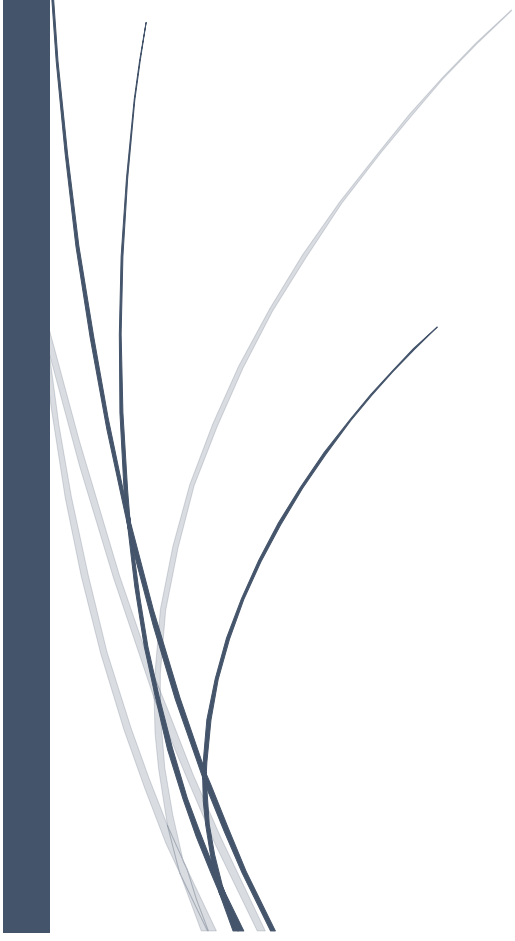
Engenharia de Software

26/9/2025

AT

Design Patterns eDomain-Driven Design(DDD) com Java

Professor(a): Armênio Torres Santiago Cardoso



Samuel Hermany
INSTITUTO INFNET

LINK GITHUB

1.1. QUAL É O PAPEL DOS DESIGN PATTERNS NA SOLUÇÃO DE PROBLEMAS DE SOFTWARE?

São um conjunto de abordagens de codificação reutilizáveis que resolvem os problemas mais comuns encontrados durante o desenvolvimento de aplicativos, tendo como Benefícios Principais:

- **Reutilização:** Aplicam soluções já testadas e comprovadas pela comunidade
- **Comunicação:** Criam vocabulário técnico comum entre desenvolvedores
- **Qualidade:** Promovem boas práticas como baixo acoplamento e alta coesão
- **Flexibilidade:** Tornam código mais adaptável a mudanças futuras
- **Manutenibilidade:** Facilitam modificações sem quebrar código existente
- **Documentação:** Tornam código mais autodocumentado e compreensível

Função Prática:

- Eles facilitam mudanças e evolução: padrões bem aplicados ajudam o sistema a absorver novos requisitos sem precisar reescrever grandes blocos de código.
- Cada pattern resolve problemas específicos - Singleton controla instanciação, Observer facilita comunicação desacoplada, Factory Method gerencia criação de objetos, etc.

Observações:

- Não são receitas prontas: funcionam como “modelos de pensamento”, que precisam ser adaptados ao contexto e ao problema.
- Devem ser usados apenas quando realmente resolvem um problema, não por serem "elegantes". Uso inadequado gera complexidade desnecessária.

Exemplo:

Em um sistema de agendamento de consultas, temos duas entidades, o médico e o paciente, os quais não se comunicam diretamente, pois entre os dois temos o agendamento que não sabe como a ação é feita, desacoplando todas as informações do médico e do paciente, torando o software mais extensível futuramente. Como Exemplo abaixo um dia podemos e adicionar enviar WhatsApp ou notificar convênio, basta criar um novo Observer, sem mudar a lógica central.

```
// Exemplo de uso
class Program {
    static void Main() {
        var sistema = new ConsultaAgendada();

        sistema.RegistrarObserver(new NotificacaoEmail());
        sistema.RegistrarObserver(new AtualizarAgendaMedico());
        sistema.RegistrarObserver(new RelatorioEstatistico());

        sistema.AgendarConsulta(new Consulta { Paciente = "João", Medico = "Dra.
Maria" });
    }
}
```

1.2. QUAL É O PAPEL DAS FACHADAS NA INTEGRAÇÃO DE CONTEXTOS LIMITADOS?

Simplificação da comunicação: a fachada expõe uma interface única e simplificada para interações entre dois contextos, escondendo a complexidade interna.

- **Isolamento de mudanças:** se o modelo interno de um contexto mudar, a fachada absorve essas mudanças sem quebrar o contrato com os outros contextos.
- **Redução de acoplamento:** em vez de outros contextos conhecerem os detalhes internos, eles falam apenas com a fachada.

- **Organização e clareza:** serve como “ponto de contato oficial” entre contextos, deixando explícito como eles se comunicam.

Exemplo:
A Fachada atua como ponto único de entrada para operações complexas entre contextos limitados.
Ela simplifica a integração, expondo apenas métodos necessários como AgendarConsulta.
Também reduz o acoplamento, isolando detalhes internos de cada contexto.
Assim, garante clareza, estabilidade e flexibilidade na comunicação entre subsistemas.

```
var facade = new AgendamentoFacade(
        new PacienteService(),
        new MedicoService(),
        new ConsultaService()
);

var consulta = facade.AgendarConsulta(1, 10, DateTime.Now.AddDays(1));
Console.WriteLine($"Consulta agendada para {consulta.Paciente.Nome} com
{consulta.Medico.Nome}");
```

2.1. Como o Domain-Driven Design (DDD) auxilia na gestão da complexidade de projetos de software?

O DDD ajuda a quebrar o domínio quando complexo em partes menores (bounded contexts), cada uma com regras próprias e linguagem específica (linguagem ubíqua).
Evitando acoplamento de uma ou mais entidades com funções diferentes, evitando assim Isso que o sistema vire um “monólito gigante/caótico ”, facilitando manutenção, evolução e entendimento devido a separações de funções desacopladas.
Ele também cria modelos ricos do negócio, alinhando equipe técnica e especialistas do domínio.

2.2. Explique a diferença entre design estratégico e tático.

- **Estratégico:** alto nível, foca em como dividir o domínio (subdomínios, bounded contexts, integrações). É o “mapa” do negócio.

  No contexto de agendamento de consultas temos como exemplo a separação do sistema em subdomínios/contextos limitados:
  - Agendamento (core domain).
  - Pacientes (suporte).
  - Médicos (suporte).
  - Pagamentos (genérico, pode ser terceirizado).

  Além disso, devemos definir como esses contextos se integram (ex.: via fachadas ou eventos).

- **Tático:** baixo nível, foca em como implementar dentro de cada contexto (entidades, agregados, repositórios, eventos, etc.). É o contexto do “código” em si e ou como implementar o código.

  No contexto de Agendamento:
  - Entidade Consulta (tem médico, paciente, data/hora).
  - Agregado: Consulta como raiz do agregado, garantindo regras (um médico não pode ter duas consultas no mesmo horário).
  - Repositório: ConsultaRepository para salvar e buscar consultas.
  - Evento de domínio: ConsultaAgendada para disparar notificação ao paciente.

2.3. Qual é a importância da Linguagem Ubíqua, mesmo em projetos que não usam DDD?

A Linguagem Ubíqua garante que desenvolvedores e especialistas falem a mesma língua, evitando ambiguidades.
Mesmo sem aplicar DDD completo, ela ajuda na clareza dos requisitos, documentação mais fiel ao negócio, redução de retrabalho, sendo assim uma ferramenta de alinhamento organizacional que vai muito além de metodologias específicas de desenvolvimento.

- **Comunicação Efetiva**

Elimina o "telefone sem fio" entre negócio e tecnologia. Quando todos usam os mesmos termos, as especificações são transmitidas com maior fidelidade do especialista do domínio até o código final.

- **Redução de Ambiguidades**

Evita interpretações múltiplas de um mesmo conceito. Por exemplo, "cliente" pode significar pessoa física, empresa, ou conta ativa - a linguagem ubíqua define exatamente qual o sentido em cada contexto.

- **Onboarding Acelerado**

Novos membros da equipe (técnicos ou não) conseguem entender o sistema mais rapidamente quando a terminologia é consistente em todos os artefatos.

- **Testes Mais Significativos**

Cenários de teste escritos na linguagem do domínio são mais facilmente validados pelos especialistas do negócio, melhorando a qualidade da cobertura dos testes.

- **Manutenção Simplificada**

Mudanças de requisitos são implementadas com menor risco de mal-entendidos, pois a semântica é clara e compartilhada.

2.4. O que são contextos principais, de suporte e genéricos? Por que os contextos genéricos, em geral, são contratados externamente?

CONTEXTOS PRINCIPAIS (Core Domain)
Onde está o diferencial competitivo do negócio (ex.: agendamento em um app médico).
É área que oferece vantagem competitiva e diferenciação no mercado, tendo como características a alta complexidade, conhecimento específico, valor estratégico máximo
Exemplos:

- Sistema de recomendação da Netflix
- Algoritmo de busca do Google
- Engine de precificação dinâmica do Uber

CONTEXTOS DE SUPORTE (Supporting Domain)
São necessários para o negócio funcionar, mas não oferecem diferenciação, e tem como característica uma complexidade média, conhecimento do domínio, valor tático, menor uso de recursos.
Exemplos:

- Sistema de faturamento
- Gestão de usuários e permissões
- Relatórios gerenciais

CONTEXTOS GENÉRICOS (Generic Domain)
Possuem funcionalidades comuns, sem diferenciação competitiva, tendo como características investimento mínimo, terceirizações, baixa complexidade, existência de soluções padronizadas disponíveis.
Exemplos:

- Autenticação e autorização

- Envio de emails
- Processamento de pagamentos
- Backup e monitoramento

Por que Contextos Genéricos são Terceirizados?

- **Economia de Recursos:**

Desenvolver internamente funcionalidades comuns desperdiça recursos que poderiam ser investidos no core domain.

- **Expertise Especializada:**

Fornecedores focados nessas soluções possuem conhecimento mais profundo e experiência acumulada (ex: Stripe para pagamentos, Auth0 para autenticação).

- **Manutenção e Evolução:**

Terceiros assumem responsabilidade por atualizações, segurança, conformidade regulatória e melhorias contínuas.

Soluções prontas aceleram desenvolvimento, permitindo foco no que realmente diferencia o negócio.

- **Exemplo Prático:**

Uma fintech deve investir pesado em seu algoritmo de análise de crédito (core), desenvolver internamente seu sistema de suporte, mas usar Stripe para pagamentos e AWS para autenticação (genéricos).
A estratégia é maximizar investimento onde há diferenciação e minimizar onde há uso comum a mais sistemas.

3.1. Desenhe o Mapa de Contexto do Projeto Pet Friends (use o modelo do TP3 com as melhorias que achar necessário) e classifique os contextos encontrados em “Principal”, “Genérico” e “Suporte” (neste cenário do AT o seu Bounded Context principal é o de Agendamento). Indique no mapa os tipos de relacionamentos.

3.2. Elabore uma lista com as estratégias a serem adotadas para as comunicações / integrações entre o contexto de agendamento e os contextos relacionados e aponte os tipos de comunicações entre contextos.

4.1. Explique a diferença entre Entidade e Objeto de Valor e dê um exemplo de cada usando o Pet Friends.

4.2. Modele os agregados que encontrar no contexto de Agendamento.