**DESENVOLVIMENTO DE SOFTWARE EM CAMADAS**

## Resumo

Este material procura descrever os principais conceitos envolvidos com a técnica de desenvolvimento de software em camadas, metodologia largamente aceita e utilizada no desenvolvimento de softwares corporativos.

## Conceitos

Existem diferentes tipos de software para diferentes tipos de aplicações, cada uma com suas características e complexidades.

*Aplicações que manipulam uma grande quantidade de dados que devem ser persistidos, às vezes por muitos anos, que são acessadas por uma grande quantidade de pessoas concorrentemente, que possuem uma quantidade significante de código e que se destina a mostrar esses dados de diferentes maneiras e com diferentes propósitos, geralmente precisam se integrar com outros sistemas, construídos em diferentes tempos com diferentes tecnologias e trabalham com as regras de negócio da instituição para a qual estão sendo desenvolvidos, que geralmente estão em constante mudança.*

*(Martin Fowler, 2002)*

Por causa das características citadas acima, é visível que aplicações corporativas têm um alto grau de complexidade e precisam estar preparadas para sofrerem mudanças constantemente, por isso, muitos padrões foram criados para facilitar as mudanças nas estruturas e regras sobre as quais o software foi construído.

## Camadas - principal padrão utilizado para aplicações corporativas

A maioria dos padrões utilizados no desenvolvimento de aplicações corporativas está relacionada com a separação do código em camadas, isto é, eles procuram definir como essas camadas estão organizadas e como é feita a comunicação entre elas.

### Coesão e Acoplamento

Atualmente existem duas métricas básicas de qualidade que devem ser seguidas por um módulo de software (método, classe, *namespace* ou qualquer divisão que se faz em um software dependendo do tamanho do mesmo): coesão e acoplamento.

Módulos de software devem ter uma alta coesão, isto é, todos os componentes desse módulo **deveriam** ter as mesmas responsabilidades e, idealmente, cada módulo deveria está focado em resolver um problema específico. Além disso, devem ter um baixo acoplamento, ou seja, todo módulo deveria ser o mais independente possível de outro e uma alteração da implementação de um módulo não deveria afetar nenhum outro.

Um alto grau de acoplamento entre muitos objetos é sinal de um design pobre porque deixa o código difícil de ler e entender e, mais importante, torna o código pouco flexível a mudanças.

*“O uso de Camadas é um padrão arquitetural que ajuda na tarefa de separar responsabilidades, promovendo baixo acoplamento e alta coesão em um sistema”.*

*(Philipe Calçado, 2005)*

Idealmente, Camadas se comunicam apenas com Camadas adjacentes.

### Arquitetura em duas camadas

A necessidade de se separar uma aplicação em camadas surgiu no começo da década de 90 com os sistemas com arquitetura cliente-servidor. Existem geralmente duas camadas nesses sistemas: a interface gráfica com o usuário e a camada de persistência (geralmente um Banco de Dados relacional).

Há duas possibilidades de implementação para esses sistemas: “cliente rico” ou “cliente pobre” (conhecidas também como: “cliente gordo” ou “cliente magro”).

#### Cliente rico

No cliente rico a lógica da aplicação está embutida junto com o código da interface gráfica com o usuário. Na medida em que os sistemas vão crescendo e se tornando mais complexos, essa técnica gera um grande número de problemas, como uma grande quantidade de código repetido. Uma pequena mudança pode ter um impacto enorme na aplicação (e isso é o que exatamente se quer evitar em sistemas corporativos).

#### Cliente pobre

A outra abordagem seria colocar a lógica da aplicação em *Stored Procedures* no banco de dados relacional. Mas essa solução também tem suas desvantagens como, por exemplo, a perda de portabilidade, a dificuldade de trabalhar com transações quando está envolvido mais de um SGDB e não se poder trabalhar com orientação a objetos e suas vantagens.

### Arquitetura em três camadas

Diante dos problemas encontrados na Arquitetura 2 Camadas, surgiu a necessidade de se evoluir para a chamada programação em três camadas, na qual se permanece com a camada de interface gráfica com o usuário (também conhecida como camada de apresentação) e se manteve também a camada de persistência, porém foi **criada uma nova camada** entre as duas anteriores, que modela o domínio da aplicação e onde ficam as regras de negócio da mesma; chamada por muitos autores de camada de negócio.

Com a introdução dessa camada, resolveu-se grande parte dos problemas citados anteriormente. As camadas de persistência e a de negócio nunca deveriam ser dependentes da camada de apresentação, pois essa dependência geraria muita duplicação de código, visto que os dados de uma aplicação podem ser mostrados em diferentes partes e de diferentes maneiras e, consequentemente, acarretaria muito trabalho para se alterar qualquer parte desse sistema.

A separação dessas três camadas não precisa ser fixa. Para sistemas simples podem ser três métodos diferentes, para sistemas mais complexos pode-se separar as três camadas em três classes diferentes. Com o aumento da complexidade, faz-se necessária a divisão das três camadas em três *assemblies* diferentes. A forma de realizar a separação não é única e depende de cada caso, mas o importante é que se separe o sistema em três camadas.

## Vantagens e desvantagens

Abaixo estão listadas algumas vantagens e desvantagens em desenvolver sistemas utilizando uma arquitetura em camadas:

### Vantagens

* Reduzem complexidade: agrupam componentes e simplificam a comunicação entre eles;
* Reduzem dependência/acoplamento: a regra de comunicação evita dependências diretas entre componentes de Camadas diferentes;
* Favorecem a coesão: componentes de responsabilidades relacionadas são agrupados;
* Promovem a reusabilidade: camadas podem ser reutilizadas em outros sistemas ou podem ser substituídas;
* É um padrão arquitetural conhecido: facilita a comunicação e entendimento entre desenvolvedores.

### Desvantagens

* Apenas complicam um sistema muito simples: não é qualquer sistema que exige o uso de Camadas;
* Possibilidade de overdose: muitos arquitetos acabam criando Camadas demais e tornando a aplicação extremamente complexa.

## DDD – Introdução a Domain Driven Design

Domain Driven Design significa Projeto Orientado a Domínio. Ele veio do título do livro escrito por Eric Evans, dono da Domain Language, uma empresa especializada em treinamento e consultoria para desenvolvimento de software. O livro de Evans é um grande catálogo de Padrões, baseados em experiências do autor ao longo de mais de 20 anos desenvolvendo software utilizando técnicas de Orientação a Objetos. O que seria um Padrão?

Um padrão é uma regra de três partes que expressa a relação entre um contexto (1), um problema (2) e uma solução (3).

DDD pode ser visto por alguns como a volta da orientação a objetos. É verdade que o livro é um chamado às boas práticas de programação que já existem desde a época remota do SmallTalk. Quando se fala em Orientação a Objetos pensa-se logo em classes, heranças, polimorfismo, encapsulamento. Mas a essência da Orientação a Objetos também tem coisas como:

### Alinhamento do código com o negócio:

O contato dos desenvolvedores com os especialistas do domínio é algo essencial quando se faz DDD (o pessoal de métodos ágeis já sabe disso faz tempo);

### Favorecer reutilização

Os blocos de construção, que veremos adiante, facilitam aproveitar um mesmo conceito de domínio ou um mesmo código em vários lugares;

### Mínimo de acoplamento

Com um modelo bem feito, organizado, as várias partes de um sistema interagem sem que haja muita dependência entre módulos ou classes de objetos de conceitos distintos;

### Independência da Tecnologia

DDD não foca em tecnologia, mas sim em entender as regras de negócio e como elas devem estar refletidas no código e no modelo de domínio. Não que a tecnologia usada não seja importante, mas essa não é uma preocupação de DDD.

Todas essas coisas são bem exemplificadas e mostradas na forma de vários padrões em DDD. Mas o livro também mostra muitos padrões que não dizem respeito a código ou modelagem. Aparecem coisas que estão mais ligadas a processos (como Integração Contínua) ou a formas de relacionamento entre times que fazem parte do desenvolvimento de um sistema complexo.

## Colocando o modelo de domínio para funcionar

Para ter um software que atenda perfeitamente a um determinado domínio, é necessário que se estabeleça, em primeiro lugar, uma Linguagem Ubíqua (devman: Linguagem comum, com termos bem definidos, que fazem parte do domínio do negócio e que são usados por todas as pessoas que fazem parte do processo de desenvolvimento de software). Nessa linguagem estão termos que fazem parte das conversas diárias entre especialistas de negócio e times de desenvolvimento. Todos devem usar os mesmos termos tanto na linguagem falada quanto no código. Isso significa que, se durante uma conversa com um cliente e ele deseje uma aplicação de controle de tarefas para colocar em seu Smartphone e por exemplo, ele disser: “Quando os **subitens** de uma **tarefa** estiverem com **percentual** igual a 100%, significa que a tarefa **estará concluída**”, vamos ter no nosso código alguma coisa do tipo:

* Uma classe para a entidade Tarefa;
* Uma classe para a entidade Subitem;
* Alguma classe que tenha um método Atualizar Percentual;
* Algum atributo com o nome de Esta Concluído

Essa linguagem ubíqua deve ser compreendida por todos e não pode haver ambiguidades. Toda vez que alguém perceber que um determinado conceito do domínio possui várias palavras que o represente, essa pessoa deve tentar readequar tanto a linguagem falada e escrita, quanto o código.

## Arquitetura defendida pelo DDD

Uma vez que decidimos criar um modelo usando DDD, precisamos, inicialmente, isolar o modelo de domínio das demais partes que compõem o sistema. Essa separação pode ser feita utilizando-se uma arquitetura em camadas (Figura 1), que dividirá nossa aplicação em quatro partes:

### Interface de Usuário

Parte responsável pela exibição de informações do sistema ao usuário e também por interpretar comandos do usuário;

### Aplicação

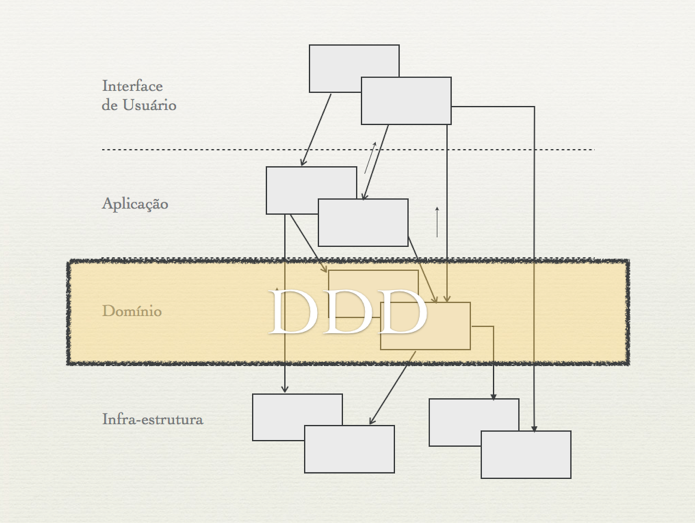
Essa camada não possui lógica de negócio. Ela é apenas uma camada fina, responsável por conectar a Interface de Usuário às camadas inferiores;

### Domínio

Representa os conceitos, regras e lógicas de negócio. Todo o foco de DDD está nessa camada. Nosso trabalho, daqui para frente, será aperfeiçoar e compreender profundamente essa parte;

### Infra-estrutura

Fornece recursos técnicos que darão suporte às camadas superiores. São normalmente as partes de um sistema responsáveis por persistência de dados, conexões com bancos de dados, envio de mensagens por redes, gravação e leitura de discos, etc.



**Figura 1.** Arquitetura em camadas, utilizada para separar o domínio do resto da aplicação.

Depois de dividirmos o sistema em camadas, nos preocuparemos apenas com a camada de domínio. Para modelar essa parte, utilizamos alguns Padrões propostos em DDD. Esses padrões são chamados de blocos de construção e serão utilizados para representar nosso modelo abstrato. Esses blocos podem ser:

### Entidades

Classes de objetos que necessitam de uma identidade. Normalmente são elementos do domínio que possuem ciclo de vida dentro de nossa aplicação: um Cliente, por exemplo, se cadastra no sistema, faz compras, se torna inativo, é excluído, etc.;

### Objetos de Valores

Objetos que só carregam valores, mas que não possuem distinção de identidade. Bons exemplos de objetos de valores seriam: strings, números ou cores. Por exemplo: se o lápis de cor da criança acabar e você der um novo lápis a ela, da mesma cor, só que de outra caixa, ela não vai se importar. Para a criança, o lápis vermelho de uma caixa é igual ao lápis vermelho de outra caixa. As instâncias de Objetos de Valores são imutáveis, isto é, uma vez criados, seus atributos internos não poderão mais ser modificados. Em Java, temos, por exemplo, a classe BigDecimal, muito utilizada para fazer cálculos com valores grandes. Na Listagem 1 observamos que, para multiplicar dois valores representados pela classe BigDecimal, não alteramos os objetos com os valores dos fatores da multiplicação. Para calcular 5 milhões vezes 30 milhões construímos cada um dos fatores e então obtemos o resultado, que será armazenado numa terceira variável. Após o cálculo, cada um dos fatores continuará armazenando o valor original. A saída do código será:

### Agregados

Compostos de Entidades ou Objetos de Valores que são encapsulados numa única classe. O Agregado serve para manter a integridade do modelo. Elegemos uma classe para servir de raiz do Agregado. Quando algum cliente quiser manipular dados de uma das classes que compõem o Agregado, essa manipulação só poderá ser feita através da raiz;

### Fábricas

Classes responsáveis pelo processo de criação dos Agregados ou dos Objetos de Valores. Algumas vezes, Agregados são relativamente complexos e não queremos manter a lógica de criação desses Agregados nas classes que o compõem. Extraímos então as regras de criação para uma classe externa: a fábrica;

### Serviços

Classes que contém lógica de negócio, mas que não pertence a nenhuma Entidade ou Objetos de Valores. É importante ressaltar que Serviços não guardam estado, ou seja, toda chamada a um mesmo serviço, dada uma mesma pré-condição, deve retornar sempre o mesmo resultado;

### Repositórios

Classes responsáveis por administrar o ciclo de vida dos outros objetos, normalmente Entidades, Objetos de Valor e Agregados. Os repositórios são classes que centralizam operações de criação, alteração e remoção de objetos. Em linguagens como Java e .NET, repositórios são comumente implementados usando-se frameworks como Hibernate ou Nhibernate. Já em RubyOnRails, o ActiveRecord faz o papel de repositório;

### Módulos

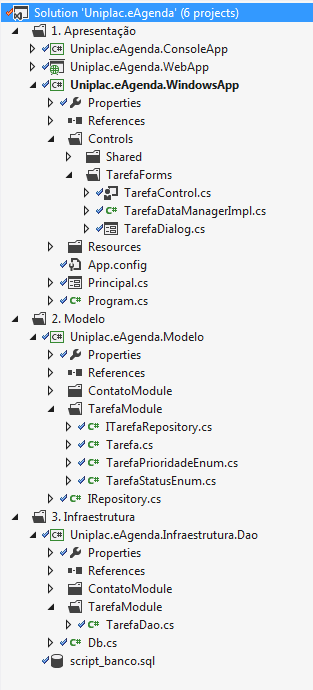
Abstrações que têm por objetivos agrupar classes por um determinado conceito do domínio. A maioria das linguagens de programação oferecem suporte a módulos (pacotes em Java, namespaces em .NET ou módulos em Ruby). Um anti-padrão comum é a criação de módulos que agrupam as classes segundo conceitos de infra-estrutura. Um exemplo seria, ao se trabalhar com Struts, em Java, criar um pacote que conterá todas as Actions do sistema. Ao usar DDD devemos agrupar classes se esse agrupamento faz sentido do ponto de vista do domínio, ou seja, do negócio. Se tivermos, por exemplo, várias classes que compõem informações de Paciente num sistema médico, podemos criar um módulo chamado paciente e colocar classes como Ficha, PrescricaoMedica, RegistroDeConsulta e HistoricoDeCirurgias num mesmo pacote.

## Exemplo do uso

Nesta seção é apresentado um exemplo, para efeito de esclarecimento, da implicação do uso do padrão de desenvolvimento em camadas. É apresentada uma aplicação bem simples que foi projetada em uma arquitetura multicamadas.

### Estrutura do projeto

Na Figura abaixo, é mostrada a estrutura do projeto desenhada para ter quatro camadas: Apresentação, Aplicação, Modelo e Persistência.

Explicando a figura ao lado, podemos ver no diretório “1. Apresentação” os projetos **Uniplac.eAgenda.WindowsApp, Uniplac.eAgenda.ConsoleApp e Uniplac.eAgenda.WebApp** que representam a camada de Apresentação do software, esta camada é responsável por construir a interface gráfica com o usuário e tratar seus eventos.

No segundo diretório mostra-se a camada de Aplicação, esta é a responsável por coordenar a sequência das tarefas do aplicativo.

A camada de Modelo, que possui uma modelagem do domínio do aplicativo está representada nesse exemplo pelo objeto Tarefa, nesse caso este objeto não possui Regras de Negócio, apenas os dados do aplicativo.

A camada de infraestrutura/persistência foi implementada usando-se o padrão DAO, que é um *design pattern* para acessar dados em um Banco de Dados (veremos mais adiante).