Material Completo sobre Domain-Driven Design (DDD) com Exemplo Prático

Este material visa apresentar os conceitos fundamentais do Domain-Driven Design (DDD) e demonstrar sua aplicação prática através de um exemplo de sistema de Catálogo de Filmes, implementado em Python sem o uso de frameworks complexos. O objetivo é fornecer uma compreensão didática sobre como as regras de negócio e o comportamento do domínio devem ser o centro do desenvolvimento de software.

Sumário

---

1. Introdução ao Domain-Driven Design (DDD)

O **Domain-Driven Design (DDD)** é uma abordagem de desenvolvimento de software que coloca o **domínio** — a área de conhecimento e as regras de negócio do problema que o software busca resolver — no centro do processo de design. Em vez de focar primeiramente na tecnologia ou na infraestrutura, o DDD enfatiza a compreensão profunda e a modelagem precisa do negócio.

O que é DDD?

Proposto por Eric Evans em seu livro "Domain-Driven Design: Tackling Complexity in the Heart of Software" [1], o DDD é uma filosofia que busca alinhar o código com a linguagem e a lógica do negócio. Isso é feito através da criação de um **Modelo de Domínio** rico, que encapsula as regras, comportamentos e conceitos do negócio. O objetivo é criar um software que seja uma representação clara e expressiva do domínio, facilitando a comunicação entre especialistas de domínio e desenvolvedores.

Por que usar DDD?

A adoção do DDD é particularmente benéfica em projetos com alta complexidade de negócio. Seus principais benefícios incluem:

- **Clareza e Expressividade:** O código reflete diretamente o negócio, tornando-o mais fácil de entender e manter.
- **Manutenibilidade:** Mudanças nas regras de negócio podem ser implementadas de forma mais isolada e controlada, reduzindo o risco de efeitos colaterais.
- **Colaboração:** Promove uma linguagem comum e compartilhada entre as equipes de negócio e desenvolvimento, eliminando ambiguidades.
- **Qualidade do Software:** Ajuda a construir sistemas mais robustos, testáveis e menos propensos a erros de lógica de negócio.

2. Padrões Estratégicos do DDD

Os padrões estratégicos do DDD ajudam a lidar com a complexidade em larga escala, focando na organização do modelo de domínio e na colaboração entre equipes.

Domínio

O **Domínio** é o **coração das regras de negócio** que o sistema precisa respeitar. Ele representa a área de conhecimento específica para a qual o software está sendo construído. No contexto do nosso exemplo, o domínio é o **Catálogo de Filmes**, onde as regras sobre categorias, filmes, atores e classificações são centrais.

Linguagem Ubíqua (Ubiquitous Language)

A **Linguagem Ubíqua** é a linguagem comum e compartilhada, desenvolvida e usada por todos os membros da equipe (desenvolvedores, especialistas de domínio, gerentes) dentro de um contexto específico. É a linguagem que se reflete diretamente no código do domínio, nos testes, na documentação e nas conversas. O uso de uma linguagem ubíqua é fundamental para evitar a "perda de tradução" entre o que o negócio precisa e o que o software implementa.

Contextos Delimitados (Bounded Contexts)

Um **Contexto Delimitado** é um limite lógico dentro de um sistema onde um modelo de domínio específico é definido e aplicado. Dentro de um contexto, os termos da Linguagem Ubíqua têm um significado preciso e consistente. Fora desse contexto, os mesmos termos podem ter significados diferentes ou não existir.

Exemplo: Em um sistema de streaming, o termo "Filme" pode ter um significado no contexto de "Catálogo" (com atributos como título, sinopse, elenco) e um significado diferente no contexto de "Licenciamento" (com atributos como data de expiração da licença, custo, região).

3. Padrões Táticos do DDD

Os padrões táticos fornecem um conjunto de blocos de construção para a criação de um modelo de domínio rico e expressivo.

Entidades (Entities)

Uma **Entidade** é um objeto que possui uma **identidade única** e contínua ao longo do tempo, independentemente de seus atributos. Mesmo que os valores de seus atributos mudem, a entidade permanece a mesma se sua identidade for a mesma.

- **Identidade:** Geralmente representada por um ID (como um UUID).
- **Ciclo de Vida:** Entidades são criadas, modificadas e, eventualmente, removidas.

- **Exemplo:** No nosso Catálogo de Filmes, `Category` e `Movie` são entidades, pois cada uma possui um `id` único que as distingue.

Objetos de Valor (Value Objects)

Ao contrário das Entidades, os **Objetos de Valor** são objetos caracterizados apenas por seus atributos. Eles não possuem uma identidade conceitual própria e são **imutáveis**. Dois objetos de valor são considerados iguais se todos os seus atributos forem iguais.

- **Imutabilidade:** Uma vez criados, seus atributos não podem ser alterados. Qualquer modificação resulta em uma nova instância.
- **Comparação por Valor:** A igualdade é baseada nos valores de seus atributos.
- **Exemplo:** Um endereço (`Rua` , `Número` , `Cidade`) ou, no nosso domínio, uma `Classificacao` (e.g., "Livre", "18 anos") poderia ser um Objeto de Valor.

Agregados (Aggregates)

Um **Agregado** é um cluster de objetos de domínio (Entidades e Objetos de Valor) que são tratados como uma única unidade transacional. Ele possui uma **Raiz do Agregado** (Aggregate Root), que é uma Entidade que garante a consistência do agregado como um todo. Todas as operações externas ao agregado devem passar pela Raiz do Agregado.

- **Consistência Transacional:** Garante que as regras de negócio (invariantes) que abrangem múltiplos objetos dentro do agregado sejam sempre mantidas.
- **Exemplo:** Um `Filme` poderia ser a raiz de um agregado que inclui uma lista de `Atores` (Entidades) e `Premios` (Objetos de Valor). Qualquer alteração nos atores ou prêmios de um filme seria feita através da entidade `Filme` .

Serviços de Domínio (Domain Services)

Serviços de Domínio são usados para lógica de negócio que não se encaixa naturalmente em uma Entidade ou Objeto de Valor. São operações que envolvem múltiplos objetos de domínio ou que orquestram uma ação complexa.

- **Stateless:** Não mantêm estado.
- **Foco na Operação:** Descrevem uma ação ou comportamento do domínio.
- **Exemplo:** Uma operação para "Sugerir filmes similares" poderia ser um Serviço de Domínio, pois precisaria analisar vários filmes e categorias.

Repositórios (Repositories)

O padrão **Repositório** atua como uma camada de abstração entre o domínio e a persistência de dados. Ele fornece uma interface de coleção para acessar os Agregados,

escondendo os detalhes de como eles são armazenados e recuperados.

- **Abstração da Persistência:** Permite que o domínio interaja com os dados de forma agnóstica à tecnologia (SQL, NoSQL, arquivos).
- **Foco em Agregados:** Geralmente, existe um repositório por tipo de Agregado.

Fábricas (Factories)

Fábricas são responsáveis por encapsular a lógica de criação de objetos complexos, como Entidades e Agregados. Elas garantem que o objeto seja criado em um estado válido, satisfazendo todas as suas invariantes.

- **Criação Consistente:** Centraliza a lógica de construção de objetos, especialmente quando a criação é complexa.
- **Exemplo:** Uma `MovieFactory` poderia ser usada para criar um objeto `Movie` a partir de dados brutos, garantindo que todas as validações e associações necessárias sejam feitas corretamente.

4. Arquitetura e Camadas no DDD

Separação de Responsabilidades (SOLID)

O DDD promove fortemente a **separação de responsabilidades**, alinhada com o Princípio da Responsabilidade Única (SRP) do SOLID. Cada componente do sistema deve ter uma única razão para mudar.

Camada de Domínio, Aplicação e Infraestrutura

Uma arquitetura típica baseada em DDD possui as seguintes camadas:

- **Camada de Domínio (Domain Layer):** Contém as entidades, objetos de valor, agregados e serviços de domínio. É o coração do sistema e não deve ter dependências de infraestrutura.
- **Camada de Aplicação (Application Layer):** Orquestra as operações do domínio, traduzindo requisições externas (casos de uso) em comandos de domínio e vice-versa. Ela não contém regras de negócio, apenas coordena a execução.
- **Camada de Infraestrutura (Infrastructure Layer):** Lida com a persistência de dados (implementação dos repositórios), comunicação externa (APIs, mensageria) e outros detalhes técnicos.
- **Camada de Apresentação (Presentation Layer):** Responsável pela interface com o usuário (UI) ou por expor uma API (como uma API REST).

Adaptadores (HTTP)

Os **Adaptadores** (parte da camada de apresentação/infraestrutura) traduzem as interações externas para o formato que a camada de aplicação entende. No contexto de uma API web, o servidor HTTP atua como um adaptador. Ele recebe requisições HTTP, as passa para a camada de aplicação e traduz as respostas de volta para o formato HTTP. É crucial que o adaptador **não contenha regras de negócio**.

5. Exemplo Prático: Catálogo de Filmes

Para ilustrar os conceitos do DDD, utilizaremos um exemplo de um sistema de Catálogo de Filmes. Este sistema é intencionalmente construído sem o uso de frameworks web robustos (como Flask ou FastAPI) para demonstrar como os princípios do DDD podem ser aplicados usando apenas a biblioteca padrão do Python, focando na separação de responsabilidades e na centralidade do domínio.

Visão Geral e Estrutura do Projeto

O projeto implementa um backend simples para gerenciar categorias e filmes. Ele expõe uma API REST básica para operações CRUD (Create, Read, Update, Delete) e demonstra como as regras de negócio são encapsuladas nas entidades de domínio, enquanto a persistência e a interface HTTP são tratadas em camadas separadas.

A estrutura de diretórios reflete a separação de responsabilidades proposta pelo DDD:

Plain Text

```
movies_backend_no_framework/
├── app/
│   └── server.py        # Adaptador HTTP com http.server + roteamento por
regex
├── domain/
│   ├── category.py      # Entidade Category (regras, validações,
comportamentos)
│   └── movie.py         # Entidade Movie (título, ano, categoria)
├── infra/
│   └── json_repo.py     # Repositório genérico persistindo em arquivos JSON
├── data/                # Diretório para arquivos JSON de persistência
(criado automaticamente)
└── README.md            # Documentação e exemplos de uso
```

Implementação Detalhada (com código Python)

`domain/category.py` - Entidade `Category`

Esta é a entidade central para as categorias de filmes. Ela encapsula as regras de negócio relacionadas a uma categoria.

Python

```python
# domain/category.py
import uuid
from dataclasses import dataclass, field
from typing import Optional

MAX_NAME = 255

@dataclass
class Category:
    """
    Entidade Category (sem framework).

    Regras:
    - name obrigatório e <= 255 chars
    - id/description/is_active opcionais
    - is_active default True
    - gera id (uuid4) se não for informado
    - permite update(name/description) e (des)ativar
    """
    name: str
    description: str = ""
    is_active: bool = True
    id: Optional[str] = field(default=None)

    def __post_init__(self):
        # Gera id se não vier um
        if not self.id:
            self.id = str(uuid.uuid4())
        # Valida e normaliza
        self.name = self._validate_name(self.name)
        self.description = self.description or ""
        self.is_active = bool(self.is_active)

    # ---- Validação de regras de negócio ----
    @staticmethod
    def _validate_name(name: str) -> str:
        if not isinstance(name, str):
            raise TypeError("name deve ser string")
        n = name.strip()
        if not n:
            raise ValueError("name é obrigatório")
        if len(n) > MAX_NAME:
            raise ValueError(f"name deve ter no máximo {MAX_NAME}
```

```
caracteres")
        return n

    # ---- Comportamentos do domínio ----
    def update(self, *, name: Optional[str] = None, description:
Optional[str] = None) -> None:
        if name is not None:
            self.name = self._validate_name(name)
        if description is not None:
            self.description = description

    def activate(self) -> None:
        self.is_active = True

    def deactivate(self) -> None:
        self.is_active = False

    # Representações úteis para depuração e logs
    def __str__(self) -> str:
        return f"{self.name} | {self.description} ({self.is_active})"

    def __repr__(self) -> str:
        return f"<Category {self.name} ({self.id})>"
```

`domain/movie.py` - Entidade `Movie`

Esta entidade representa um filme e suas regras associadas, incluindo a referência a uma categoria.

Python

```python
# domain/movie.py
import uuid
from dataclasses import dataclass, field
from typing import Optional

MAX_TITLE = 255

@dataclass
class Movie:
    """Entidade Movie simples; relaciona com Category via category_id."""
    title: str
    year: int
    category_id: str
    description: str = ""
    is_active: bool = True
    id: Optional[str] = field(default=None)
```

```
    def __post_init__(self):
        if not self.id:
            self.id = str(uuid.uuid4())
        self.title = self._validate_title(self.title)
        self.year = self._validate_year(self.year)
        self.description = self.description or ""
        self.is_active = bool(self.is_active)

    @staticmethod
    def _validate_title(title: str) -> str:
        if not isinstance(title, str):
            raise TypeError("title deve ser string")
        t = title.strip()
        if not t:
            raise ValueError("title é obrigatório")
        if len(t) > MAX_TITLE:
            raise ValueError(f"title deve ter no máximo {MAX_TITLE}
caracteres")
        return t

    @staticmethod
    def _validate_year(year: int) -> int:
        if not isinstance(year, int):
            raise TypeError("year deve ser int")
        if year < 1880 or year > 2100:
            raise ValueError("year fora do intervalo razoável (1880..2100)")
        return year

    def update(
        self, *,
        title: Optional[str] = None,
        description: Optional[str] = None,
        year: Optional[int] = None,
        category_id: Optional[str] = None
    ):
        if title is not None:
            self.title = self._validate_title(title)
        if description is not None:
            self.description = description
        if year is not None:
            self.year = self._validate_year(year)
        if category_id is not None:
            self.category_id = category_id

    def activate(self): self.is_active = True
    def deactivate(self): self.is_active = False
```

```
    def __str__(self) -> str:
        return f"{self.title} ({self.year}) | cat={self.category_id}
({self.is_active})"

    def __repr__(self) -> str:
        return f"<Movie {self.title} ({self.id})>"
```

`infra/json_repo.py` - Repositório Genérico JSON

Este repositório é responsável por persistir e recuperar objetos de domínio em arquivos JSON. Ele demonstra o princípio da inversão de dependência, sendo agnóstico aos tipos de entidades que armazena.

Python

```python
# infra/json_repo.py
from __future__ import annotations
import json, os
from typing import Dict, List, TypeVar, Callable

T = TypeVar("T")

class JsonRepo:
    def __init__(self, *, path: str, to_dict: Callable[[T], dict],
from_dict: Callable[[dict], T]):
        self.path = path
        self.to_dict = to_dict          # como serializar o objeto
        self.from_dict = from_dict      # como reconstruir o objeto
        self._data: Dict[str, T] = {}
        self._load()

    # ---- Persistência ----
    def _load(self):
        if os.path.exists(self.path):
            with open(self.path, 'r', encoding='utf-8') as f:
                raw = json.load(f)
            self._data = {k: self.from_dict(v) for k, v in raw.items()}

    def _save(self):
        raw = {k: self.to_dict(v) for k, v in self._data.items()}
        os.makedirs(os.path.dirname(self.path), exist_ok=True)
        with open(self.path, 'w', encoding='utf-8') as f:
            json.dump(raw, f, ensure_ascii=False, indent=2)

    # ---- CRUD ----
    def add(self, obj: T):
        oid = getattr(obj, 'id')
```

```
        if oid in self._data:
            raise ValueError(f"objeto com id {oid} já existe")
        self._data[oid] = obj
        self._save()
        return obj

    def get(self, oid: str) -> T | None:
        return self._data.get(oid)

    def list(self) -> List[T]:
        return list(self._data.values())

    def update(self, obj: T):
        oid = getattr(obj, 'id')
        if oid not in self._data:
            raise KeyError(f"não encontrado: {oid}")
        self._data[oid] = obj
        self._save()
        return obj

    def delete(self, oid: str):
        if oid in self._data:
            del self._data[oid]
            self._save()
```

`app/server.py` - API HTTP usando `http.server`

Este arquivo atua como o adaptador HTTP, traduzindo requisições web para chamadas ao domínio e vice-versa. Ele usa o módulo `http.server` da biblioteca padrão do Python.

Python

```python
# app/server.py
from __future__ import annotations
from http.server import BaseHTTPRequestHandler, HTTPServer
import json, re, os

from domain.category import Category
from domain.movie import Movie
from infra.json_repo import JsonRepo

# ---- Repositórios com persistência em ./data/*.json ----
DATA_DIR = os.path.join(os.path.dirname(__file__), '..', 'data')
CAT_PATH = os.path.join(DATA_DIR, 'categories.json')
MOV_PATH = os.path.join(DATA_DIR, 'movies.json')

cat_repo = JsonRepo(
```

```
    path=CAT_PATH,
    to_dict=lambda c: c.__dict__,
    from_dict=lambda d: Category(**d),
)
mov_repo = JsonRepo(
    path=MOV_PATH,
    to_dict=lambda m: m.__dict__,
    from_dict=lambda d: Movie(**d),
)

class JsonHandler(BaseHTTPRequestHandler):
    # ---------- utilitários ----------
    def _json(self, status: int, payload: dict | list):
        body = json.dumps(payload, ensure_ascii=False).encode('utf-8')
        self.send_response(status)
        self.send_header('Content-Type', 'application/json; charset=utf-8')
        self.send_header('Content-Length', str(len(body)))
        self.end_headers()
        self.wfile.write(body)

    def _parse_body(self):
        length = int(self.headers.get('Content-Length', 0))
        if length == 0:
            return {}
        raw = self.rfile.read(length)
        try:
            return json.loads(raw.decode('utf-8'))
        except json.JSONDecodeError:
            return {}

    # ---------- GET ----------
    def do_GET(self):
        path = self.path

        # /categories e /categories/{id}
        if path == '/categories':
            return self._json(200, [c.__dict__ for c in cat_repo.list()])

        m = re.match(r'^/categories/([\w-]+)$', path)
        if m:
            cid = m.group(1)
            c = cat_repo.get(cid)
            return self._json(200, c.__dict__) if c else self._json(404,
{"error": "category not found"})

        # /movies e /movies/{id}
        if path == '/movies':
            return self._json(200, [m.__dict__ for m in mov_repo.list()])
```

```
        m = re.match(r'^/movies/([\w-]+)$', path)
        if m:
            mid = m.group(1)
            mv = mov_repo.get(mid)
            return self._json(200, mv.__dict__) if mv else self._json(404,
{"error": "movie not found"})

        return self._json(404, {"error": "not found"})

    # ---------- POST ----------
    def do_POST(self):
        path = self.path
        data = self._parse_body()

        # Criar Category
        if path == '/categories':
            try:
                # NÃO passamos id quando não vier -> Category gera sozinho
                kwargs = {
                    "name": data.get("name", ""),
                    "description": data.get("description", ""),
                    "is_active": data.get("is_active", True),
                }
                if data.get("id"):  # se quiser criar com id fixo em testes
                    kwargs["id"] = data["id"]
                c = Category(**kwargs)
                cat_repo.add(c)
                return self._json(201, c.__dict__)
            except Exception as e:
                return self._json(400, {"error": str(e)})

        # Ativar/Desativar Category
        m = re.match(r'^/categories/([\w-]+)/(activate|deactivate)$', path)
        if m:
            cid, action = m.group(1), m.group(2)
            c = cat_repo.get(cid)
            if not c:
                return self._json(404, {"error": "category not found"})
            c.activate() if action == 'activate' else c.deactivate()
            cat_repo.update(c)
            return self._json(200, c.__dict__)

        # Criar Movie
        if path == '/movies':
            try:
                cat_id = data.get('category_id')
                if not cat_id or not cat_repo.get(cat_id):
```

```
                    return self._json(400, {"error": "category_id
inválido/inexistente"})
                kwargs = {
                    "title": data.get("title", ""),
                    "year": int(data.get("year", 0)),
                    "category_id": cat_id,
                    "description": data.get("description", ""),
                    "is_active": data.get("is_active", True),
                }
                if data.get("id"):
                    kwargs["id"] = data["id"]
                mobj = Movie(**kwargs)
                mov_repo.add(mobj)
                return self._json(201, mobj.__dict__)
            except Exception as e:
                return self._json(400, {"error": str(e)})

        return self._json(404, {"error": "not found"})

    # ---------- PUT ----------
    def do_PUT(self):
        # Atualizar Category (name/description)
        m = re.match(r'^/categories/([\w-]+)$', self.path)
        if m:
            cid = m.group(1)
            c = cat_repo.get(cid)
            if not c:
                return self._json(404, {"error": "category not found"})
            data = self._parse_body()
            try:
                c.update(name=data.get('name'),
description=data.get('description'))
                cat_repo.update(c)
                return self._json(200, c.__dict__)
            except Exception as e:
                return self._json(400, {"error": str(e)})

        # Atualizar Movie
        m = re.match(r'^/movies/([\w-]+)$', self.path)
        if m:
            mid = m.group(1)
            mv = mov_repo.get(mid)
            if not mv:
                return self._json(404, {"error": "movie not found"})
            data = self._parse_body()
            try:
                cat_id = data.get('category_id')
                if cat_id and not cat_repo.get(cat_id):
```

```
                    return self._json(400, {"error": "category_id
inválido/inexistente"})
                mv.update(
                    title=data.get('title'),
                    description=data.get('description'),
                    year=data.get('year'),
                    category_id=cat_id
                )
                mov_repo.update(mv)
                return self._json(200, mv.__dict__)
            except Exception as e:
                return self._json(400, {"error": str(e)})

        return self._json(404, {"error": "not found"})

    # ---------- DELETE ----------
    def do_DELETE(self):
        # Deletar Category
        m = re.match(r'^/categories/([\w-]+)$', self.path)
        if m:
            cid = m.group(1)
            # Regra de negócio: não pode deletar categoria com filmes
associados
            if any(m.category_id == cid for m in mov_repo.list()):
                return self._json(400, {"error": "não pode deletar categoria
com filmes associados"})
            cat_repo.delete(cid)
            return self._json(204, {})

        # Deletar Movie
        m = re.match(r'^/movies/([\w-]+)$', self.path)
        if m:
            mid = m.group(1)
            mov_repo.delete(mid)
            return self._json(204, {})

        return self._json(404, {"error": "not found"})

def run(server_class=HTTPServer, handler_class=JsonHandler, port=8000):
    server_address = ('', port)
    httpd = server_class(server_address, handler_class)
    print(f'Servidor HTTP rodando em http://localhost:{port}')
    httpd.serve_forever()

if __name__ == '__main__':
    run()
```

6. Dicas Didáticas para Explicação

Ao apresentar este material, considere as seguintes dicas para tornar a explicação mais eficaz e didática:

- **Por que as regras no domínio?** Enfatize que a validação e as regras de negócio devem estar nas entidades (`__post_init__` , métodos de validação), e não nos controladores HTTP. Isso garante que as regras sejam sempre aplicadas, independentes da interface de acesso.
- **Passo a passo da `Category` :** Demonstre a criação de uma `Category` , mostre sua representação (`__str__`), ative/desative, atualize o nome e, crucialmente, tente violar as regras (nome vazio, nome muito longo) para mostrar como as validações do domínio funcionam.
- **Relação `Movie` → `Category` :** Explique que a criação de um `Movie` falha se o `category_id` não corresponder a uma `Category` existente. Isso ilustra uma regra de integridade que é aplicada na camada de aplicação, mas que depende da existência de entidades no domínio.
- **Troca de Persistência:** Ressalte que o `JsonRepo` pode ser facilmente substituído por uma implementação de repositório para SQLite (ou qualquer outro banco de dados) sem a necessidade de alterar as entidades de domínio (`Category` , `Movie`). Isso demonstra a flexibilidade e a separação de responsabilidades do DDD.
- **"Sem framework ≠ gambiarra":** Deixe claro que construir um sistema sem um framework web completo não significa criar um código desorganizado. Pelo contrário, este exemplo mostra como aplicar princípios de design (camadas, responsabilidades claras, testes possíveis) para construir um sistema robusto e bem estruturado, mesmo com ferramentas básicas.
- **Extensões:** Mencione que o projeto pode ser estendido para usar SQLite (`sqlite3`) ou incluir testes unitários (`unittest`), mantendo a mesma arquitetura baseada em DDD.

7. Referências

[1] Evans, E. (2004). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional.