André Hideki Gashu Nishimura
William Abe Fukushima

Ferramenta de modelagem de software para ensino de programação com Design Patterns

São Paulo, SP

2023

André Hideki Gashu Nishimura
William Abe Fukushima

Ferramenta de modelagem de software para ensino de programação com Design Patterns

Trabalho de conclusão de curso apresentado ao Departamento de Engenharia de Computação e Sistemas Digitais da Escola Politécnica da Universidade de São Paulo para obtenção do Título de Engenheiro.

Universidade de São Paulo – USP

Escola Politécnica

Departamento de Engenharia de Computação e Sistemas Digitais (PCS)

Orientador: Prof. Dr. Jorge Luis Risco Becerra

São Paulo, SP

2023

Catalogação-na-publicação

Nishimura, André
Ferramenta de modelagem de software para ensino de programação com Design Patterns / A. Nishimura, W. Fukushima -- São Paulo, 2023.
54 p.

Trabalho de Formatura - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1.GERAÇÃO DE CÓDIGO 2.PADRÕES DE SOFTWARE 3.DESENVOLVIMENTO DE SOFTWARE I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II.t. III.Fukushima, William

Este trabalho é dedicado às nossas queridas famílias e amigos,
cujo apoio incondicional e amor são fontes inestimáveis de força e inspiração.

Agradecimentos

Expressamos nossa gratidão a todos os profissionais da Escola Politécnica, em especial ao Departamento de Engenharia de Computação e Sistemas Digitais do curso de Engenharia de Computação da Universidade de São Paulo, por toda a experiência que nos repassaram ao longo do curso.

Resumo

Padrões de Design oferecem aos desenvolvedores soluções comuns para aprimorar a qualidade e organização do código. No entanto, dominar sua aplicação representa um desafio, exigindo um alto nível de abstração e prática consistente. Este projeto busca abordar essa questão ao criar uma ferramenta educacional de geração de código, acompanhada por diagramas UML. Seu objetivo é facilitar a transição da documentação arquitetural para o desenvolvimento de pequenos módulos de desenvolvimento web, além de fornecer exemplos de estudo instrutivos.

Palavras-chave: Padrões de Codificação. UML. Geração de Código. Code Scaffolding. Typescript. Web Development. Low-Code. MDA. Arquitetura Orientada a Modelos.

Abstract

Design Patterns offer developers common solutions to enhance code quality and organization. Yet, mastering their application poses a challenge, demanding a high level of abstraction and consistent practice. This project seeks to address this issue by creating an educational code generation tool, accompanied by UML diagrams. Its goal is to facilitate the transition from architectural documentation to development for small modules of web development, while also offering instructive study examples.

Keywords: Design Patterns. UML. Code Generation. Code Scaffolding. Typescript. Web Development. Low-Code. MDA. Model-Driven Architecture.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

API	Application Programming Interface
BPMN	Business Process Model Notation
CRUD	Create-Read-Update-Delete
HTTPS	Hyper Text Transfer Protocol Secure
IDE	Integrated Development Environment
IEEE	Instituto de Engenheiros Eletricistas e Eletrônicos
JSON	JavaScript Object Notation
MDA	Model Driven Architecture
MVC	Model View Controller
REST	Representational State Transfer
SysML	Systems Modeling Language
UML	Unified Modeling Language

Sumário

1 Introdução

1.1 Motivação

Em desenvolvimento de software é comum a utilização de padrões de projeto (Design Patterns) durante as fases de programação e refatoração. Eles são úteis principalmente por serem soluções consideradas satisfatórias para problemas muito conhecidos e que se repetem em aplicações de software. Como ressaltado no artigo de Rafal (2021), apesar de serem conceitos velhos, design patterns ainda são relevantes hoje em dia devido principalmente à constante presença de padrões em softwares. Um programador no mercado possivelmente não deve utilizar diretamente muitos desses padrões de solução, mas sim frameworks e bibliotecas disponíveis. Porém essas ferramentas, por sua vez, provavelmente utilizam um conjunto considerável de design patterns em seu núcleo. Dessa forma, além de fornecerem robustez e qualidade para a arquitetura do código, os design patterns servem também como um bom ponto de partida para estudos de programadores iniciantes.

No entanto, é amplamente considerado (HUANG; YANG, 2008; PILLAY, 2010) que design patterns são conceitos difíceis tanto de aprender quanto de ensinar. No quesito de aprendizagem, muitos desses padrões apresentam um grau de abstração muito elevado que acaba prejudicando os estudantes de programação a absorverem os conceitos facilmente. A principal dificuldade se dá em saber adequadamente qual padrão utilizar em um denominado contexto, necessitando adquirir uma compreensão muito boa sobre as diferenças de cada padrão. Enquanto no ensino, os métodos tradicionais se mostram insuficientes para propriamente ensinar todos os detalhes dos padrões e facilmente capacitar os estudantes. Os design patterns requerem, além de uma boa instrução teórica sobre as motivações e o funcionamento de cada um, de vários exemplos e exercícios práticos que variem em diversos graus de dificuldade a fim de evitar que o aluno se depare com uma longa distância associativa entre a teoria e a aplicação.

1.2 Objetivos

Tendo em vista o contexto apresentado, este trabalho tem como propósito criar uma ferramenta de modelagem de sistemas que utiliza diagramas UML e gera códigos template baseados em design patterns associados às classes propostas no diagrama. O escopo da ferramenta consiste em projetos educativos e de pequeno porte de desenvolvimento de sistemas web. E com isso, deseja-se averiguar a praticidade deste tipo de ferramenta no aprendizado de desenvolvimento de softwares (se é intuitiva e eficiente em ensinar os conceitos) e avaliar os benefícios da internalização do conhecimento passado.

1.3 Justificativa

1.3.1 Importância e necessidade da ferramenta

Como citado anteriormente pelos trabalhos de Huang e Yang (2008) e Pillay (2010), design patterns possuem uma parede de aprendizado alta para se superar, mesmo que sejam conceitos importantes no desenvolvimento de aplicações digitais, principalmente em projetos que utilizam linguagens orientadas a objetos. Desse modo, muitos programadores iniciantes aprendem diretamente a utilizar os frameworks e as bibliotecas, que abstraem as complexidades das implementações. Visto que pode-se desenvolver funcionalidades básicas apenas com o conhecimento de uso desses frameworks, tende-se a se focar no aprendizado deles sem se preocupar com a base conceitual que os compõem, a qual é formada, além de outros elementos, de design patterns (RAFAL, 2021). Posteriormente, ao adentrarem em projetos reais no mercado, estes desenvolvedores juniores programam seus códigos sem os fundamentos necessários para utilizarem adequadamente as ferramentas oferecidas por esses frameworks e bibliotecas, construindo assim arquiteturas de software que não se beneficiam de todo o potencial dos design patterns.

Com isso, pode-se atrelar uma correlação entre design patterns e arquiteturas robustas de software, já que os padrões de projeto proporcionam certas vantagens que aumentam a qualidade da arquitetura, em especial a manutenção e refatoração do código. Ao utilizar esses padrões, outros programadores familiarizados com os conceitos podem facilmente compreender o código e aplicar alterações, caso necessário. Além disso, os design patterns já passaram por diversos testes e são considerados soluções satisfatórias para seus problemas, provendo assim maiores opções de códigos refatorados. Portanto, é importante que haja formas mais estruturas e viáveis de ensino que possibilitem estudantes de programação a aprenderem melhor e realmente entenderem os conceitos dos padrões mais relevantes.

Como concluído na pesquisa de Pillay (2010), existem algumas abordagens de ensino mais compatíveis com as características desses conceitos, entre elas estão o uso de abordagem de aprendizado baseado em problema (Problem-Based Learning) e, consequentemente, a ilustração frequente da teoria em exemplos práticos. Portanto, a ferramenta de modelagem a ser desenvolvida visa abranger esta necessidade do exercício prático no processo de aprendizagem de design patterns. O usuário, ao modelar seu sistema pelos diagramas selecionando os padrões a serem usados, poderá gerar templates de códigos deixando à disposição exemplos semi-prontos. Dessa forma, têm-se uma ferramenta que possibilita praticidade e experimentação para o aluno aplicar os conceitos aprendidos na prática.

1.3.2 Relevância das plataformas Low Code no mercado

Um conceito emergente no mercado de ferramentas de modelagem é o Low Code, que consiste no desenvolvimento de softwares com pouca codificação. As plataformas de Low Code utilizam interfaces drag and drop com elementos visuais e representativos de componentes padrões de sistemas que podem ser associados para se construir um novo sistema. Porém, elas possuem a limitação de gerarem softwares para usos restritos sem muita personalização, ou seja, são muito boas para projetos mais comuns. Um requisito importante para essas plataformas é a abstração de conceitos avançados de programação, mantendo a interface e a usabilidade simples e de fácil manejo.

Segundo uma pesquisa de mercado da Fortune Business Insights (FBI, 2022), projetada para o período entre 2017-2028, ferramentas Low Code tem experienciado um grande crescimento de demanda, impulsionado pela demanda de transformação digital e tecnológica das indústrias. Soluções já utilizadas no mercado incluem aplicativos como Appian Multi-experience Development, Low-Code Pega Platform, Microsoft Power App, ServiceNow App Engine, Zoho Creator, Oracle APEX, Outsystems Low Code Platform, etc.

Na região Norte-Americana, 8 a cada 10 negócios aceleraram suas transformações digitais em 2020 e segundo o relatório de Low Code de 2021, 77% das empresas nos Estados Unidos comparadas de maneira global, estão implementando plataformas Low Code e 3 a cada 5 funcionários estão as utilizando para criar aplicativos. Dentro da América do Sul, o Brasil é o país que experiencia maior participação de mercado.

Pode-se, então, tomar algumas das ideias das plataformas Low Code como base para o desenvolvimento deste trabalho visto que é necessário que a ferramenta a ser desenvolvida seja utilizada para fins educativos. Ou seja, os fatores de simplicidade da interface e de boa usabilidade das plataformas Low Code são muito bem aplicáveis para o aspecto educativo do trabalho.

1.4 Organização do Trabalho

A seção 2 descreve conceitos teóricos de desenvolvimento de software utilizados e referenciados ao longo de nosso projeto.

A seção 3 explica a metodologia de desenvolvimento utilizada no projeto.

A seção 4 mostra o contexto da aplicação, incluindo os requisitos levantados.

A seção 5 descreve a estrutura da proposta de solução, explicando os seus elementos e sua operação.

A seção 6 apresenta as contribuições do projeto e possíveis melhorias futuras.

2 Aspectos Conceituais

2.1 Modeling languages

Dentro do tópico de engenharia de software e sistemas, há linguagens diagramáticas padronizadas para o levantamento e documentação da arquitetura do software, como por exemplo UML, BPMNs, SysML. Estas linguagens promovem diversos níveis de abstração e formas de descrever processos e sistemas. Ao entrarmos em níveis de abstração mais baixos, aproxima-se a linguagem de descrição da implementação técnica dos códigos.

2.2 Low Code/No Code frameworks

No mercado, atualmente algumas alternativas para geração de código por diagramas são por exemplo o Enterprise Architect, o Eclipse Modeling Framework, Microsoft's DSL tools e MetaEdit+ tool. Estas ferramentas providenciam interfaces de programação por meio de diagramas, sendo necessário pouco ou nenhum código escrito por parte do programador para o desenvolvimento.

2.3 Design Patterns

Segundo o artigo no site Refactoring.Guru (SHVETS), Design patterns são soluções típicas para problemas comuns na área de design e arquitetura de software. São classes e estruturas com funcionalidades genéricas e cabe ao programador reconhecer a ocorrência destas funcionalidades e implementá-las de maneira estruturada.

A maior parte dos Design Patterns são descritos de forma a facilitar a reprodução em diversos contextos. As descrições incluem a intenção do padrão que descreve brevemente o problema e a solução, a motivação que explica ainda mais o problema e sua solução que torna o padrão possível, a estrutura das classes que mostra cada parte do padrão e como elas estão relacionadas, o exemplo de código em uma das linguagens de programação populares, facilitando a compreensão da ideia por trás do padrão e exemplos de aplicações.

2.4 Model-Driven Architecture

De acordo com o site da organização de padrões de desenvolvimento OMG, a MDA Trata-se de técnicas de desenvolvimento onde são utilizados modelos de domínio, definindo as atividades que são realizadas em uma determinada aplicação, separando estas

da implementação técnica de algoritmos, permitindo que as lógicas de negócio evoluam interfaceando, porém separadamente à tecnologia.

2.5 Code Scaffolding

Esta técnica recebe o nome fazendo uma analogia com andaimes de construções civis. A idéia é gerar uma arquitetura inicial bem estruturada para que o programador possa trabalhar em cima. A dissertação de Magno (2015) apresenta como a técnica se popularizou por aumentar a produtividade de desenvolvedores no framework Ruby on Rails, que utiliza geradores de padrões MVC inicializados com telas para cada operação de bancos de dados (CRUD) de cada classe. Dentre algumas técnicas utilizadas na implementação de ferramentas de scaffolding, estão o uso de templates e scripts de geração de código que implementam metaprogramação.

3 Método do trabalho

3.1 Design Science

A base teórica do projeto é a metodologia da Ciência do Design. Esta abordagem é frequentemente adotada em disciplinas onde a ação prática e a criação de soluções concretas são centrais para o processo de pesquisa como projetos de engenharia, design e ciência da computação.

Segundo o livro (WIERINGA, 2014), é explicado que a ciência do design consiste no design e investigação de artefatos em contextos. É apresentado um framework para a aplicação da Ciência do Design, que consiste em uma forma de pensamento que guia a elaboração de artefatos para o projeto. Estes artefatos são as próprias soluções ou partes da solução que, ao interagir dentro de um contexto, são capazes de resolver algum problema ou promover melhorias neste contexto.

O framework citado se estrutura primordialmente em um contexto de problema, que molda todas as características do problema com as quais o artefato em estudo (solução) deve interagir. Dentro do contexto de problema estão os contextos social e de conhecimento. O primeiro contexto representa as necessidades dos stakeholders do problema, os quais proporcionam objetivos e restrições ao projeto; enquanto que o contexto de conhecimento abrange todos os conceitos e resultados já existentes sobre o problema que podem ser usados para resolvê-lo.

Dessa forma, a ciência do design possui dois principais tipos de processos que são feitos paralelamente sobre o contexto de problema. Um deles é o processo de design, que consiste na elaboração de artefatos para resolução do problema dentro do seu contexto. Este processo é orientado por conceitos denominados de “problemas de design” formadas por declarações que clamam por mudanças dentro do contexto, possuindo mais de uma possível solução. O segundo tipo é a investigação de “perguntas de conhecimento”, um conceito que visa responder questões sobre os artefatos e o contexto. Portanto, as respostas para essas perguntas servem de suporte teórico para a elaboração dos artefatos.

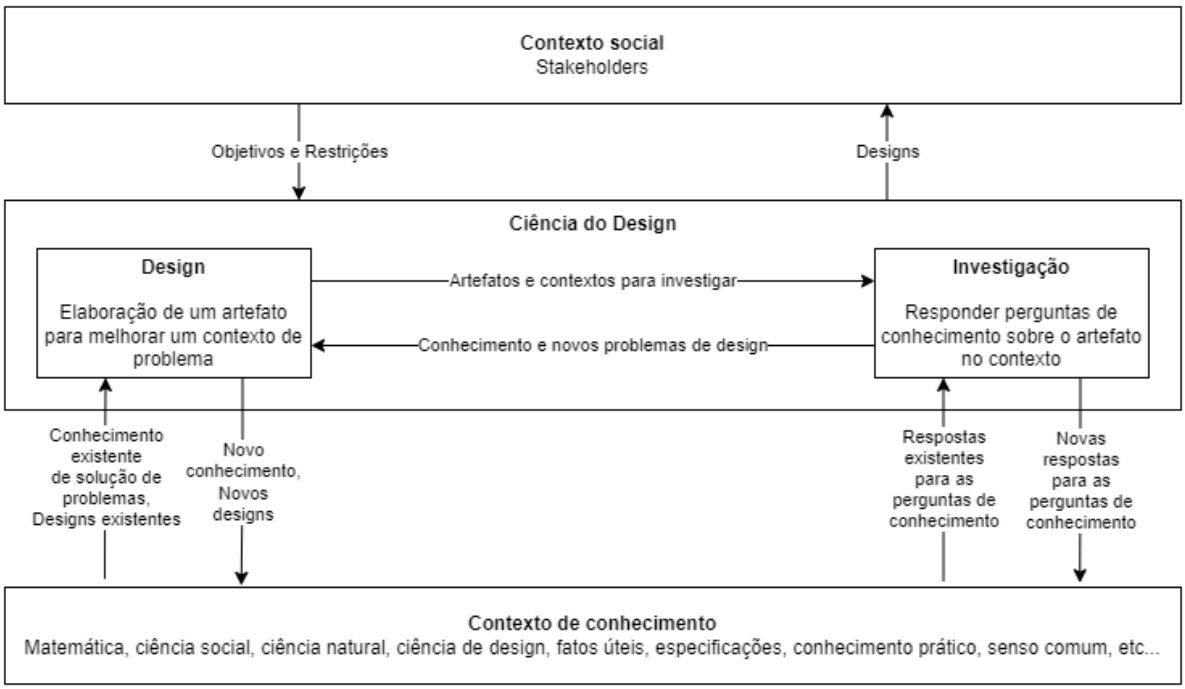
Resumidamente, a metodologia consiste na seguinte sequência de estudos:

- Problema de Design: Identificação clara de um problema ou oportunidade de design.
- Investigação de perguntas de conhecimento: Pesquisas sobre o problema de design que auxiliam o desenvolvimento de soluções.
- Criação de Artefatos: Desenvolvimento prático de soluções (artefatos) para resolver o

problema identificado. Esses artefatos podem assumir a forma de produtos, sistemas, algoritmos, entre outros.

- Avaliação e Refinamento: Avaliação rigorosa dos artefatos para garantir que atendam aos requisitos e solucionem efetivamente o problema. O processo inclui o refinamento contínuo dos artefatos com base nos resultados da avaliação.
- Contribuição para a Teoria: Enquanto os artefatos são desenvolvidos, a pesquisa busca compreender e explicar os princípios subjacentes envolvidos no design.

Figura 1 – Framework de Ciência do Design



Fonte: Wieringa (2014)

A figura 1 mostra o funcionamento do framework. Primeiramente, o contexto social delimita os objetivos que devem ser atingidos e as restrições a serem seguidas dentro do problema. A partir disso, os processos de design e investigação são executados para gerar possíveis soluções ao problema imposto. O processo de investigação obtém respostas através do contexto de conhecimento que são utilizadas pela elaboração de artefatos e, neste meio tempo, pode-se descobrir novos problemas de design que devem ser solucionados. Em retorno, ao desenvolver os artefatos, podem ser descobertos novas perguntas de conhecimento que devem ser respondidas e novos conceitos que são agregados tanto à investigação quanto ao contexto de conhecimento. Deste modo, a investigação do contexto social e do contexto de conhecimento juntamente à investigação das questões de natureza intrínseca de um artefato promovem a validação do processo de desenvolvimento do mesmo.

É importante destacar que a Ciência do Design adota um processo iterativo, no qual os artefatos são aprimorados continuamente com base em feedback constante e na identificação de novos insights ao longo do tempo. Além disso, vale ressaltar que essa abordagem se diferencia do modelo convencional de pesquisa, o qual frequentemente prioriza a geração de conhecimento teórico. Em vez disso, seu foco reside na criação de soluções tangíveis, com aplicabilidade prática em situações concretas específicas. Para alcançar esse objetivo, a metodologia direciona uma análise aprofundada da usabilidade do artefato, com especial ênfase na relevância prática e utilidade.

3.2 Modelo de desenvolvimento

Para a implementação do artefato deste trabalho, foi utilizado o modelo cascata de desenvolvimento de software com elementos da ciência de design apresentada anteriormente. Dessa forma, para cada fase de desenvolvimento, foram levantados problemas de design e perguntas de conhecimento que guiaram o pensamento e a concepção do projeto. Foram consideradas quatro principais etapas:

- Levantamento de requisitos;
- Projeto de arquitetura;
- Implementação;
- Teste e validação.

Na primeira etapa de levantamento de requisitos, deve-se identificar os stakeholders do sistema e suas necessidades dentro do contexto estabelecido, definindo os requisitos funcionais e não funcionais que as satisfazem adequadamente. Em seguida, para o projeto de arquitetura, é elaborada a estrutura geral do sistema, seja de dados, interface, algoritmos, etc. Assim como decide-se as tecnologias que serão utilizadas, tudo conforme os requisitos levantados anteriormente. Já a implementação representa a fase de construção do sistema, neste caso, a codificação da ferramenta. Por fim, têm-se os testes, que são previamente elaborados e executados na última etapa para verificar se o sistema atinge todos os seus objetivos como esperado.

4 Contexto da aplicação

4.1 Stakeholders

Como o objetivo do projeto é facilitar a aprendizagem de design patterns para estudantes de programação, qualquer indivíduo que deseja adquirir conhecimento em programação, em outras palavras um programador, é o principal interessado em utilizar a ferramenta. Considerou-se então que o usuário da aplicação terá o papel de um programador, desenvolvedor de códigos, que pode apresentar as seguintes necessidades dentro do contexto de problema:

- Necessidade de documentação da arquitetura de código;
- Aplicar boas práticas de programação em versões iniciais do código;
- Capacidade de modelar sua própria aplicação adequadamente;
- Certificar que o código está de acordo com a modelagem;
- Agilidade no desenvolvimento do código;
- Testar seu código.

Estas necessidades serviram como base para a elaboração dos processos de negócio e, posteriormente, para o levantamento de requisitos deste projeto.

4.2 Definição de escopo e processos de negócio

Este projeto busca facilitar o desenvolvimento de projetos de pequeno porte, possuindo teor mais educativo, que familiariza os programadores com o uso de design patterns. Inicialmente, o projeto fornece suporte a Javascript por ser a linguagem mais familiar ao desenvolvimento Web, podendo se estender a outras linguagens futuramente.

Deste modo, considerando as necessidades do stakeholder e o escopo delimitado, foram listados abaixo os processos de negócio que este projeto visa incluir:

- Criar uma nova modelagem para aplicação de software: o usuário pode criar um novo modelo em branco para construir sua modelagem e, através de uma interface gráfica, pode adicionar elementos visuais representativos de sua aplicação. Quando desejar ele deve ser capaz de salvar seu progresso de alguma forma;

- Editar modelagens existentes para aplicações de software: o usuário pode carregar uma modelagem que já tenha criado para alterá-la, possuindo as mesmas capacidades que o processo anterior;
- Gerar imagens da modelagem para documentação: o usuário pode exportar qualquer modelo que tenha elaborado como uma imagem, para ser utilizada em documentações;
- Gerar códigos representativos da modelagem: o usuário pode gerar estruturas de códigos que representem adequadamente os modelos que tenha elaborado.

4.3 Especificação de requisitos

Inicialmente, delineou-se uma lista preliminar de requisitos, fundamentada na visão e nos objetivos almejados. Para aprimorar a eficiência e a viabilidade do projeto, procedeu-se à hierarquização de cada requisito em termos de prioridade, antecipando a possibilidade de abandono de determinadas funcionalidades e características, conforme necessário, por meio de ajustes no escopo do projeto.

Seguindo o framework da ciência do design, foram investigadas as perguntas de conhecimento do problema estudando algumas aplicações e conceitos já previamente consolidados em ferramentas existentes.

Pesquisas foram feitas buscando por ferramentas no mercado que já fazem a geração de código em design pattern por diagramas e pelas palavras chave “Low-Code”, “UML”, “Code Generation”, “Modeling tools” e “Design Patterns” em artigos do IEEE Xplore.

Estas pesquisas possibilitaram verificar o estado de pesquisa atual dentro desta área de desenvolvimento e também listar problemas relacionados com as soluções já implementadas. A pesquisa também trouxe mais termos frequentemente utilizados como Model-Driven Architecture e Code Scaffolding que renderam resultados mais relevantes em relação às técnicas necessárias para a elaboração deste tipo de ferramenta. Os conceitos encontrados durante esta pesquisa encontram-se na seção 2.

Na sequência, prosseguiu-se para a fase subsequente da metodologia, centrada na investigação do contexto do problema. Esta etapa fundamentou-se em extensivas investigações acerca de arquiteturas típicas de aplicações análogas ao escopo de nosso projeto. Foram conduzidas pesquisas sobre conceitos associados a plataformas Low/No Code, especificamente aquelas que englobam funcionalidades de transpilação e geração de código, elementos essenciais ao projeto. Desse modo, utilizou-se como base um artigo de estudo de um framework arquitetural de plataformas Low Code para construir a arquitetura própria do projeto, que será detalhada em seções futuras.

Neste estágio, foram conduzidas discussões acerca do projeto, focalizando principalmente na delimitação de seu escopo. Algumas exigências foram excluídas do ciclo de

desenvolvimento, sendo consideradas como possíveis extensões e adições em etapas futuras. Com a arquitetura devidamente concebida e a especificação de requisitos devidamente revisada, tornou-se viável iniciar a implementação do projeto.

A tabela 1 a seguir apresenta os requisitos finais levantados para a aplicação.

Tabela 1 – Lista de requisitos funcionais e não funcionais

	Classificação	Requisito
R1	Funcional	Modelagem de diagramas de classe UML
R2	Funcional	Persistência de diagramas em arquivos locais do usuário
R3	Funcional	Exportação de diagramas em imagens
R4	Funcional	Geração de templates de códigos a partir dos diagramas
R5	Não funcional	Licenciamento open source
R6	Não funcional	Eficiência educativa: promover aprendizado sobre design patterns
R7	Não funcional	Usabilidade simples e fácil

O requisito R1 é funcional e diz respeito à capacidade do sistema atuar como uma ferramenta de modelagem. Assim ele deve possuir uma interface gráfica que possibilite a construção de diagramas visuais. Por se tratar de uma ferramenta para desenvolvimento de softwares, a linguagem de modelagem a ser tratada será a Unified Modeling Language (UML), uma linguagem específica para modelagem de sistemas, comumente usada para representar modelos de software. Como a UML abrange diversos tipos de diagramas, cada um com sua especificidade e nível de abstração sobre o software, optou-se por focar inicialmente em apenas um tipo de diagrama para este projeto.

Desse modo, foi determinado que diagramas de classe seriam o tipo mais adequado por apresentar elementos, como classes, atributos, métodos e relacionamentos, que representam minimamente a estrutura e o comportamento do software. Apesar de não ter detalhes sobre a funcionalidade de cada elemento das classes, é possível inferir a base estrutural do software e os componentes principais que o formam a partir das informações delas e os relacionamentos entre elas. Em outras palavras, este tipo de diagrama é de um nível de abstração baixo o suficiente que permite a construção das estruturas dos códigos que vão implementar os modelos.

Além disso, aplica-se neste requisito todas as funcionalidades padrões de uma ferramenta de modelagem, como por exemplo: interface "drag and drop", adição, remoção e edição de elementos visuais, movimentação desses elementos dentro do quadro, etc.

Quanto ao requisito funcional R2, ele indica que o sistema deve possuir alguma forma para que os modelos criados pelo usuário persistidos. Isto é necessário para que o progresso de modelagem não seja perdido e que exista a possibilidade de restaurar e alterar qualquer modelo previamente elaborado. A forma mais simples de persistência é o armazenamento local em arquivo na própria máquina do usuário. Com isso, se têm um

maior controle sobre como se armazena esses arquivos, seja mantendo-o localmente, seja subindo em um drive na nuvem.

Satisfazendo a necessidade de gerar documentações sobre o software, o requisito funcional R3 estabelece a funcionalidade de converter os diagramas produzidos em imagens, que por sua vez podem ser utilizadas para quaisquer tipos de documentação.

O requisito funcional R4 representa uma das principais funções deste projeto: a funcionalidade de gerar códigos a partir dos diagramas. Utilizando os dados contidos nos modelos e considerando a correspondência um para um entre uma classe UML e um arquivo de código, a ferramenta deve ter a capacidade de gerar trechos de códigos, que serviriam como templates das respectivas classes usadas na modelagem para o desenvolvedor posteriormente completar com o que desejar. Como citado anteriormente, os códigos devem ser escritos em Javascript, assim, as classes já devem ser pré-preenchidas com os devidos nomes, atributos e métodos, porém sem suas implementações. Cabe ao usuário completar o código com o comportamento desejado.

Em vista disso, o sistema consegue fornecer uma aproximação entre os conceitos teóricos de design patterns e sua aplicação prática. Os usuários terão em mãos arquivos de códigos com estruturas características dos design patterns que foram modelados nos diagramas, agilizando o processo de codificação e facilitando o aprendizado.

Outro ponto é que os códigos e arquivos gerados deverão seguir as normas e conceitos de arquitetura e código limpos para que o desenvolvedor produza os seus projetos sobre uma estrutura robusta e concisa. Deste modo, pode-se proporcionar aos usuários os benefícios de uma base de códigos organizada de forma quase automática desde o início da implementação.

O primeiro requisito não funcional listado R5 descreve que o sistema deve ser concebido utilizando tecnologias de código aberto. Isto é interessante para que seja de uso gratuito e expansível, de modo que qualquer desenvolvedor interessado consiga contribuir para o projeto. Assim, pode-se ter no futuro uma maior gama de suportes a Design Patterns de diversos tipos aplicações e não apenas de desenvolvimento Web ou até mesmo maior suporte a geração de outras linguagens de código.

A grande missão deste sistema em relação ao usuário é facilitar o seu aprendizado sobre os conceitos e usos de Design Patterns. Desta forma, o requisito não funcional R6 ressalta que o sistema deve ser eficiente nesta função didática, ou seja, garantir que o material apresentado é pertinente e correto e promover benefícios ao processo de estudo.

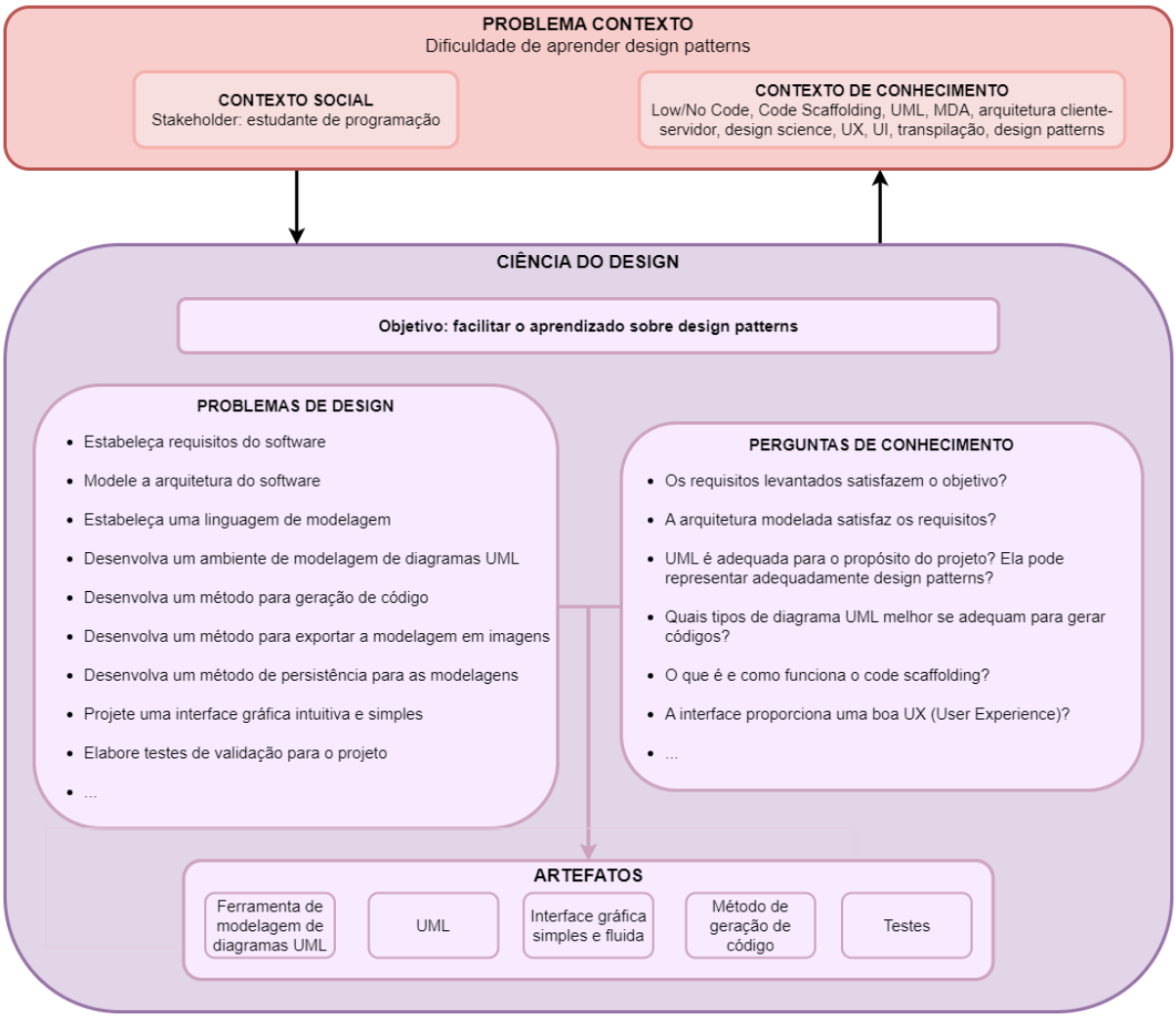
Em adição à isso, o último requisito não funcional R7 enuncia que a usabilidade do sistema deve ser simples e fácil. Interfaces com má usabilidade geram barreiras, com as quais o usuário pode se deparar, que aumentam o nível de esforço necessário para se utilizar a ferramenta e que, consequentemente, prejudicam o processo de estudo. Assim, um

trabalho minimamente satisfatório de design gráfico para a interface é necessária para que se minimize a ocorrência dessas barreiras de usabilidade, proporcionando uma experiência fluida ao usuário que impacte positivamente o seu aprendizado.

4.4 Modelo conceitual

Esta seção visa descrever os elementos conceituais da ciência de design que compõem o projeto desenvolvido. A figura 2 demonstra o modelo conceitual elaborado.

Figura 2 – Modelo conceitual da proposta sob os moldes da ciência do design



Fonte: autoral

Começando com a definição do problema contexto, de modo resumido este projeto se insere na questão de que aspirantes a programadores possuem dificuldades em absorver rapidamente as utilidades dos design patterns. A seção 1 explica adequadamente todo o contexto e a motivação. Nesse sentido, o problema contexto é formado pelo contexto social, que delimita os stakeholders como o estudante de programação, e o contexto de

conhecimento, composto por todos os conceitos teóricos e técnicos utilizados na concepção do projeto.

Como explicado anteriormente, para cada fase do desenvolvimento, foram levantados diversos problemas de design e perguntas de conhecimento. Alguns problemas de design estão descritos na figura e incluem: “estabeleça requisitos de software”, “estabeleça uma linguagem de modelagem” e “desenvolva um método de geração de código”. O mesmo foi feito para algumas das perguntas de conhecimento: “os requisitos levantados satisfazem o objetivo?”; “o que é e como funciona o code scaffolding?”; etc.

Assim, ambos os contextos, social e de conhecimento, são consumidos para solucionar os problema de design e responder às perguntas de conhecimento, gerando como resultado os artefatos desta aplicação. Eles interagem com o problema contexto a fim de atingir o objetivo definido. Alguns desses artefatos estão listados: ferramenta de modelagem de diagramas UML, linguagem UML, métodos de geração de código e entre outros.

5 Desenvolvimento do Trabalho

5.1 Processo de desenvolvimento da proposta

5.2 Tecnologias Utilizadas

Para a programação do código, foi utilizado o Visual Studio Code (VS Code), que é um ambiente de desenvolvimento integrado (IDE) leve e de código-fonte aberto desenvolvido pela Microsoft. Ele é projetado para ser uma ferramenta poderosa e flexível para desenvolvedores que trabalham em uma variedade de linguagens de programação. Algumas características-chave incluem: extensibilidade, suporte a diversas linguagens, integração com Git e entre outros que são extremamente úteis para o processo de programação.

Quanto à linguagem, foi utilizado o JavaScript juntamente com o NodeJS, um ambiente de execução de código JavaScript do lado do servidor, construído sobre o motor V8 da Google Chrome. Ele permite que os desenvolvedores usem JavaScript para escrever scripts do lado do servidor, o que tradicionalmente era realizado principalmente no lado do cliente, no navegador. Uma vantagem do NodeJS é a grande variedade de bibliotecas e pacotes de serviços em código aberto disponível para uso por desenvolvedores Web.

Foram utilizados também o modelo de API REST utilizando NodeJS para o back-end e a ferramenta Apollon (KRUSCHE et al., 2020) que utiliza o framework ReactJS.

5.3 Solução

5.3.1 Arquitetura

Baseando-se no framework apresentado pelo artigo Cruz et al. (2021), foi utilizado o modelo cliente-servidor para a arquitetura da aplicação. A arquitetura cliente-servidor para este gerador de código segue o princípio fundamental de dividir a aplicação em dois componentes principais: o cliente e o servidor. Cada componente tem responsabilidades distintas e interage entre si para fornecer funcionalidades específicas.

Foram realizadas pesquisas por opções de codificação de diagramas UML no formato JSON para a implementação da funcionalidade de salvar e importar diagramas (persistência de arquivo) assim como a fácil transmissão dos dados formatados para o back-end. Esta funcionalidade será detalhada na seção 5.3.3.

O módulo de cliente é representado pelo Front-End, que caracteriza primordialmente a interface com a qual o usuário interage. Consiste em dois sub-módulos: o editor de

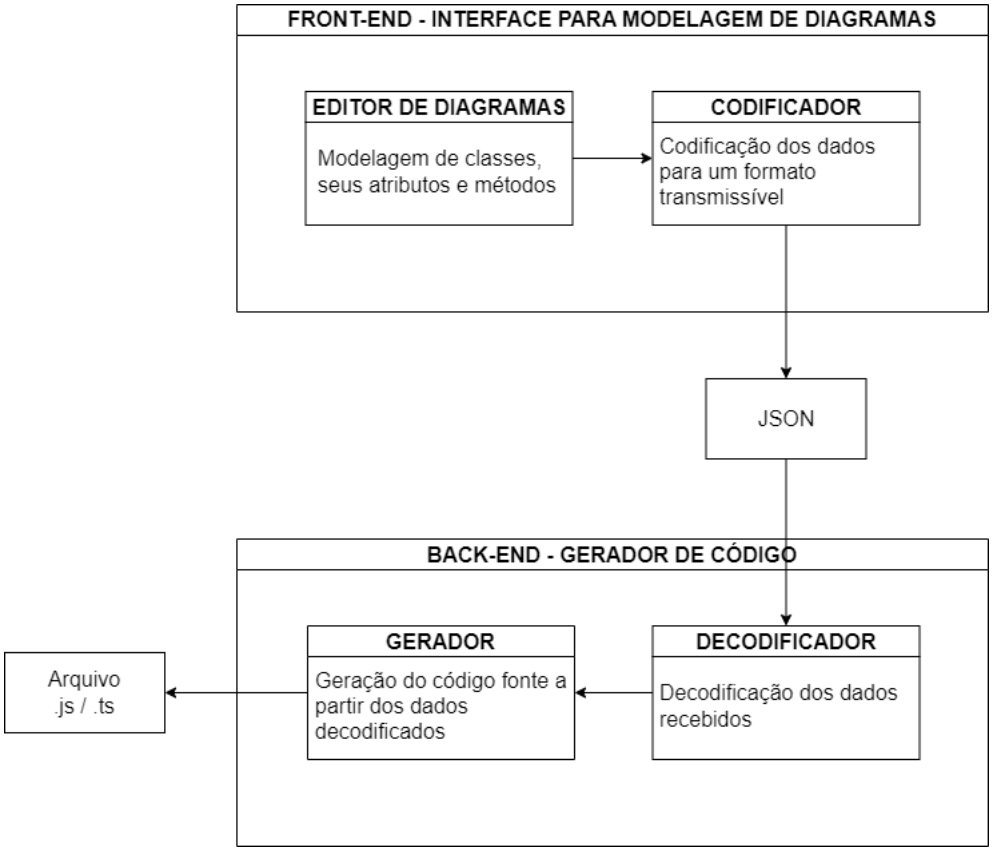


Figura 3 – Arquitetura cliente-servidor da aplicação

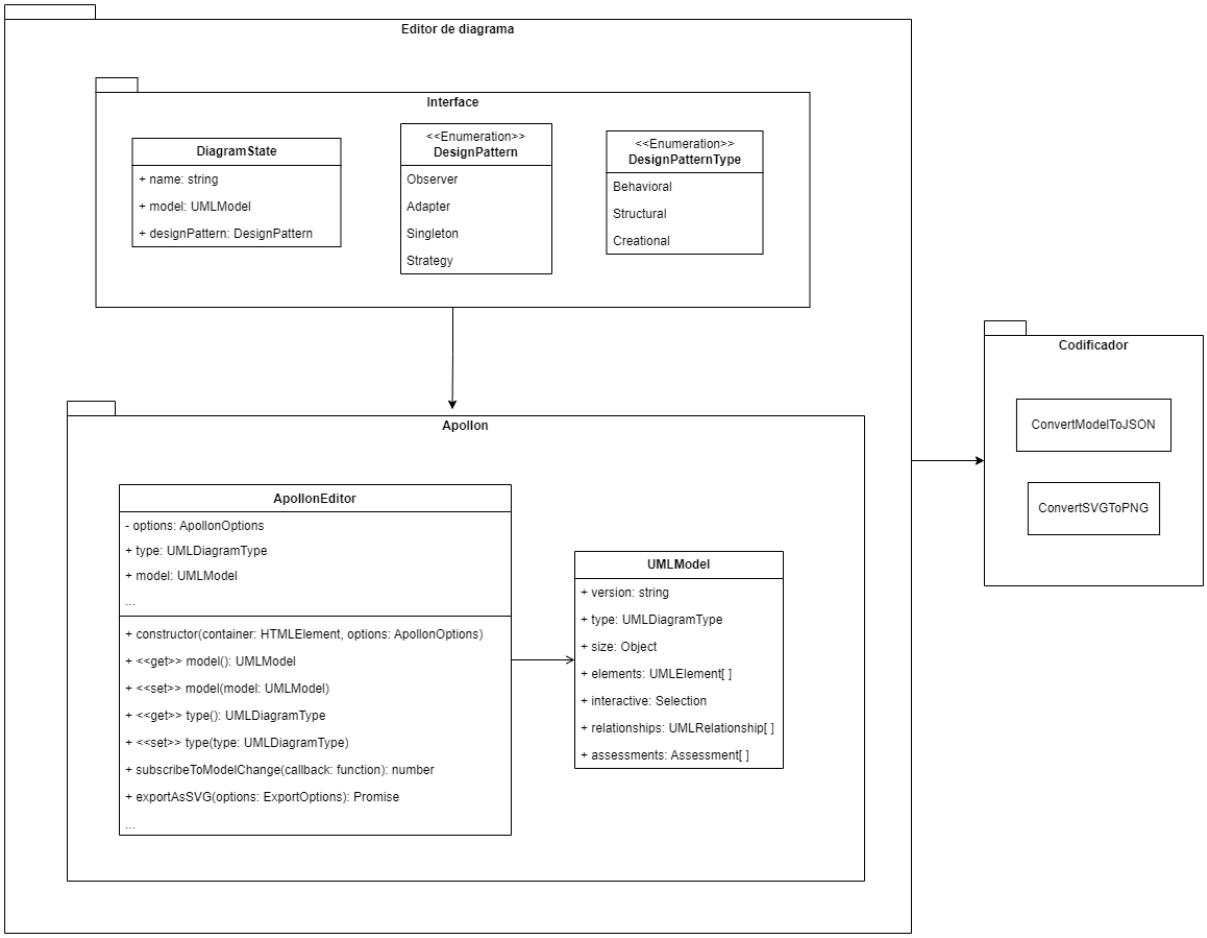
diagramas e o codificador. O editor de diagramas executa as principais funcionalidades ligadas à modelagem do diagrama UML, incluindo manipulação dos elementos visuais, definição das classes, atributos e métodos, gerenciamento de arquivos, etc. Em relação ao codificador, será responsável por manipular as informações do diagrama em um formato que seja transmissível pela rede, no caso, no formato JSON.

O módulo do servidor, dado pelo Back-End, por sua vez, é composto pelos sub-módulos do decodificador e do gerador. O decodificador, no caso, recebe os dados do diagrama em JSON transmitido, e converte em um formato de símbolos que seja reconhecido pelo gerador de código. Enquanto isso, o gerador lê a informação passada e gera os arquivos de código fonte correspondentes em javascript.

5.3.2 Cliente (front-end)

O módulo do cliente foi desenvolvido em ReactJS, portanto não é viável apresentar a sua hierarquia de classes completa. Porém, a figura 4 apresenta a estrutura simplificada dos principais módulos internos que compõem o frontend. Cada um será explicado a seguir.

Figura 4 – Classes e arquitetura simplificada do Frontend da aplicação



Fonte: autoral

5.3.2.1 Apollon

Como o núcleo do editor de diagramas, utilizou-se uma aplicação de código aberto denominada de Apollon (KRUSCHE et al., 2020), desenvolvido pela Universidade Técnica de Munique em ReactJS. O Apollon pode ser instalado através do gerenciador de pacotes do Node.js e deixa disponível uma simples API para ser utilizada dentro de outras aplicações. Ele fornece as principais funcionalidades de um editor de diagramas, permitindo que o usuário instancie e altere os elementos. Além disso, possui suporte a diversos tipos de diagramas UML, incluindo diagramas de classe, de atividade e de sequência.

Neste projeto, o Apollon foi configurado especificamente para modelagem de diagramas de classe, como definido pelo requisito R1. Desta maneira, o usuário é capaz de adicionar classes ao diagrama, definindo seu nome, seus atributos e métodos, assim como especificar o seu tipo entre uma classe normal, abstrata, de interface ou de enumeração. Tanto os atributos quanto os métodos podem ser definidos como privados, públicos e protegidos.

O usuário também pode definir os relacionamentos entre as classes adicionadas,

podendo ser caracterizados pelos tipos: associação unidirecional ou bidirecional, agregação, herança, composição, dependência e realização. As multiplicidades e os papéis que as classes envolvidas assumem também podem ser especificados.

Segundo a documentação da API do Apollon (KRUSCHE, 2023), uma instância do editor pode ser criada a partir da inicialização da classe *ApollonEditor* dado como parâmetro um conjunto de opções. Assim, neste projeto o editor foi instanciado e renderizado dentro de um elemento divisor HTML (div).

O editor gerencia os elementos visuais do diagrama através do seu atributo *model*, instância da classe *UMLModel*. Essa classe possui somente atributos públicos, armazenando as informações dos elementos instanciados e dos relacionamentos entre eles, de forma que podem ser facilmente convertidos em strings JSON. Toda alteração que é feita no diagrama é refletida no objeto *model*.

5.3.2.2 Interface

Por enquanto, o Apollon sozinho apenas inclui as funções básicas de criações de diagramas, então foi construída uma interface que possa implementar as seguintes funcionalidades necessárias deste projeto:

- Persistência em arquivo local;
- Carregamento de diagrama já salvo;
- Exportação em imagem PNG;
- Associação de um design pattern ao diagrama;
- Acionamento da geração de código.

A fim de implementar todas elas, foi necessário gerenciar o estado do diagrama externamente ao editor, feito pela classe *DiagramState*. Ela possui um nome e um *designPattern* para o diagrama que podem ser atribuídos pelo usuário e o respectivo *UMLModel* dado pelo editor.

5.3.2.3 Persistência em arquivo local

Esta funcionalidade foi desenvolvida com o uso da File System API, uma Web API disponível abertamente para uso em grande parte dos navegadores atuais. Ela é utilizada para acessar o sistema de arquivos do dispositivo utilizado, fornecendo capacidades de leitura e escrita de arquivos locais.

Ela faz uso de objetos denominados de *FileHandle*, que representam um arquivo no sistema do usuário. Eles permitem que a aplicação consiga ler o conteúdo de um arquivo e

alterá-lo. Assim, caso já exista um FileHandle instanciado na ferramenta, ou seja, o usuário carregou um arquivo, é utilizado um fluxo de escrita através de seu método *createWritable()*. A escrita do conteúdo é então, realizada pela função *write()*. Caso o FileHandle não exista, ou seja, não há nenhum arquivo carregado e o usuário modelou um diagrama novo, ele é criado através do método *showSaveFilePicker()*, o qual abre uma janela do sistema de arquivos para que seja selecionado o local no qual o novo arquivo será salvo, retornando o respectivo FileHandle à aplicação. Mais detalhes podem ser encontrados na documentação da API (MOZILLA, 2023).

O conteúdo do arquivo a ser escrito é justamente o estado do diagrama *DiagramState*, que por sua vez é convertido para uma string JSON pelo método *ConvertModelToJSON* do módulo codificador. Assim, um arquivo JSON é persistido localmente contendo o nome, o design pattern e o modelo do diagrama.

5.3.2.4 Carregamento de diagrama existente

O carregamento é realizado também com o uso da File System API, utilizando agora o método *showOpenFilePicker()*, que requisita ao usuário que seja selecionado um arquivo de seu dispositivo. Ao selecionar, o método retorna o FileHandle associado, que por sua vez permite que o arquivo seja lido pela aplicação. Dessa forma, para restaurar o diagrama dentro do editor basta ler a string JSON contida no arquivo selecionado, convertê-lo para objeto e por fim atualizar o estado *DiagramState*.

5.3.2.5 Exportação do diagrama em imagem

Em relação à exportação do diagrama em formato de imagem, o editor Apollon já possui em sua API um método denominado de *exportAsSVG()* que converte os dados do diagrama em imagem SVG. Porém para proporcionar uma variedade maior, foi desenvolvido a possibilidade de exportá-lo em formato PNG através de um simples método de conversão de dados de SVG para PNG contido no módulo codificador. Agora, para salvar a imagem no dispositivo do usuário basta executar o mesmo procedimento de persistência de arquivo escrevendo como seu conteúdo os dados de imagem gerados.

5.3.2.6 Associação de design pattern ao diagrama

Esta funcionalidade é interessante tanto para a questão organizacional quanto para o processo de aprendizagem do usuário. Ao associar um design pattern específico para o diagrama elaborado, pode-se pressupor que ele já terá uma estrutura de modelagem pré-determinada. Desse modo, a aplicação pode auxiliar a construção do diagrama através de templates, sobre os quais o usuário pode se basear ao invés de modelar a partir de um quadro vazio.

A interface deve possuir então uma espécie de biblioteca de design patterns com um objeto *UMLModel* pré-determinado para cada um deles. Esses objetos são armazenados na aplicação em arquivos JSON de fácil acesso. Com a lista enumerada *DesignPattern* pode-se especificar quais padrões são suportadas pela aplicação. Elas ainda são dividas em três categorias definidas pela enumeração *DesignPatternType*: comportamentais, criacionais e estruturais. Portanto, cada design pattern é representado dentro da aplicação pelo seu nome, seu tipo e o respectivo diagrama de template. Para expandir o repertório de padrões suportados, basta adicionar o nome à lista, especificar seu tipo e adicionar o arquivo JSON contendo o template desejado.

5.3.2.7 Acionamento da geração do código

Um elemento visual deve ser suficiente para acionar a geração do código. Tudo o que a interface faz é apenas converter o modelo atual do diagrama, ou seja, o atributo *model* do estado *DiagramState*, em uma string JSON. Novamente, isto é feito pelo módulo codificador da aplicação, a qual envia os dados para o servidor por meio de uma requisição HTTPS. Em retorno, o cliente espera pela transmissão dos arquivos de código gerados.

5.3.3 Servidor (back-end)

A arquitetura idealizada do backend com seus módulos internos encontra-se detalhada na figura 5.

5.3.3.1 Recepção das requisições em formato JSON

A requisição entre o cliente e o servidor é feita pela transferência de um arquivo JSON correspondente ao diagrama, utilizando o padrão de persistência de arquivo do Apollon (KRUSCHE et al., 2020). Neste padrão, verificamos que cada elemento do diagrama possui um identificador único e que caixas, métodos e atributos são elementos enquanto setas são relacionamentos. Dentro do campo de atributos e métodos de um elemento, há referências aos ID de outros elementos.

5.3.3.2 Geração de ficheiros de arquivos customizados de acordo com a arquitetura especificada

Inicialmente, tomando como referência implementações de scaffolding e geração de código, foi desenvolvida a funcionalidade de gerar código a partir de um modelo predefinido contido em um arquivo, proporcionando a capacidade de realizar o download do arquivo personalizado.

Um componente essencial para esta funcionalidade é representado por uma classe responsável por executar operações no sistema de arquivos, tais como criação, exclusão e compressão. Esta classe recebe uma string dos pacotes de scaffold.

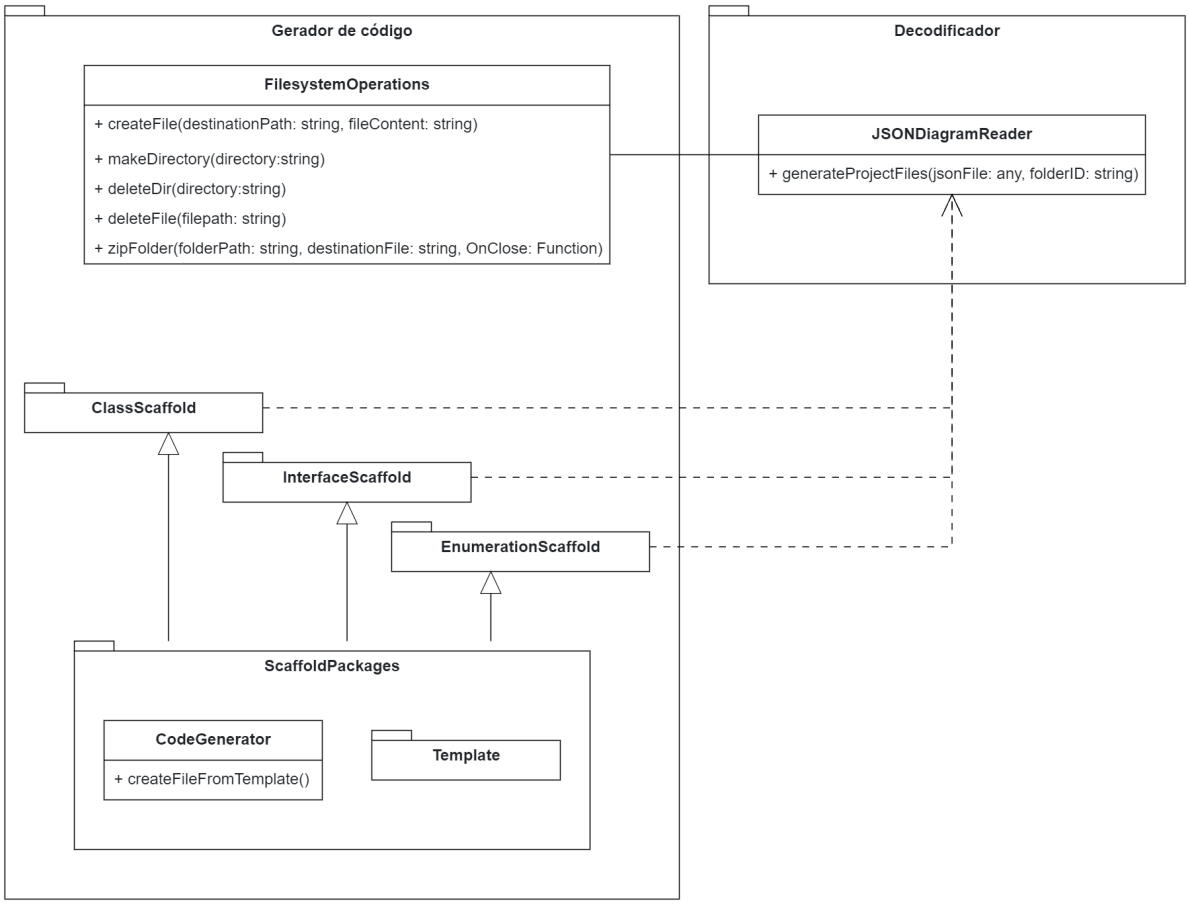


Figura 5 – Classes do Backend da aplicação

Os pacotes de scaffolding são compostos por um arquivo modelo e um script encarregado da personalização do referido modelo. A saída desses pacotes de scaffolding é formatada como uma string que pode ser salva por meio da classe de operações do sistema de arquivos. É designado um pacote para cada categoria de arquivo.

As informações que uma classe do diagrama contém e serão customizadas são: nome da classe, atributos e métodos.

A identificação das seções e termos a serem substituídos ocorre por meio de expressões regulares (RegEx). Por meio dos RegEx's, o nome da classe é customizado e são detectadas as regiões "Attributes" e "Methods".

No processamento dos atributos e métodos, os símbolos "+", "#", e "-" exercem a função de determinar o modificador de encapsulamento do campo, seja ele um atributo ou método. Os atributos do diagrama são incorporados à seção "Attributes", enquanto os métodos são inseridos na seção "Methods" do modelo.

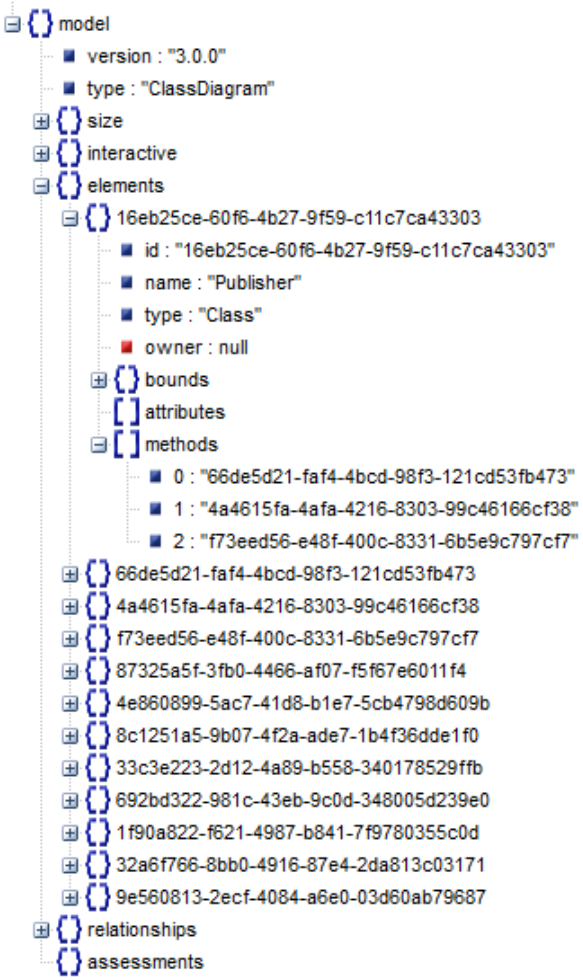


Figura 6 – Exemplo da estrutura do JSON do diagrama

5.3.3.3 Decodificação do diagrama

Este módulo de geração de código por sua vez, é empregado pelo módulo decodificador, incumbido de analisar o arquivo JSON contendo o diagrama e de executar o processo de transpilação deste.

O procedimento de transpilação do diagrama compreende a interpretação do arquivo modelo associado ao tipo de declaração presente em cada elemento do diagrama. Nesse contexto, propriedades como o nome da classe, atributos e métodos são sujeitas a modificações.

Como o serviço de geração de arquivos é executado para cada requisição de usuário, um identificador único é gerado para cada requisição, e uma pasta associada a este identificador é criada dentro do diretório público da aplicação para acomodar os arquivos pertinentes do respectivo usuário. Essa abordagem assegura que duas requisições de usuários simultâneas não entrem em conflito durante o processo de criação de arquivos.

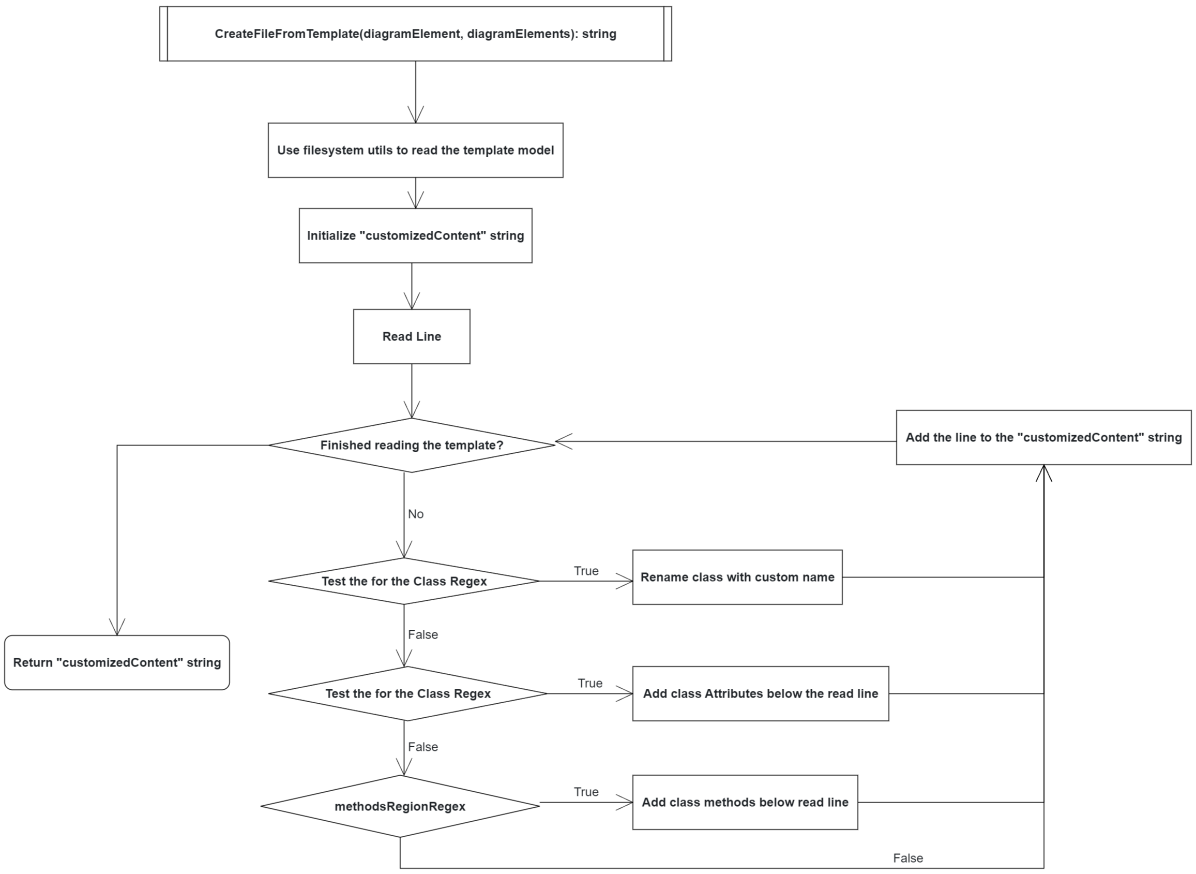


Figura 7 – Fluxo geral dos scripts geradores de código

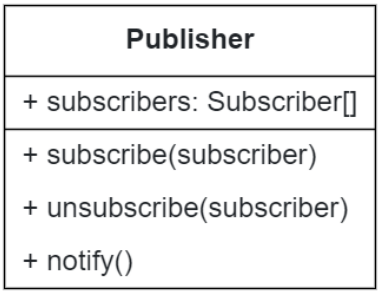


Figura 8 – Exemplo de uma classe do diagrama válida

Após a geração bem-sucedida do código correspondente ao diagrama, os arquivos resultantes são compactados e enviados ao usuário que realizou a requisição finalizando o processo.

5.3.4 Fluxo de operação

Dentro do ciclo de vida de sistemas (SEBOK, 2023), esta ferramenta é utilizada para acelerar o fluxo entre a fase de definição do sistema e a fase de realização que sofre algumas iterações com testes até a obtenção de um sistema satisfatório:

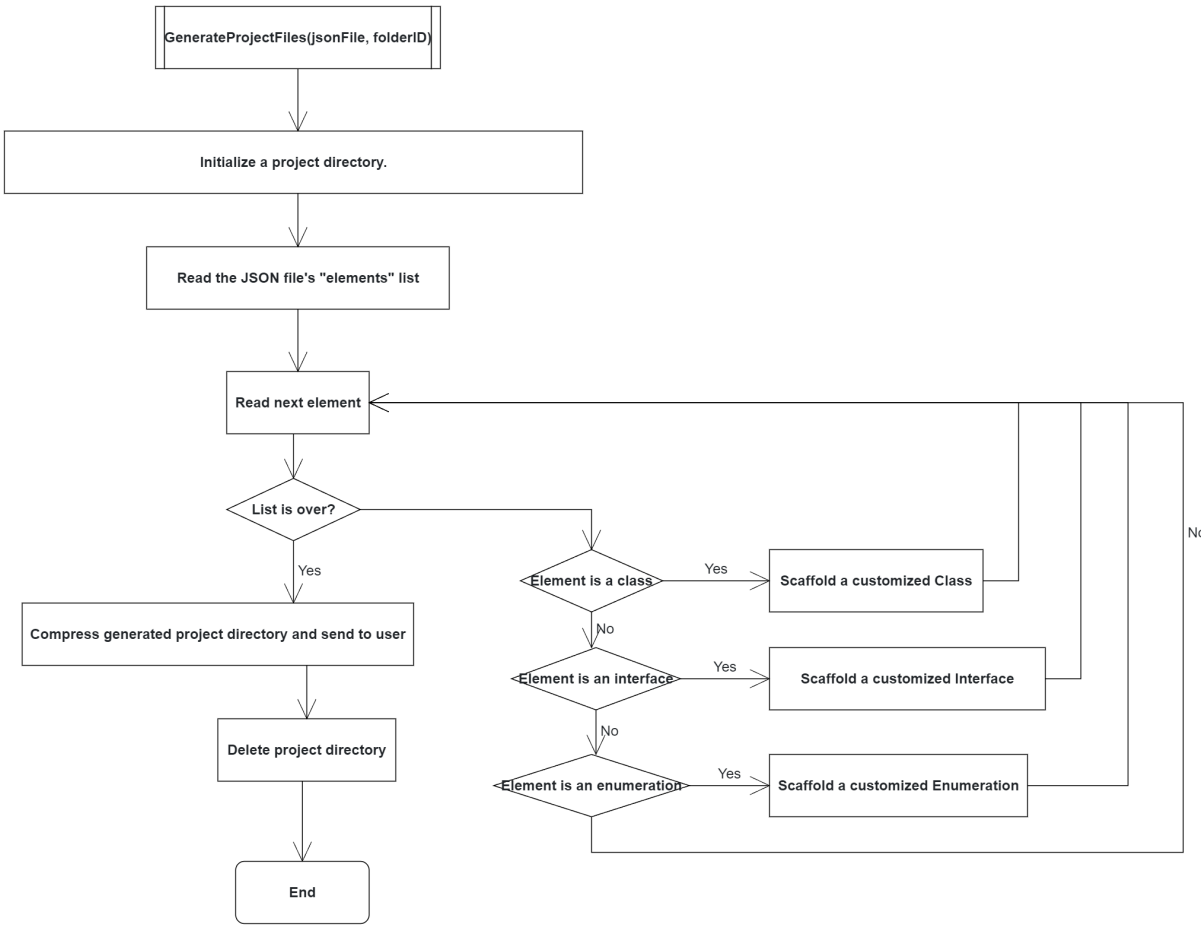
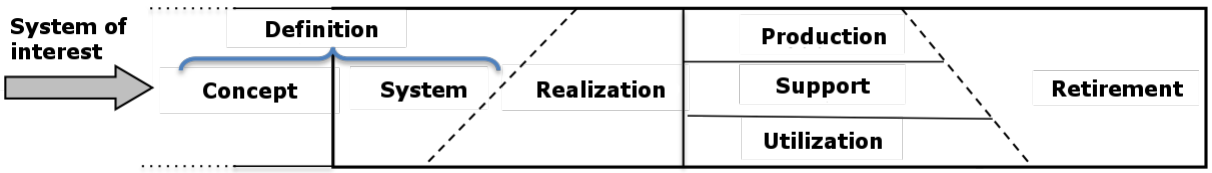


Figura 9 – Fluxo da geração de código através da leitura do diagrama

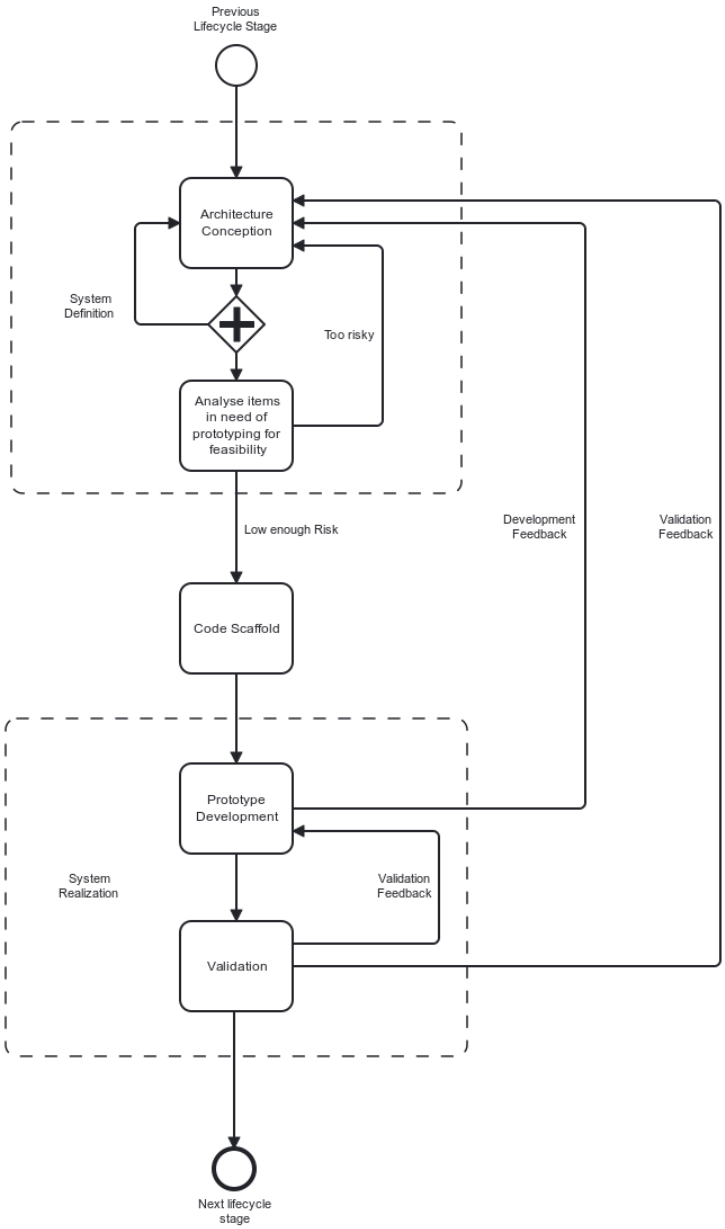
Figura 10 – Ciclo de vida de sistemas



Fonte: SEBoK (2023)

O usuário desenvolve a arquitetura de seu sistema podendo se inspirar nos exemplos de design patterns já implementados ou elaborar um próprio e acelera a prototipação com a geração da estrutura do código das classes.

Figura 11 – Uso de geradores de código no ciclo de Definição/Realização de um sistema

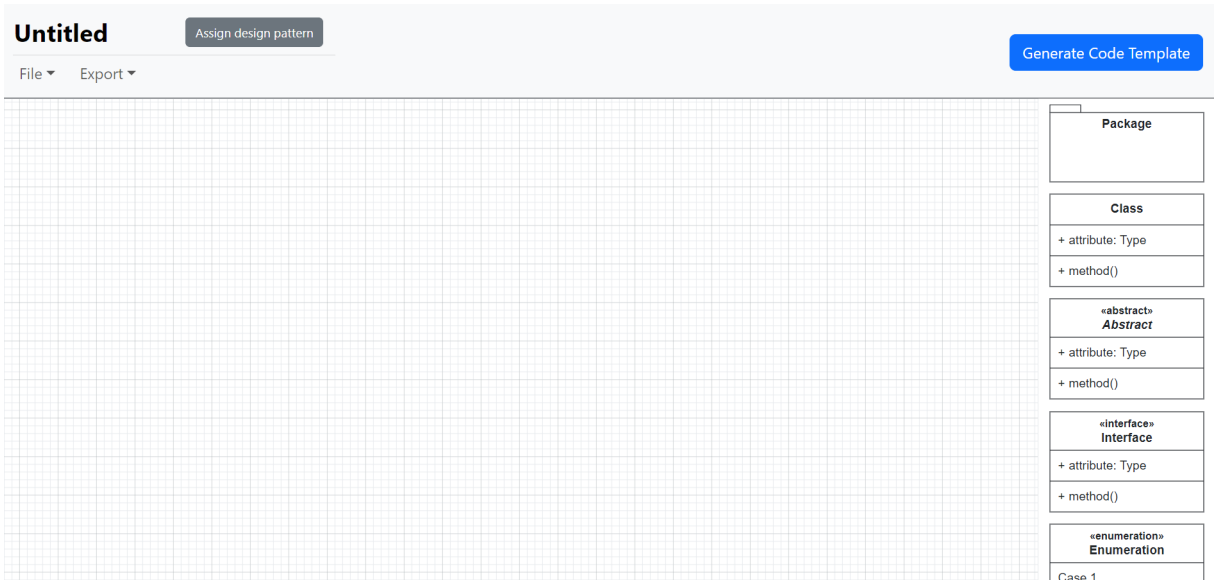


Usuários podem também desenvolver diagramas de projetos como exercícios e disponibilizá-los para o estudo, comparando a implementação do código com outros programadores, abordando o estudo de maneira mais prática.

5.4 Resultados

Seguindo essas especificações, a implementação da ferramenta se deu bem sucedida. Como o frontend foi desenvolvido em ReactJS, foi utilizado o pacote React Bootstrap para acelerar o processo de estilização e garantir que a interface tenha um design minimamente satisfatório, com uma hierarquia e propriedades de elementos pré-estabelecidos e prontos para uso. Assim, a principal tela da aplicação é mostrada na figura 12.

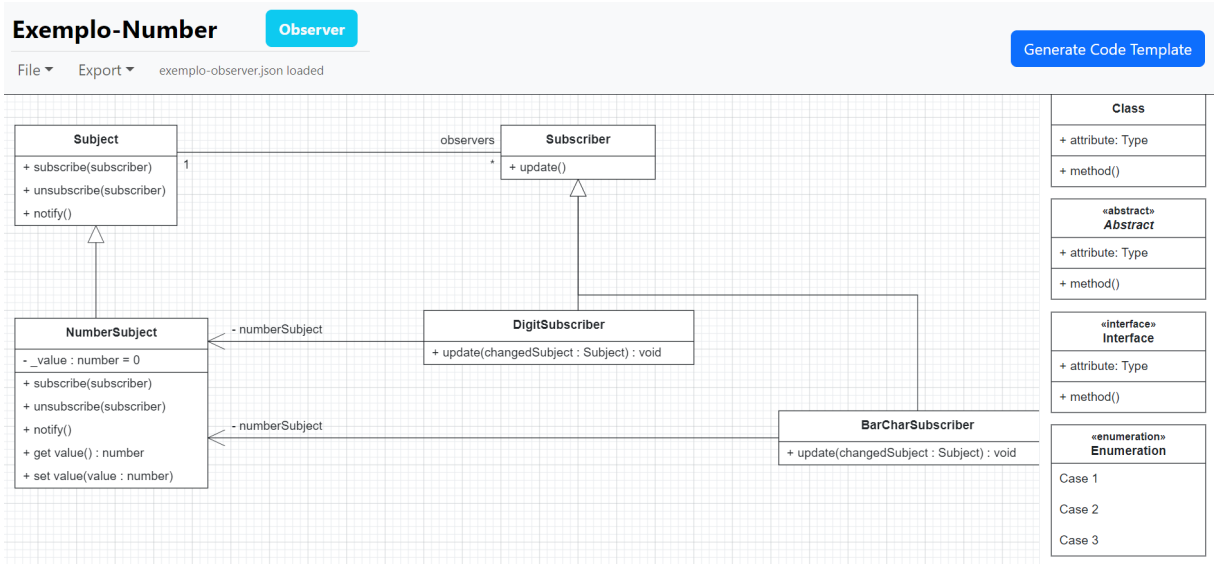
Figura 12 – Tela principal da ferramenta de modelagem



Fonte: autoral

Na região de cima está localizada a barra de ferramentas, em que o usuário pode realizar algumas funções: nomear o diagrama, associar o diagrama a um design pattern, gerenciar o arquivo, exportar o diagrama em imagens e gerar os templates de código. Abaixo se encontra o quadro no qual é possível modelar os diagramas. Na barra lateral direita estão dispostos elementos de classes UML genéricos que podem ser arrastados para dentro do quadro para serem adicionados.

Figura 13 – Tela principal da ferramenta com diagrama modelado



Fonte: autoral

A imagem 13 apresenta a tela com um diagrama modelado e o design pattern *Observer* associado a ele, que é indicado pelo botão ao lado do nome do diagrama. O usuário pode alterar o design pattern designado quando desejar clicando no botão, abrindo

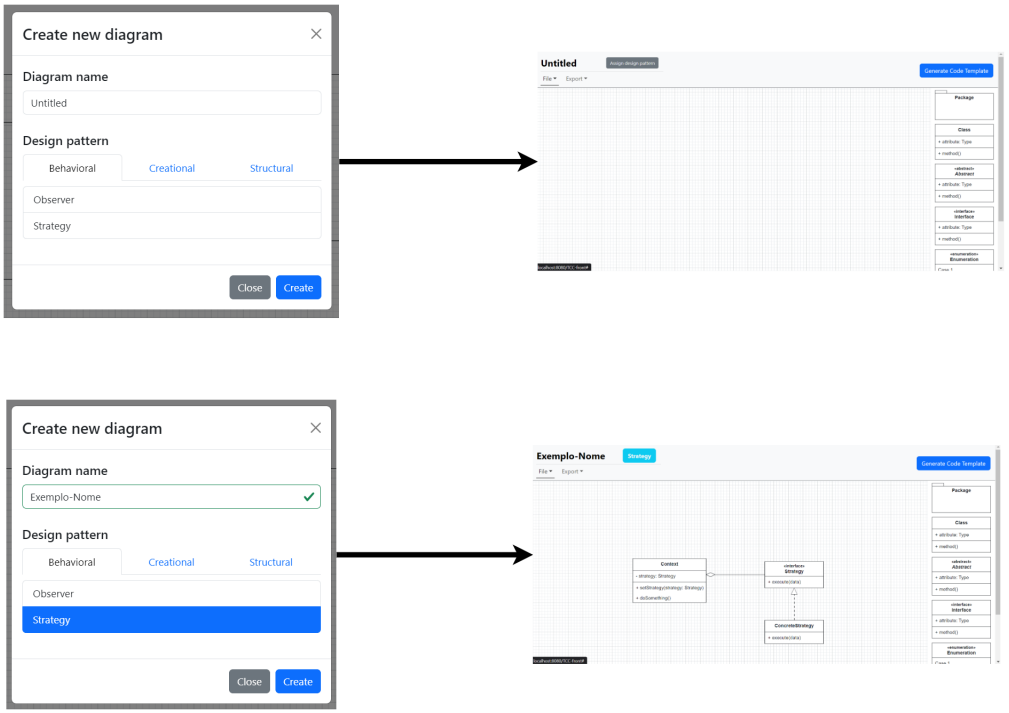
o modal de associação de design pattern (figura 16).

Figura 14 – Menu de gerenciamento de arquivo e menu de exportação

Fonte: autoral

Os menus de gerenciamento de arquivo e de exportação de imagens estão expostos na figura 14 acima. Através do primeiro menu o usuário tem as seguintes opções: criar um novo diagrama, o que abre o modal de criação de diagrama (figura 15); carregar um diagrama existente, o que requisita ao usuário a seleção de um arquivo no seu sistema de arquivo; salvar o arquivo, caso um já esteja carregado; e "salvar como", o que permite que ele escolha o nome e o local em que o arquivo será salvo. Enquanto o menu de exportação possui duas opções de formato de imagem, PNG e SVG.

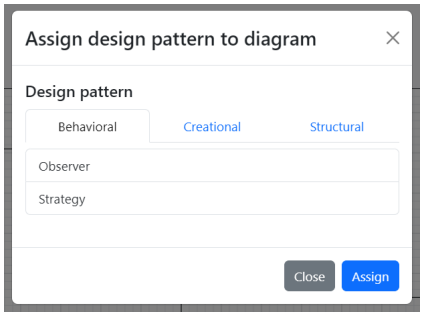
Figura 15 – Modal de criação de novo diagrama



Fonte: autoral

O modal de criação de diagrama permite que o usuário defina um nome inicial e especifique um design pattern que queira modelar. Se ele não for especificado o diagrama é criado em branco, caso contrário o template do padrão selecionado é renderizado no quadro.

Figura 16 – Modal de associação de design pattern



Fonte: autoral

Quanto ao modal de associação de design pattern, comentado anteriormente, ele possui uma estrutura semelhante ao modal anterior. Através dele, é possível alterar o padrão associado ao diagrama se já tiver ou, caso contrário, atribuir um.

Figura 17 – Exemplo de códigos gerados a partir de um diagrama

```
export class Subject {
    //#region Attributes
    //#endregion

    constructor() {
    }

    //#region Methods
    public subscribe(subscriber){
      # Implement This Method
    };
    public unsubscribe(subscriber){
      # Implement This Method
    };
    public notify(){
      # Implement This Method
    };
    //#endregion
  }
```

```
export class Subscriber {
    //#region Attributes
    //#endregion

    constructor() {
    }

    //#region Methods
    public update(){
      # Implement This Method
    };
    //#endregion
  }
```

```
export class NumberSubject {
    //#region Attributes
    private _value : number = 0;
    //#endregion

    constructor() {
    }

    //#region Methods
    public subscribe(subscriber){
      # Implement This Method
    };
    public unsubscribe(subscriber){
      # Implement This Method
    };
    public notify(){
      # Implement This Method
    };
    public get value() : number{
      # Implement This Method
    };
    public set value(value : number){
      # Implement This Method
    };
    //#endregion
  }
```

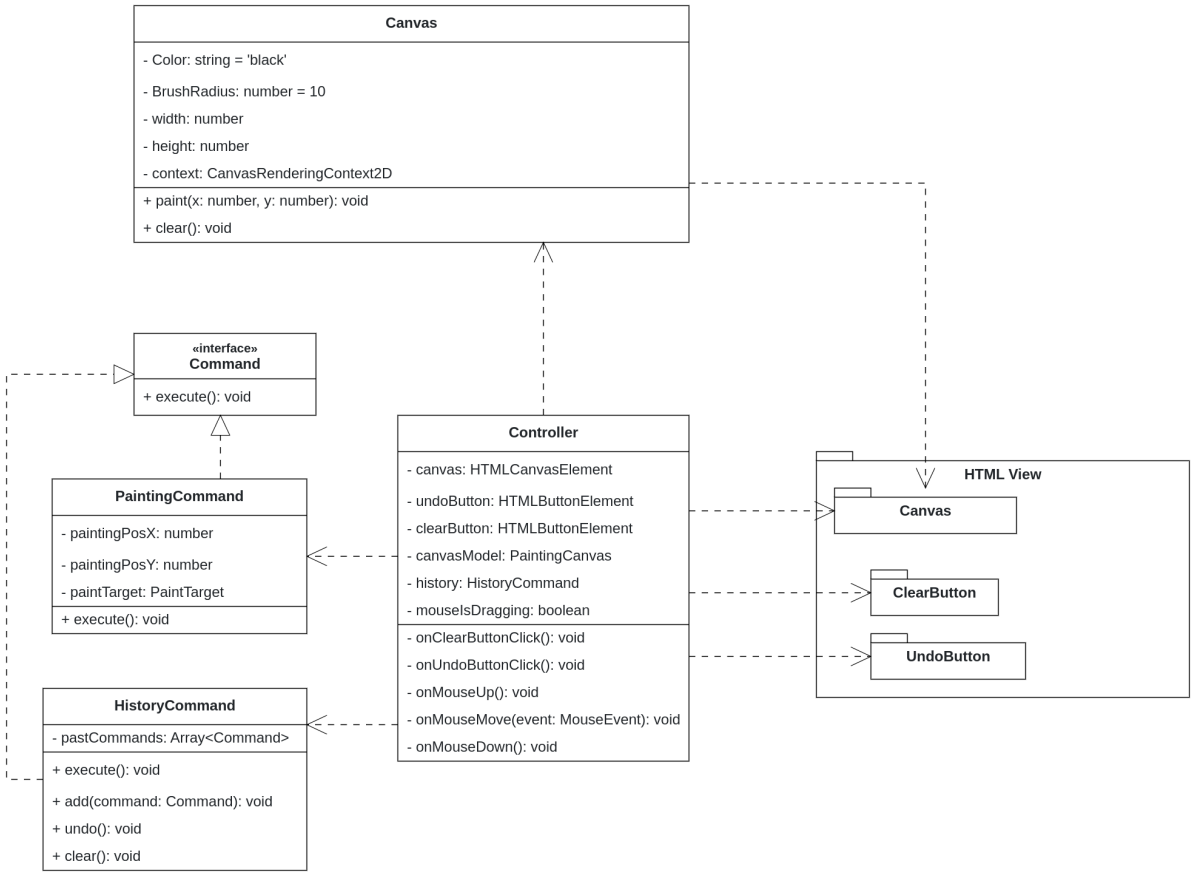
Fonte: autoral

Por fim, quando o usuário clica no botão “Generate Code Template”, os arquivos de código relacionados ao diagrama são gerados e salvos no sistema. A figura 17 demonstra alguns exemplos de códigos gerados a partir do diagrama modelado na figura 13. Vale ressaltar que tanto os nomes das classes, como seus atributos e métodos, foram transcritos adequadamente para seus códigos correspondentes, inserindo lacunas para que o usuário complete a implementação.

5.5 Testes e Avaliação

A fim de validar se o projeto cumpre com os objetivos estipulados, foi desenvolvido um pequeno projeto Web simples utilizando o código gerado pela ferramenta. Utilizamos

Figura 18 – Diagrama da aplicação



Fonte: autoral

alguns exemplos de diagrams UML de Design Patterns fornecidos pelo repositório Teshima (2021).

O teste consistiu na recriação do projeto referente ao modelo do Design Pattern "Command". Nessa implementação, foi desenvolvida uma interface que permite ao usuário desenhar livremente utilizando o mouse, além de oferecer funcionalidades para desfazer e limpar o desenho.

A arquitetura da aplicação foi simplificada e seu diagrama está apresentado na figura 18. O projeto produzido pelo gerador de código está listado no apêndice A e o código implementado através da modificação dos códigos gerados está listado no apêndice B.

A recriação do projeto foi bem-sucedida, evidenciando que esse método de estudo possibilita ao usuário uma visão que requer um nível de abstração menor ao examinar o panorama geral das interações da arquitetura durante a documentação no diagrama, ao mesmo tempo em que permite focar mais a atenção para a lógica dos algoritmos implementados dentro dos métodos durante o desenvolvimento do código.

6 Considerações Finais

6.1 Conclusões do Projeto de Formatura

Neste projeto de formatura, foi concluido o ciclo de design de um artefato de metaprogramação, empregando tecnologias e conceitos atuais e relevantes. Esta realização representa uma contribuição para o progresso e aperfeiçoamento do campo de estudo abordado, provando a relevância das abordagens adotadas no desenvolvimento do artefato.

6.2 Contribuições

No âmbito deste trabalho, foi realizada uma análise do panorama de pesquisa relacionado a ferramentas que integram a documentação arquitetural, através de diagramas UML, com o processo de geração de código. Além disso, foi concretizada uma contribuição prática ao adaptar a interface e o modelo de transferência de dados da ferramenta de diagramas UML Apollon (KRUSCHE et al., 2020) elaborando uma solução própria de um servidor para a geração de código e integrando as tecnologias. Essa iniciativa resultou em uma ferramenta que promove a sinergia entre a representação visual da arquitetura e o processo de geração de código.

6.3 Perspectivas de Continuidade

Extensões deste projeto abrangem as seguintes ramificações:

6.3.1 Ampliação de Suporte para Outras Linguagens de Programação

Este avanço seria alcançado mediante a incorporação de templates e scripts de geração de código específicos para classes e estruturas inerentes a diferentes linguagens de programação.

6.3.2 Implementação de Fluxo Reverso de Operação

Considera-se a viabilidade de desenvolver a capacidade de gerar um diagrama, funcionando como documentação, a partir da análise da base de código existente, proporcionando uma perspectiva inversa do processo de transpilação.

6.3.3 Pesquisa e desenvolvimento de Design Patterns

A pesquisa e inclusão de novos design patterns como modelos de exemplo ao projeto é uma possibilidade, sendo necessário a padronização, documentação e validação por outros usuários, isto aumenta as opções disponíveis para inicializar os diagramas e promove práticas para manter o estudo de padrões de design robustos e atualizados.

6.3.4 Desenvolvimento de Ferramentas Padronizadas para Testes de Módulos

Com o intuito de aprimorar a qualidade e confiabilidade do software gerado, contempla-se a criação de ferramentas padronizadas para a realização de testes nos diversos módulos do projeto.

Referências

CRUZ, M. A. A. da et al. Olp—a restful open low-code platform. *Future Internet*, v. 13, n. 10, 2021. ISSN 1999-5903. Disponível em: <https://www.mdpi.com/1999-5903/13/10/249>. Citado na página 26.

FBI. *Low Code Development Platform Market Size, Share*. [S.l.]: Fortune Business Insights, 2022. <https://www.fortunebusinessinsights.com/low-code-development-platform-market-102972>. Acesso em: 20 mar. 2023. Citado na página 14.

HUANG, H.; YANG, D. Teaching design patterns: A modified pbl approach. *IEEE*, 2008. Disponível em: <https://ieeexplore.ieee.org/document/4709353>. Citado 2 vezes nas páginas 12 e 13.

KRUSCHE, S. *Apollon API*. 2023. [Online; accessed 3-December-2023]. Disponível em: <https://apollon-library.readthedocs.io/en/latest/user/api.html>. Citado na página 29.

KRUSCHE, S. et al. An Interactive Learning Method to Engage Students in Modeling. In: . [S.l.]: ACM, 2020. (42nd International Conference on Software Engineering, Software Engineering Education and Training), p. 12–22. Citado 4 vezes nas páginas 26, 28, 31 e 41.

MAGNO, D. G. Aplicação da técnica de scaffolding para a criação de sistemas crud. *UNIFEI Itajubá*, 2015. Disponível em: <https://repositorio.unifei.edu.br/xmlui/bitstream/handle/123456789/197/dissertacao_magno_2015.pdf>. Citado na página 16.

MOZILLA. *File System API*. 2023. [Online; accessed 3-December-2023]. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/API/File_System_API>. Citado na página 30.

OMG. *MDA® - The Architecture Of Choice For A Changing World*. [S.l.]: OMG. <http://www.omg.org/mda/index.htm>. Acesso em: 20 mar. 2023. Citado na página 15.

PILLAY, N. Teaching design patterns. *Universidade de KwaZulu-Natal*, 2010. Disponível em: <https://www.researchgate.net/publication/229049551_Teaching_Design_Patterns>. Citado 2 vezes nas páginas 12 e 13.

RAFAL. *Design patterns – do we still need them?* Mestwin Blog, 2021. Disponível em: <https://blog.mestwin.net/design-patterns-do-we-still-need-them/>. Acesso em: 19 abr. 2023. Citado 2 vezes nas páginas 12 e 13.

SEBOK. *Generic Life Cycle Model — SEBoK,*. 2023. [Online; accessed 2-December-2023]. Disponível em: <https://sebokwiki.org/w/index.php?title=Generic_Life_Cycle_Model&oldid=70215>. Citado 2 vezes nas páginas 34 e 35.

SHVETS, A. *What's a design pattern?* [S.l.]: Refactoring.Guru. <https://refactoring.guru/design-patterns/what-is-pattern>. Acesso em: 20 mar. 2023. Citado na página 15.

TESHIMA, T. *UML Diagram for TypeScript Design Pattern Examples*. 2021. <https://github.com/takaakit/uml-diagram-for-typescript-design-pattern-examples>. Acesso em: 6 dez. 2023. Citado na página 40.

WIERINGA, R. J. *Design Science Methodology for Information Systems and Software Engineering*. Enschede, Holanda: Springer, 2014. 3-11 p. Citado 2 vezes nas páginas 17 e 18.

Apêndices

APÊNDICE A – Códigos gerados no teste

Listagem A.1 – Código gerado para a classe Controller.ts

```
export class Controller {
  //#region Attributes
    private canvas: HTMLCanvasElement;
    private undoButton: HTMLButtonElement;
    private clearButton: HTMLButtonElement;
    private canvasModel: PaintingCanvas;
    private history: HistoryCommand;
    private mouseIsDragging: boolean;
  //#endregion

  constructor() {
  }

  //#region Methods
    private onClearButtonClick(): void
    {
      //# Implement This Method
    };
    private onUndoButtonClick(): void
    {
      //# Implement This Method
    };
    private onMouseUp(): void
    {
      //# Implement This Method
    };
    private onMouseMove(event: MouseEvent): void
    {
      //# Implement This Method
    };
    private onMouseDown(): void
    {
      //# Implement This Method
    };
  //#endregion
}
```

Listagem A.2 – Código gerado para a classe Canvas.ts

```
export class Canvas {
  //#region Attributes
    private Color: string = 'black';
    private BrushRadius: number = 10;
    private width: number;
    private height: number;
    private context: CanvasRenderingContext2D;
  //#endregion

  constructor() {
  }

  //#region Methods
    public paint(x: number, y: number): void
    {
      //# Implement This Method
    };
    public clear(): void
    {
      //# Implement This Method
    };
  //#endregion
}
```

Listagem A.3 – Código gerado para a classe Command.ts

```
export interface Command {
    //#region Attributes
    //#endregion

    //#region Methods
    execute(): void;
    //#endregion
}
```

Listagem A.4 – Código gerado para a classe HistoryCommand.ts

```
export class HistoryCommand {
  //#region Attributes
    private pastCommands: Array<Command>;
  //#endregion

  constructor() {
  }

  //#region Methods
    public execute(): void
    {
      //# Implement This Method
    };
    public add(command: Command): void
    {
      //# Implement This Method
    };
    public undo(): void
    {
      //# Implement This Method
    };
    public clear(): void
    {
      //# Implement This Method
    };
  //#endregion
}
```

Listagem A.5 – Código gerado para a classe PaintingCommand.ts

```
export class PaintingCommand {
  //#region Attributes
    private paintingPosX: number;
    private paintingPosY: number;
    private paintTarget: PaintTarget;
  //#endregion

  constructor() {
  }

  //#region Methods
    public execute(): void
    {
      //# Implement This Method
    };
```

```
    //#endregion
}
```

APÊNDICE B – Códigos implementados no teste

Listagem B.1 – Código implementado para a classe Controller.ts

```
import { Command } from "../Interfaces/Command";
import { Canvas } from "./Canvas";
import { HistoryCommand } from "./HistoryCommand";
import { PaintingCommand } from "./PaintingCommand";

export class Controller {
  //#region Attributes
    private canvas: HTMLCanvasElement;
    private undoButton: HTMLButtonElement;
    private clearButton: HTMLButtonElement;
    private canvasModel: Canvas;
    private history: HistoryCommand;
    private mouseIsDragging: boolean;
  //#endregion

  constructor() {
    this.mouseIsDragging = false;
    this.canvas = <HTMLCanvasElement>document.getElementById(`canvas`);
    this.undoButton = <HTMLButtonElement>document.getElementById(`
        undoButton`)
    this.clearButton = <HTMLButtonElement>document.getElementById(`
        clearButton`)
    this.history = new HistoryCommand();
    this.canvasModel = new Canvas(this.canvas.width, this.canvas.height,
        this.canvas.getContext(`2d`));
    this.canvas.addEventListener(`mousedown`, () => this.onMouseDown());
    this.canvas.addEventListener(`mousemove`, (event) => this.onMouseMove(
        event));
    this.canvas.addEventListener(`mouseup`, () => this.onMouseUp());
    this.undoButton.addEventListener(`click`, () => this.onUndoButtonClick
        ());
    this.clearButton.addEventListener(`click`, () => this.
        onClearButtonClick());
  }
```

```
  //#region Methods
    public onClearButtonClick(): void
    {
      this.canvasModel.clear();
      this.history.clear();
    };
    public onUndoButtonClick(): void
    {
      this.canvasModel.clear();
      this.history.undo();
      this.history.execute();
    };
    public onMouseUp(): void
    {
      this.mouseIsDragging = false;
    };
    public onMouseMove(event: MouseEvent): void
    {
      if(this.mouseIsDragging){
        const paintingPosX = event.clientX - this.canvas.
            getBoundingClientRect().left;
        const paintingPosY = event.clientY - this.canvas.
            getBoundingClientRect().top ;
        const command: Command = new PaintingCommand(paintingPosX,
            paintingPosY, this.canvasModel);
        this.history.add(command);
        command.execute();
      }
    };
    public onMouseDown(): void
    {
      this.mouseIsDragging = true;
    }
  //#endregion
}
```

Listagem B.2 – Código implementado para a classe Canvas.ts

```
export class Canvas {
  //#region Attributes
    private Color: string = 'black';
    private BrushRadius: number = 10;
    private width: number;
    private height: number;
    private context: CanvasRenderingContext2D;
```

```
  //#endregion

  public constructor(width: number, height: number, context:
      CanvasRenderingContext2D)
  {
    this.Color = 'black';
    this.BrushRadius = 10;
    this.context = context;
    this.width = width;
    this.height = height;
  };

  //#region Methods
  public paint(x: number, y: number): void
  {
    this.context.beginPath();
    this.context.arc(x, y, this.BrushRadius, 0, Math.PI*2, false);
    this.context.fillStyle = this.Color;
    this.context.fill();
  };
  public clear(): void
  {
    this.context.clearRect(0, 0, this.width, this.height);
  };
  //#endregion
}
```

Listagem B.3 – Código implementado para a classe Command.ts

```
export interface Command {
    //#region Attributes
    //#endregion

    //#region Methods
    execute(): void;
    //#endregion
}
```

Listagem B.4 – Código implementado para a classe HistoryCommand.ts

```
import { Command } from "../Interfaces/Command";
```

```
export class HistoryCommand {
  //#region Attributes
    private pastCommands: Array<Command>;
  //#endregion

  constructor() {
    this.pastCommands = new Array<Command>();
  }

  //#region Methods
    public execute()
    {
      for (let command of this.pastCommands) {
        command.execute();
      }
    };
    public add(command: Command): void
    {
      this.pastCommands.push(command);
    };
    public undo(): void
    {
      if (this.pastCommands.length !== 0) {
        this.pastCommands.pop();
      }
    };
    public clear(): void
    {
      this.pastCommands.length = 0;
    };
    //#endregion
}
```

Listagem B.5 – Código implementado para a classe PaintingCommand.ts

```
import { Canvas } from "./Canvas";

export class PaintingCommand {
  //#region Attributes
    private paintingPosX: number;
    private paintingPosY: number;
    private paintTarget: Canvas;
  //#endregion
```

```
  constructor(paintingPosX: number, paintingPosY: number, paintTarget:
      Canvas) {
    this.paintingPosX = paintingPosX;
    this.paintingPosY = paintingPosY;
    this.paintTarget = paintTarget;
  }

  //#region Methods
    public execute()
    {
      this.paintTarget.paint(this.paintingPosX, this.paintingPosY);
    };
  //#endregion
}
```