UNIVERSIDADE TÉCNICA DIOGO EUGÉNIO GUILANDE

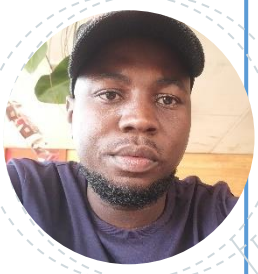
Unidade Curricular: Complementos de Engenharia de Software

Docente Regente: Alírio Rungo, PhD

Manual da Disciplina de Complementos de Engenharia de Software

2026

Alírio A. Rungo



Orientações Gerais para o Estudo

Antes de iniciar o estudo da disciplina, estamos propondo algumas dicas que irão facilitar o aproveitamento de seus estudos. Se preferir, e se você as considerar relevantes, inclua-as em seu próprio método de estudo.

Compromisso

Siga o ditado: "não deixe para amanhã o que pode fazer hoje!" Assuma o compromisso, independente da vontade de fazer ou não.

Não espere que o tutor, ou outra pessoa de seu relacionamento, chame sua atenção ou lembre que está na hora de estudar. É importante se conscientizar que você está aqui para adquirir conhecimento e que isso depende exclusivamente de você!

Agenda de Estudos

Organize a sua agenda de estudos. Faça disso um compromisso.

Não substitua a atividade agendada por qualquer outra. Compromisso assumido é uma obrigação! Estabeleça um horário padrão de estudos e busque cumprir essa meta. Somente impedimentos graves poderão mudar o que foi planejado. Observe a carga horária e o cronograma da disciplina.

Elabore o seu cronograma! Entregue as atividades na data estipulada, pois a pontualidade faz parte do sistema de avaliação!

Preparação para o Estudo

Assim como no seu ambiente de trabalho, prepare o ambiente de estudo. Reúna com antecedência o material necessário: livro, caderno de anotações, computador, lápis, borracha e caneta, entre outros. Sente- se confortavelmente e comece as leituras e atividades agendadas para aquele momento.

Dúvidas e Apoio

Não se acanhe em pedir ajuda a mim. Anote as dúvidas e dificuldades. **Não acumule dúvidas e busque orientação assim que elas surgirem.**

Exercícios e Avaliações

As atividades e avaliações determinadas no cronograma da disciplina são importantes para você. É uma forma de ensino-aprendizagem. Não deixe de resolvê-las e entregá-las na data estipulada. Procure manter a concentração em todos os momentos. Sempre surgem novas curiosidades. Busque respondê-las!

Com as avaliações corrigidas, procure aprender com os erros cometidos. Busque compreender onde está o erro. Esta também é uma forma de aprender.

Bom estudo e sucesso!

Prof. Alírio A. Rungo

Índice

UNIDADE TEMÁTICA 1 - ENGENHARIA DE SOFTWARE (CONCEITO & PARADIGMAS)

Bem-vindos à primeira unidade da nossa disciplina de Complementos de Engenharia de Software! Aqui, embarcaremos numa jornada para compreender o que é a Complementos de Engenharia de Software, por que ela é tão crucial no mundo tecnológico actual e como ela surgiu para resolver os desafios complexos do desenvolvimento de sistemas. Exploraremos a famosa "Crise do Software" e entenderemos como a aplicação de princípios de engenharia transformou a forma como construímos soluções digitais. Além disso, mergulharemos nos diferentes paradigmas de desenvolvimento, como o clássico modelo em Cascata, as abordagens Incremental e Espiral, e as populares metodologias Ágeis. Compreender estes conceitos fundamentais é como aprender o alfabeto antes de escrever um livro: eles são a base sobre a qual todo o seu conhecimento em Engenharia de Software será construído, permitindo-lhe escolher a melhor abordagem para cada projecto e garantir a qualidade do software que irá desenvolver.

1. Introdução à Engenharia de Software

1.1. O que é Engenharia de Software?

A **Engenharia de Software** é uma disciplina da engenharia que se dedica à aplicação de uma **abordagem sistemática**, **disciplinada** e **quantificável** ao desenvolvimento, operação e manutenção de software. Em outras palavras, é a **aplicação da engenharia ao software**.

O objectivo principal é **produzir software de alta qualidade**, que **atenda às necessidades dos utilizadores**, **dentro do prazo** e do **orçamento estabelecidos**.

> "*Engenharia de Software é a aplicação de uma abordagem sistemática, disciplinada e quantificável para o desenvolvimento, operação e manutenção de software*."

Por que precisamos da Engenharia de Software?

Historicamente, o desenvolvimento de software era muitas vezes visto como uma arte ou um ofício, resultando em projetos com atrasos, estouros de orçamento, falhas e insatisfação do cliente. Este cenário ficou conhecido como a **Crise do Software**.

Exemplo: Imagine um projecto de software para um sistema de controlo de tráfego aéreo. Se este projeto for desenvolvido sem uma abordagem de engenharia, as chances de erros críticos são altíssimas, podendo levar a consequências desastrosas. A Engenharia de Software fornece as **ferramentas e metodologias** para gerir essa complexidade e garantir a segurança e confiabilidade do sistema.

1.2. Atributos de um Bom Software

Um software de qualidade deve possuir uma série de atributos que garantam sua utilidade e longevidade. Sommerville (2011) destaca alguns atributos essenciais:

- ✓ **Confiabilidade:** O software deve operar sem falhas por um período razoável e sob condições normais de uso.
- ✓ **Eficiência:** O software deve utilizar os recursos do sistema (memória, processador, disco) de forma otimizada.
- ✓ **Usabilidade:** O software deve ser fácil de aprender e usar pelos seus utilizadores.
- ✓ **Manutenibilidade:** O software deve ser fácil de modificar para corrigir erros, adaptar-se a novos requisitos ou melhorar o desempenho.
- ✓ **Segurança:** O software deve proteger-se contra acessos não autorizados e ataques maliciosos.
- ✓ **Portabilidade:** O software deve ser capaz de operar em diferentes ambientes de hardware e software.

Exemplo: Um aplicativo bancário online precisa ser confiável (**transações seguras**), seguro (**proteção de dados do cliente**) e usável (**interface intuitiva para o utilizador**). Se falhar em qualquer um desses pontos, perderá a confiança dos utilizadores.

2. Paradigmas de Desenvolvimento de Software

Um **paradigma de desenvolvimento de software** é um **modelo ou padrão** que fornece uma estrutura para o processo de desenvolvimento de software. Ele define a abordagem geral, as fases, as actividades e as ferramentas utilizadas.

2.1. Paradigma Clássico (Modelo em Cascata - Waterfall)

O modelo em cascata é um dos mais antigos e tradicionais paradigmas. Ele propõe uma abordagem sequencial e linear, onde cada **fase** deve ser concluída antes que a próxima possa começar.

Fases:

- ✓ **1.Requisitos:** Levantamento e documentação detalhada de todas as necessidades do sistema.
- ✓ **2.Análise:** Estudo dos requisitos para entender o problema e definir a arquitetura do sistema.
- ✓ **3.Design:** Projeto da estrutura do software, incluindo arquitetura, componentes e interfaces.
- ✓ **4.Implementação:** Codificação do software com base no design.
- ✓ **5.Testes:** Verificação e validação do software para garantir que atende aos requisitos.
- ✓ **6.Manutenção:** Correção de erros, adaptação a novas necessidades e melhorias após a entrega.

Vantagens:

a) Simples e fácil de entender.
b) Boa documentação em cada fase.
c) Adequado para projetos com requisitos bem definidos e estáveis.

Desvantagens:

a) Pouca flexibilidade a mudanças nos requisitos.
b) Erros descobertos tardiamente são caros para corrigir.
c) O cliente só vê o produto final no final do ciclo.

Exemplo: O desenvolvimento de software para sistemas de controlo de equipamentos médicos, onde os requisitos são extremamente rigorosos e a mudança é indesejável após a fase de design, pode se beneficiar de uma abordagem mais próxima à cascata.

2.2. Paradigma Incremental

O paradigma incremental combina elementos do modelo em cascata com a natureza iteractiva da prototipagem. O sistema é construído em **pequenas partes (incrementos),** que são desenvolvidas e entregues ao cliente em ciclos sucessivos.

Características:

- ✓ Cada incremento adiciona funcionalidades ao sistema.
- ✓ Feedback do cliente é incorporado nos incrementos subsequentes.
- ✓ Reduz o risco de falha do projeto, pois o cliente vê o progresso regularmente.

Exemplo: Um sistema de gestão de biblioteca pode ser desenvolvido incrementalmente. O primeiro incremento pode ser a funcionalidade de pesquisa de livros. O segundo, a funcionalidade de empréstimo e devolução. O terceiro, a gestão de utilizadores, e assim por diante.

2.3. Paradigma Espiral

Proposto por Boehm (1988), o modelo espiral é um paradigma de desenvolvimento de software que enfatiza a análise e mitigação de riscos. Ele combina a natureza iteractiva da prototipagem com o controlo sistemático do modelo em cascata.

Fases (iterativas):

1. **Determinação de Objetivos:** Definir objectivos para a iteração actual.
2. **Análise de Riscos:** Identificar e mitigar riscos associados aos objectivos.
3. **Desenvolvimento e Testes:** Construir e testar um protótipo ou incremento.
4. **Planeamento:** Planear a próxima iteracção.

Vantagens:

- ✓ Foco na gestão de riscos.
- ✓ Adequado para projectos grandes e complexos com requisitos incertos.
- ✓ Permite a incorporação de feedback do cliente em cada ciclo.

Desvantagens:

- ✓ Complexo de gerir.
- ✓ Requer experiência em análise de riscos.
- ✓ Pode ser caro para projectos pequenos.

Exemplo: O desenvolvimento de um novo sistema de inteligência artificial para análise de dados financeiros, onde os riscos tecnológicos e de mercado são altos, seria um bom candidato para o modelo espiral.

2.4. Paradigma Ágil

Os métodos ágeis surgiram como uma resposta às limitações dos modelos mais tradicionais, especialmente em ambientes onde os requisitos mudam rapidamente. O Manifesto Ágil (2001) define os valores e princípios que guiam este paradigma.

Valores do Manifesto Ágil:

- ✓ Indivíduos e interacções mais que processos e ferramentas.
- ✓ Software a funcionar mais que documentação abrangente.
- ✓ Colaboração com o cliente mais que negociação de contratos.
- ✓ Responder à mudança mais que seguir um plano.

Metodologias Ágeis Comuns:

a) **Scrum:** Framework iterativo e incremental para gerir o desenvolvimento de produtos complexos. Baseia-se em ciclos curtos (Sprints), papéis definidos (Product Owner, Scrum Master, Development Team) e eventos (Daily Scrum, Sprint Review, Sprint Retrospective).

b) **Kanban:** Método para geir e melhorar o fluxo de trabalho. Visualiza o trabalho, limita o trabalho em progresso (WIP) e foca na melhoria contínua.

Vantagens:

- ✓ Alta adaptabilidade a mudanças.
- ✓ Entrega contínua de valor ao cliente.
- ✓ Maior satisfação do cliente e da equipa.

Desvantagens:

- ✓ Pode ser difícil para grandes equipas ou organizações com cultura tradicional.
- ✓ Requer forte envolvimento do cliente.
- ✓ A documentação pode ser menos formal.

Exemplo: O desenvolvimento de um aplicativo móvel para uma *startup*, onde a velocidade de entrega e a capacidade de adaptação às necessidades do mercado são cruciais, é ideal para metodologias ágeis como o Scrum.

2.5Exercícios Práticos: Engenharia de Software I - Aula 1: Engenharia de Software (Conceito & Paradigmas)

Estes exercícios visam consolidar a compreensão dos conceitos fundamentais da Engenharia de Software, a sua importância e os diferentes paradigmas de desenvolvimento.

Exercício 1.1: A Crise do Software e a Necessidade da Engenharia

Cenário: Nos anos 60 e 70, o desenvolvimento de software enfrentava problemas graves como atrasos, estouros de orçamento, falhas frequentes e software que não atendia às expectativas dos utilizadores. Este período ficou conhecido como a "Crise do Software" .

Questões:

a) Descreva, com suas próprias palavras, o que foi a Crise do Software e quais foram as suas principais manifestações.
b) Explique como o surgimento da Engenharia de Software se propôs a resolver os problemas da Crise do Software. Quais princípios ou abordagens foram introduzidos para mitigar esses desafios?
c) Dê um exemplo contemporâneo (pode ser hipotético) de um projeto de software que poderia estar a enfrentar uma "mini-crise" devido à falta de aplicação de princípios de Engenharia de Software. Justifique.

Exercício 1.2: Atributos de Qualidade de Software em Contextos Diferentes

Cenário: Um software de qualidade deve possuir diversos atributos que garantam sua utilidade e longevidade. No entanto, a importância de cada atributo pode variar dependendo do contexto de aplicação do software.

Questões:

a) Considere um sistema de controlo de voo para uma aeronave comercial. Identifique e justifique os três atributos de qualidade de software que seriam absolutamente críticos para este sistema. Explique as consequências da falha em cada um desses atributos.
b) Agora, considere um aplicativo de rede social para adolescentes. Identifique e justifique os três atributos de qualidade de software que seriam mais importantes

para o sucesso deste aplicativo. Compare a sua lista com a do sistema de controlo de voo e discuta as diferenças.

c) Para ambos os cenários, qual a importância da manutenibilidade? Explique como a facilidade de manutenção impacta cada um desses sistemas a longo prazo.

d) *Para um sistema de controlo de tráfego aéreo, identifique e descreva três atributos de qualidade de software (ex: confiabilidade, eficiência, segurança, manutenibilidade, usabilidade) que seriam críticos. Explique por que cada atributo é importante neste contexto e como a sua ausência poderia impactar negativamente o sistema*.

Exercício 1.3: Análise Comparativa de Paradigmas de Desenvolvimento

Cenário: Uma empresa de desenvolvimento de software está a decidir a melhor abordagem para dois novos projetos:

- ✓ **Projeto A:** Desenvolvimento de um sistema de gestão de base de dados para um banco central, com requisitos extremamente rigorosos, regulamentação pesada e pouca tolerância a erros. Os requisitos são bem definidos desde o início.
- ✓ **Projeto B:** Desenvolvimento de um novo jogo para dispositivos móveis, onde a inovação é constante, o feedback dos utilizadores é crucial e os requisitos podem mudar frequentemente durante o desenvolvimento.

Questões:

1. Para o Projeto A, qual paradigma de desenvolvimento de software (Cascata, Incremental, Espiral ou Ágil) você recomendaria? Justifique sua escolha, destacando as vantagens do paradigma selecionado para este tipo de projeto e as desvantagens dos outros paradigmas neste contexto.
2. Para o Projeto B, qual paradigma de desenvolvimento de software você recomendaria? Justifique sua escolha, explicando como o paradigma selecionado se alinha com as características de um projeto de jogo móvel e por que os outros paradigmas seriam menos adequados.
3. Discuta como a comunicação com o cliente e a capacidade de adaptação a mudanças diferem entre os paradigmas que você escolheu para o Projeto A e o Projeto B.
4. *Considere o desenvolvimento de um novo sistema de gestão académica para uma universidade. Discuta qual paradigma de desenvolvimento de software (Cascata, Ágil, Espiral) seria mais adequado para este projeto, justificando a sua escolha com base nas características, vantagens e desvantagens de cada um. Apresente um cenário onde cada um dos outros paradigmas poderiam ser mais apropriado.*

Exercício 1.4: Reflexão sobre o Manifesto Ágil

Cenário: O Manifesto para o Desenvolvimento Ágil de Software (2001) propõe quatro valores fundamentais que priorizam:

- ✓ Indivíduos e interações mais que processos e ferramentas.
- ✓ Software a funcionar mais que documentação abrangente.
- ✓ Colaboração com o cliente mais que negociação de contratos.
- ✓ Responder à mudança mais que seguir um plano.

Questões:

1. Escolha um dos valores do Manifesto Ágil e explique o seu significado em um contexto prático de desenvolvimento de software. Dê um exemplo de como este valor pode ser aplicado no dia a dia de uma equipa.
2. Pense em uma situação onde a priorização de "processos e ferramentas" sobre "indivíduos e interações" poderia levar a problemas em um projeto de software. Descreva essa situação e as possíveis consequências.
3. Na sua opinião, qual é o valor mais desafiador de ser implementado em uma organização tradicional de desenvolvimento de software? Justifique sua resposta.

UNIDADE TEMÁTICA 2: Arquiteturas Avançadas de Software

2.1 Introdução às Arquiteturas Avançadas

A arquitetura de software é a base fundamental sobre a qual os sistemas informáticos são construídos. Ela define a estrutura do sistema, os seus componentes, as propriedades visíveis externamente desses componentes e as relações entre eles. No contexto atual de desenvolvimento de software, onde a escalabilidade, a resiliência e a rápida adaptação às mudanças do mercado são cruciais, o estudo de arquiteturas avançadas torna-se imprescindível para os engenheiros de software.

Esta unidade temática explora os principais padrões arquiteturais modernos, contrastando abordagens tradicionais com paradigmas emergentes. O objetivo é dotar o estudante de conhecimentos sólidos para tomar decisões arquiteturais adequadas aos requisitos específicos de cada projeto, considerando os compromissos (trade-offs) inerentes a cada escolha.

2.2 Arquitetura em Camadas (Layered Architecture)

A arquitetura em camadas, frequentemente designada por arquitetura n-tier, é um dos padrões arquiteturais mais comuns e amplamente adotados na indústria de software. Este padrão organiza os componentes do sistema em camadas horizontais, onde cada camada desempenha um papel específico e tem responsabilidades bem definidas dentro da aplicação.

2.2.1 Estrutura e Componentes

Na sua forma mais clássica, a arquitetura em camadas é composta por quatro camadas principais, embora este número possa variar consoante a complexidade do sistema:

1. **Camada de Apresentação (Presentation Layer):** Responsável por interagir com o utilizador ou com sistemas externos. Esta camada gere a interface do utilizador (UI) e a experiência do utilizador (UX), traduzindo as ações do utilizador em pedidos compreensíveis para as camadas inferiores.
2. **Camada de Negócio (Business Layer):** Contém a lógica central da aplicação. É nesta camada que as regras de negócio, os cálculos e as validações são executados.

Ela atua como o "cérebro" do sistema, processando os dados recebidos da camada de apresentação e coordenando as ações necessárias.

3. **Camada de Persistência (Persistence Layer):** Responsável por gerir a comunicação com o armazenamento de dados. Esta camada abstrai a complexidade das operações de base de dados (como consultas SQL), fornecendo uma interface simplificada para a camada de negócio aceder e manipular os dados.
4. **Camada de Base de Dados (Database Layer):** O sistema de armazenamento físico dos dados, que pode ser uma base de dados relacional (como MySQL ou PostgreSQL) ou não relacional (como MongoDB).

2.2.2 Princípios e Características

Um princípio fundamental da arquitetura em camadas é a separação de preocupações ***(Separation of Concerns***). Cada camada deve focar-se exclusivamente na sua responsabilidade, sem necessitar de conhecer os detalhes de implementação das outras camadas. Além disso, a comunicação entre as camadas ocorre tipicamente de forma sequencial, de cima para baixo. A camada de apresentação comunica com a camada de negócio, que por sua vez comunica com a camada de persistência.

Este isolamento facilita a manutenção e a evolução do software. Por exemplo, se for necessário alterar a interface do utilizador de uma aplicação web para uma aplicação móvel, apenas a camada de apresentação precisa de ser modificada, mantendo as camadas de negócio e persistência intactas.

2.2.3 Aplicabilidade no Contexto Empresarial e Exemplos

No contexto empresarial, a arquitetura em camadas é a escolha padrão para sistemas de informação internos, portais corporativos e aplicações de gestão (ERP, CRM) de pequena a média dimensão. A sua aplicabilidade é alta quando a equipa de desenvolvimento é estruturada por especialidades (ex: equipa de frontend, equipa de backend, administradores de base de dados).

Exemplo Prático e Didático: Sistema de Gestão de Recursos Humanos (RH)

Imagine que uma empresa está a desenvolver um sistema interno para gerir as férias dos funcionários.

- **Camada de Apresentação:** O ecrã web onde o funcionário clica no botão "Solicitar Férias" e preenche as datas.
- **Camada de Negócio:** O código que verifica se o funcionário tem dias de férias disponíveis suficientes e se as datas não coincidem com um período de bloqueio da empresa.
- **Camada de Persistência:** O código (ex: usando Hibernate ou Entity Framework) que pega no objeto "Pedido de Férias" aprovado e o transforma numa instrução INSERT para a base de dados.
- **Camada de Base de Dados:** O servidor MySQL onde a tabela pedidos_ferias reside fisicamente.

2.2.4 Vantagens e Desvantagens

Vantagens	Desvantagens
Simplicidade: É um padrão intuitivo e fácil de compreender, alinhando-se frequentemente com a estrutura organizacional das equipas de desenvolvimento.	**Acoplamento**: Pode gerar um forte acoplamento entre as camadas, dificultando a extração de componentes isolados.
Testabilidade: As camadas podem ser testadas de forma independente, utilizando mocks ou stubs para simular o comportamento das camadas adjacentes.	**Desempenho:** A passagem de dados por múltiplas camadas pode introduzir latência, especialmente se houver transformações de dados em cada etapa.
Facilidade de Desenvolvimento: Permite que diferentes equipas trabalhem em camadas distintas simultaneamente, acelerando o processo de desenvolvimento inicial.	**Escalabilidade Monolítica:** Tipicamente, a aplicação é implantada como um único bloco (monólito), o que significa que para escalar uma funcionalidade específica, é necessário escalar toda a aplicação.

2.3 Arquitetura Orientada a Serviços (SOA)

A **Arquitetura Orientada a Serviços (SOA - Service-Oriented Architecture)** representa uma evolução significativa em relação aos sistemas monolíticos tradicionais. A SOA é um paradigma de design de software onde as funcionalidades da aplicação são disponibilizadas como serviços discretos, independentes e reutilizáveis, que comunicam entre si através de uma rede.

2.3.1 Conceitos Fundamentais

Na **SOA**, um serviço é uma unidade lógica de funcionalidade que representa uma atividade de negócio repetível com um resultado especificado. Os serviços são concebidos

para serem auto-contidos e atuarem como "caixas-pretas" para os seus consumidores, ocultando a complexidade da sua implementação interna.

A comunicação entre os serviços na SOA é frequentemente mediada por um Enterprise Service Bus (ESB), que atua como um canal central de comunicação, lidando com o roteamento de mensagens, a transformação de protocolos e a segurança.

2.3.2 Princípios da SOA

A implementação eficaz de uma arquitetura SOA baseia-se em vários princípios fundamentais:

- **Contrato de Serviço Padronizado:** Os serviços aderem a um acordo de comunicação padrão, definido por documentos de descrição de serviço (como WSDL em serviços SOAP).
- **Abstração de Serviço:** A lógica interna do serviço é oculta dos consumidores, expondo apenas a interface necessária para a interação.
- **Autonomia de Serviço:** Os serviços têm controlo sobre a funcionalidade que encapsulam, operando de forma independente.
- **Ausência de Estado (Statelessness):** Idealmente, os serviços não mantêm o estado das interações entre os pedidos, o que melhora a escalabilidade e a resiliência.
- **Composibilidade:** Os serviços podem ser combinados para criar novos serviços mais complexos ou processos de negócio completos.
- **Descoberta de Serviço:** Os serviços são complementados com metadados que permitem a sua descoberta e interpretação dinâmica por outros sistemas.

2.3.3 Aplicabilidade no Contexto Empresarial e Exemplos

A SOA é altamente aplicável em grandes corporações (bancos, seguradoras, telecomunicações) que possuem múltiplos sistemas legados (legacy systems) desenvolvidos ao longo de décadas em diferentes tecnologias (Mainframes, Java, .NET). A SOA permite que estes sistemas comuniquem entre si sem precisarem de ser reescritos, expondo as suas funcionalidades como serviços através de um ESB.

Exemplo: Integração Num Banco

Um banco tem um sistema antigo em Cobol que gere as contas à ordem, e um sistema mais recente em Java que gere os cartões de crédito. O banco quer lançar uma nova aplicação móvel que mostre o saldo total do cliente.

Em vez de a aplicação móvel tentar ligar-se diretamente ao Cobol e ao Java (o que seria um pesadelo técnico), o banco implementa uma arquitetura SOA:

- 1.Cria-se um serviço "ObterSaldoConta" (que fala com o Cobol).
- 2.Cria-se um serviço "ObterSaldoCartao" (que fala com o Java).
- 3.Cria-se um serviço composto "ObterPosicaoGlobal" que chama os dois serviços anteriores, soma os valores e devolve o resultado à aplicação móvel num formato padrão (ex: XML ou JSON).

2.3.4 Vantagens e Desvantagens

Vantagens	Desvantagens
Reutilização: Os serviços podem ser partilhados por múltiplas aplicações, reduzindo a duplicação de esforço e promovendo a consistência.	**Complexidade:** A introdução de um ESB e a gestão de múltiplos serviços aumentam a complexidade da infraestrutura e das operações.
Interoperabilidade: Facilita a integração de sistemas heterogéneos, desenvolvidos em diferentes tecnologias e plataformas.	**Sobrecarga de Comunicação**: A comunicação através da rede e a transformação de mensagens no ESB podem introduzir latência e afetar o desempenho.
Alinhamento com o Negócio: Os serviços são frequentemente modelados em torno de processos de negócio reais, melhorando a comunicação entre as equipas técnicas e de negócio.	**Custo Inicial:** A implementação de uma arquitetura SOA requer um investimento significativo em planeamento, infraestrutura e governação.

2.4 Microserviços (Microservices Architecture)

A arquitetura de microserviços é uma abordagem contemporânea que leva os princípios da SOA ao extremo. Em vez de construir uma aplicação como um único monólito ou um conjunto de grandes serviços, a arquitetura de microserviços estrutura a aplicação como uma coleção de serviços pequenos, fracamente acoplados e independentemente implementáveis.

2.4.1 Características Distintivas

Os microserviços distinguem-se de outras arquiteturas por várias características chave:

- **Granularidade Fina:** Cada microserviço foca-se numa única capacidade de negócio bem definida (princípio da responsabilidade única).
- **Implementação Independente:** Os microserviços podem ser atualizados, testados e implementados sem afetar o resto do sistema. Isto permite ciclos de lançamento mais rápidos e contínuos.
- **Descentralização de Dados:** Ao contrário das arquiteturas monolíticas que partilham uma única base de dados, nos microserviços, cada serviço gere a sua própria base de dados. Isto garante o isolamento e evita o acoplamento ao nível dos dados.
- **Comunicação Leve:** Os microserviços comunicam entre si utilizando protocolos leves, como HTTP/REST ou mensagens assíncronas (ex: RabbitMQ, Kafka), em vez de dependerem de um ESB pesado.
- **Diversidade Tecnológica (Polyglot Persistence/Programming):** As equipas têm a liberdade de escolher a melhor linguagem de programação, framework e base de dados para cada microserviço específico, otimizando a solução para o problema em questão.

2.4.2 Aplicabilidade no Contexto Empresarial e Exemplos

Os microserviços são a arquitetura de eleição para empresas tecnológicas modernas (como Netflix, Uber, Spotify) e startups que necessitam de escalar rapidamente, lançar novas funcionalidades várias vezes ao dia e manter uma alta disponibilidade. São ideais para aplicações complexas baseadas na nuvem *(cloud-native).*

Exemplo: Plataforma de E-commerce (ex: Amazon)

Se a Amazon fosse um monólito e a funcionalidade de "Recomendações" falhasse devido a um erro de código, toda a loja online poderia ir abaixo. Com microserviços, a arquitetura é dividida:

- **Microserviço de Catálogo:** Gere os produtos (escrito em Go, usa base de dados NoSQL para leitura rápida).

- **Microserviço de Carrinho:** Gere o que o utilizador quer comprar (escrito em Node.js, usa Redis para cache em memória).
- **Microserviço de Pagamento:** Processa o cartão de crédito (escrito em Java, usa base de dados relacional rigorosa para transações financeiras).

Se o serviço de Recomendações falhar, o utilizador simplesmente não vê as recomendações, mas pode continuar a navegar no catálogo, adicionar ao carrinho e pagar. Além disso, na Black Friday, a equipa de infraestrutura pode escalar (adicionar mais servidores) apenas ao "Microserviço de Pagamento" e "Carrinho", sem desperdiçar recursos a escalar o serviço de "Envio de Emails".

2.4.3 Desafios e Complexidades

Apesar dos seus inúmeros benefícios, a arquitetura de microserviços introduz desafios substanciais:

- **Complexidade Distribuída:** Gerir dezenas ou centenas de serviços independentes requer ferramentas avançadas de orquestração (como Kubernetes), monitorização e registo centralizado.
- **Consistência de Dados:** Como cada serviço tem a sua própria base de dados, manter a consistência dos dados em transações que envolvem múltiplos serviços torna-se complexo, exigindo padrões como o Saga Pattern.
- **Testes:** Testar as interações entre múltiplos serviços distribuídos é consideravelmente mais difícil do que testar um sistema monolítico.

2.5 Arquiteturas Orientadas a Eventos (Event-Driven Architecture - EDA)

A Arquitetura Orientada a Eventos (EDA) é um paradigma de design de software focado na produção, deteção, consumo e reação a eventos. Um evento é definido como uma "mudança significativa de estado" no sistema.

2.5.1 Componentes e Fluxo de Eventos

Numa arquitetura EDA, os componentes do sistema comunicam de forma assíncrona através da emissão e receção de eventos. O sistema é tipicamente composto por:

1. **Produtores de Eventos (Event Producers):** Componentes que detetam uma mudança de estado e publicam um evento. O produtor não tem conhecimento de quem irá consumir o evento ou de como este será processado.
2. **Canais de Eventos (Event Channels / Event Brokers):** O mecanismo de transporte que recebe os eventos dos produtores e os distribui aos consumidores interessados. Exemplos incluem Apache Kafka, Amazon EventBridge ou RabbitMQ.
3. **Consumidores de Eventos (Event Consumers / Sinks):** Componentes que subscrevem tipos específicos de eventos e reagem a eles quando são recebidos.

2.5.2 Aplicabilidade no Contexto Empresarial e Exemplos

A EDA é crucial em cenários empresariais que exigem processamento em tempo real, alta escalabilidade e reatividade. É amplamente utilizada em sistemas de Internet das Coisas (IoT), deteção de fraudes financeiras, análise de dados em tempo real e sistemas de logística complexos.

Exemplo: Aplicação de Partilha de Viagens (ex: Uber)

Quando um passageiro clica em "Pedir Viagem", o telemóvel não fica bloqueado à espera que um motorista aceite. Em vez disso, a aplicação funciona por eventos:

1. **Produtor:** A app do passageiro emite um evento: ViagemSolicitada {id_passageiro: 123, local: "Maputo", destino: "Matola"}.
2. **Canal (Broker):** O evento entra no Apache Kafka da empresa.
3. **Consumidores** (Reações simultâneas e independentes):
   - O Serviço de Alocação ouve o evento e começa a procurar motoristas num raio de 5km.
   - O Serviço de Preços ouve o evento e calcula a tarifa dinâmica com base no trânsito atual.
   - O Serviço de Notificações ouve o evento e prepara-se para enviar um SMS ao passageiro assim que um motorista for encontrado.

Tudo isto acontece de forma assíncrona. Se o Serviço de Notificações estiver temporariamente em baixo, o evento fica guardado no Broker e será processado assim que o serviço recuperar, não impedindo que a viagem seja alocada.

2.5.3 Vantagens e Desvantagens

Vantagens	Desvantagens
Acoplamento Extremamente Fraco: Os produtores e consumidores são totalmente independentes, não necessitando de conhecer a existência uns dos outros.	**Complexidade de Rastreamento:** Seguir o fluxo de execução de um processo de negócio que é desencadeado por múltiplos eventos assíncronos pode ser muito difícil de depurar.
Escalabilidade e Desempenho: A natureza assíncrona permite que o sistema lide com grandes volumes de dados e picos de tráfego de forma eficiente.	**Garantia de Entrega:** Garantir que os eventos são entregues e processados na ordem correta, especialmente em cenários de falha de rede, requer infraestrutura robusta.
Reatividade em Tempo Real: Ideal para sistemas que necessitam de reagir instantaneamente a mudanças de estado ou a ações do utilizador.	**Curva de Aprendizagem:** Requer uma mudança de mentalidade em relação à programação síncrona tradicional, exigindo que os programadores pensem em termos de fluxos de dados e reações assíncronas.

2.6 Exercícios de Avaliação

Para consolidar os conhecimentos adquiridos nesta unidade, propõem-se os seguintes exercícios, divididos em três categorias:

2.6.1 Exercícios Teóricos

1. **Definição:** Explique, por palavras suas, a diferença fundamental entre a Arquitetura Orientada a Serviços (SOA) e a Arquitetura de Microserviços no que diz respeito à gestão de bases de dados.
2. **Comparação:** Liste duas vantagens e duas desvantagens da Arquitetura em Camadas comparativamente à Arquitetura Orientada a Eventos.
3. **Conceitos:** Num sistema baseado em eventos (EDA), qual é o papel do Event Broker (Canal de Eventos)? Dê um exemplo de uma tecnologia que cumpra este papel.

2.6.2 Exercícios Práticos

1. **Modelação de Camadas:** Desenhe um diagrama de blocos simples (pode ser em papel) representando a arquitetura em camadas para um sistema de "Reserva de Bilhetes de Cinema". Identifique claramente as 4 camadas clássicas e descreva

uma ação específica que ocorreria em cada camada quando um utilizador tenta comprar um bilhete.

2. **Decomposição em Microserviços:** Considere um sistema monolítico de uma "Universidade Online" que gere: Alunos, Professores, Cursos, Inscrições, Pagamentos e Fóruns de Discussão. Proponha uma divisão deste monólito em pelo menos 4 microserviços distintos. Justifique a sua escolha com base no princípio da responsabilidade única.

2.6.3 Exercícios Reflexivos (Estudo de Caso)

- **Cenário:** A empresa "FastDelivery", uma startup de entrega de comida ao domicílio, começou há 2 anos com uma aplicação monolítica em arquitetura de 3 camadas. Atualmente, têm milhares de pedidos por hora à hora de almoço. O sistema está a ficar lento, e sempre que a equipa tenta adicionar uma nova funcionalidade (como um novo método de pagamento), o sistema inteiro tem de ser reiniciado, causando interrupções no serviço. Além disso, a base de dados única está a atingir o seu limite de capacidade de leitura/escrita.
- **Questão para Reflexão:** Como Engenheiro de Software Sénior contratado pela FastDelivery, que arquitetura avançada (ou combinação de arquiteturas) recomendaria para resolver estes problemas? Elabore um pequeno relatório (1-2 parágrafos) justificando a sua escolha com base nos problemas apresentados no cenário e nos benefícios da arquitetura proposta.

UNIDADE TEMÁTICA 3: Padrões de Projeto e Refatoração

3.1 Introdução aos Padrões de Projeto

O desenvolvimento de software é uma atividade complexa que frequentemente se depara com problemas recorrentes. Em vez de reinventar a roda a cada novo desafio, os engenheiros de software recorrem a soluções comprovadas e testadas pelo tempo. Estas soluções são conhecidas como Padrões de Projeto (Design Patterns).

Um padrão de projeto não é um código finalizado que pode ser diretamente copiado e colado num programa. É, antes, uma descrição ou modelo geral para resolver um problema que ocorre frequentemente num determinado contexto de design de software. Os padrões fornecem um vocabulário comum para os programadores, facilitando a comunicação e a documentação da arquitetura do sistema.

A formalização dos padrões de projeto na engenharia de software é amplamente atribuída ao livro seminal "Design Patterns: Elements of Reusable Object-Oriented Software", publicado em 1994 por Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. Este grupo de autores ficou mundialmente conhecido como a Gang of Four (GoF) .

3.2 Padrões Clássicos (GoF)

O catálogo original da GoF documenta 23 padrões de projeto clássicos, que são categorizados em três grupos principais com base no seu propósito: Criacionais, Estruturais e Comportamentais.

3.2.1 Padrões Criacionais (Creational Patterns)

Os padrões criacionais abstraem o processo de instanciação de objetos. Eles ajudam a tornar um sistema independentemente de como os seus objetos são criados, compostos e representados. O foco principal destes padrões é a flexibilidade na criação de objetos, decidindo quais objetos precisam de ser criados para um determinado caso de uso.

Aplicabilidade no Contexto Empresarial:

- Em sistemas empresariais complexos, a criação de objetos pode envolver lógica condicional pesada (ex: criar diferentes tipos de faturas dependendo do país do

cliente). Os padrões criacionais centralizam esta lógica, evitando que o código de negócio fique poluído com instruções new e if/else espalhadas por todo o lado.

Padrão	Descrição e Propósito	Exemplo Prático e Didático
Singleton	Garante que uma classe tenha apenas uma única instância e fornece um ponto global de acesso a ela.	Gestor de Configurações: Uma aplicação empresarial precisa de ler as configurações da base de dados (URL, username, password). O Singleton garante que o ficheiro de configuração é lido apenas uma vez e partilhado por toda a aplicação, poupando memória e tempo de processamento.
Factory Method	Define uma interface para criar um objeto, mas permite que as subclasses decidam qual classe instanciar.	Sistema de Logística: Uma empresa de transportes tem uma classe Transporte. O Factory Method permite criar subclasses TransporteTerrestre (que cria objetos Camião) e TransporteMaritimo (que cria objetos Navio), sem alterar o código principal que agenda as entregas.
Abstract Factory	Fornece uma interface para criar famílias de objetos relacionados ou dependentes sem especificar as suas classes concretas.	UI Multi-plataforma: Uma aplicação precisa de funcionar em Windows e macOS. A Abstract Factory cria "famílias" de botões e caixas de texto que combinam visualmente entre si, garantindo que um botão estilo Windows não apareça numa janela estilo macOS.
Builder	Separa a construção de um objeto complexo da sua representação, permitindo que o mesmo processo de construção crie diferentes representações.	Gerador de Relatórios: Um sistema financeiro precisa de gerar relatórios mensais. O Builder permite usar o mesmo processo de extração de dados para construir o relatório final em formato PDF, Excel ou HTML, dependendo do pedido do utilizador.
Prototype	Especifica os tipos de objetos a criar usando uma instância prototípica e cria novos objetos copiando este protótipo.	Editor Gráfico: Num software de desenho (como o Visio), quando o utilizador copia e cola uma forma complexa (ex: um diagrama de rede com várias ligações), o sistema usa o Prototype para clonar o objeto existente em vez de o recriar do zero.

3.2.2 Padrões Estruturais (Structural Patterns)

Os padrões estruturais preocupam-se com a forma como as classes e os objetos são compostos para formar estruturas maiores. Eles utilizam a herança para compor interfaces ou implementações e definem formas de compor objetos para obter novas funcionalidades, garantindo que a estrutura se mantenha flexível e eficiente.

Aplicabilidade no Contexto Empresarial:

- Estes padrões são vitais quando empresas adquirem outras empresas ou compram software de terceiros (APIs/Bibliotecas) e precisam de integrar esses novos sistemas com o seu código legado, sem reescrever tudo do zero.

Padrão	Descrição e Propósito	Exemplo Prático e Didático
Adapter	Converte a interface de uma classe noutra interface esperada pelos clientes. Permite que classes com	Integração de Pagamentos: O seu e-commerce usa uma interface ProcessarPagamento. A empresa decide mudar do PayPal para o Stripe. O Stripe tem

	interfaces incompatíveis trabalhem em conjunto.	métodos diferentes. Em vez de reescrever o e-commerce, cria-se um StripeAdapter que traduz os pedidos do seu sistema para o formato que o Stripe entende.
Decorator	Anexa responsabilidades adicionais a um objeto dinamicamente. Fornece uma alternativa flexível à herança para estender a funcionalidade.	Sistema de Notificações: Tem um serviço básico que envia notificações por Email. Com o Decorator, pode "embrulhar" este serviço com funcionalidades extra em tempo de execução: adicionar um decorador para enviar também por SMS, e outro para enviar para o Slack, sem criar subclasses complexas como EmailE_SMS_Notifier.
Facade	Fornece uma interface unificada para um conjunto de interfaces num subsistema. Define uma interface de nível mais alto que torna o subsistema mais fácil de usar.	Processamento de Encomendas: Quando um cliente clica em "Comprar", o sistema tem de verificar stock, cobrar o cartão, gerar fatura e notificar o armazém. O Facade cria um método simples finalizarCompra() que oculta toda esta complexidade do frontend.
Composite	Compõe objetos em estruturas de árvore para representar hierarquias parte-todo. Permite que os clientes tratem objetos individuais e composições de objetos de forma uniforme.	Catálogo de Produtos: Numa loja de informática, pode vender um "Rato" (objeto individual) ou um "Kit Escritório" (que contém um Rato, Teclado e Monitor). O Composite permite calcular o preço total chamando getPreco() de forma idêntica, quer seja num item único ou num kit complexo.

3.2.3 Padrões Comportamentais (Behavioral Patterns)

Os padrões comportamentais concentram-se nos algoritmos e na atribuição de responsabilidades entre os objetos. Eles não descrevem apenas padrões de objetos ou classes, mas também os padrões de comunicação entre eles, caracterizando fluxos de controlo complexos.

Aplicabilidade no Contexto Empresarial:

- São cruciais para implementar regras de negócio dinâmicas e fluxos de trabalho (workflows) onde o comportamento do sistema precisa de mudar com base no estado atual ou em eventos externos, mantendo o código limpo e testável.

Padrão	Descrição e Propósito	Exemplo Prático e Didático
Observer	Define uma dependência um-para-muitos entre objetos, de modo que quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente.	Dashboard Financeiro: O preço de uma ação (Subject) muda na bolsa. Múltiplos gráficos e tabelas no ecrã do corretor (Observers) são atualizados instantaneamente e automaticamente, sem precisarem de perguntar constantemente "o preço já mudou?".
Strategy	Define uma família de algoritmos, encapsula cada um deles e torna-os intercambiáveis. Permite que o algoritmo varie independentemente dos clientes que o utilizam.	Cálculo de Portes de Envio: Um e-commerce tem de calcular portes. Em vez de um método gigante com if/else para CTT, DHL e FedEx, cria-se uma estratégia para cada transportadora. O sistema seleciona a estratégia correta em tempo de execução com base na escolha do cliente.
Command	Encapsula uma solicitação como um objeto, permitindo parametrizar clientes com diferentes solicitações, enfileirar ou registar solicitações e suportar operações que podem ser desfeitas.	Sistema de Automação Residencial: Uma app de Smart Home tem botões. O botão não sabe como ligar a luz. Ele apenas executa um objeto ComandoLigarLuz. Isto permite criar macros (executar vários comandos de uma vez) ou agendar comandos para mais tarde.
State	Permite que um objeto altere o seu comportamento quando o seu estado	Máquina de Venda Automática: O comportamento do botão "Tirar Café" muda

	interno muda. O objeto parecerá ter mudado de classe.	dependendo do estado da máquina: se o estado for "Sem Moeda", pede dinheiro; se for "Com Moeda", tira o café; se for "Sem Água", mostra erro. O padrão State evita dezenas de if/else espalhados pelo código.

3.3 Code Smells (Cheiros de Código)

O termo Code Smell (Cheiro de Código) foi popularizado por Kent Beck e Martin Fowler no final da década de 1990. Um code smell não é um erro técnico (bug) que impede o programa de funcionar. Em vez disso, é uma característica no código-fonte que indica uma fraqueza no design e que pode atrasar o desenvolvimento ou aumentar o risco de falhas no futuro.

Os code smells são frequentemente indicadores de Dívida Técnica (Technical Debt) acumulada. A pressão para entregar funcionalidades rapidamente, priorizando o tempo de colocação no mercado (time-to-market) em detrimento da qualidade do código, é uma das principais causas do aparecimento destes "cheiros" .

3.3.1 Categorias de Code Smells e Exemplos Empresariais

Os code smells podem ser classificados em diferentes níveis de granularidade dentro do sistema de software:

Smells ao Nível da Aplicação

- **Duplicated Code (Código Duplicado):** Ocorre quando blocos de código idênticos ou muito semelhantes existem em múltiplos locais. **Exemplo Empresarial:** O código que valida se um NIF (Número de Identificação Fiscal) é válido foi copiado e colado no módulo de Faturação, no módulo de Registo de Clientes e no módulo de Recursos Humanos. Se a regra de validação do NIF mudar por lei, o programador terá de se lembrar de alterar nos três locais.
- **Shotgun Surgery (Cirurgia de Caçadeira):** Acontece quando uma única alteração conceptual exige que o programador faça pequenas modificações em muitas classes diferentes. **Exemplo Empresarial:** A empresa decide adicionar um novo tipo de utilizador ("Auditor"). Para implementar isto, o programador tem de alterar a classe de Login, a classe de Permissões, a classe de Relatórios e a classe de Base de Dados. A responsabilidade está demasiado fragmentada.

Smells ao Nível da Classe

- **Large Class / God Object (Classe Grande / Objeto Deus):** Uma classe que assumiu demasiadas responsabilidades, contendo um número excessivo de campos e métodos. **Exemplo Empresarial:** Uma classe GestorDeEmpresa que tem 5000 linhas de código e lida com a contratação de funcionários, pagamento de salários, gestão de stock e emissão de faturas. É impossível de testar e qualquer alteração tem um alto risco de quebrar outra funcionalidade.
- **Data Clump (Aglomerado de Dados):** Ocorre quando um grupo de variáveis é frequentemente passado em conjunto como parâmetros para métodos em várias partes do programa. **Exemplo Empresarial:** Métodos como criarEncomenda(rua, cidade, codigoPostal, pais) e atualizarMorada(rua, cidade, codigoPostal, pais). Isto sugere que estas variáveis deveriam ser agrupadas num único objeto Endereco.

Smells ao Nível do Método

- **Long Method (Método Longo):** Métodos com demasiadas linhas de código. Quanto mais longo for um método, mais difícil será compreender o que ele faz. **Exemplo Empresarial:** Um método processarFechoDoMes() que tem 300 linhas de código, misturando cálculos matemáticos complexos, chamadas à base de dados e formatação de texto para o relatório final.
- **Too Many Parameters (Demasiados Parâmetros):** Uma lista longa de parâmetros torna a chamada e o teste da função complicados. **Exemplo Empresarial:** pesquisarClientes(nome, idadeMinima, idadeMaxima, cidade, estadoAtivo, dataRegistoInicio, dataRegistoFim). É fácil o programador trocar a ordem dos parâmetros ao chamar a função, causando bugs difíceis de detetar.

3.4 Refatoração de Código (Code Refactoring)

A Refatoração é o processo de reestruturar o código-fonte existente, alterando a sua estrutura interna sem modificar o seu comportamento externo observável . O objetivo principal da refatoração é melhorar os atributos não-funcionais do software, como a legibilidade, a manutenibilidade e a extensibilidade, combatendo ativamente os code smells e reduzindo a dívida técnica.

"Ao melhorar continuamente o design do código, tornamo-lo cada vez mais fácil de trabalhar. Isto contrasta fortemente com o que acontece tipicamente: pouca refatoração

e muita atenção prestada à adição expedita de novas funcionalidades." — Joshua Kerievsky

3.4.1 O Processo de Refatoração e Testes

A refatoração não deve ser um evento isolado e massivo (ex: "vamos parar o desenvolvimento durante um mês para refatorar"). Deve ser uma prática contínua (regra do escuteiro: "deixe o código mais limpo do que o encontrou"). O processo de refatoração é intrinsecamente dependente de Testes Unitários (Unit Tests) automatizados .

O ciclo ideal de refatoração segue estes passos:

1. Garantir que existe uma suite de testes unitários abrangente e que todos os testes passam (Verde).
2. Identificar um code smell.
3. Aplicar uma pequena transformação de refatoração (Micro-refactoring).
4. Executar os testes novamente. Se falharem, a alteração é revertida imediatamente. Se passarem, a transformação é consolidada.
5. Repetir o processo.

3.4.2 Técnicas Comuns de Refatoração

Existem dezenas de técnicas de refatoração catalogadas. Algumas das mais comuns incluem:

1. ***Extract Method (Extrair Método):*** Transforma um fragmento de código dentro de um método longo num novo método independente, com um nome que explica o seu propósito. Resolve o smell "Long Method".

    •**Antes:** Um método gigante que calcula o salário e depois imprime o recibo.

    •**Depois:** O método principal chama calcularSalario() e depois imprimirRecibo().

2. ***Extract Class (Extrair Classe):*** Quando uma classe está a fazer o trabalho de duas, cria-se uma nova classe e movem-se os campos e métodos relevantes da classe antiga para a nova. Resolve o smell "Large Class".

    •**Antes:** Classe Funcionario que contém dados pessoais e também a lógica de cálculo de impostos.

•**Depois:** Classe Funcionario (dados pessoais) e classe CalculadoraDeImpostos (lógica financeira).

3. ***Rename Method/Variable (Renomear Método/Variável):*** Alterar o nome de um método ou variável para que revele claramente a sua intenção. Melhora drasticamente a legibilidade.

    •**Antes:** int d; // tempo decorrido em dias

    •**Depois:** int tempoDecorridoEmDias;

4. ***Introduce Parameter Object (Introduzir Objeto de Parâmetro):*** Substitui um grupo de parâmetros que frequentemente ocorrem juntos por um único objeto. Resolve o smell "Data Clump" e "Too Many Parameters".

    •**Antes**: marcarReuniao(dataInicio, dataFim)

    •**Depois:** marcarReuniao(IntervaloDeTempo)

3.5 Exercícios de Avaliação

Para consolidar os conhecimentos adquiridos nesta unidade, propõem-se os seguintes exercícios:

3.5.1 Exercícios Teóricos

1. Conceitos GoF: Explique a diferença fundamental entre os padrões criacionais Factory Method e Abstract Factory. Em que cenário escolheria um em detrimento do outro?
2. Identificação de Smells: O que é a "Dívida Técnica" (Technical Debt) e como é que a presença de Code Smells se relaciona com este conceito?
3. Princípios de Refatoração: Porque é que a existência de Testes Unitários automatizados é considerada um pré-requisito obrigatório antes de iniciar um processo de refatoração de código?

3.5.2 Exercícios Práticos

1. **Aplicação de Padrões**: Imagine que está a desenvolver um sistema de exportação de dados para uma empresa. O sistema precisa de exportar os dados dos clientes em formato CSV, XML ou JSON, dependendo da escolha do utilizador. Qual o padrão de projeto comportamental (GoF) mais adequado para resolver este problema sem usar múltiplos if/else? Desenhe um pequeno diagrama de classes ou escreva pseudo-código demonstrando a solução.

3.5.3 Exercícios Reflexivos (Estudo de Caso)

- **Cenário:** Foi contratado como líder técnico para modernizar o software de faturação de uma clínica médica. O código atual foi escrito há 10 anos por um único programador que já saiu da empresa. Ao analisar o código, encontra uma classe

chamada SistemaClinica com 12.000 linhas de código. Esta classe faz a gestão de marcações, calcula os salários dos médicos, gera as faturas dos pacientes e envia emails de lembrete. Sempre que a equipa tenta alterar o formato da fatura, o sistema de envio de emails deixa de funcionar misteriosamente.

- **Questão para Reflexão:** Identifique pelo menos dois Code Smells presentes neste cenário. Que padrões de projeto e técnicas de refatoração utilizaria para resolver esta situação de forma progressiva e segura? Elabore um plano de ação (2-3 parágrafos) justificando as suas escolhas arquiteturais e metodológicas.