|  |
| --- |
| Análise e Projeto de Sistemas    **ANÁLISE E DESENVOLVIMENTO DE SISTEMAS**    RELATÓRIO DE AULAS PRÁTICAS    Nome: Igor Pereira da Silva      RA: 2537577    Polo de matrícula: São Paulo- Anchieta – Polo Próprio        Local da realização da Aula Prática: Unip- Anchieta        Ano da postagem: 2025  PROFESSOR DA AULA PRATICA: Fábio Assis |

|  |  |  |
| --- | --- | --- |
| **Instituto de Ciências Exatas e Tecnologia** | **Disciplina:** Análise e Projeto de Sistemas    **Título da Aula:** Introdução à Análise e Projeto de Sistemas | **Relatório 1** |

# 1. Resumo Teórico

(Escreva aqui um texto de 8 a 10 linhas abordando:)

* O que é o ciclo de vida do desenvolvimento de software.
* Diferença entre abordagens tradicional (cascata) e ágil (Scrum/XP).
* Exemplifique com situações reais (como desenvolvimento de aplicativos ou sistemas web).

O ciclo de vida do desenvolvimento de software é o conjunto de etapas que guiam a criação de um sistema, desde o levantamento de requisitos até sua manutenção. Na abordagem tradicional, como o modelo em cascata, as fases são sequenciais e rígidas: análise, projeto, implementação e testes só começam após a conclusão da etapa anterior, o que pode gerar dificuldades caso haja mudanças no meio do processo. Já nas metodologias ágeis, como Scrum e XP, o desenvolvimento é iterativo e incremental, permitindo entregas rápidas e contínuas melhorias de acordo com o feedback do cliente. Um exemplo prático é no desenvolvimento de um aplicativo de delivery: em cascata, o app só seria liberado após todo o sistema estar concluído, enquanto em Scrum versões parciais (como cadastro de usuários e pedidos) já seriam entregues e testadas em curtos ciclos, garantindo maior adaptação às necessidades do mercado.

# 2. Estudo de Caso

Sistema escolhido: (exemplo: sistema de entregas de comida online).

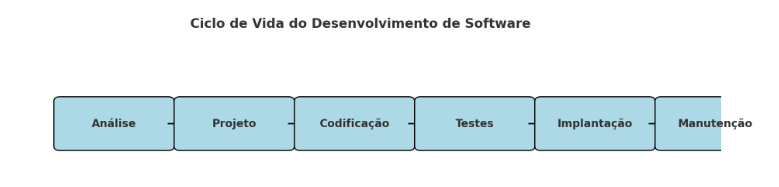
Descrição:

* Breve resumo do sistema.
* Quais fases do ciclo de vida ele percorreu (análise, projeto, codificação, testes, implantação, manutenção).
* Quais pontos se beneficiariam de metodologias ágeis.

O sistema escolhido é um **sistema de entregas de comida online**, que conecta restaurantes, entregadores e clientes por meio de um aplicativo ou site. Ele permite que o usuário visualize cardápios, faça pedidos, acompanhe o status da entrega em tempo real e efetue pagamentos digitais. Durante seu ciclo de vida, o sistema percorreu as fases de **análise**, onde foram levantados os requisitos de usabilidade e segurança; **projeto**, em que foram definidos layout, arquitetura e banco de dados; **codificação**, com o desenvolvimento do aplicativo e do painel web; **testes**, para verificar usabilidade e falhas; **implantação**, ao disponibilizar a plataforma ao público; e **manutenção**, garantindo atualizações e correções constantes. As metodologias ágeis seriam especialmente benéficas nas etapas de **codificação, testes e manutenção**, pois permitem entregar versões parciais (como login, carrinho e rastreamento) de forma incremental, além de ajustar rapidamente o sistema de acordo com feedback dos usuários e novas demandas do mercado.

# 3. Diagrama do Ciclo de Vida

(Inserir aqui a imagem do diagrama/fluxograma ou linha do tempo criada na ferramenta escolhida – Draw.io, Astah ou Lucidchart.)



# 4. Reflexões Finais

* Quais foram as maiores dificuldades em identificar o ciclo de vida?

As maiores dificuldades em identificar o ciclo de vida do sistema foram distinguir com clareza cada fase e compreender como elas se interligam, já que algumas atividades podem se sobrepor ou ocorrer de forma contínua, como testes e manutenção.

* Qual abordagem você considera mais eficiente (tradicional ou ágil) para o sistema analisado?

No caso do sistema de entregas de comida online, considero que a **abordagem ágil** é mais eficiente, pois o mercado exige adaptações rápidas, novas funcionalidades e correções constantes baseadas no feedback dos usuários.

* O que você aprendeu nesta prática?

Nesta prática, aprendi a importância de entender o ciclo de vida como um guia para estruturar o desenvolvimento, a diferença entre metodologias tradicionais e ágeis e como aplicar esses conceitos em situações reais, tornando o processo mais organizado e adaptado às necessidades do usuário.

# 5. Referências:

Livros e artigos básicos sobre ciclo de vida e metodologias de desenvolvimento:

PRESSMAN, Roger S. *Engenharia de Software*. 7. ed. São Paulo: McGraw-Hill, 2011.

SOMMERVILLE, Ian. *Engenharia de Software*. 9. ed. São Paulo: Pearson Education, 2011.

SCHWABER, Ken; BEEDLE, Mike. *Agile Software Development with Scrum*. Upper Saddle River, NJ: Prentice Hall, 2002.

BECK, Kent et al. *Extreme Programming Explained: Embrace Change*. 2. ed. Boston: Addison-Wesley, 2004.

Artigos e materiais online:

ALVES, Thais; GONÇALVES, Roberto. Ciclo de Vida de Software: Uma Visão Geral. Revista Científica, v. 4, n. 1, p. 45-56, 2017. Disponível em: https://www.examplejournal.com/ciclo-vida-software. Acesso em: 27 set. 2025.

|  |  |  |
| --- | --- | --- |
| **Instituto de Ciências Exatas e Tecnologia** | **Disciplina:** Análise e Projeto de Sistemas    **Título da Aula:** Levantamento de Requisitos | **Relatório 2** |

# 1. Resumo Teórico

(Escreva aqui um texto de 8 a 10 linhas abordando:)

* O conceito de levantamento de requisitos.
* Diferença entre requisitos funcionais e não funcionais.
* Principais técnicas de elicitação de requisitos (entrevistas, brainstorming, prototipação etc.).

O levantamento de requisitos é a etapa inicial do desenvolvimento de software que busca identificar e registrar as necessidades do cliente e dos usuários finais, servindo como base para a construção do sistema. Os **requisitos funcionais** descrevem o que o sistema deve fazer, como cadastro de usuários, processamento de pedidos ou geração de relatórios. Já os **requisitos não funcionais** tratam de características de qualidade, como desempenho, segurança, usabilidade e confiabilidade. Para elicitar requisitos, diferentes técnicas podem ser utilizadas: **entrevistas**, que permitem compreender em detalhes as necessidades do cliente; **brainstorming**, útil para gerar ideias em grupo; **prototipação**, que possibilita visualizar telas e fluxos antes da implementação; além de **questionários** e **observação direta**, que ajudam a coletar informações práticas. Um bom levantamento garante que o sistema atenda de forma eficaz às expectativas e objetivos estabelecidos

# 2. Lista de Requisitos

Liste aqui:

* Pelo menos 5 requisitos funcionais.
* Pelo menos 3 requisitos não funcionais.

Exemplo:

RF01 – O sistema deve permitir login de usuários.

RNF01 – O tempo de resposta deve ser inferior a 2 segundos.

**Requisitos Funcionais (RF):**

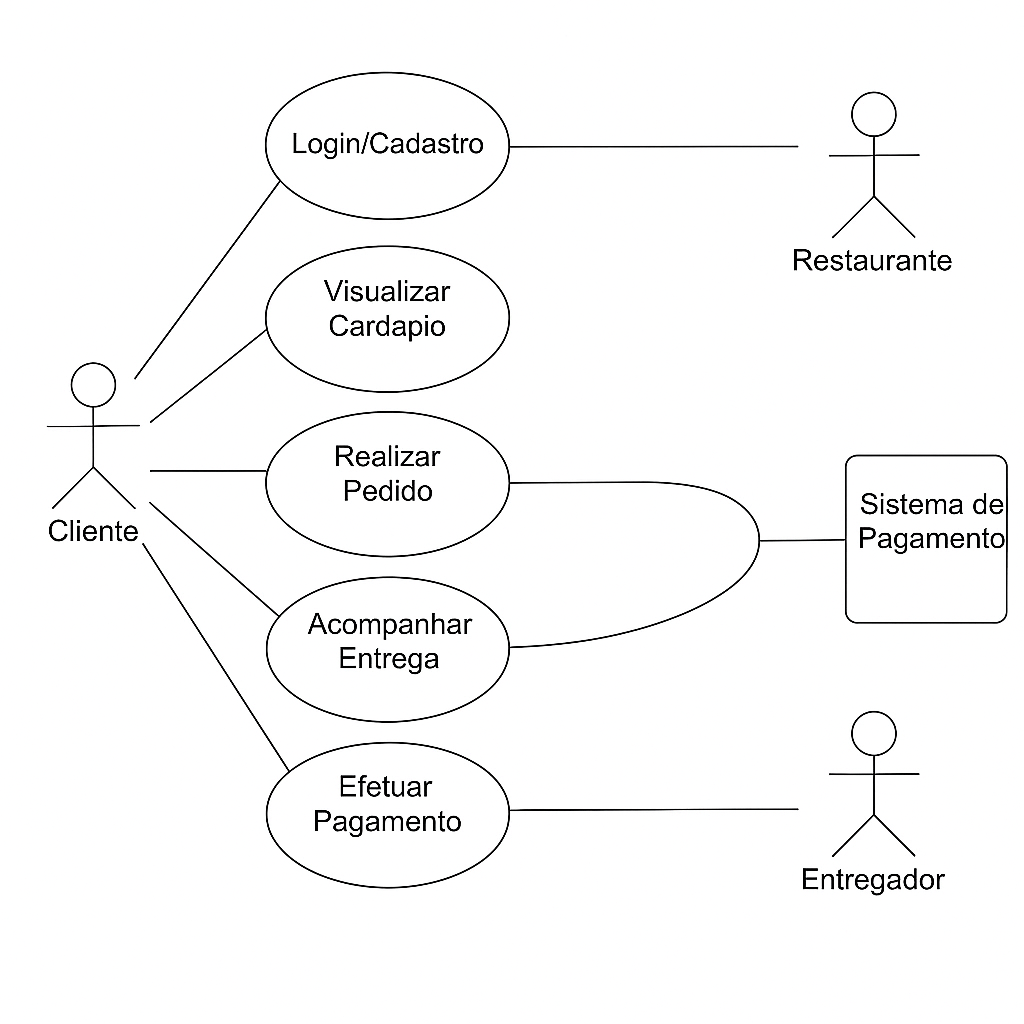
* **RF01** – O sistema deve permitir login de usuários com e-mail e senha.
* **RF02** – O sistema deve permitir o cadastro de restaurantes, clientes e entregadores.
* **RF03** – O usuário deve poder visualizar o cardápio de restaurantes e realizar pedidos.
* **RF04** – O sistema deve disponibilizar rastreamento em tempo real da entrega.
* **RF05** – O sistema deve permitir pagamentos online via cartão de crédito, débito e PIX.

**Requisitos Não Funcionais (RNF):**

* **RNF01** – O tempo de resposta do sistema deve ser inferior a 3 segundos por operação.
* **RNF02** – O sistema deve estar disponível 24 horas por dia, 7 dias por semana.
* **RNF03** – Os dados dos usuários devem ser armazenados de forma segura, seguindo padrões de criptografia.

# 3. Diagrama de Caso de Uso

# Criar diagrama de caso de uso



# 4. Reflexões Finais

* Quais foram as maiores dificuldades em levantar requisitos?

As maiores dificuldades em levantar requisitos foram compreender exatamente as necessidades dos diferentes atores do sistema (clientes, restaurantes e entregadores) e traduzi-las em funcionalidades claras e objetivas, evitando ambiguidades. Além disso, identificar requisitos não funcionais, como desempenho e segurança, exigiu uma análise mais detalhada.

* Qual técnica de licitação se mostrou mais eficaz?

A técnica de **prototipação** se mostrou a mais eficaz, pois permitiu que os usuários visualizassem telas e fluxos, dando feedback imediato sobre ajustes necessários.

Também as entrevistas ajudaram a captar expectativas de forma mais direta.

* O que você aprendeu nesta prática?

Nesta prática, aprendi que o levantamento de requisitos é essencial para alinhar expectativas entre equipe técnica e usuários, reduzindo riscos de retrabalho e garantindo que o sistema realmente atenda às demandas do negócio e dos usuários finais.

# 5. Referências:

PRESSMAN, Roger S.; MAXIM, Bruce R. *Engenharia de Software: uma abordagem profissional*. 9. ed. Porto Alegre: AMGH, 2021.

SOMMERVILLE, Ian. *Engenharia de Software*. 10. ed. São Paulo: Pearson, 2019.

BEZERRA, Eduardo. *Princípios de Análise e Projeto de Sistemas com UML*. 3. ed. Rio de Janeiro: Elsevier, 2016.

IEEE. *IEEE Std 830-1998 – IEEE Recommended Practice for Software Requirements Specifications*. New York: IEEE Computer Society, 1998.

PÁDUA, Wilson de Pádua Paula Filho. *Engenharia de Software: Fundamentos, Métodos e Padrões*. 3. ed. Rio de Janeiro: LTC, 2012.

|  |  |  |
| --- | --- | --- |
| **Instituto de Ciências Exatas e Tecnologia** | **Disciplina:** Análise e Projeto de Sistemas    **Título da Aula:** Modelagem de Sistemas com UML | **Relatório 3** |

# 1. Resumo Teórico

(Escreva aqui um resumo de 8 a 10 linhas abordando:)

* O que é UML e sua importância.
* Diferença entre diagramas estruturais e comportamentais.
* Aplicações práticas de diagramas de caso de uso, classes e sequência.

A UML (Unified Modeling Language) é uma linguagem padrão de modelagem utilizada na engenharia de software para representar, visualizar e documentar sistemas de forma clara e padronizada. Sua importância está em facilitar a comunicação entre desenvolvedores, analistas e clientes, garantindo uma compreensão comum sobre o sistema. Os diagramas UML podem ser divididos em **estruturais**, que mostram a organização estática dos elementos (como diagramas de classes, objetos e componentes), e **comportamentais**, que representam a dinâmica do sistema e suas interações ao longo do tempo (como casos de uso, sequência e atividades). O **diagrama de caso de uso** é aplicado para identificar e descrever as funcionalidades do sistema do ponto de vista do usuário. O **diagrama de classes** detalha a estrutura do sistema, mostrando atributos, métodos e relações entre classes. Já o **diagrama de sequência** é usado para representar a ordem e o fluxo das mensagens trocadas entre objetos em um processo. Assim, a UML contribui diretamente para a qualidade e clareza no desenvolvimento de sistemas

# 2. Sistema Modelado

Descreva brevemente o sistema fictício escolhido, suas principais funcionalidades e atores.

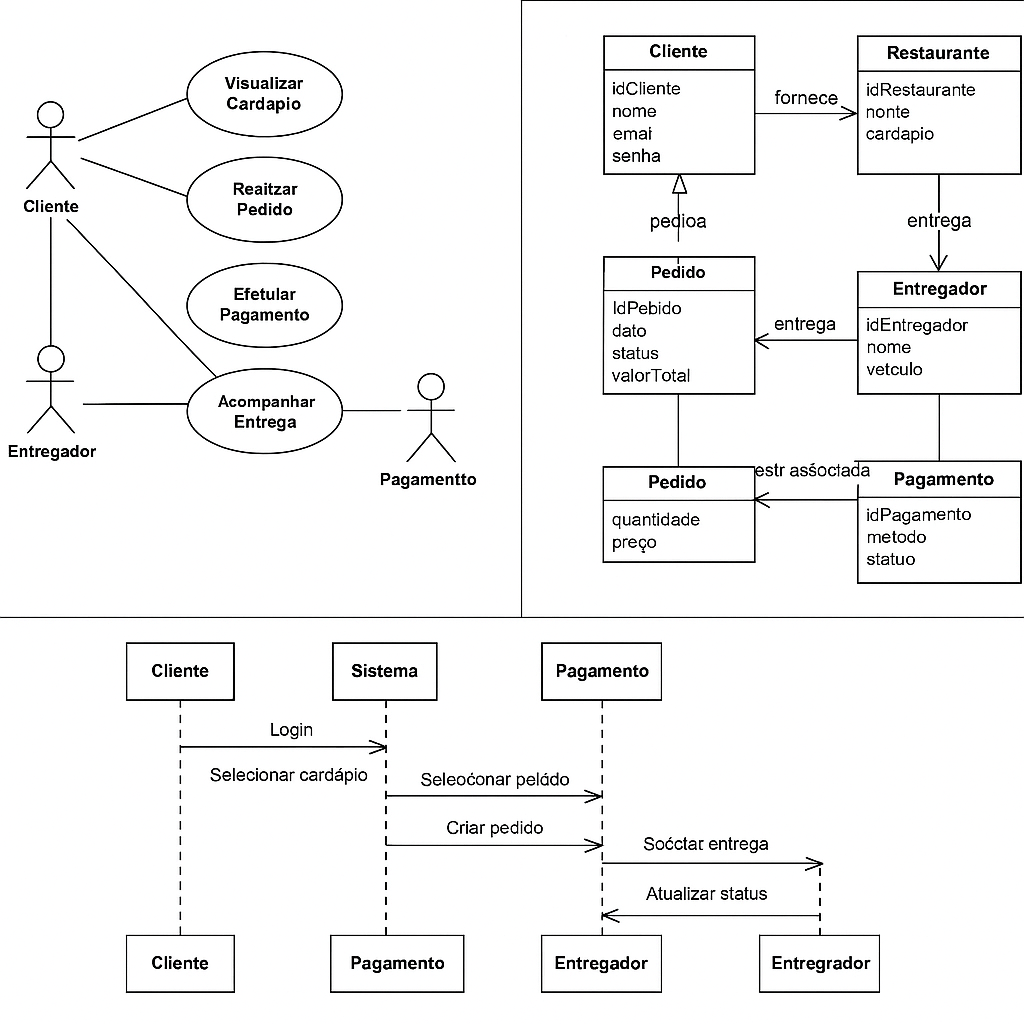
O sistema fictício escolhido é uma **plataforma de delivery online**, cujo objetivo é intermediar pedidos entre clientes, restaurantes e entregadores. Suas principais funcionalidades incluem: cadastro e autenticação de usuários, visualização de cardápios, realização de pedidos, processamento de pagamentos online e rastreamento em tempo real das entregas. O sistema também possibilita que restaurantes cadastrem seus produtos e gerenciem pedidos recebidos, enquanto os entregadores podem atualizar o status da entrega durante o processo.

Os principais **atores** identificados são:

* **Cliente:** acessa cardápio, realiza pedidos, efetua pagamentos e acompanha entregas.
* **Restaurante**: cadastra cardápio e administra pedidos.
* **Entregador:** recebe solicitações de entrega e atualiza o status do pedido.
* **Sistema de Pagamento**: ator externo responsável pelo processamento financeiro.

**Diagramas Criados**

(Inserir os diagramas criados: caso de uso, classes e sequência.)

****

# 4. Reflexões Finais

* Quais foram as dificuldades na criação dos diagramas?

As principais dificuldades na criação dos diagramas foram representar de forma clara as relações entre os atores e os casos de uso, além de organizar corretamente as classes e seus relacionamentos no diagrama estrutural. No diagrama de sequência, também foi desafiador definir a ordem das interações sem perder a clareza do fluxo.

* Como os diagramas ajudam no entendimento do sistema?

Os diagramas ajudam no entendimento do sistema ao fornecer uma visão visual e padronizada das funcionalidades, da estrutura e do comportamento, facilitando a comunicação entre desenvolvedores, analistas e clientes. Isso garante que todos compreendam melhor como o sistema deve funcionar antes da implementação.

* O que você aprendeu nesta prática?

Nesta prática, aprendi que a modelagem com UML é essencial para documentar e validar requisitos, além de reduzir ambiguidades no desenvolvimento. Também percebi que cada tipo de diagrama tem um papel específico e complementar, sendo fundamental para o planejamento e a qualidade do sistema de software.

# 5. Referências

PRESSMAN, Roger S.; MAXIM, Bruce R. *Engenharia de Software: uma abordagem profissional*. 9. ed. Porto Alegre: AMGH, 2021.

SOMMERVILLE, Ian. *Engenharia de Software*. 10. ed. São Paulo: Pearson, 2019.

BEZERRA, Eduardo. *Princípios de Análise e Projeto de Sistemas com UML*. 3. ed. Rio de Janeiro: Elsevier, 2016.

PÁDUA, Wilson de Pádua Paula Filho. *Engenharia de Software: Fundamentos, Métodos e Padrões*. 3. ed. Rio de Janeiro: LTC, 2012.

IEEE. *IEEE Std 830-1998 – IEEE Recommended Practice for Software Requirements Specifications*. New York: IEEE Computer Society, 1998.

|  |  |  |
| --- | --- | --- |
| **Instituto de Ciências Exatas e Tecnologia** | **Disciplina:** Análise e Projeto de Sistemas    **Título da Aula:** Design de Software | **Relatório 4** |

# 1. Resumo Teórico

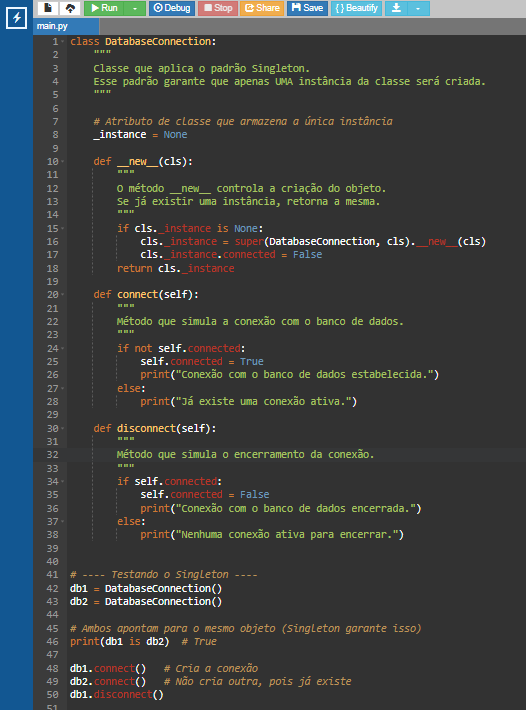
(Escreva aqui um resumo de 8 a 10 linhas abordando:)

* O que são padrões de design e sua importância.
* Exemplos de padrões (Factory, Singleton, Observer etc.).
* Como a arquitetura multicamadas se relaciona ao design de software.

Os **padrões de design** são soluções reutilizáveis para problemas recorrentes no desenvolvimento de software, que auxiliam a criar sistemas mais organizados, flexíveis e de fácil manutenção. Sua importância está em fornecer boas práticas consolidadas, reduzindo erros e aumentando a produtividade das equipes. Entre os exemplos mais conhecidos estão o **Factory Method**, que centraliza a criação de objetos sem expor a lógica de instância; o **Singleton**, que garante a existência de apenas uma instância de uma classe em todo o sistema; e o **Observer**, que permite a comunicação reativa entre objetos, útil em sistemas com notificações e eventos. Já a **arquitetura multicamadas** relaciona-se ao design de software por dividir a aplicação em camadas independentes, como apresentação, negócio e persistência, facilitando a aplicação de padrões de design e promovendo modularidade, reutilização de código e maior escalabilidade no desenvolvimento de sistemas.

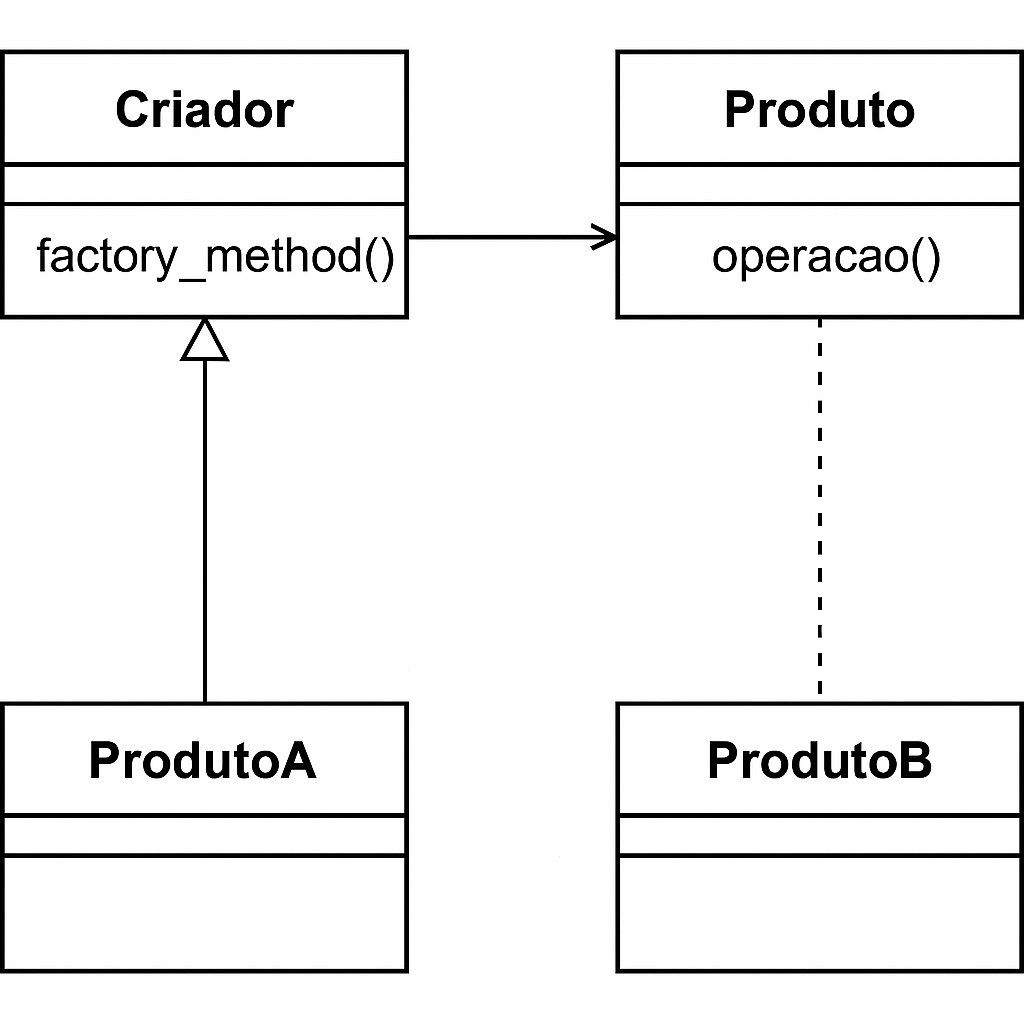
# 2. Código Desenvolvido

(Cole aqui o código desenvolvido em Python ou C, comentando as principais partes que aplicam o padrão de projeto escolhido.)



# Diagrama UML

(Inserir aqui o diagrama UML criado para representar a implementação do padrão de design.)



# 4. Reflexões Finais

* Quais foram as dificuldades na implementação do padrão?

Durante a implementação do padrão de design, a principal dificuldade foi compreender como aplicá-lo corretamente ao contexto do problema. Alguns desafios específicos incluíram identificar onde criar as abstrações necessárias, evitar dependências excessivas entre classes e garantir que o padrão realmente resolvesse o problema sem complicar o código. Além disso, adaptar conceitos teóricos para uma solução prática exigiu atenção aos detalhes e testes constantes.

* Como os padrões de design ajudam na manutenção do software?

Os padrões de design facilitam a manutenção do software porque fornecem soluções testadas e reconhecidas para problemas recorrentes, promovendo código mais organizado, legível e modular. Eles ajudam a reduzir a duplicação de código, facilitam a extensão de funcionalidades e tornam mais simples a identificação e correção de erros. Com padrões bem aplicados, novos desenvolvedores podem compreender o sistema mais rapidamente, garantindo consistência e qualidade ao longo do tempo.

 O que você aprendeu nesta prática?

Nesta prática, aprendi a importância de pensar no design antes de codificar, identificando cenários em que a aplicação de padrões de design traz benefícios claros. Também compreendi melhor como cada padrão resolve problemas específicos e como escolher o padrão adequado dependendo do contexto. Além disso, a experiência prática reforçou a importância de escrever código limpo e modular, considerando sempre a manutenção futura do sistema.

# 5. Referências

GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. Design Patterns: Elements of Reusable Object-Oriented Software. Boston: Addison-Wesley, 1995.

FREEMAN, Eric; FREEMAN, Elisabeth. Head First Design Patterns. 2. ed. O’Reilly Media, 2020.

SILVA, João. Padrões de Projeto em Java: Guia Prático. São Paulo: Novatec, 2018.

W3Schools. Python Programming Tutorials. Disponível em: https://www.w3schools.com/python/. Acesso em: 27 set. 2025.

|  |  |  |
| --- | --- | --- |
| **Instituto de Ciências Exatas e Tecnologia** | **Disciplina:** Análise e Projeto de Sistemas    **Título da Aula:** Arquitetura de Software | **Relatório 5** |

# 1. Resumo Teórico

(Escreva aqui um resumo de 8 a 10 linhas abordando:)

* Conceito de arquitetura de software.
* Principais modelos arquiteturais (Monolítico, SOA, Microsserviços).

Relação entre arquitetura, escalabilidade e manutenção.

A arquitetura de software representa a estrutura fundamental de um sistema, definindo como seus componentes interagem, se comunicam e se integram para atender aos requisitos funcionais e não funcionais. Ela fornece uma visão de alto nível que orienta as decisões de design e implementação.

Os principais modelos arquiteturais incluem o monolítico, em que todo o sistema é implementado como um único bloco de código; o SOA (Service-Oriented Architecture), que organiza funcionalidades em serviços independentes e interoperáveis; e a arquitetura de microsserviços, que fragmenta o sistema em pequenos serviços autônomos e escaláveis.

A arquitetura influencia diretamente a escalabilidade, a manutenção e a evolução do sistema. Arquiteturas bem definidas reduzem a complexidade, facilitam a adição de novas funcionalidades e tornam o sistema mais resiliente a falhas. Dessa forma, a escolha do modelo arquitetural deve considerar o porte do sistema, a demanda de usuários e o ciclo de vida do produto.

# 2. Sistema Modelado

Descreva brevemente o sistema fictício escolhido e a arquitetura proposta.

O sistema fictício modelado é o “SmartDelivery”, uma plataforma de entregas inteligentes que conecta restaurantes, entregadores e clientes por meio de um aplicativo web e mobile.

A arquitetura proposta é baseada em microsserviços, pois permite o desenvolvimento independente dos módulos de usuários, pedidos, pagamentos e logística. Cada serviço possui seu próprio banco de dados e se comunica por meio de APIs REST, garantindo isolamento de falhas e escalabilidade horizontal.

O sistema conta com um Gateway API, responsável pelo roteamento das requisições, um serviço de autenticação, e um módulo de monitoramento que coleta métricas de desempenho. Essa abordagem facilita o deploy contínuo e reduz o tempo de manutenção.

**Diagramas Criados**

(Inserir aqui os diagramas de componentes e implantação criados.)

Diagrama de Componentes (visão lógica do sistema)

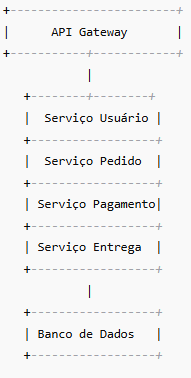
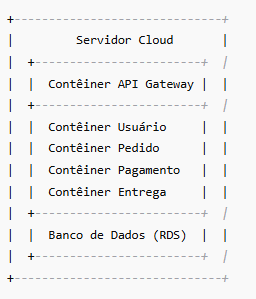


Diagrama de Implantação (visão física do sistema)



Esses diagramas mostram a **separação de responsabilidades** entre serviços e sua implantação em **contêineres independentes**, característica essencial de microsserviços.

# 4. Reflexões Finais

* Quais foram as dificuldades na modelagem da arquitetura?

Durante a modelagem da arquitetura, a principal dificuldade foi definir corretamente os limites entre os serviços e suas responsabilidades, evitando redundâncias e dependências circulares. Também foi desafiador estabelecer a comunicação eficiente entre os módulos sem comprometer o desempenho.

* Qual modelo arquitetural você considera mais adequado para este sistema?

O modelo de microsserviços foi considerado o mais adequado para o sistema “SmartDelivery”, pois oferece flexibilidade, resiliência a falhas e facilidade de escalonamento — fatores cruciais em aplicações distribuídas e com alto volume de requisições.

* O que você aprendeu nesta prática?

A prática permitiu compreender melhor como a arquitetura influencia diretamente a qualidade do software, tornando-o mais sustentável e preparado para o crescimento contínuo. Além disso, reforçou a importância do planejamento arquitetural antes do início do desenvolvimento.

# 5. Referências

PRESSMAN, Roger S.; MAXIM, Bruce R. Engenharia de Software: uma abordagem profissional. 9. ed. Porto Alegre: AMGH, 2021.

SOMMERVILLE, Ian. Engenharia de Software. 10. ed. São Paulo: Pearson Education, 2020.

ALURA. Arquitetura de Microsserviços em Python. Disponível em: https://www.alura.com.br/

. Acesso em: 18 out. 2025.

MICROSOFT DOCS. Architectural styles and patterns. Disponível em: https://learn.microsoft.com/

. Acesso em: 18 out. 2025.

|  |  |  |
| --- | --- | --- |
| **Instituto de Ciências Exatas e Tecnologia** | **Disciplina:** Análise e Projeto de Sistemas    **Título da Aula:** Ferramentas de Modelagem e Case | **Relatório 6** |

# 1. Resumo Teórico

(Escreva aqui um resumo de 8 a 10 linhas abordando:)

* O que são ferramentas CASE.
* Exemplos de ferramentas (Astah, Visual Paradigm, Draw.io).
* Benefícios do uso dessas ferramentas no desenvolvimento de sistemas.

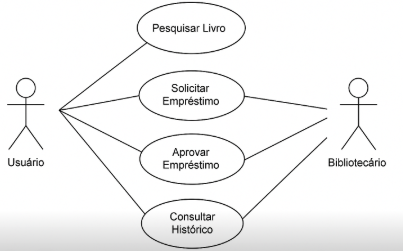
As ferramentas CASE (Computer-Aided Software Engineering) são aplicações desenvolvidas para auxiliar engenheiros e analistas de software nas etapas de modelagem, documentação, análise e projeto de sistemas. Elas automatizam e padronizam a criação de diagramas e artefatos utilizados na Engenharia de Software, como diagramas UML, fluxogramas e modelos de banco de dados.

Entre as principais ferramentas CASE destacam-se: Astah, Visual Paradigm e Draw.io. O Astah é amplamente utilizado em ambientes acadêmicos pela simplicidade de modelagem UML; o Visual Paradigm oferece integração com linguagens de programação e geração automática de código; e o Draw.io é uma alternativa online gratuita e colaborativa.

O uso de ferramentas CASE traz benefícios significativos, como a redução de erros de modelagem, melhor comunicação entre equipes, padronização da documentação e agilidade na manutenção de projetos. Além disso, proporcionam melhor rastreabilidade e consistência entre os modelos, contribuindo para o desenvolvimento de sistemas de maior qualidade e confiabilidade.

# 2. Diagramas Criados

(Inserir as capturas de tela dos diagramas de caso de uso, classes e sequência gerados na ferramenta.)



# Descrição do Processo

Descreva o processo de modelagem: quais recursos da ferramenta foram utilizados e quais foram as dificuldades.

Durante a prática, utilizei recursos da ferramenta como a criação de entidades, associação de relacionamentos e a geração automática de diagramas.

A modelagem iniciou pelo diagrama de casos de uso, identificando os atores e as funcionalidades principais. Em seguida, criei o diagrama de classes, definindo atributos e métodos. Por fim, foi gerado o diagrama de sequência para detalhar a interação entre objetos.

Dificuldades encontradas incluíram a organização visual dos diagramas mais complexos, que exigiu ajuste manual das posições das entidades para melhorar a leitura, e a escolha adequada de tipos de relacionamento entre classes.

# 4. Reflexões Finais

* Quais foram os maiores benefícios ao usar a ferramenta?

O uso da ferramenta trouxe clareza na modelagem do sistema, padronização dos diagramas e facilidade na comunicação entre membros da equipe.

* Como as ferramentas CASE auxiliam na documentação e no desenvolvimento?

Ferramentas CASE auxiliam na documentação completa do sistema, garantindo rastreabilidade e permitindo alterações mais rápidas sem comprometer a consistência do projeto.

 O que você aprendeu nesta prática?

Aprendi a utilizar diferentes tipos de diagramas UML, a organizar informações de forma estruturada e a compreender melhor a importância da padronização na Engenharia de Software.

# 5. Referências

Pressman, R. S. Engenharia de Software: Uma Abordagem Profissional. 8ª edição. McGraw-Hill, 2016.

Sommerville, I. Software Engineering. 10th edition. Pearson, 2015.

Astah. Disponível em: https://astah.net

Visual Paradigm. Disponível em: https://www.visual-paradigm.com

Draw.io. Disponível em: <https://app.diagrams.net>

|  |  |  |
| --- | --- | --- |
| **Instituto de Ciências Exatas e Tecnologia** | **Disciplina:** Análise e Projeto de Sistemas    **Título da Aula:** Avaliação e Validação de Projetos de Sistemas | **Relatório 7** |

# 1. Resumo Teórico

(Escreva aqui um resumo de 8 a 10 linhas abordando:)

* A importância da avaliação e validação em projetos de sistemas.
* Diferença entre verificação e validação.
* Exemplos de métodos de revisão colaborativa.

A avaliação e validação de projetos de sistemas são etapas fundamentais para garantir que o software atenda aos requisitos esperados e funcione corretamente. A verificação consiste em assegurar que o sistema está sendo construído de acordo com as especificações, enquanto a validação garante que o sistema atende às necessidades do usuário final.

Métodos de revisão colaborativa, como inspeções de código, revisões de pares (peer review) e walkthroughs, permitem identificar erros, inconsistências e lacunas antes da entrega final. Essas práticas promovem melhoria contínua, aumento da qualidade do software e maior alinhamento entre a equipe de desenvolvimento e os objetivos do projeto.

A aplicação consistente de avaliação e validação reduz retrabalho, evita falhas críticas e contribui para projetos mais confiáveis, robustos e bem documentados.

# 2. Checklist de Avaliação

Inclua o checklist utilizado para avaliar os diagramas ou documentos de outro grupo.

Exemplo de itens:

* Diagramas estão completos e coerentes?
* Requisitos estão claros e documentados?
* Há problemas de consistência ou duplicação?

 Diagramas estão completos e coerentes?

 Requisitos estão claros e documentados?

 Há problemas de consistência ou duplicação?

 Relacionamentos entre entidades e classes estão corretos?

 Nomes de elementos e atributos estão padronizados?

 Fluxos de processos e interações estão claros e compreensíveis?

 Todas as funcionalidades exigidas estão representadas nos diagramas?

 Há clareza na representação de atores e casos de uso?

 Possíveis erros de lógica ou omissões foram identificados?

 Documentação está organizada e de fácil compreensão?

# 3. Resultados da Avaliação

Descreva os problemas encontrados, as correções sugeridas e os pontos fortes do projeto avaliado.

Durante a avaliação do projeto do grupo, foram encontrados alguns problemas, como omissão de certos casos de uso, inconsistências na nomenclatura das classes e relações duplicadas entre entidades.

As correções sugeridas incluem padronização dos nomes, complementação dos diagramas faltantes e revisão dos relacionamentos para eliminar duplicidades.

Entre os pontos fortes do projeto estão a clareza na representação dos fluxos principais, boa organização visual dos diagramas e documentação detalhada dos requisitos funcionais.

# 4. Reflexões Finais

* Quais foram os principais aprendizados durante a avaliação?

A prática reforçou a importância de revisar cuidadosamente diagramas e documentação para garantir qualidade e consistência.

* Como o feedback em grupo contribui para a melhoria do projeto?

O feedback colaborativo ajuda a identificar erros que passam despercebidos individualmente e promove melhoria contínua do projeto.

* O que você aprendeu nesta prática?

Aprendi técnicas de avaliação e validação, a importância de checklists padronizados e como a colaboração entre membros melhora a qualidade do desenvolvimento de sistemas.

# 5. Referências

Pressman, R. S. Engenharia de Software: Uma Abordagem Profissional. 8ª edição. McGraw-Hill, 2016.

Sommerville, I. Software Engineering. 10th edition. Pearson, 2015.

IEEE. IEEE Standard for Software Reviews and Audits, 2017.

|  |  |  |
| --- | --- | --- |
| **Instituto de Ciências Exatas e Tecnologia** | **Disciplina:** Análise e Projeto de Sistemas    **Título da Aula:** Manutenção e Evolução de Sistemas | **Relatório 8** |

# 1. Resumo Teórico

(Escreva aqui um resumo de 8 a 10 linhas abordando:)

* Tipos de manutenção de software (corretiva, adaptativa, perfectiva e preventiva).
* Importância da refatoração e da reengenharia.
* Exemplos práticos de evolução de sistemas.

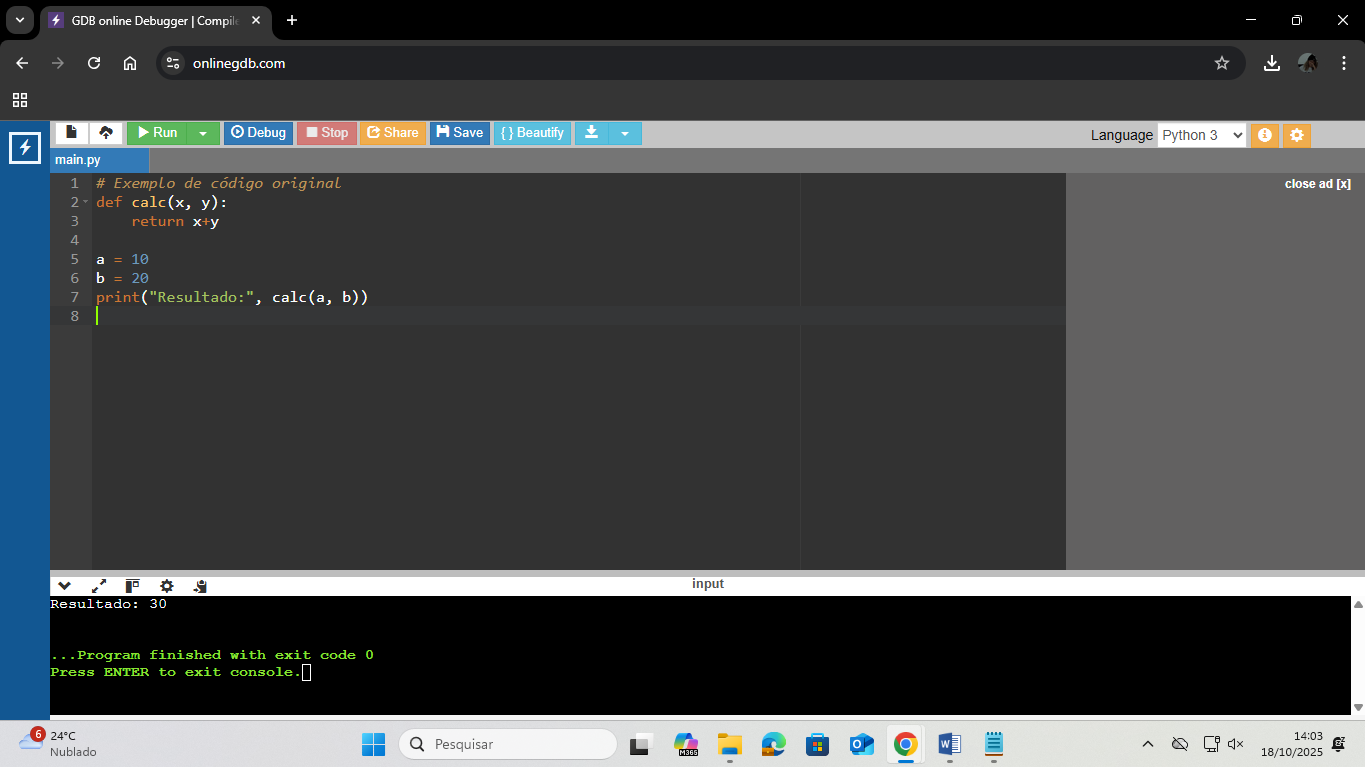
A manutenção de software é essencial para garantir a continuidade, eficiência e qualidade dos sistemas ao longo do tempo. Existem quatro tipos principais: corretiva, que corrige erros identificados; adaptativa, que ajusta o sistema a novas condições ou ambientes; perfectiva, que melhora funcionalidades existentes; e preventiva, que evita futuros problemas.

A refatoração consiste em reorganizar o código sem alterar seu comportamento, melhorando legibilidade e manutenção futura. Já a reengenharia envolve a revisão completa do sistema para modernização, simplificação ou migração tecnológica.

Exemplos práticos incluem a atualização de sistemas bancários para novos padrões de segurança, a melhoria de aplicativos móveis para suportar novos dispositivos e a correção de bugs em softwares corporativos críticos, garantindo maior confiabilidade e desempenho.

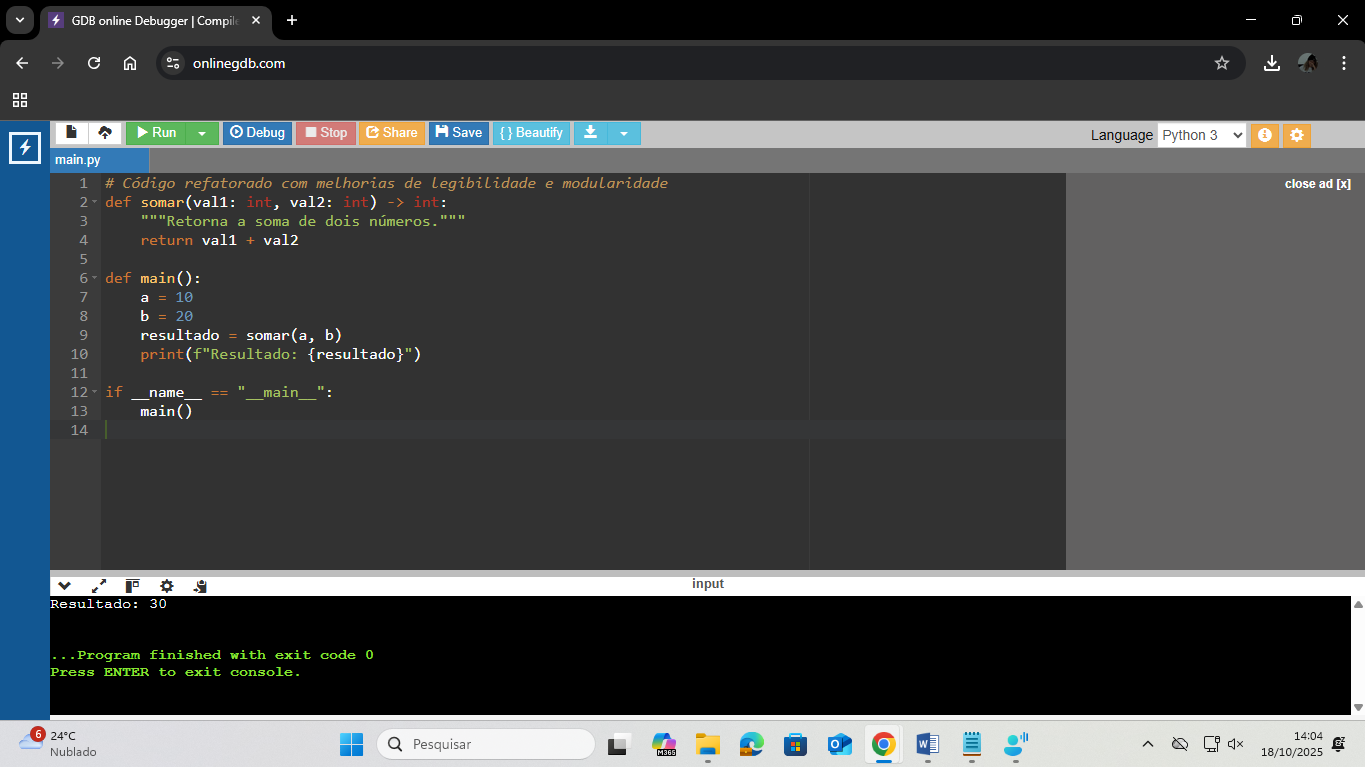
**2. Código Original**

(Cole aqui o código-fonte original antes da refatoração.)



# 3. Código Refatorado

(Cole aqui o código-fonte atualizado após a refatoração, destacando melhorias aplicadas.)



Melhorias aplicadas:

Função com nome descritivo (somar) e docstring explicativa.

Criação de função main() para organizar o fluxo do programa.

Uso de f-string para impressão mais clara e moderna.

Tipagem de variáveis e retorno para melhor entendimento e manutenção futura.

# 4. Reflexões Finais

* Quais foram as dificuldades ao realizar a manutenção?

Identificar trechos de código redundantes e decidir quais alterações não afetariam o comportamento do sistema

* Que melhorias foram alcançadas?

Código mais legível, modular e fácil de manter, com menor risco de introduzir erros futuros.

* O que você aprendeu nesta prática?

Compreendi a importância de boas práticas de programação, refatoração constante e manutenção preventiva para prolongar a vida útil do software.

# 5. Referências

Pressman, R. S. Engenharia de Software: Uma Abordagem Profissional. 8ª edição. McGraw-Hill, 2016.

Sommerville, I. Software Engineering. 10th edition. Pearson, 2015.

Fowler, M. Refactoring: Improving the Design of Existing Code. 2nd edition. Addison-Wesley, 2018.