Engenharia de Software Moderna

Marco Tulio Valente

Domain-Driven Design (DDD): Um Resumo

Introdução

Domain-Driven Design (DDD) é um conjunto de princípios para projeto de software, organizados e sistematizados em 2003, por Eric Evans, em um livro com o mesmo nome.

Os princípios defendidos por DDD têm, no seu conjunto, um objetivo central: permitir o desenvolvimento de sistemas cujo design é centrado em conceitos próximos e alinhados com um domínio de negócio.

O **domínio** de um sistema consiste da área e problema de negócio que ele pretende resolver. Neste artigo, para explicar DDD, vamos usar como exemplo um sistema para gerenciar uma biblioteca. Logo, esse problema – gerenciamento de bibliotecas – constitui o domínio do nosso sistema de exemplo.

DDD defende que os **desenvolvedores** devem ter um profundo conhecimento do domínio do sistema que eles desenvolvem. Esse conhecimento deve ser obtido por meio de conversas e discussões frequentes com **especialistas no domínio** (ou no negócio). Portanto, o design do sistema deve ser norteado para atender ao seu domínio. E não, por exemplo, para se moldar a uma determinada tecnologia de programação. Em suma, o design é dirigido pelo domínio, e não por frameworks, arquiteturas, linguagens de programação, etc.

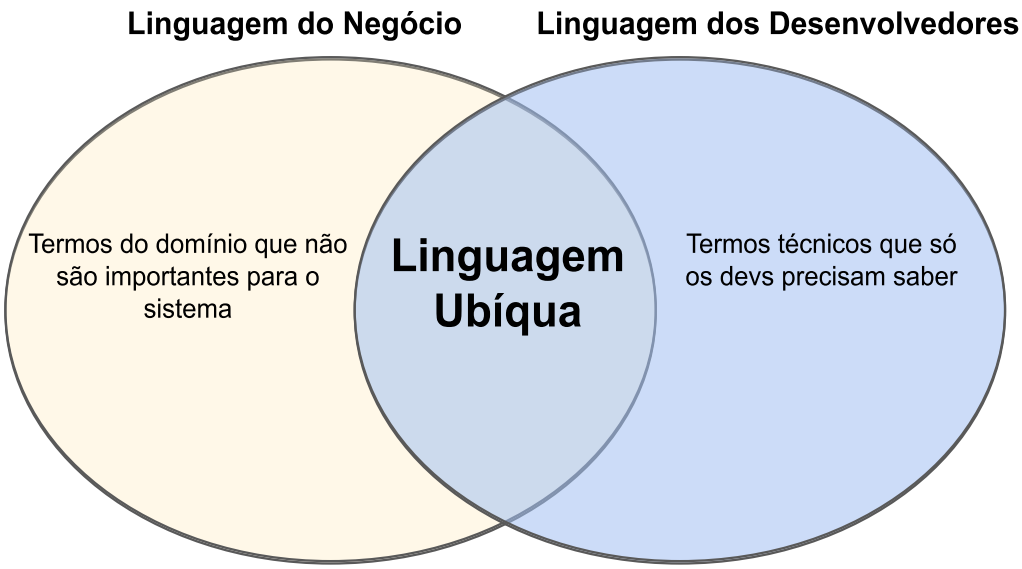
DDD defende que a separação entre domínio e tecnologia deve ser promovida e expressa na arquitetura do sistema. Para tanto, padrões como Arquitetura em Camadas (estudado no Capítulo 7), Arquitetura Limpa (tratada neste outro artigo didático) ou Arquitetura Hexagonal (também coberta em um artigo separado) podem ser usados.

Antes de avançarmos, é importante mencionar também que DDD se sobressai quando é usado em sistemas para domínios complexos, cujas regras de negócio são mais difíceis de serem imediatamente entendidas e implementadas pelos desenvolvedores.

Linguagem Ubíqua

Linguagem Ubíqua (ou **Linguagem Onipresente**) é um conceito central de DDD. Ela consiste de um conjunto de termos que devem ser plenamente entendidos tanto por especialistas no domínio (usuários do sistema) como por desenvolvedores (implementadores do sistema).

Para um projeto de software dar certo, DDD defende que esses dois papéis – especialistas no domínio e desenvolvedores – devem falar a mesma língua, que vai constituir a chamada Linguagem Ubíqua do sistema. Essa ideia é ilustrada na seguinte figura:



A figura deixa claro que existem termos que só os especialistas de domínio conhecem. Já outros termos, de cunho tecnológico, são do conhecimento apenas dos desenvolvedores. Porém, existe um conjunto de termos que devem ser do conhecimento de ambos, os quais formam a Linguagem Ubíqua do sistema.

Os termos da Linguagem Ubíqua são usados com dois propósitos:

- Para possibilitar uma comunicação fluida entre desenvolvedores e especialistas no domínio.
- Para nomear entidades do código do sistema, como classes, métodos, atributos, pacotes, módulos, tabelas de bancos de dados, rotas de APIs, etc.

Além de clarificar o significado dos termos da linguagem ubíqua, é importante que se definam os **relacionamentos** e **associações** que existem entre eles.

Exemplo: no nosso sistema de bibliotecas, a Linguagem Ubíqua inclui termos como os seguintes:

> Livro, Exemplar, ISBN, Bibliotecária, Usuário, Acervo, Reserva, Empréstimo, Multa, Catálogo

Por outro lado, alguns termos são de domínio apenas dos desenvolvedores, tais como proxy, observadores, cache, camadas, rotas, dentre outros. Existem ainda termos que são do conhecimento apenas de bibliotecárias, como certos formatos para definição de ISBNs, que não são usados no Brasil.

Devemos definir também os relacionamentos e associações entre esses termos, como exemplificado a seguir:

- Um `Livro` pode ter um ou mais `Exemplares`.
- Uma `Reserva` pode ser feita para no máximo três `Livros`.
- Existem três tipos de `Usuário`: `Aluno`, `Professor` e `UsuárioExterno`.
- O `Acervo` da biblioteca é formado por um conjunto de `Livros`.

Para documentar de forma visual esses relacionamentos pode ser usado um **Diagrama de Classes** de UML, conforme estudamos no Capítulo 4.

Objetos de Domínio

DDD foi proposto pensando em sistemas implementados em linguagens orientadas a objetos. Então, quando se define o design desses sistemas, alguns tipos importantes de objetos se destacam. Dentre eles, DDD lista os seguintes:

- Entidades
- Objetos de Valor
- Serviços
- Agregados
- Repositórios

Esses tipos de objetos de domínio devem ser entendidos como as ferramentas conceituais que um projetista deve lançar mão para projetar com sucesso um determinado sistema. Por isso, eles são chamados também dos **building blocks** de DDD. Iremos comentar sobre cada um deles a seguir.

Entidades e Objetos de Valor

Uma **entidade** é um objeto que possui uma identidade única, que o distingue dos demais objetos da mesma classe. Por exemplo, cada `Usuário` da nossa biblioteca é uma entidade, cujo identificador é o seu número de matrícula na universidade.

Por outro lado, **objetos de valor** (*value objects*) não possuem um identificador único. Assim, eles são caracterizados apenas por seu estado, isto é, pelos valores de seus atributos. Por exemplo, o `Endereço` de um `Usuário` da biblioteca é um objeto de valor. Veja que se dois `Endereços` tiverem exatamente os mesmos valores para `rua`, `número`, `cidade`, `CEP`, etc, eles serão idênticos.

Outros exemplos de objetos de valor incluem: `Moeda`, `Data`, `Fone`, `Email`, `Hora`, `Cor`, etc.

Por que distinguir entre entidades e objetos de valor? Entidades são objetos mais importantes e devemos, por exemplo, projetar com cuidado como eles serão persistidos e depois recuperados de um banco de dados. Devemos também tomar cuidado com o ciclo de vida de entidades. Por exemplo, podem existir regras que governam a criação e remoção de entidades. No caso da nossa bibliotecas, não se pode remover um `Usuário` se ele tiver um `Empréstimo` pendente.

Já objetos de valor são mais simples. E também eles devem ser **imutáveis**, ou seja, uma vez criados, não deve ser possível alterar seus valores internos. Por exemplo, para alterar o `Endereço` de um `Usuário` devemos abandonar o objeto antigo e criar um objeto com o `Endereço` novo. Os benefícios de objetos imutáveis já foram discutidos no Capítulo 9.

É interessante mencionar também que, recentemente, algumas linguagens de programação passaram a oferecer suporte sintático para implementação de objetos de valor. Por exemplo, nas versões mais novas de Java, eles podem ser implementados por meio de records.

Serviços

Existem operações importantes do domínio que não se encaixam em entidades e objetos de valor. Assim, o ideal é criar objetos específicos para implementar essas operações. No jargão de DDD, esses objetos são chamados de **serviços**. Em alguns sistemas, é comum ver esses objetos sendo chamados também de gerenciadores ou controladores.

A assinatura das operações de um objeto de serviço pode incluir entidades e objetos de valor. No entanto, objetos de serviço não devem possuir estado, isto é, eles devem ser **stateless**. Por isso, eles não costumam ter atributos, mas apenas métodos.

Serviços normalmente são implementados como **singletons**, ou seja, possuem uma única instância durante a execução do sistema. Mais detalhes sobre esse padrão de projeto no Capítulo 6.

Exemplo: no nosso sistema de bibliotecas, podemos ter um serviço que implementa as seguintes operações:

```
class ServicoDeEmprestimo {
  ...
  void emprestarLivro(Usuario, Livro) {...}
  void devolverLivro(Usuario, Livro)  {...}
  ...
}
```

Na primeira operação, realiza-se o empréstimo de um `Livro` para um certo `Usuário`. Na segunda operação, um `Usuário` devolve um `Livro` que ele tenha sob empréstimo.

Ambas as operações não são específicas nem de `Usuário`, nem de `Livro`. Logo, a recomendação de criar um objeto de serviço para acomodá-las.

Agregados

Agregados (*aggregates*) são coleções de entidades e objetos de valor. Ou seja, algumas vezes não faz sentido raciocinar sobre entidades e objetos de valor de forma individual. Em vez disso, temos que pensar em grupos de objetos para ter uma visão consistente com o domínio que estamos modelando.

Um agregado possui um objeto raiz, que deve ser uma entidade. Externamente, o agregado é acessado a partir dessa raiz. A raiz, por sua vez, referencia os objetos internos do agregado. Porém, esses objetos internos não devem ser visíveis para o resto do sistema, ou seja, apenas a raiz pode referenciá-los.

Como formam uma unidade coerente, agregados são persistidos em conjunto em bancos de dados. A deleção de um agregado, da memória principal ou de um banco de dados, implica na deleção da sua raiz e de todos os objetos internos.

Como eles são objetos mais complexos e com objetos internos, pode ser interessante implementar métodos especificamente para criação de agregados, os quais são chamados de **fábricas**. Ou seja, tais métodos são implementações do padrão de projeto de mesmo nome.

Exemplo: No sistema de bibliotecas, um `Empréstimo` possui um `Usuário` (que é uma entidade), uma data de realização (que é um objeto de valor) e uma lista de `Itens Emprestados`. Cada `Item Emprestado` contém informações sobre um certo `Livro` que foi emprestado e sua data de devolução (estamos pressupondo que alguns livros devem ser devolvidos mais rapidamente do que outros, por exemplo).

Logo, `Empréstimo` e `Itens de Empréstimo` formam um agregado. Isto é, uma entidade única do ponto de vista conceitual. `Empréstimo` é a raiz do agregado e `Itens Emprestados` são os seus objetos internos, que não podem ser manipulados sem passar antes pela raiz.

Observe que `Itens Emprestados` referenciam `Livros`, porém esses últimos não fazem parte do agregado, pois eles têm vida própria, isto é, eles existem independentemente de estarem emprestados ou não.

Repositórios

Para implementar certos serviços do domínio precisamos antes obter referências para determinados objetos.

Por exemplo, suponha um serviço que lista os `Empréstimos` realizados por um `Usuário`. Para implementá-lo, não podemos assumir que todos os agregados do tipo `Empréstimo` estão na memória principal. Na verdade, em qualquer sistema real, eles estão armazenados em um banco de dados.

Um **repositório** é então um objeto usado para recuperar outros objetos de domínio de um banco de dados. Seu objetivo é prover uma abstração que blinde os desenvolvedores de preocupações relacionadas com acesso a bancos de dados. Normalmente, repositórios são criados para recuperar entidades ou agregados.

Em outras palavras, um repositório oferece uma abstração para o banco de dados usado pelo sistema e, assim, permite que os desenvolvedores continuem focados no domínio, em vez de ter sua atenção desviada, em certos momentos, para uma tecnologia de armazenamento de dados. Em termos mais concretos, um repositório permite manipular objetos de domínio como se eles fossem listas (ou coleções) armazenadas na memória principal. A implementação interna do repositório cuida então de ler e salvar essas listas no banco de dados.

Exemplo: No sistema de bibliotecas, existe um repositório com métodos para recuperar `Empréstimos` salvos em um banco de dados:

```
class RepositorioDeEmprestimos {
  List<Emprestimo> findEmprestimosDeUsuario(Usuario u) {...}
  List<Emprestimo> findEmprestimosPorData(Data inicio, Data fim) {...}
  List<Emprestimo> findEmprestimosVencidos() {...}
  ...
}
```

Além dos métodos `find*`, um repositório pode implementar métodos para salvar, atualizar e remover objetos:

```
class RepositorioDeEmprestimos {

  // métodos find* (veja acima)

  void salvar(Emprestimo e) {...}
  void atualizar(Emprestimo e) {...}
  void remover(Emprestimo e) {...}
}
```

Contextos Delimitados

Com o tempo, sistemas de software ficam mais complexos e abrangentes. Por isso, é irrealista imaginar que sistemas de organizações grandes e complexas vão possuir um modelo de domínio único e baseado na mesma linguagem ubíqua.

Em vez disso, é natural que tais organizações tenham sistemas que atendem a usuários com perfis e necessidades diferentes, o que complica a definição de uma linguagem ubíqua. A solução para esse problema consiste em quebrar tais domínios complexos em domínios menores, os quais em DDD são chamados de **Contextos Delimitados** (*Bounded Contexts*).

Exemplo: Suponha que a nossa biblioteca tenha um setor financeiro. Esse setor tem necessidades específicas, que começam a justificar um projeto separado, com uma linguagem própria. Por exemplo, nesse domínio financeiro, a classe `Usuário` pode, inclusive, ser chamada de `Cliente` e ter novos atributos.

Camada Anticorrupção

Às vezes, temos que integrar sistemas que estão em contextos delimitados diferentes. Por exemplo, um sistema A precisa usar serviços de um sistema B, que pode inclusive ser um sistema externo, isto é, de uma outra organização. Para evitar que A tenha que se adaptar e usar, mesmo que parcialmente, a linguagem ubíqua de B, pode-se usar uma **Camada Anticorrupção** para mediar essa comunicação.

Essa camada é formada por três tipos principais de classes:

- Classes de Serviço, cujos métodos serão chamados por A e que, portanto, seguem a linguagem ubíqua desse sistema.
- Classes Adaptadoras, que convertem o modelo e os tipos de dados de B para o modelo e tipos de dados de A. Ou seja, essas classes vão isolar elementos próprios de B e evitar que eles cheguem até o sistema A.
- Uma Classe de Fachada, usada para acessar o sistema B. O papel dessa classe é facilitar o uso de B, principalmente quando ele é um sistema legado com uma interface complexa e antiga.

Logo, o fluxo costuma ser o seguinte:

```
Sistema A -> [ Serviços -> Adaptadores -> Fachada ] -> Sistema B
```

Nesse fluxo, as classes entre colchetes constituem a Camada Anticorrupção que foi construída para integrar os sistemas A e B.

Comentários Finais

Em um material de referência, que escreveu em 2014, Eric Evans define assim DDD:

> DDD é uma abordagem para desenvolvimento de sistemas de software complexos, em que: (1) o foco está no domínio do sistema; (2) desenvolvedores e especialistas no negócio devem explorar esse domínio de forma colaborativa; (3) como resultado, eles devem se comunicar usando uma linguagem ubíqua, mas dentro de um contexto delimitado.

A linguagem ubíqua do sistema deve ser usada também no seu código, para nomear variáveis, parâmetros, métodos, classes, pacotes, etc. Especificamente, um projeto DDD deve fazer uso dos seguintes tipos de objetos principais: entidades, objetos de valor, serviços, agregados e repositórios.

Referência

VALENTE, M.T. Domain-Driven Design (DDD): Um Resumo. Engenharia de Software Moderna. 2022. Disponível em <https://engsoftmoderna.info/artigos/ddd.html>