocking {
30             handlers[event.name()]?.forEach { handler ->
```

```
                launch {
                    handler.handle(event)
                }
            }
        }
    }

    override fun has(handler: EventHandler<Event>): Boolean {
        return this.handlers.containsKey(handler.eventName()) && this.handlers[handler.eventName()]!!.contains(handler)
    }

    override fun clear() {
        this.handlers.clear()
    }

    override fun handlers() = this.handlers.toMap()
}
```

```
package app.educaverso.commons.domain.events

import junit.framework.TestCase.assertEquals
import junit.framework.TestCase.assertTrue
import org.junit.Test
import org.mockito.Mockito.times
import org.mockito.Mockito.verify
import org.mockito.kotlin.doReturn
import org.mockito.kotlin.mock

class DomainEventDispatcherTest {
```

```
12
13      @Test
14      fun `should register an event handle`() {
15
16          // Given
17          val dispatcher = DomainEventDispatcher
18
19          val eventHandler = mock<EventHandler<Event>> {
20              on { eventName() } doReturn "event.created"
21          }
22
23          // When
24          dispatcher.register(eventHandler)
25
26          // Then
27          assertTrue(dispatcher.has(eventHandler))
28
29      }
30
31      @Test
32      fun `should unregister an event handle`() {
33          // Given
34          val dispatcher = DomainEventDispatcher
35
36          val eventHandler = mock<EventHandler<Event>> {
37              on { eventName() } doReturn "event.created"
38          }
39
40          // When
41          dispatcher.register(eventHandler)
42          dispatcher.remove(eventHandler)
43
```

```
44          // Then
45          assertTrue(!dispatcher.has(eventHandler))
46          assertTrue(!dispatcher.handlers().containsKey(
                eventHandler.eventName()))
47          assertTrue(dispatcher.handlers().isEmpty())
48      }
49
50
51      @Test
52      fun `should dispatch an event and notify the handler`()
            {
53          // Given
54          val dispatcher = DomainEventDispatcher
55
56          val eventHandler = mock<EventHandler<Event>> {
57              on { eventName() } doReturn "event.created"
58          }
59
60          val event = mock<Event> {
61              on { name() } doReturn "event.created"
62          }
63
64          // When
65          dispatcher.register(eventHandler)
66          dispatcher.dispatch(event)
67
68          // Then
69          verify(eventHandler, times(1)).handle(event)
70
71      }
72
73      @Test
```

```
74      fun `should clear all handlers`() {
75          // Given
76          val dispatcher = DomainEventDispatcher
77
78          val eventHandler = mock<EventHandler<Event>> {
79              on { eventName() } doReturn "event.created"
80          }
81
82          // When
83          dispatcher.register(eventHandler)
84          dispatcher.clear()
85
86          // Then
87          assertEquals(0, dispatcher.handlers().size)
88
89      }
90
91      @Test
92      fun `should not register an event handle twice`() {
93          // Given
94          val dispatcher = DomainEventDispatcher
95
96          val eventHandler = mock<EventHandler<Event>> {
97              on { eventName() } doReturn "event.created"
98          }
99
100         // When
101         dispatcher.register(eventHandler)
102         dispatcher.register(eventHandler)
103
104         // Then
105         assertEquals(1, dispatcher.handlers().size)
```

```
106         assertEquals(1, dispatcher.handlers()[eventHandler.
                eventName()]?.size)
107     }
108
109 }
```

```
1  package app.educaverso.commons.domain.events
2
3  interface EventDispatcher<T : Event> {
4      fun register(handler: EventHandler<T>)
5      fun remove(handler: EventHandler<T>)
6      fun dispatch(event: T)
7      fun has(handler: EventHandler<T>): Boolean
8      fun clear()
9      fun handlers(): Map<String, List<EventHandler<T>>>
10 }
```

```
1  package app.educaverso.commons.domain.events
2
3  import app.educaverso.commons.domain.value.objects.EventId
4  import java.time.Instant
5
6  abstract class Event(
7      val eventId: EventId = EventId.generate(),
8      val occurredOn: Instant = Instant.now()
9  ) {
10     abstract fun name(): String
11 }
```

```
1  package app.educaverso.commons.domain.events
2
3  interface EventHandler<T : Event> {
```

```
    fun handle(event: T)
    fun eventName(): String

}
```

```
package app.educaverso.commons.domain.commands

import app.educaverso.commons.domain.entities.BaseEntity
import app.educaverso.commons.domain.events.Event

abstract class ActionCommand<T : BaseEntity<*>, E : Event>
   : EventDispatcherCommand<E>() {
    abstract fun execute(entity: T)
}
```

```
package app.educaverso.commons.domain.commands

import app.educaverso.commons.domain.events.
   DomainEventDispatcher
import app.educaverso.commons.domain.events.Event

abstract class EventDispatcherCommand<T : Event> {

    var event: T? = null

    fun dispatch() {
        event?.let {
            DomainEventDispatcher.dispatch(it)
        }
    }

}
```

```
package app.educaverso.commons.domain.commands

import app.educaverso.commons.domain.entities.BaseEntity
import app.educaverso.commons.domain.events.Event

abstract class FactoryMethodDispatcherCommand<T :
    BaseEntity<*>, E : Event> : EventDispatcherCommand<E>()
    {
      abstract fun execute(): T
}
```

```
package app.educaverso.course.service.core.courses

import app.educaverso.commons.domain.entities.AggregateRoot
import app.educaverso.commons.domain.value.objects.CourseId
import app.educaverso.course.service.core.courses.fields.
    CourseName

class Course(
    id: CourseId = CourseId.generate(),
    var name: CourseName,
    var published: Boolean
) : AggregateRoot<CourseId>(id)
```

```
package app.educaverso.course.service.core.courses

import app.educaverso.commons.domain.value.objects.CourseId

interface CourseRepository {
    fun persist(course: Course)
    fun update(course: Course)
```

```
8       fun findBy(id: CourseId): Course
9   }
```

```
1   package app.educaverso.course.service.core.courses.fields
2
3   class CourseName(val value: String) {
4       init {
5           require(value.isNotEmpty()) { "Course name cannot
                be empty" }
6           require(value.length <= 255) { "Course name cannot
                be longer than 255 characters" }
7       }
8   }
```

```
1   package app.educaverso.course.service.core.courses.fields
2
3   import org.junit.Test
4   import kotlin.test.assertEquals
5   import kotlin.test.assertFails
6
7   class CourseNameTest {
8
9       @Test
10      fun `should create a new course name`() {
11          // Given
12          var aName = "Course name"
13
14          // When
15          var aCourseName = CourseName(aName)
16
17          // Then
18          assertEquals(aCourseName.value, aName)
```

```
19      }
20
21
22      @Test
23      fun `should launch an exception when the course name is
            empty`() {
24          // Given
25          var aName = ""
26
27          // When
28          var aError = assertFails { CourseName(aName) }
29
30          // Then
31          assertEquals(aError.message, "Course name cannot be
                empty")
32      }
33
34
35      @Test
36      fun `should launch an exception when the course name is
            greater than 255 characters`() {
37          // Given
38          var aName = "a".repeat(256)
39
40          // When
41          var aError = assertFails { CourseName(aName) }
42
43          // Then
44          assertEquals(aError.message, "Course name cannot be
                longer than 255 characters")
45      }
46
```

```
}
```

```
package app.educaverso.course.service.core.courses.commands
    .create

import app.educaverso.commons.domain.events.Event

data class CourseCreated(
    val courseId: String,
    val courseName: String
) : Event() {
    override fun name() = "course.created"
}
```

```
package app.educaverso.course.service.core.courses.commands
    .create

import app.educaverso.commons.domain.value.objects.CourseId
import junit.framework.TestCase.assertEquals
import org.junit.Test

class CourseCreatedTest {

    @Test
    fun `should create a course created event`() {
        // Given
        val courseId = CourseId.generate().value.toString()
        val courseName = "Course Name"

        // When
        val event = CourseCreated(
            courseId = courseId,
```

```
            courseName = courseName
        )

        // Then
        assertEquals(event.name(), "course.created")
        assertEquals(event.courseId, courseId)
        assertEquals(event.courseName, courseName)
    }

}
```

```
package app.educaverso.course.service.core.courses.commands
    .create

import app.educaverso.commons.domain.commands.
    FactoryMethodDispatcherCommand
import app.educaverso.course.service.core.courses.Course
import app.educaverso.course.service.core.courses.fields.
    CourseName

data class CreateCourse(val name: String) :
    FactoryMethodDispatcherCommand<Course, CourseCreated>()
    {
     override fun execute(): Course {

        val course = Course(
            name = CourseName(name),
            published = false
        )

        this.event = CourseCreated(
            courseId = course.id.value.toString(),
```

```
            courseName = name
        )

        return course
    }
}
```

```
package app.educaverso.course.service.core.courses.commands
    .create

import junit.framework.TestCase.*
import org.junit.Test

class CreateCourseTest {

    @Test
    fun `should create a new course`() {
        // Given
        val command = CreateCourse(
            name = "Course name"
        )

        // When
        val aCourse = command.execute()

        // Then
        assertNotNull(aCourse.id)
        assertEquals(aCourse.name.value, command.name)
        assertFalse(aCourse.published)

        assertNotNull(command.event)
        assertNotNull(command.event?.courseId)
```

```
25          assertNotNull(command.event?.courseName)
26          assertEquals(command.event?.name(), "course.created
               ")
27          assertNotNull(command.event?.occurredOn)
28          assertEquals(command.event?.courseId, aCourse.id.
               value.toString())
29          assertEquals(command.event?.courseName, aCourse.
               name.value)
30      }
31
32  }
```

```
1   package app.educaverso.course.service.core.courses.commands
       .publish
2
3   import app.educaverso.commons.domain.events.Event
4
5   data class CoursePublished(
6       val courseId: String
7   ) : Event() {
8
9       override fun name(): String {
10          return "course.published"
11      }
12  }
```

```
1   package app.educaverso.course.service.core.courses.commands
       .publish
2
3   import app.educaverso.commons.domain.value.objects.CourseId
4   import org.junit.Test
5   import kotlin.test.assertEquals
```

```
6
7  class CoursePublishedTest {
8
9      @Test
10     fun `should create a course published event`() {
11         // Given
12         val courseId = CourseId.generate().value.toString()
13
14         // When
15         val event = CoursePublished(
16             courseId
17         )
18
19         // Then
20         assertEquals(event.name(), "course.published")
21         assertEquals(event.courseId, courseId)
22
23     }
24
25 }
```

```
1  package app.educaverso.course.service.core.courses.commands
       .publish
2
3  import app.educaverso.commons.domain.commands.ActionCommand
4  import app.educaverso.course.service.core.courses.Course
5
6  class PublishCourse : ActionCommand<Course, CoursePublished
       >() {
7
8      override fun execute(entity: Course) {
9          entity.published = true
```

```
10
11          this.event = CoursePublished(
12              courseId = entity.id.value.toString()
13          )
14      }
15
16  }
```

```
1   package app.educaverso.course.service.core.courses.commands
        .publish
2
3   import app.educaverso.course.service.core.courses.commands.
        create.CreateCourse
4   import org.junit.Test
5   import kotlin.test.assertTrue
6
7   class PublishCourseTest {
8
9       @Test
10      fun `should publish a course`() {
11          // Given
12          var course = CreateCourse(
13              name = "Course name"
14          ).execute()
15
16
17          // When
18          course.execute(PublishCourse())
19
20          // Then
21          assertTrue(course.published)
22
```

```
    }

}
```

```
package app.educaverso.course.service.core.courses.commands
    .unpublish

import app.educaverso.commons.domain.events.Event

data class CourseUnpublished(
    val courseId: String
) : Event() {

    override fun name(): String {
        return "course.unpublished"
    }
}
```

```
package app.educaverso.course.service.core.courses.commands
    .unpublish

import app.educaverso.commons.domain.value.objects.CourseId
import org.junit.Test
import kotlin.test.assertEquals

class CourseUnpublishedTest {

    @Test
    fun `should unpublish a course event`() {

        // Given
        val courseId = CourseId.generate().value.toString()
```

```
14
15          // When
16          val event = CourseUnpublished(
17              courseId = courseId
18          )
19
20          // Then
21          assertEquals(event.name(), "course.unpublished")
22          assertEquals(event.courseId, courseId)
23
24      }
25  }
```

```
1   package app.educaverso.course.service.core.courses.commands
        .unpublish
2
3   import app.educaverso.commons.domain.commands.ActionCommand
4   import app.educaverso.course.service.core.courses.Course
5
6   class UnpublishCourse : ActionCommand<Course,
        CourseUnpublished>() {
7
8       override fun execute(entity: Course) {
9           entity.published = false
10
11          this.event = CourseUnpublished(
12              courseId = entity.id.value.toString()
13          )
14      }
15
16  }
```

```kotlin
package app.educaverso.course.service.core.courses.commands.unpublish

import app.educaverso.course.service.core.courses.commands.create.CreateCourse
import org.junit.Test
import kotlin.test.assertFalse

class UnpublishCourseTest {
    @Test
    fun `should unpublish a course`() {

        // Given
        var course = CreateCourse(
            name = "Course name"
        ).execute()


        // When
        course.execute(UnpublishCourse())

        // Then
        assertFalse(course.published)

    }
}
```

```kotlin
package app.educaverso.course.service.core.courses.commands.update

import app.educaverso.commons.domain.events.Event

```

```
data class CourseUpdated(
    val courseId: String,
    val name: String
) : Event() {

    override fun name(): String {
        return "course.updated"
    }
}
```

```
package app.educaverso.course.service.core.courses.commands
    .update

import app.educaverso.commons.domain.commands.ActionCommand
import app.educaverso.course.service.core.courses.Course
import app.educaverso.course.service.core.courses.fields.
    CourseName

class UpdateCourse(val name: String) : ActionCommand<Course
    , CourseUpdated>() {
     override fun execute(entity: Course) {

         entity.name = CourseName(name)

         this.event = CourseUpdated(
             courseId = entity.id.value.toString(),
             name = name
         )
     }

}
```

```
package app.educaverso.course.service.core.modules

import app.educaverso.commons.domain.entities.AggregateRoot
import app.educaverso.commons.domain.value.objects.CourseId
import app.educaverso.commons.domain.value.objects.ModuleId
import app.educaverso.course.service.core.modules.fields.
   ModuleName

class Module(
    id: ModuleId = ModuleId.generate(),
    val courseId: CourseId,
    var name: ModuleName,
    var activated: Boolean
): AggregateRoot<ModuleId>(id)
```

```
package app.educaverso.course.service.core.modules.fields

class ModuleName(val value: String) {
    init {
        require(value.isNotEmpty()) { "Module name cannot
           be empty" }
        require(value.length <= 100) { "Module name cannot
           be longer than 100 characters" }

    }
}
```

```
package app.educaverso.course.service.core.courses.fields

import org.junit.Test
import kotlin.test.assertEquals
import kotlin.test.assertFails
```

```
6
7  class CourseNameTest {
8
9      @Test
10     fun `should create a new course name`() {
11         // Given
12         var aName = "Course name"
13
14         // When
15         var aCourseName = CourseName(aName)
16
17         // Then
18         assertEquals(aCourseName.value, aName)
19     }
20
21
22     @Test
23     fun `should launch an exception when the course name is
           empty`() {
24         // Given
25         var aName = ""
26
27         // When
28         var aError = assertFails { CourseName(aName) }
29
30         // Then
31         assertEquals(aError.message, "Course name cannot be
               empty")
32     }
33
34
35     @Test
```

```
36      fun `should launch an exception when the course name is
            greater than 255 characters`() {
37          // Given
38          var aName = "a".repeat(256)
39
40          // When
41          var aError = assertFails { CourseName(aName) }
42
43          // Then
44          assertEquals(aError.message, "Course name cannot be
                longer than 255 characters")
45      }
46
47  }
```

```
1   package app.educaverso.course.service.core.modules.commands
        .activate
2
3   import app.educaverso.commons.domain.commands.ActionCommand
4   import app.educaverso.course.service.core.modules.Module
5   class ActivateModule: ActionCommand<Module, ModuleActivated
        >() {
6       override fun execute(entity: Module) {
7           entity.activated = true
8
9           this.event = ModuleActivated(
10              moduleId = entity.id.value.toString()
11          )
12      }
13  }
```

```
1   package app.educaverso.course.service.core.modules.commands
```

```
    .activate

import app.educaverso.commons.domain.events.Event

data class ModuleActivated(
    val moduleId: String
) : Event() {
    override fun name(): String {
        return "module.activated"
    }
}
```

```
package app.educaverso.course.service.core.modules.commands
    .create

import app.educaverso.commons.domain.commands.
    FactoryMethodDispatcherCommand
import app.educaverso.commons.domain.value.objects.CourseId
import app.educaverso.course.service.core.modules.Module
import app.educaverso.course.service.core.modules.fields.
    ModuleName

data class CreateModule(val name: String, val courseId:
    String) :
    FactoryMethodDispatcherCommand<Module, ModuleCreated>()
        {

    override fun execute(): Module {

        val module = Module(
            courseId = CourseId.fromString(courseId),
            name = ModuleName(name),
```

```
            activated = false
        )

        this.event = ModuleCreated(
            moduleId = module.id.value.toString(),
            moduleName = name,
            courseId = courseId
        )

        return module
    }

}
```

```
package app.educaverso.course.service.core.modules.commands
    .create

import app.educaverso.commons.domain.events.Event

data class ModuleCreated(
    val moduleId: String,
    val moduleName: String,
    val courseId: String,
) : Event() {
    override fun name() = "module.created"
}
```

```
package app.educaverso.course.service.core.modules.commands
    .deactivate

import app.educaverso.commons.domain.commands.ActionCommand
import app.educaverso.course.service.core.modules.Module
```

```
5
6  class DeactivateModule : ActionCommand<Module,
       ModuleDeactivated>() {
7      override fun execute(entity: Module) {
8          entity.activated = false
9
10         this.event = ModuleDeactivated(
11             moduleId = entity.id.value.toString()
12         )
13     }
14
15 }
```

```
1  package app.educaverso.course.service.core.modules.commands
       .deactivate
2
3  import app.educaverso.commons.domain.events.Event
4
5  data class ModuleDeactivated(
6      val moduleId: String
7  ) : Event() {
8      override fun name(): String {
9          return "module.deactivated"
10     }
11 }
```

```
1  package app.educaverso.course.service.core.modules.commands
       .update
2
3  import app.educaverso.commons.domain.events.Event
4
5  data class ModuleUpdated(
```

```
    val moduleId: String,
    val name: String
) : Event() {
    override fun name(): String {
        return "module.updated"
    }
}
```

```
package app.educaverso.course.service.core.modules.commands
    .update

import app.educaverso.commons.domain.commands.ActionCommand
import app.educaverso.course.service.core.modules.Module
import app.educaverso.course.service.core.modules.fields.
    ModuleName

class UpdateModule(val name: String): ActionCommand<Module,
    ModuleUpdated>() {
    override fun execute(entity: Module) {
        entity.name = ModuleName(name)

        this.event = ModuleUpdated(
            moduleId = entity.id.value.toString(),
            name = name
        )
    }
}
```

```
package app.educaverso.course.service.application.courses.
    create

data class CreateCourseOutput(
```

```
    val id: String
)
```

```
package app.educaverso.course.service.application.courses.
    create

import app.educaverso.course.service.core.courses.commands.
    create.CreateCourse

interface CreateCourseUseCase {
    fun execute(createCourse: CreateCourse):
        CreateCourseOutput

}
```

```
package app.educaverso.course.service.application.courses.
    create

import app.educaverso.course.service.core.courses.Course
import app.educaverso.course.service.core.courses.
    CourseRepository
import app.educaverso.course.service.core.courses.commands.
    create.CreateCourse

class CreateCourseUseCaseImpl(private val repository:
    CourseRepository) : CreateCourseUseCase {
    override fun execute(command: CreateCourse):
        CreateCourseOutput {

        val course = command.execute()

        repository.persist(course)
```

```
13
14          command.dispatch()
15
16          return mapToCreateCourseOutput(course)
17      }
18
19      private fun mapToCreateCourseOutput(course: Course) =
            CreateCourseOutput(
20          id = course.id.value.toString()
21      )
22
23  }
```

```
1   package app.educaverso.course.service.application.courses.
        publish
2
3   import app.educaverso.commons.domain.value.objects.CourseId
4   import app.educaverso.course.service.core.courses.commands.
        publish.PublishCourse
5
6   interface PublishCourseUseCase {
7
8       fun execute(courseId: CourseId, command: PublishCourse)
9
10  }
```

```
1   package app.educaverso.course.service.application.courses.
        publish
2
3   import app.educaverso.commons.domain.value.objects.CourseId
4   import app.educaverso.course.service.core.courses.
        CourseRepository
```

```
5  import app.educaverso.course.service.core.courses.commands.
       publish.PublishCourse
6
7  class PublishCourseUseCaseImpl(private val repository:
       CourseRepository) : PublishCourseUseCase {
8
9      override fun execute(courseId: CourseId, command:
           PublishCourse) {
10
11         val course = repository.findBy(courseId)
12
13         course.execute(command)
14
15         repository.update(course)
16
17         command.dispatch()
18
19     }
20
21 }
```

```
1  package app.educaverso.course.service.application.courses.
       unpublish
2
3  import app.educaverso.commons.domain.value.objects.CourseId
4  import app.educaverso.course.service.core.courses.commands.
       unpublish.UnpublishCourse
5
6  interface UnpublishCourseUseCase {
7
8      fun execute(courseId: CourseId, command:
           UnpublishCourse)
```

```
9
10 }
```

```
1  package app.educaverso.course.service.application.courses.
       unpublish
2
3  import app.educaverso.commons.domain.value.objects.CourseId
4  import app.educaverso.course.service.core.courses.
       CourseRepository
5  import app.educaverso.course.service.core.courses.commands.
       unpublish.UnpublishCourse
6
7  class UnpublishCourseUseCaseImpl(private val repository:
       CourseRepository) : UnpublishCourseUseCase {
8
9      override fun execute(courseId: CourseId, command:
           UnpublishCourse) {
10
11         val course = repository.findBy(courseId)
12
13         course.execute(command)
14
15         repository.update(course)
16
17         command.dispatch()
18
19     }
20
21 }
```

```
1  package app.educaverso.course.service.application.courses.
       update
```

```
2
3  import app.educaverso.commons.domain.value.objects.CourseId
4  import app.educaverso.course.service.core.courses.commands.
       update.UpdateCourse
5
6  interface UpdateCourseUseCase {
7
8      fun execute(courseId: CourseId, updateCourse:
           UpdateCourse)
9
10 }
```

```
1  package app.educaverso.course.service.application.courses.
       update
2
3  import app.educaverso.commons.domain.value.objects.CourseId
4  import app.educaverso.course.service.core.courses.
       CourseRepository
5  import app.educaverso.course.service.core.courses.commands.
       update.UpdateCourse
6
7  class UpdateCourseUseCaseImpl(private val repository:
       CourseRepository) : UpdateCourseUseCase {
8
9      override fun execute(courseId: CourseId, updateCourse:
           UpdateCourse) {
10
11         val course = repository.findBy(courseId)
12
13         course.execute(updateCourse)
14
15         repository.update(course)
```

```
16
17          updateCourse.dispatch()
18
19      }
20
21  }
```

```
1  package app.educaverso.course.service.infrastructure.
       mongodb.config
2
3  import com.mongodb.kotlin.client.coroutine.MongoClient
4
5  private val URI = System.getenv("MONGODB_URI") ?: "mongodb
       ://localhost:27017"
6  private const val DATABASE_NAME = "educaverso"
7  private val client = MongoClient.create(URI)
8  private val database = client.getDatabase(DATABASE_NAME)
9
10 object MongodbConnection {
11     fun getDatabase() = database
12 }
```

```
1  package app.educaverso.course.service.infrastructure.
       mongodb.repository
2
3  import app.educaverso.commons.domain.value.objects.CourseId
4  import app.educaverso.course.service.core.courses.Course
5  import app.educaverso.course.service.core.courses.fields.
       CourseName
6  import org.bson.codecs.pojo.annotations.BsonId
7
8  data class CourseDocument(
```

```
    @BsonId val id: String,
    val name: String,
    val published: Boolean
) {

    fun toCourse(): Course {
        return Course(
            id = CourseId.fromString(id),
            name = CourseName(name),
            published = published
        )
    }

}
```

```
package app.educaverso.course.service.infrastructure.
    mongodb.repository

import app.educaverso.commons.domain.value.objects.CourseId
import app.educaverso.course.service.core.courses.Course
import app.educaverso.course.service.core.courses.
    CourseRepository
import app.educaverso.course.service.infrastructure.mongodb
    .config.MongodbConnection
import com.mongodb.client.model.Filters.eq
import kotlinx.coroutines.flow.firstOrNull
import kotlinx.coroutines.runBlocking

class CourseMongodbRepository : CourseRepository {

    private val collection = MongodbConnection.getDatabase
        ().getCollection<CourseDocument>("courses")
```

```
14
15      override fun persist(course: Course) {
16
17          runBlocking {
18              collection.insertOne(
19                  CourseDocument(
20                      id = course.id.value.toString(),
21                      name = course.name.value,
22                      published = course.published
23                  )
24              )
25          }
26      }
27
28      override fun update(course: Course) {
29
30          runBlocking {
31              collection.replaceOne(
32                  eq("_id", course.id.value.toString()),
33                  CourseDocument(
34                      id = course.id.value.toString(),
35                      name = course.name.value,
36                      published = course.published
37                  )
38              )
39          }
40      }
41
42      override fun findBy(id: CourseId): Course = runBlocking
            {
43
44          val result = collection.find(
```

```
            eq("_id", id.value.toString())
        ).firstOrNull()?.toCourse()

        require(result != null) { "Course not found" }

        result
    }

}
```

```
package app.educaverso.course.service.api

import io.ktor.server.netty.EngineMain.main

fun main(args: Array<String>): Unit = main(args)
```

```
package app.educaverso.course.service.api.plugins

import io.ktor.http.*
import io.ktor.serialization.gson.*
import io.ktor.server.application.*
import io.ktor.server.plugins.compression.*
import io.ktor.server.plugins.contentnegotiation.*
import io.ktor.server.plugins.statuspages.*
import io.ktor.server.response.*
import org.koin.ktor.plugin.Koin

fun Application.module() {

    configureRouting()

    install(Compression) {
```

```
17          gzip()
18      }
19
20      install(ContentNegotiation) {
21
22          gson {
23              setPrettyPrinting()
24              setLenient()
25          }
26
27      }
28
29      install(Koin) {
30          modules(repositories)
31          modules(useCases)
32      }
33
34      install(StatusPages) {
35          exception<Throwable> { call, cause ->
36
37              when (cause) {
38                  is IllegalArgumentException -> call.respond
                        (
39                      message = "{\"msg\": ${cause.message}}"
                            ,
40                      status = HttpStatusCode.BadRequest
41                  )
42
43                  else -> call.respond(
44                      message = "{\"msg\": ${cause.message}}"
                            ,
45                      status = HttpStatusCode.
```

```
                        InternalServerError
                )
            }
        }
    }
}
```

```
package app.educaverso.course.service.api.plugins

import app.educaverso.course.service.core.courses.
    CourseRepository
import app.educaverso.course.service.infrastructure.mongodb
    .repository.CourseMongodbRepository
import org.koin.core.module.dsl.bind
import org.koin.core.module.dsl.singleOf
import org.koin.dsl.module

val repositories = module {
    singleOf(::CourseMongodbRepository) { bind<
        CourseRepository>() }
}
```

```
package app.educaverso.course.service.api.plugins

import app.educaverso.course.service.api.routes.courses.
    courses
import io.ktor.server.application.*
import io.ktor.server.routing.*

fun Application.configureRouting() {
    routing {
        courses()
```

```
    }
}
```

```
package app.educaverso.course.service.api.plugins

import app.educaverso.course.service.application.courses.
   create.CreateCourseUseCase
import app.educaverso.course.service.application.courses.
   create.CreateCourseUseCaseImpl
import app.educaverso.course.service.application.courses.
   publish.PublishCourseUseCase
import app.educaverso.course.service.application.courses.
   publish.PublishCourseUseCaseImpl
import app.educaverso.course.service.application.courses.
   unpublish.UnpublishCourseUseCase
import app.educaverso.course.service.application.courses.
   unpublish.UnpublishCourseUseCaseImpl
import app.educaverso.course.service.application.courses.
   update.UpdateCourseUseCase
import app.educaverso.course.service.application.courses.
   update.UpdateCourseUseCaseImpl
import org.koin.core.module.dsl.bind
import org.koin.core.module.dsl.singleOf
import org.koin.dsl.module

val useCases = module {
    singleOf(::CreateCourseUseCaseImpl) { bind<
       CreateCourseUseCase>() }
    singleOf(::UpdateCourseUseCaseImpl) { bind<
       UpdateCourseUseCase>() }
    singleOf(::PublishCourseUseCaseImpl) { bind<
       PublishCourseUseCase>() }
```

```
    singleOf(::UnpublishCourseUseCaseImpl) { bind<
        UnpublishCourseUseCase>() }
}
```

```
package app.educaverso.course.service.api.routes.courses

import app.educaverso.commons.domain.value.objects.CourseId
import app.educaverso.course.service.application.courses.
    create.CreateCourseUseCase
import app.educaverso.course.service.application.courses.
    publish.PublishCourseUseCase
import app.educaverso.course.service.application.courses.
    unpublish.UnpublishCourseUseCase
import app.educaverso.course.service.application.courses.
    update.UpdateCourseUseCase
import app.educaverso.course.service.core.courses.commands.
    create.CreateCourse
import app.educaverso.course.service.core.courses.commands.
    publish.PublishCourse
import app.educaverso.course.service.core.courses.commands.
    unpublish.UnpublishCourse
import app.educaverso.course.service.core.courses.commands.
    update.UpdateCourse
import io.ktor.http.*
import io.ktor.server.application.*
import io.ktor.server.request.*
import io.ktor.server.response.*
import io.ktor.server.routing.*
import org.koin.ktor.ext.inject

fun Route.courses() {

```

```
21      val createCourse by inject<CreateCourseUseCase>()
22      val updateCourse by inject<UpdateCourseUseCase>()
23      val publishCourse by inject<PublishCourseUseCase>()
24      val unpublishCourse by inject<UnpublishCourseUseCase>()
25
26      route("courses") {
27
28          post {
29              val command = call.receive<CreateCourse>()
30
31              val output = createCourse.execute(command)
32
33              call.respond(
34                  status = HttpStatusCode.Created,
35                  message = output
36              )
37          }
38
39          put("{courseId}") {
40
41              val courseId = call.parameters["courseId"]
42
43              val command = call.receive<UpdateCourse>()
44
45              updateCourse.execute(CourseId.fromString(
                    courseId.toString()), command)
46
47              call.respond("Course updated")
48          }
49
50          put("{courseId}/publish") {
51
```

```
52              val courseId = call.parameters["courseId"]
53
54              val command = PublishCourse()
55
56              publishCourse.execute(CourseId.fromString(
                    courseId.toString()), command)
57
58              call.respond("Course published")
59          }
60
61          put("{courseId}/unpublish") {
62
63              val courseId = call.parameters["courseId"]
64
65              val command = UnpublishCourse()
66
67              unpublishCourse.execute(CourseId.fromString(
                    courseId.toString()), command)
68
69              call.respond("Course unpublished")
70          }
71      }
72  }
```

```
1   ktor {
2
3       deployment {
4           port = 8080
5           port = ${?PORT}
6       }
7
8       application {
```

```
9           modules = [ app.educaverso.course.service.api.
                plugins.ApplicationKt.module ]
10      }
11  }
```