INSTITUTO FEDERAL DE ALAGOAS

***CAMPUS* ARAPIRACA**

CURSO DE BACHARELADO EM SISTEMAS DE INFORMAÇÃO

GABRIEL SOARES DA SILVA SANTOS

ISMAEL FELIX DA ROCHA

TOTH: GERAÇÃO DE CÓDIGO BASEADA NA ARQUITETURA DE PORTAS E ADAPTADORES COM GOLANG

ARAPIRACA, AL

2023

GABRIEL SOARES DA SILVA SANTOS
ISMAEL FELIX DA ROCHA

TOTH: GERAÇÃO DE CÓDIGO BASEADA NA ARQUITETURA DE PORTAS E ADAPTADORES COM GOLANG

Trabalho de Conclusão de Curso apresentado ao Curso Superior de Sistemas de Informação do Instituto Federal de Alagoas, Campus Arapiraca, como requisito parcial para obtenção de grau de Bacharel em Sistemas de Informação.

Orientador: Dr. Társis Marinho de Souza.

ARAPIRACA, AL
2023

Dados Internacionais de Catalogação na Publicação
Instituto Federal de Alagoas
***Campus* Arapiraca**

S237t

Santos, Gabriel Soares da Silva.
Toth: geração de código baseada na arquitetura de portas e adaptadores em Golang / Gabriel Soares da Silva Santos, Ismael Felix da Rocha. – 2023.
1 PDF: il., color. (1 aquivo : 1,1 MB).

Arquivo digital no formato PDF do trabalho acadêmico com 69 folhas.

Orientação: Prof. Dr. Társis Marinho de Souza.
Trabalho de Conclusão de Curso (Graduação, Bacharelado em Sistemas de Informação) – Instituto Federal de Alagoas, *Campus* Arapiraca, Arapiraca, 2023.

1. Geração de código. 2. Arquitetura de portas e adaptadores. 3. Linguagem Go (Golang) – linguagem de programação. 4. Desenvolvimento orientado a modelo. 5. Flexibilidade de código. I. Rocha, Ismael Felix da. II. Título.

CDD: 005.1

Luciete B. da Silva
Bibliotecária – CRB-4/1739

GABRIEL SOARES DA SILVA SANTOS
ISMAEL FELIX DA ROCHA

TOTH: GERAÇÃO DE CÓDIGO BASEADA NA ARQUITETURA DE PORTAS E ADAPTADORES COM GOLANG

Trabalho apresentado ao Curso Superior de Sistemas de Informação do Instituto Federal de Alagoas, campus Arapiraca, como requisito parcial para a obtenção do grau de Bacharel em Sistemas de Informação.

Aprovado(a) em: 13/12/2023.

BANCA EXAMINADORA:

gov.br Documento assinado digitalmente
TARSIS MARINHO DE SOUZA
Data: 15/12/2023 02:26:19-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. Társis Marinho de Souza (Orientador)
Instituto Federal de Alagoas - IFAL

gov.br Documento assinado digitalmente
BRUNO RAFAEL FERREIRA SOUZA BARBOSA DA
Data: 15/12/2023 08:47:42-0300
Verifique em https://validar.iti.gov.br

Prof. MSc. Bruno Rafael Ferreira Souza Barbosa da Silva
Polarsoft

gov.br Documento assinado digitalmente
ELVYS ALVES SOARES
Data: 15/12/2023 08:28:16-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. Elvys Alves Soares
Instituto Federal de Alagoas - IFAL

AGRADECIMENTOS

Agradeço ao meu orientador, professor Dr. Társis Marinho, pelos conselhos, pela amizade e por todas as oportunidades que me foram oferecidas, pois através delas pude aprender, evoluir como profissional e mudar completamente a realidade em que eu estava inserido.

Agradeço à minha mãe, Erineidy, por todo o amor, dedicação e luta para superar muitas barreiras e me dar o privilégio de crescer num ambiente de segurança, por meio do qual eu consegui me desenvolver tanto profissionalmente quanto como ser humano.

Sou grato aos amigos formados durante o curso de graduação pelo companheirismo e apoio durante os últimos anos.

GABRIEL SOARES DA SILVA SANTOS

AGRADECIMENTOS

Ao Professor Dr. Társis Marinho, pela paciência na orientação e incentivo que tornaram possível a conclusão desse projeto além de toda a parceria ao longo do curso e por todas as oportunidades ofertadas, as quais contribuíram profundamente para o meu desenvolvimento profissional e humano.

À minha família e à minha noiva, por estarem sempre do meu lado e me incentivaram a continuar perseverando por entre os desafios, por contribuírem para que eu sempre tivesse o suporte e o apoio necessário.

Aos meus amigos, por sempre estarem presentes e por terem compartilhado comigo de muitas experiências memoráveis ao longo desses anos de graduação.

ISMAEL FELIX DA ROCHA

RESUMO

Em um cenário de busca incessante por soluções tecnológicas, a rapidez no desenvolvimento de software é crucial. Contudo, um processo acelerado sem a devida atenção e que possui problemas de planejamento pode comprometer a manutenção e a qualidade do software, gerando problemas como a inserção de complexidade acidental, o atraso de entregas e a inutilização do produto a longo prazo. A geração de código baseada em modelo apresenta-se como um fator atenuante da problemática, uma vez que promove a aceleração do processo de desenvolvimento, mantendo o código padronizado, reduzindo a inserção de erros humanos e promovendo a adoção de boas práticas de software. Este trabalho concentra-se na pesquisa e desenvolvimento de uma ferramenta para a geração de código em linguagem Go, integrando-a à arquitetura de portas e adaptadores, representando uma contribuição significativa para a pesquisa em geração de código. A proposta visa oferecer uma solução prática e eficaz, enquanto também se esforça para otimizar a eficiência e escalabilidade do processo, aprimorando ainda mais a aplicabilidade da ferramenta desenvolvida.

Palavras-chave: geração de código; arquitetura de portas e adaptadores; desenvolvimento orientado a modelo; linguagem Go (Golang); flexibilidade de código.

ABSTRACT

In a scenario of relentless pursuit of technological solutions, speed in software development is crucial. However, an accelerated process without proper attention and with planning issues can compromise software maintenance and quality, leading to problems such as the introduction of accidental complexity, delays in deliveries, and the long-term obsolescence of the product. Model-based code generation emerges as a mitigating factor for these challenges, accelerating the development process, maintaining standardized code, reducing the introduction of human errors, and promoting the adoption of good software practices. This work focuses on the research and development of a tool for code generation in the Go language, integrating it into the architecture of ports and adapters, representing a significant contribution to research in code generation. The proposal aims to provide a practical and effective solution while also striving to optimize the efficiency and scalability of the process, further enhancing the applicability of the developed tool.

Keywords: code generation; ports and adapters architecture; model-driven development; Go language (Golang); code flexibility.

LISTA DE ILUSTRAÇÕES

LISTA DE CÓDIGOS-FONTE

LISTA DE QUADROS E TABELAS

SUMÁRIO

1 INTRODUÇÃO

A busca frenética por soluções tecnológicas a fim de atender demandas e dores apresentadas pela sociedade no menor tempo e com o menor custo possíveis, tem estimulado a investigação e o desenvolvimento de ferramentas capazes de acelerar o processo de desenvolvimento de software, as quais têm assumido um papel fundamental nesse processo (Gurung, Shah e Jaiswal, 2020).

Nesse contexto, uma série de ferramentas destinadas a facilitar o processo de produção de software tem emergido, incluindo as abordagens de baixo código (*low-code*), sem código (*no-code*) e a geração de código baseada em modelo. Pinho, Aguiar e Amaral (2022) analisam os aspectos positivos e negativos do desenvolvimento baixo de código (*Low-Code* Development) e o comparam com a abordagem sem desenvolvimento código (*No-Code* Development), revelando que ambas as abordagens promovem a produção de software com pouca ou nenhuma codificação manual, embora imponham limitações nas capacidades do software, abrangendo questões de desempenho, escalabilidade, alcance de domínio e problemas de segurança.

Sob outra perspectiva, há a geração de código baseada em modelo, a qual fornece uma flexibilidade altamente desejada por equipes de desenvolvimento, pois atuam efetivamente na dor dos desenvolvedores de *software*, agilizando tarefas repetitivas ao passo em que fornece autonomia de alterar e estender o código existente (Paolone *et al*, 2020). Dentre os benefícios fornecidos pela geração de código, destacam-se a mitigação da introdução de erros humanos no software (Huning & Pulvermueller, 2021), a padronização dos componentes do código, mantendo a qualidade do software (Mukhtar & Galadanci, 2018) e a amenização dos problemas citados anteriormente. Em adição a isso, também é relevante destacar a presença de software usado diariamente por profissionais do desenvolvimento de software que contém uma gama de diferentes recursos que envolvem a geração de código, como, por exemplo, ambientes de desenvolvimento integrado (IDE), como mencionado por XU, Vasislescu & Neubig (2022) e Nam *et al* (2023).

Contudo, essa busca por agilidade produz influência substancial na qualidade do ciclo de desenvolvimento de software e no sucesso do lançamento de produtos no mercado, conforme enfatizado por De Souza & De Moraes (2021). Nessa perspectiva, arquitetos de software desempenham um papel crucial, sendo responsáveis por criar e avaliar o design de software para assegurar que o trabalho

produzido pela equipe de desenvolvimento esteja alinhado com as necessidades do cliente e atenda aos requisitos de qualidade necessários para garantir aspectos fundamentais da engenharia de software como manutenibilidade, escalabilidade, segurança e reutilização de código (Richards & Ford, 2020).

No entanto, é importante salientar que, em muitas equipes de desenvolvimento, o processo de planejamento pode ser negligenciado ou mal executado, resultando em problemas ocultos no curto e médio prazo que podem se transformar em obstáculos significativos mais a frente no processo (Robert, 2019). Com frequência, a falta de especificações decorrentes de um planejamento deficiente leva os engenheiros de software a introduzir complexidade acidental no software, comprometendo a vida útil do projeto (Balaban, Khitron & Maraee, 2022).

Desse modo, a pesquisa em questão tem como foco a abordagem da geração de código baseada em modelo (*Model Driven Development*), tendo como produto soluções baseadas em arquitetura de software. Ao explorar as possibilidades e desafios relacionados à geração de código, esta pesquisa busca detalhar a construção de uma ferramenta capaz de gerar código, levando em consideração os principais aspectos de engenharia e de arquitetura de software e sobre como essa técnica pode ser aplicada de forma a aprimorar o processo de desenvolvimento de software, possibilitando uma maior flexibilidade e eficiência na evolução e manutenção dos sistemas (Mukhtar & Galadanci, 2018). Além disso, este trabalho concentra-se especificamente na linguagem Go (*Golang*[1]) e no suporte à arquitetura de software de portas e adaptadores. A escolha da linguagem Go se deve à sua eficiência (Chabbi & Ramanathan, 2022), simplicidade e a crescente popularidade no desenvolvimento de software (Marchuk, Melnyk & Melnyk, 2023). Quanto à adoção da arquitetura de portas e adaptadores, esta proporciona uma estrutura flexível e extensível que se alinha com os objetivos de agilidade e qualidade do desenvolvimento de software (Cockburn, 2005). Essa combinação de elementos busca otimizar a geração de código e a arquitetura de software para atender às demandas do mercado de forma eficaz, conferindo a flexibilidade de desenvolvimento sobre a base de código gerada e mantendo o processo mais veloz do que o processo inteiramente manual.

Em destaque, a abordagem proposta neste trabalho confere ao time de desenvolvimento liberdade para avançar no processo de desenvolvimento de

[1] https://go.dev/

maneira independente após a geração inicial de código, contrariando muitos frameworks utilizados cotidianamente em contextos de desenvolvimento de software, os quais estabelecem uma dependência contínua de seu uso ao longo de todo o ciclo de desenvolvimento (Moströ & Hyerberg, 2022). A dependência entre equipe e ferramenta dá-se, principalmente, no excesso de abstração que essas ferramentas possuem, distanciando a equipe dos processos que ocorrem internamente e limitando o poder de modificação do código (Møller & Veileborg, 2020).

Como ponto de partida para esta pesquisa, conduziram-se operações de coleta de informações. Foi realizada uma pesquisa preliminar com o objetivo de identificar os termos-chave associados ao tema, tanto em português quanto em inglês. Em seguida, foi realizada uma categorização protocolada, objetivando aprofundar a base de conhecimento sobre a qual o trabalho foi fundamentado.

Posteriormente, foram conduzidos os procedimentos de desenvolvimento da solução Toth, os quais envolveram o estabelecimento de documentações iniciais, como documento de requisitos, casos de uso, diagramas de sequência e cronograma de produção, instauração da dinâmica de uma metodologia ágil no processo de desenvolvimento da ferramenta e o ciclo de desenvolvimento propriamente dito, contando com a escolha das tecnologias utilizadas e as decisões técnicas de implementação da solução.

Desse modo, foi possível estabelecer os objetivos da pesquisa da seguinte forma.

1.1 OBJETIVO GERAL

Propor uma solução para apoiar a geração de códigos de base extensível apoiado por uma arquitetura de software baseada em portas e adaptadores.

1.2 OBJETIVOS ESPECÍFICOS

a. Promover o desenvolvimento de soluções que adotam as boas práticas de engenharia de software;

b. Acelerar o processo de desenvolvimento de soluções baseadas em uma arquitetura de portas e adaptadores que adotam a linguagem Go.

1.3 ORGANIZAÇÃO DO TRABALHO

Sobre o restante desta pesquisa, a estrutura está feita como descrito a seguir. No capítulo 2, tem-se o referencial teórico, em que estão destacados os recursos teóricos basais sobre os quais se estrutura este trabalho. No capítulo 3, estão pontuados os principais trabalhos que se relacionam à temática desta pesquisa. No capítulo 4, são apresentados os aspectos metodológicos deste trabalho. No capítulo 5, explicam-se os detalhes da solução proposta. No capítulo 6, apresentam-se os resultados observados após uma avaliação da ferramenta. Finalizando, no capítulo 7, são apresentadas as considerações finais e as observações a serem consideradas em trabalhos futuros.

2 REFERENCIAL TEÓRICO

A fim de realizar a apresentação do conhecimento necessário para a compreensão dos desenvolvimentos desta pesquisa, esta seção trabalha os conceitos norteadores de geração de código, arquitetura de portas e adaptadores, desenvolvimento de software orientado a modelo (MDD), o modelo cliente-servidor e a arquitetura de transferência de estado representacional.

2.1 GERAÇÃO DE CÓDIGO

Grande parte do processo de desenvolvimento de software envolve a execução de tarefas de escrita de código repetitivas. Essas tarefas não apenas reduzem a velocidade de produção dos desenvolvedores, mas também afetam a qualidade do software, impactando negativamente a produtividade, a motivação dos programadores e o cumprimento de prazos estipulados para o produto (Errington, 2003).

Nesse contexto, a geração de código é apresentada como uma solução para mitigar esses problemas. Sua função primordial é produzir código replicável que pode ser aplicado em diversos contextos, mantendo um nível consistente de qualidade (Mukhtar & Galadanci, 2018) e prevenindo a introdução de erros humanos durante o processo (Huning & Pulvermueller, 2021).

Os geradores de código são ferramentas criadas com a finalidade de produzir código automaticamente em diferentes contextos e com diferentes propósitos. Errington (2003) descreve seis tipos distintos de geradores de código, com base em como o gerador processa os dados de entrada para gerar os dados de saída.

Dentre esses tipos, é relevante destacar o modelo de geração por camada, que foi adotado no desenvolvimento da ferramenta mencionada neste trabalho. Esse modelo opera com base na ideia de geração orientada por modelo. O processo consiste em fornecer um arquivo de definição ou configuração, juntamente com vários templates, para que o gerador de código possa utilizar essas informações e criar uma ou mais camadas da aplicação.

Czarnecki (2002) oferece diversas perspectivas sobre a reusabilidade de código e como a programação generativa tem o potencial de simplificar a manutenção de software, prevenir erros humanos por meio do uso de padrões de

projeto, tornar o software mais consistente, traduzir abstrações para código, facilitar a padronização de projetos e, de modo geral, acelerar e reduzir o custo do ciclo de desenvolvimento de software.

Diante dos benefícios que a geração de código proporciona aos engenheiros de software e a todos os *stakeholders* envolvidos no ciclo de vida do software, torna-se imprescindível que a produção de um gerador de código leve em conta as diretrizes que melhor garantem a qualidade do produto, como a realização da documentação do software, a interação amigável do produto com seu usuário, a inclusão e o acompanhamento no processo de integração com o desenvolvimento de um software real (Czarnecki, 2002).

Além disso, faz-se importante destacar a presença dos geradores de código nas ferramentas utilizadas por desenvolvedores de software ao redor do mundo. Ambientes integrados de desenvolvimento como *IntelliJ, CLion, PyCharm, Goland* (*Jetbrains*[2]) e *Visual Studio Code*[3] fazem uso de geradores de código automático tanto para gerar métodos, propriedades e construtores quanto para facilitar o processo de refatoração do código.

Outrossim, iniciativas de documentação de software automática como *Swagger*[4], inicializadores de projetos com *frameworks* famosos, como *Spring Boot*[5] e *Ruby on Rails*[6], são ferramentas altamente disseminadas e que utilizam-se dos princípios da reutilização de código, da programação generativa e dos benefícios de redução de custos de desenvolvimento de software para cumprir seus propósitos.

Desse modo, evidencia-se que a criação de geradores de código tanto enriquece o estado da arte sobre a programação generativa quanto promove novos recursos para que a indústria de software torne-se ainda mais produtiva em seus processos por meio da garantia de que a força de trabalho estará voltada a resolução de problemas de maior importância e não a execução de tarefas repetitivas (Errington, 2003).

---

[2] https://www.jetbrains.com/
[3] https://code.visualstudio.com/
[4] https://swagger.io/
[5] https://spring.io/projects/spring-boot
[6] https://rubyonrails.org/

2.2 ARQUITETURA DE PORTAS E ADAPTADORES

O processo de planejamento de software desempenha um papel fundamental no sucesso de um projeto. Quando esse processo é deficiente, os produtos muitas vezes se tornam difíceis de manter para a equipe de desenvolvimento, e os custos associados podem se tornar insustentáveis para os proprietários a longo prazo. Isso ocorre porque, sem um planejamento adequado, erros ocultos no curto e médio prazo podem se tornar verdadeiros obstáculos no longo prazo (Robert, 2019).

Uma etapa crucial nesse processo de planejamento é a arquiteturação do software, onde são estabelecidas diretrizes de comunicação entre os componentes e a estrutura geral do sistema é definida com antecedência (Richards & Ford, 2020). Isso desempenha um papel essencial na prevenção de problemas futuros e na redução de custos. A arquitetura do software é o alicerce sobre o qual todo o desenvolvimento é construído (Garlan & Perry, 1995).

Uma abordagem que tem se mostrado eficaz na busca por uma arquitetura bem definida é o padrão arquitetural de Portas e Adaptadores, também conhecido como arquitetura hexagonal. Esse padrão enfatiza a separação clara entre a lógica de negócio da aplicação e o mundo externo, incluindo tecnologias de suporte. Para alcançar essa separação, utiliza-se o conceito de "portas" como interfaces de comunicação e "adaptadores" que implementam essas portas, conectando a aplicação ao mundo externo. Isso torna a aplicação mais flexível, facilita a manutenção e ajuda a evitar custos excessivos a longo prazo (Cockburn, 2005).

Sob uma outra perspectiva, Vernon (2013) apresenta o relacionamento entre o padrão arquitetural de portas e adaptadores e o conceito de *design* orientado a domínio (DDD). Tanto a arquitetura hexagonal quanto o DDD reconhecem a imprescindibilidade do conhecimento de negócio na construção de um software.

De um lado, a arquitetura hexagonal advoga pela separação do núcleo de negócios da aplicação do mundo externo, composto por tecnologias sujeitas a mudanças frequentes. Essa abordagem evita a confusão entre a lógica de negócios e as demandas tecnológicas instáveis e variáveis do sistema, conferindo ao time de desenvolvimento a possibilidade de troca de tecnologias com custo reduzido.

Por outro lado, o *design* orientado a domínio enfatiza que o manuseio cuidadoso das informações de negócios confere um imenso poder a qualquer equipe de desenvolvimento de software. A posse de uma linguagem ubíqua, capaz de descrever o negócio de maneira unificada e compartilhada, proporciona uma flexibilidade significativa no processo de tomada de decisão para engenheiros e arquitetos de software.

Aplicando de maneira explícita os conceitos de DDD à arquitetura de portas e adaptadores, Vernon (2013) destaca a importância da criação de um modelo de domínio sólido com o apoio de especialistas de domínio, uma vez que essa abordagem explora as capacidades do padrão arquitetural ainda mais, conferindo mais flexibilidade e dinamismo ao software construído.

2.3 DESENVOLVIMENTO DE SOFTWARE ORIENTADO A MODELO

Como discutido na seção anterior, a deficiência no planejamento de um software pode ter impactos significativos no ciclo de vida do produto, nos custos de manutenção e na viabilidade contínua do software. Grande parte dessas consequências está diretamente ligada à introdução da chamada "complexidade acidental" no software.

A complexidade acidental refere-se à complexidade que não é intrínseca ao problema que o software se propõe a resolver, mas que é adicionada devido a escolhas inadequadas de projeto, falta de planejamento ou implementações desnecessariamente complexas (Antinyan, 2020). Essa forma de complexidade pode tornar o software difícil de compreender, modificar e manter ao longo do tempo.

Um dos principais catalisadores da complexidade acidental reside na tradução problemática dos requisitos abstraídos por analistas de sistema em documentação formal e, finalmente, em código implementado por desenvolvedores de software (Balaban, Khitron & Maraee, 2022). Essa etapa crítica pode envolver uma categoria de problemas relacionados à transformação de modelos de software em soluções práticas. Por exemplo, a falta de alinhamento preciso entre os requisitos definidos e sua implementação no código pode levar a divergências e complexidades indesejadas.

Para mitigar esse desafio, uma estratégia eficaz é aumentar o nível de abstração da tecnologia utilizada no desenvolvimento de software. Isso significa

adotar abordagens que permitem que a equipe de desenvolvimento se concentre mais nos aspectos conceituais e de alto nível do problema a ser resolvido, em vez de se perder em detalhes técnicos complexos. Ao elevar o nível de abstração, é possível reduzir a complexidade acidental, tornar o software mais sustentável e, em última instância, garantir a competitividade do produto de software.

Nesse sentido, Milicev (2009) apresenta o desenvolvimento de software orientado a modelo como um recurso valioso na luta contra a complexidade acidental no desenvolvimento de software. Esta abordagem coloca a ênfase na criação de modelos abstratos que servem como a espinha dorsal do sistema, permitindo que os desenvolvedores se concentrem nos conceitos e requisitos de alto nível. Ao elevar o nível de abstração, o desenvolvimento orientado a modelo simplifica a tradução dos requisitos em implementações práticas e, assim, reduz a probabilidade de complexidade acidental.

Os benefícios do desenvolvimento orientado a modelo são significativos. Esta abordagem não apenas melhora a clareza e a manutenibilidade do software, mas também acelera o ciclo de desenvolvimento. Isso ocorre porque a criação de modelos bem definidos permite a geração automática de código a partir desses modelos, economizando tempo e reduzindo erros manuais. Além disso, o MDD facilita a colaboração entre equipes de desenvolvimento e partes interessadas não técnicas, pois os modelos são mais acessíveis e compreensíveis do que código escrito em linguagens de programação.

Portanto, ao adotar o desenvolvimento orientado a modelo, as organizações podem não apenas reduzir a complexidade acidental, mas também aumentar a eficiência, a qualidade e a competitividade de seus produtos de software em um mercado em constante evolução. No entanto, vale ressaltar que a adoção bem-sucedida do MDD requer uma compreensão sólida dos princípios subjacentes e o investimento em ferramentas e treinamento adequados para a equipe de desenvolvimento.

Outrossim, o MDD possui recursos valiosos para a programação generativa implementada nesta pesquisa. A utilização de abstrações para a geração de código é uma prática extremamente comum (Paolone et al, 2020). Isso ocorre por meio da utilização de arquivos de definição ou representações diagramadas de fluxos e estruturas do software. Além disso, ao utilizar modelos abstratos, a geração

de código se torna mais eficiente, uma vez que é possível automatizar parte do processo.

2.4 MODELO CLIENTE-SERVIDOR

O modelo cliente-servidor é uma arquitetura amplamente empregada no desenvolvimento de software que desempenha um papel fundamental na organização e distribuição de recursos e funcionalidades. Essa abordagem é essencial para a construção de sistemas escaláveis, eficientes e interconectados, e é comumente adotada em uma variedade de contextos, desde aplicativos web e móveis até sistemas empresariais complexos (Kumar, 2019).

Nesta arquitetura de sistema, as responsabilidades são divididas claramente entre cliente e servidor. Enquanto o cliente é responsável por requisitar recursos e aguardar que esses recursos sejam recebidos, o servidor é responsável por processar tais requisições, operar sobre elas e, finalmente, emitir uma resposta apropriada para o cliente. Essa divisão de tarefas é essencial para o funcionamento eficaz do modelo cliente-servidor (Tanenbaum, 2007).

De um lado, o cliente, frequentemente a interface gráfica visível para os usuários finais, tem a importante função de apresentar informações de forma acessível, coletar entradas do usuário e encaminhar solicitações para o servidor. Ele é responsável por criar uma experiência interativa para o usuário, exibindo dados de maneira compreensível e transmitindo solicitações para o servidor, onde ocorre o processamento necessário. Essa clara separação de responsabilidades torna possível a atualização da interface de usuário sem afetar a lógica, permitindo a evolução da experiência do usuário de forma independente.

Por outro lado, o servidor desempenha um papel central na execução das operações subjacentes. Ele é encarregado de realizar a lógica de negócios, processar as solicitações do cliente, acessar e gerenciar dados, bem como preparar e enviar respostas adequadas. Essa camada do sistema é onde ocorre o processamento pesado, manipulação de dados e aplicação das regras de negócios.

Uma das principais vantagens dessa arquitetura é a capacidade de escalabilidade. À medida que as necessidades do sistema aumentam, mais clientes podem ser adicionados, e servidores adicionais podem ser implantados para lidar com a demanda crescente. Isso proporciona uma solução eficaz para sistemas que

precisam lidar com um grande número de usuários simultaneamente (Mühlbauer, 2020).

O modelo cliente-servidor é encontrado em muitos aspectos da vida cotidiana digital. Por exemplo, em aplicativos web, o navegador atua como cliente, enquanto o servidor web responde às solicitações. Em aplicativos móveis, o aplicativo no dispositivo do usuário é o cliente, e os servidores de *back-end* fornecem dados e funcionalidade. Além disso, em sistemas empresariais complexos, essa arquitetura permite a divisão de tarefas entre dispositivos de usuário e servidores de alto desempenho (Tanenbaum, 2007).

Mesmo com o surgimento de novas abordagens arquitetônicas, o modelo cliente-servidor continua a ser uma pedra angular do desenvolvimento de software. Sua capacidade de separar claramente as responsabilidades entre as camadas de cliente e servidor é essencial para a criação de sistemas robustos, escaláveis e eficientes. Como resultado, o modelo cliente-servidor desempenha um papel crucial na entrega de experiências de usuário aprimoradas em um mundo cada vez mais interconectado.

2.5 ARQUITETURA DE TRANSFERÊNCIA DE ESTADO REPRESENTACIONAL

A arquitetura de transferência de estado representacional, do inglês *Representational State Transfer* (*REST*), surge como um paradigma orientador para o *design* eficiente e escalável de serviços *web*, tendo como ponto chave a busca pela simplicidade, pois, em seu contexto de surgimento, a estruturação de serviços *web* era caracterizada pela complexidade e verbosidade (Fielding, 2000).

Fundamentada em quatro princípios essenciais, a proposta *REST* visa simplificar a interação entre sistemas distribuídos, centralizando o *design* da comunicação em recursos. A partir disso, torna-se possível a identificação de entidades por meio de conjuntos de identificadores uniformes de recursos (*URIs*), proporcionando consistência no processo de comunicação entre clientes e servidores (Masse, 2011).

Nesse contexto, o estado representacional apresenta-se como princípio fundamental para o entendimento da arquitetura *REST*. Este componente do paradigma é responsável pela priorização da transferência de estado da aplicação por meio das representações de recursos. Em outras palavras, ao realizar uma

requisição a um servidor *REST*, o cliente não apenas recebe os dados solicitados, mas também a representação do estado associado a esses dados, fornecendo um contexto mais abrangente.

Nesse sentido, o princípio da manipulação de recursos por representações enfatiza que as interações entre clientes e servidores na arquitetura REST ocorrem por meio da manipulação de representações dos recursos. Quando um cliente solicita um recurso ao servidor, ele recebe uma representação do estado atual do recurso, geralmente em um formato como *JavaScript Object Notation (JSON[7])* ou *Extensible Markup Language (XML[8])*. As alterações no estado do recurso são realizadas pelo cliente ao manipular e enviar representações do recurso de volta para o servidor. Esse modelo baseado em representações torna as operações uniformes e independe da implementação interna dos recursos no servidor.

Sistemas distribuídos que possuem a independência do estado armazenado no servidor e foram desenhados para se preocupar com a significação da representação dos recursos por meio de *URIs*, sendo este mais um princípio *REST,* e das ações das requisições recebidas são frequentemente apelidados de *RESTful*. Cada requisição é autocontida, o que significa que o servidor não precisa manter o contexto do cliente entre solicitações. Isso não apenas simplifica a implementação do servidor, mas também contribui significativamente para a escalabilidade do sistema, uma vez que cada requisição contém todas as informações necessárias para ser processada de forma eficiente (Richardson & Ruby, 2008).

Ao considerar o princípio de *Hypermedia as the Engine of Application State* (HATEOAS), nota-se uma inovação que vai além da simples troca de dados, redefinindo dinamicamente a interação entre clientes e servidores em ambientes RESTful. HATEOAS permite que as representações dos recursos, além de conterem dados, forneçam também informações sobre as ações ou recursos relacionados disponíveis no contexto da aplicação (Fielding, 2000). Ao incorporar links dinâmicos nas respostas enviadas aos clientes, incentiva-se a descoberta e a exploração do sistema de maneira mais autônoma. Essa abordagem não apenas simplifica a

[7] https://www.json.org/json-en.html
[8] https://www.xml.com/

implementação do cliente, reduzindo a dependência de documentação extensiva, mas também promove uma adaptabilidade contínua.

3 TRABALHOS RELACIONADOS

Visando à apresentação de pesquisas correlatas ao tema deste trabalho, esta seção é composta por uma sequência de breves análises sobre trabalhos semelhantes a este. A partir da comparação entre os acervos apresentados a seguir, é possível identificar semelhanças e distinções entre o estado da arte e a proposta aqui desenvolvida.

Sob a luz da construção de software baseada em modelos pré-definidos, Atencio et al (2021) apresentam uma ferramenta que busca acelerar os processos de prototipação de módulos de software e projetos completos com diferentes necessidades e propósitos baseada na ideia de que há muitos recursos compartilhados entre projetos que podem ser simplificados ou abstraídos por meio da geração automática de código.

Através de uma interface gráfica de usuário (GUI), os colaboradores do projeto proporcionam a usuários de sua ferramenta uma forma rápida de gerar código *JavaScript*[9] estruturado por meio dos padrões arquiteturais de software *Model-View-Controller* (MVC) e *Model-View-ViewModel* (MVVM) e separado por meio de módulos, possibilitando ao consumidor do recurso a customização posterior do código gerado pela ferramenta.

Analogamente, Paolone *et al* (2020) promoveram uma ferramenta de tradução automática de modelos estruturais e comportamentais de diagramas da Linguagem de Modelagem Unificada (UML[10]) para código *Java*[11] baseado na arquitetura de software MVC. Com sua abordagem de criação de software direcionada a modelos, os autores buscam tanto realizar a otimização do ciclo de desenvolvimento de software quanto tornar possível a customização do código a partir da base criada por sua ferramenta. Sendo assim, eles implementaram sua solução tendo em mente a necessidade de alterações manuais no programa gerado pelo computador.

Comparativamente a ideia de geração de código altamente versátil baseada em padrões arquiteturais de software anteriormente citada, Domingos (2018) trás uma abordagem mais restritiva quanto ao nicho de sua ferramenta geradora de código. Destrinchando a abordagem tomada, percebe-se a utilização de

[9] https://www.javascript.com/
[10] https://www.uml.org/
[11] https://www.java.com/

interpretadores de XML para comandar, por meio de controladores lógicos programáveis, veículos de ambientes subterrâneos, utilizando código *C#*[12] gerado automaticamente.

De modo similar, Nguyen (2018) propõe um modelo interpretador de XML e que produz um código para aplicações com controladores lógicos programáveis. Porém sua abordagem se aproxima do trabalho aqui desenvolvido em um nível mais profundo que o anterior, pois a produção de código automática em C# promovida pela pesquisa faz uso de um padrão arquitetural de software, MVVM, para garantir uma estrutura replicável e familiar para a customização do código em casos de necessidade.

Diante da importância da representação dos estados e comportamentos das entidades de um sistema, Durai et al (2022) propuseram-se a criar uma ferramenta que interpreta diagramas UML de sequência e de atividade e produz definições de classes e seus comportamentos em *Java*, objetivando a simplificação e redução de erros durante o ciclo de desenvolvimento de software.

Sob um outro espectro, Huning & Pulvermueller (2021) apresentam uma ferramenta de geração de código cujo objetivo primário enriquece a lista de benefícios adquiridos por meio da automatização do processo de desenvolvimento de software. Em seu domínio, os autores fazem uso da produção automática de código para transformar especificações de mecanismos de segurança em UML em código Java, protegendo o software da introdução de erros humanos na tradução da especificação em código.

A fim de garantir eficiência na produção de contratos inteligentes de Blockchain, Jurgelaitis et al (2022) e Mars et al (2023) optaram por utilizar a geração automática de código como ferramenta aceleradora do processo. Respectivamente, os autores trabalharam a tradução de diagramas UML e diagramas *Business Process Model And Notation* (BPMN[13]) em código Solidity[14], apoiando-se na ideia de produção de software baseada em modelo, onde partes de um sistema são replicáveis em outros.

Trazendo uma nova perspectiva para as aplicações da produção automática de código, Monteiro, Pereira & Barreto (2020) e Ramírez-Noriega et al

[12] https://learn.microsoft.com/pt-br/dotnet/csharp/
[13] https://www.bpmn.org/
[14] https://soliditylang.org/

(2022) decidiram utilizar esse recurso como ferramenta para a transmissão do aprendizado no campo da computação. Enquanto os primeiros autores trabalharam a ideia da tradução de modelos *Finite State Process* (FSP) em código Java, visando o ensino de sistemas concorrentes, os outros autores propõem uma experiência de conexão entre diagramas entidade-relacionamento (ER) e uma aplicação TypeScript[15], objetivando o aprendizado do aluno por meio da visualização de mudanças no software a partir do modelo.

A Quadro 1 exibe os aspectos mais relevantes dos trabalhos supracitados no contexto desta pesquisa. Sua responsabilidade consiste em fornecer informações que identificam o trabalho e tratam sobre as entradas e saídas do sistema de geração de código promovido bem como mostram o uso ou não de padrões arquiteturais de software na abordagem adotada pelos autores.

Quadro 1 – Trabalhos relacionados e parâmetros de contexto da pesquisa.

Autores	Ano	Entrada do sistema	Saída do sistema	Arquitetura de software
Atencio *et al*	2021	GUI	*JavaScript*	MVC/MVVM
Domingos	2018	XML	*C#*	—
Durai *et al*	2022	UML	*Java*	—
Huning & Pulvermueller	2021	UML	*Java*	—
Jurgelaitis *et al*	2022	UML	*Solidity*	—
Mars *et al*	2023	BPMN	*Solidity*	—
Monteiro, Pereira & Barreto	2020	FSP	*Java*	—
Nguyen	2018	XML	*C#*	MVVM
Paolone *et al*	2020	UML	*Java*	MVC
Ramírez-Noriega *et al*	2022	ER	*TypeScript*	MVC
Proposta	2023	CLI / YAML	*Golang*	Arquitetura Hexagonal

Fonte: Próprio autor.

[15] https://www.typescriptlang.org/

Em análise comparativa, é observado que todas as ferramentas, incluindo a ferramenta proposta, compartilham a abordagem de automação na geração de código por meio de mecanismos de entrada e possuindo como principal objetivo a aceleração do desenvolvimento de software. Apesar disso, as diferenças entre as ferramentas existentes e a solução proposta existem. Enquanto Atencio *et al* (2021) e Paolone *et al* (2020) concentram-se na geração de código *JavaScript* com ênfase nos padrões arquiteturais MVC e MVVM, respectivamente, Toth destaca-se ao adotar a linguagem *Golang* para gerar código personalizado destinado a aplicações baseadas na arquitetura hexagonal. Além disso, a ferramenta proposta foi desenhada para ser estendida e suportar outros padrões arquiteturais caso sejam implementados.

Sob outra perspectiva, a restrição de Domingos (2018) a um nicho específico, usando XML para controlar veículos subterrâneos com código C#, contrasta com a abordagem mais abrangente da Toth, que se concentra em *APIs*. Nguyen (2018) e a solução proposta compartilham a abordagem de interpretação de arquivos de configuração, mas Toth destaca-se ao oferecer uma interface de linha de comando para emissão direta de comandos em adição a outra funcionalidade de leitura e interpretação. Além disso, Durai *et al* (2022) focam na interpretação de diagramas UML para *Java*, enquanto o sistema proposto escolhe *Golang* e a arquitetura hexagonal. Outras ferramentas, como as de Huning & Pulvermueller (2021), Jurgelaitis *et al* (2022), e Mars *et al* (2023), concentram-se em traduzir diagramas UML para *Java* ou *Solidity*.

Apesar de a interpretação de diagramas UML ser uma fonte confiável de informações e promover uma forma eficiente de produção de código, essa abordagem é limitada, pois o escopo de diagramas está condicionado a representar apenas estruturas ou apenas comportamentos específicos do sistema. A Toth supera essa limitação ao adotar a arquitetura hexagonal, proporcionando uma visão mais holística e flexível na geração de código. A escolha da linguagem Golang e a capacidade de estender suporte a outros padrões arquiteturais conferem à solução proposta maior adaptabilidade, permitindo aos desenvolvedores escolherem a abordagem que melhor se adequa às necessidades de seus projetos.

4 METODOLOGIA

O ponto de partida para esta investigação foi a definição de um protocolo de pesquisa abrangente. Foram detalhados os parâmetros da pesquisa, variáveis-chave e o escopo da análise. Este protocolo atuou como bússola, orientando cada passo subsequente e proporcionando um alicerce sólido para a condução da pesquisa.

Assim procedeu-se à aplicação do protocolo estabelecido, permitindo a coleta de dados substanciais e a observação de padrões emergentes. A aplicação prática não apenas validou o protocolo, mas também enriqueceu a compreensão do fenômeno em estudo, alimentando as próximas fases do processo metodológico.

Com base nas análises dos materiais obtidos durante a aplicação do protocolo, uma ferramenta de geração de código foi desenvolvida. Esta etapa não apenas representa a aplicação prática da pesquisa, mas também destaca a interconexão entre teoria e implementação. O processo de desenvolvimento será detalhadamente explorado, destacando as decisões cruciais que moldaram a funcionalidade da ferramenta.

4.1 PROTOCOLO DE PESQUISA

Com o propósito de contextualizar os estudos relacionados à geração de código com base em arquitetura de software e fornecer o embasamento teórico fundamental para a compreensão deste trabalho, foram conduzidos procedimentos de coleta de informações. Neste contexto, realizou-se uma pesquisa preliminar com o objetivo de identificar os termos-chave associados ao tema, tanto em inglês quanto em português. De modo subsequente, foi realizada uma categorização protocolada, estabelecendo assim uma base sólida para a seleção dos conceitos essenciais e dos trabalhos relacionados.

Através da pesquisa preliminar conduzida, coletaram-se os termos-chave a serem utilizados na pesquisa, bem como seus sinônimos. Nesse sentido, foram estabelecidos alguns termos-chave norteadores, estes foram: "geração automática de código", "automatic code generation", "arquitetura de software", "software architecture". Além disso, o processo preliminar de pesquisa consultou a base de dados do *Google Scholar*[16], pois a indexação genérica da plataforma pôde contribuir

[16] https://scholar.google.com/

tanto para o estabelecimento das palavras-chave quanto para a determinação das bases de dados que foram consultadas nas etapas posteriores da pesquisa, sendo estas: *Google Scholar*, *IEEE*[17], *Springer*[18] e Capes Periódicos[19].

Com base nas definições da etapa anterior, a expressão geral de busca (*String* de busca) definida e aplicada à pesquisa foi: (("geração automática de código" OR "automatic code generation") AND ("arquitetura de software" OR "software architecture")). Após isso, foram definidas as estratégias de busca específicas para cada base de dados:

- ***Springer* e *Google Scholar:*** (("geração automática de código" | "automatic code generation") & ("arquitetura de software" | "software architecture"))
- ***IEEE:*** (("geração de código" OR "code generation" OR "geração automática de código" OR "automatic code generation") AND ("arquitetura de software" OR "software architecture"))
- **Capes Periódicos:** ("geração de código" OR "code generation" OR "geração automática de código" OR "automatic code generation") AND ("arquitetura de software" OR "software architecture")

Subsequente à execução das expressões de busca sobre cada base de dados, foram obtidos os resultados apresentados na Tabela 1.

Tabela 1 – Resultados iniciais de pesquisa com expressões de busca.

Base de dados	Resultados
Springer	500
IEEE	243
Google Scholar	3700
Capes Periódicos	162

Fonte: Próprio autor.

[17] https://ieeexplore.ieee.org/
[18] https://link.springer.com/
[19] https://www-periodicos-capes-gov-br.ezl.periodicos.capes.gov.br/

Consecutivamente, foram determinadas os critérios de inclusão e exclusão dos trabalhos encontrados nas bases de dados selecionadas.

Critérios de inclusão:

- Trabalhos que se aproximam do domínio da geração automática de código baseado em arquitetura de software.
- Trabalhos que estejam escritos no idioma inglês ou português.
- Trabalhos que reforçam que o conteúdo desta pesquisa é relevante para o contexto do desenvolvimento de software.
- Trabalhos que contribuem para a concretização dos objetivos específicos desta pesquisa.
- Trabalhos que contribuem para o embasamento teórico e contextualização das terminologias utilizadas nesta pesquisa.

Critérios de exclusão:

- Trabalhos com data de publicação inferior ao intervalo de cinco anos.
- Trabalhos duplicados.
- Trabalhos não revisados em pares.

A fim de refinar o conteúdo componente deste trabalho, os critérios supracitados foram aplicados para cada plataforma. Em primeira instância, a aplicação de filtros de tempo foi aplicada para assegurar a pertinência dos estudos, já que o campo de desenvolvimento de software é volátil e necessita de estar atualizado sobre as melhores e mais aplicadas práticas de mercado (Bhavsar, Shah e Gopalan, 2019).

Em seguida, foi aplicada uma abordagem de seleção de trabalhos componentes baseada na relevância do conteúdo para a temática. Selecionaram-se trabalhos a partir de um método gradativo de análise textual: partiu-se da análise do título e foi-se para a análise de resumo e introdução, até a leitura completa do trabalho. Desse modo, foi possível realizar a seleção e a inclusão dos trabalhos, resultando no quantitativo presente na Tabela 2.

Tabela 2 – Trabalhos incluídos após aplicação dos critérios de inclusão e exclusão.

Base de dados	Resultados	Trabalhos incluídos
Springer	500	6
IEEE	243	2
Google Scholar	3700	8
Capes Periódicos	162	4

Fonte: Próprio autor.

4.2 ASPECTOS DO DESENVOLVIMENTO DA FERRAMENTA

Na execução do desenvolvimento da ferramenta, adotou-se o Kanban (Junior & Godinho Filho, 2010) como metodologia ágil central. A implementação bem-sucedida desta metodologia para o desenvolvimento da ferramenta incorporou ferramentas adicionais de gerenciamento, como o Trello[20] que, no contexto do Kanban, proporcionou uma visão geral e detalhada do fluxo de trabalho, permitindo uma gestão eficaz das tarefas em tempo real.

No âmbito do planejamento temporal, a alocação adequada de recursos e a definição de metas realistas desempenharam papéis cruciais. A determinação de prazos claros e alcançáveis contribuiu para a eficiência geral do processo, promovendo a conclusão de marcos importantes ao longo do desenvolvimento da ferramenta.

A gestão eficaz do tempo não foi isolada; foi intrinsecamente entrelaçada com o planejamento do software. A definição clara de requisitos, a identificação de dependências e a alocação de tarefas específicas foram fundamentais para mitigar riscos e garantir uma implementação suave. O processo de planejamento de software não apenas guiou a alocação de recursos, mas também atuou como uma estrutura orientadora para as atividades de desenvolvimento.

A importância desses aspectos durante o desenvolvimento da ferramenta reside na capacidade de assegurar uma progressão coesa e controlada do projeto. O Kanban, combinado com um planejamento de tempo e software criterioso, permitiu uma abordagem sistemática e adaptativa, culminando na criação da

[20] https://trello.com/

ferramenta em questão. A atenção a esses aspectos não apenas facilitou a entrega pontual, mas também contribuiu para a qualidade intrínseca da solução desenvolvida.

5 SOLUÇÃO PROPOSTA

Neste capítulo, abordam-se as características específicas do processo de desenvolvimento do utilitário denominado Toth, concebido com o objetivo principal de otimizar a criação de software para aplicações estruturadas na arquitetura hexagonal. A proposta central do Toth é acelerar o processo de desenvolvimento por meio da geração automatizada de código personalizado, permitindo que os usuários emitam comandos diretamente por meio de uma interface de linha de comando (CLI). De modo geral, a ferramenta foi constituída para a construção de aplicações de interface de programação (*API*) utilizando o padrão REST.

Inicialmente, é pertinente enfatizar que a audiência primária desta ferramenta compreende desenvolvedores, engenheiros e arquitetos de software. Nesse contexto, os padrões adotados para a experiência do usuário levam em consideração a expectativa de que os usuários possuam conhecimento prévio em ferramentas de linha de comando.

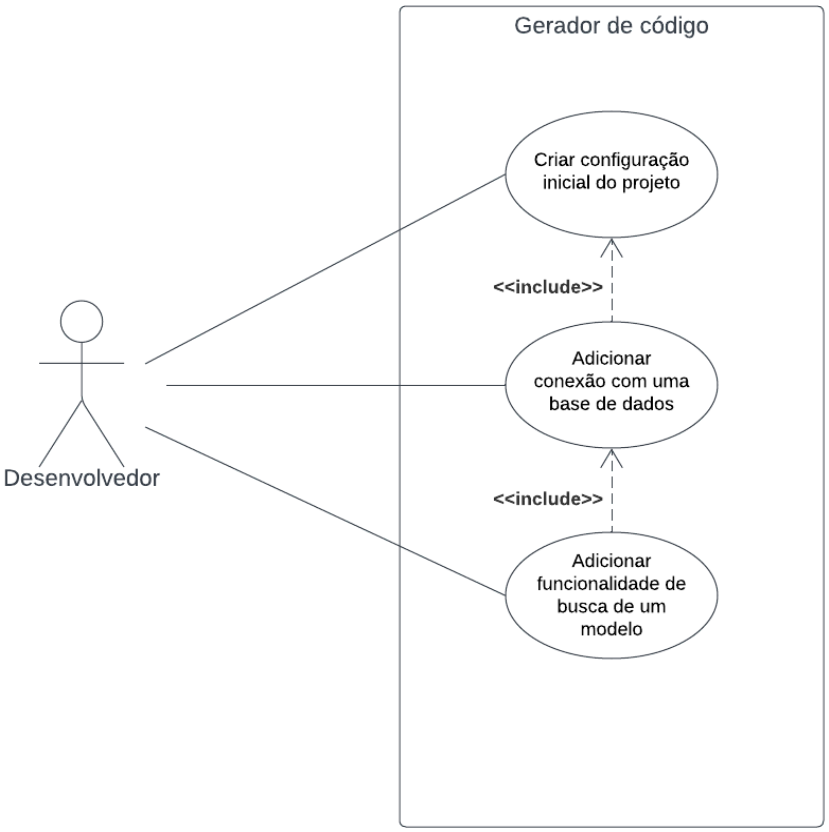
Dessa maneira, a interação foi desenhada para se alinhar com a familiaridade desses profissionais, proporcionando uma experiência eficiente e produtiva para aqueles que estão habituados a utilizar recursos similares em seu cotidiano de desenvolvimento.

Após a consolidação da linguagem de programação a ser utilizada no processo de desenvolvimento e da determinação clara do propósito geral da ferramenta, deu-se início ao processo de planejamento do software.

5.1 ESTABELECIMENTO DE REQUISITOS

Durante o processo de planejamento do software, a definição cuidadosa dos requisitos fundamentais desempenhou um papel crucial para orientar o desenvolvimento da **Toth**. Tais requisitos foram estabelecidos para assegurar que a ferramenta estivesse alinhada com as demandas específicas de aplicações web estruturadas na arquitetura hexagonal. A representação visual dos casos de uso, conforme ilustrado na Imagem 1, serve como um guia tangível que direcionou o desenvolvimento subsequente da aplicação. Esses casos de uso representam cenários práticos nos quais a ferramenta deve ser capaz de oferecer funcionalidades específicas, integrando-se de forma coesa com as exigências reais de desenvolvimento de software.

Imagem 1 – Diagrama de casos de uso para Toth.



Fonte: Próprio autor.

A partir da representação dos casos de uso, da análise dos trabalhos relacionados e das ferramentas utilizadas na indústria de software, foram produzidos os requisitos funcionais, representados na Quadro 2, e os requisitos não funcionais, representados na Quadro 3.

Quadro 2 – Requisitos funcionais da Toth.

ID	Descrição	Prioridade	Data de Criação
RF01	A ferramenta deve ser capaz de criar uma aplicação a partir de um arquivo de configuração *YAML.*	ALTA	26/10/2023

RF02	A ferramenta deve aceitar uma instrução de linha de comando que configura as bases de um projeto.	ALTA	26/10/2023
RF03	A ferramenta deve aceitar uma instrução de linha de comando que configura uma conexão com uma base de dados em aplicação existente e gerada pela ferramenta.	ALTA	27/10/2023
RF04	A ferramenta deve aceitar uma instrução de linha de comando responsável por produzir uma funcionalidade de buscar um modelo.	ALTA	27/10/2023
RF05	A ferramenta deve ser capaz de preservar código gerado anteriormente pela ferramenta ou alterações escritas pelo usuário.	ALTA	27/10/2023
RF06	A ferramenta deve ser capaz de cancelar as operações de geração de código caso ocorra um erro.	ALTA	27/10/2023

Fonte: Próprio autor.

Quadro 3 – Requisitos não funcionais da Toth.

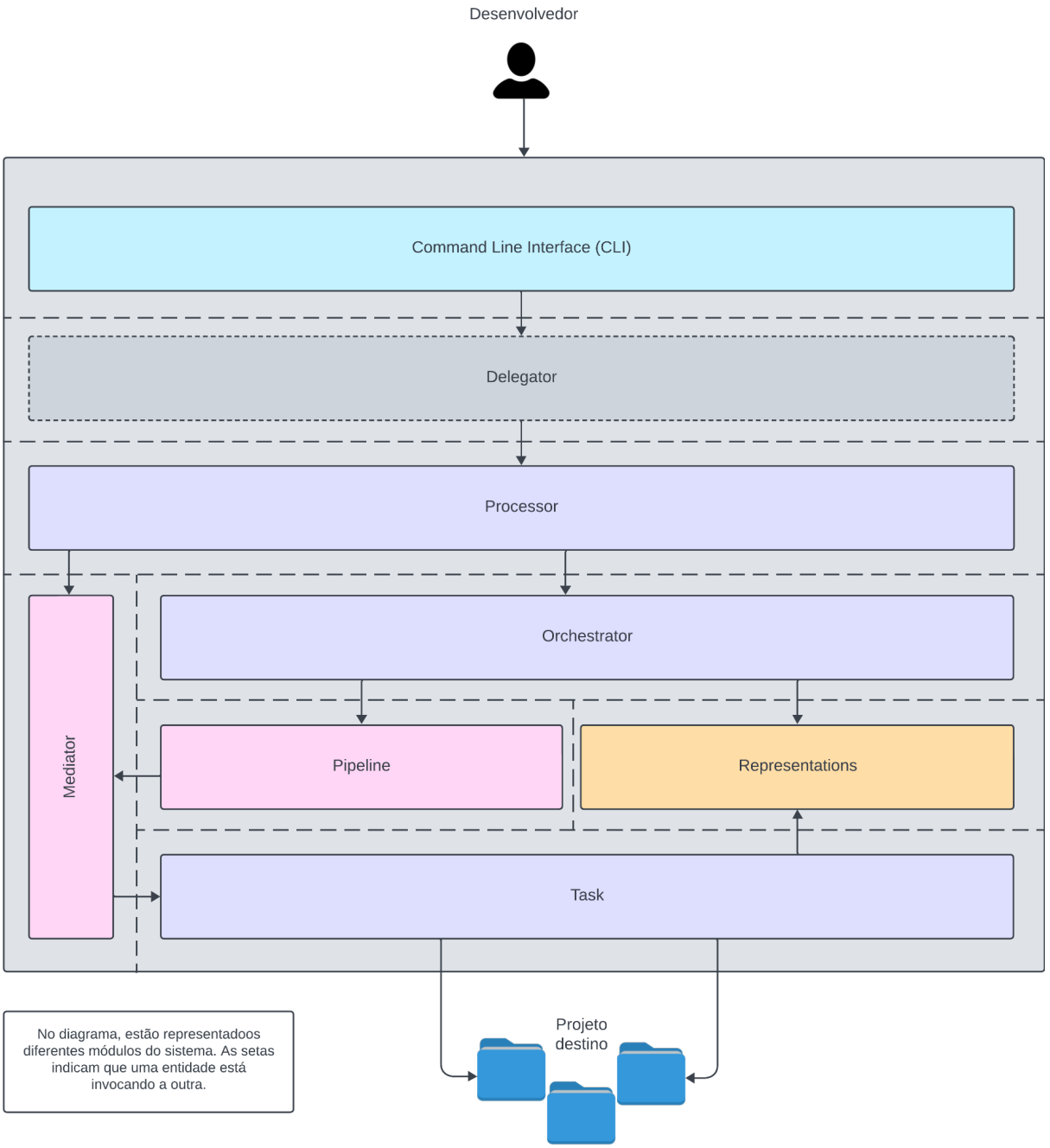
ID	Descrição	Categoria	Data de Criação
RNF01	A ferramenta deve ser capaz de executar em qualquer sistema operacional	Portabilidade	27/10/2023
RNF02	A ferramenta deve transmitir feedback informativo para os usuários enquanto gera código.	Experiência do usuário	27/10/2023

Fonte: Próprio autor.

5.2 VISÃO LÓGICA DA APLICAÇÃO

Considerando as concepções do domínio da geração de código automática, as particularidades da linguagem *Go* e os requisitos consolidados anteriormente, os componentes destacados na Imagem 2, que proporciona uma visão lógica da aplicação, foram desenvolvidos com responsabilidades específicas.

Imagem 2 – Representação lógica da ferramenta.



Fonte: Próprio autor.

O primeiro módulo mostrado na imagem, CLI, possui como responsabilidade a interação direta com o desenvolvedor. Neste módulo é feita a captação da intenção do usuário por meio da interface de linha de comando. Os dados digitados (i.e. comandos, subcomandos, opções, parâmetros) são validados a fim de garantir que sigam a sintaxe dos comandos definidos pelo Toth. Feita a

validação, as informações precisam ser estruturadas e enviadas para as camadas inferiores responsáveis pelo processamento.

Existem múltiplas abordagens para lidar com comandos dos usuários. Por exemplo, poderiam ser criados métodos para cada tipo de ação que esperasse os dados por parâmetro ou que recebessem um objeto de transferência de dados para identificar cada tipo de funcionalidade. Poderiam ainda haver classes (ou *structs*) responsáveis por um tipo de comando e estas receberiam todos os parâmetros necessários no momento da instanciação.

Embora as abordagens citadas anteriormente sejam válidas, em alguns casos, como em um sistema com muitas funcionalidades e subcomandos, faz-se necessário buscar alternativas para gerenciar a passagem e gerenciamento de comandos, evitando assim a repetição de código, reduzindo as classes no projeto e possibilitando a escalabilidade à longo prazo.

Tendo isso em vista, foi aplicado o *Command Pattern*, que permite a estruturação dos dados de modo que exista um único tipo de objeto genérico representando todos os comandos (Gamma et al, 1994). Neste objeto, estão contidos atributos sobre a ação requisitada pelo ator, assim como parâmetros para a execução. Desta forma, cada linha digitada pelo usuário é transformada em um comando genérico que é passado para o *processor* (vide Imagem 2).

Visando as boas práticas e requisitos de qualidade já citados anteriormente, foi criado um intermediador entre os comandos gerados no módulo CLI e o processador. Nesta etapa, foi aplicado o *Delegation Pattern* (padrão de delegação). Este padrão de projeto define a utilização de um *delegator* (atribuidor) e um conjunto de *delegates* (delegados) que compartilham a mesma interface pública, mas possuem especializações diferentes. Sendo assim, em vez de gerar uma dependência direta entre a entidade que quer invocar uma ação e a parte requerida, o delegator é invocado, recebe a ação desejada e atribui a execução a um *delegate* (Gamma et al, 1994). No caso do Toth, o *Delegator* é responsável por receber um comando recém construído e passá-lo para que o processador execute a requisição.

As estruturas invocadas pelo *Delegator*, os *Processors* (ou processadores), são responsáveis por ter o domínio de todas as entidades necessárias para a geração de código na sintaxe da linguagem Go utilizando a arquitetura de portas e adaptadores, ficando responsável por instanciar as *Tasks* (tarefas) necessárias para a geração e código para o projeto.

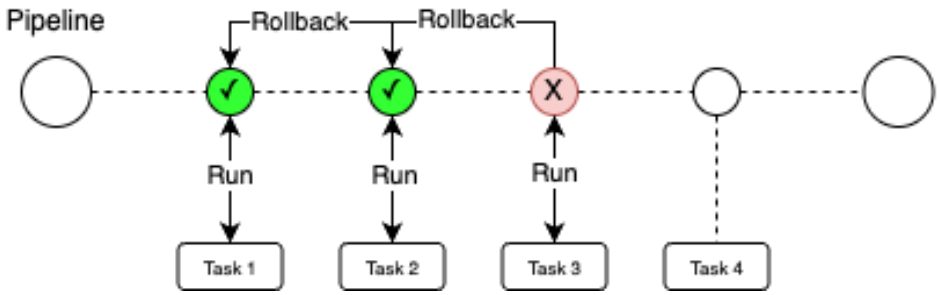
As *Tasks* (tarefas) consistem em objetos isolados responsáveis por realizar pequenas ações no âmbito da configuração do projeto e geração de código. As tarefas podem ser segregadas em *queries* (consultas), que fazem buscas a respeito do estado do projeto, e em *commands* (comandos), que possuem como objetivo a realização de alterações. Para cada tipo de *Task* existe um objeto *Handler* correspondente cuja função é executar as instruções da *Task* ou executar instruções reversas para retornar o projeto ao estado anterior à execução (*Rollback*). O processador agrupa as instâncias das tarefas e seus respectivos *handlers* através da implementação de um *Mediator* (mediador), que utiliza conceitos estabelecidos pelo *Mediator Pattern* (Gamma et al, 1994).

Além das tarefas, o processador também instancia uma coleção de orquestradores (*Orchestrator*) e delega para um deles o comando recebido. Cada orquestrador recebe uma referência do *Mediator*, por meio do qual é possível executar os *Handlers* das *Tasks*. Os orquestradores também atuam fazendo as validações necessárias que precedem a execução de uma *Task*.

Para garantir um resultado final consistente, cada orquestrador enfileira as tarefas através da instanciação e configuração de uma *Pipeline*. A implementação da *Pipeline* teve como inspiração em padrões como os *SAGA*, utilizados para manter a coesão no momento de realizar transações de longa duração em aplicações distribuídas (Richardson, 2018), além do *Pipeline Design Pattern*, definido para situações em que é preciso executar ações de forma sucessiva (Kleppmann, 2017).

A finalidade principal da *Pipeline* é receber *Tasks* para serem executadas sucessivamente e, em caso de erro na execução, notificar aos handlers anteriores que o processo já finalizado precisa ser revertido (vide Imagem 3).

Imagem 3 – Representação do funcionamento da *Pipeline.*

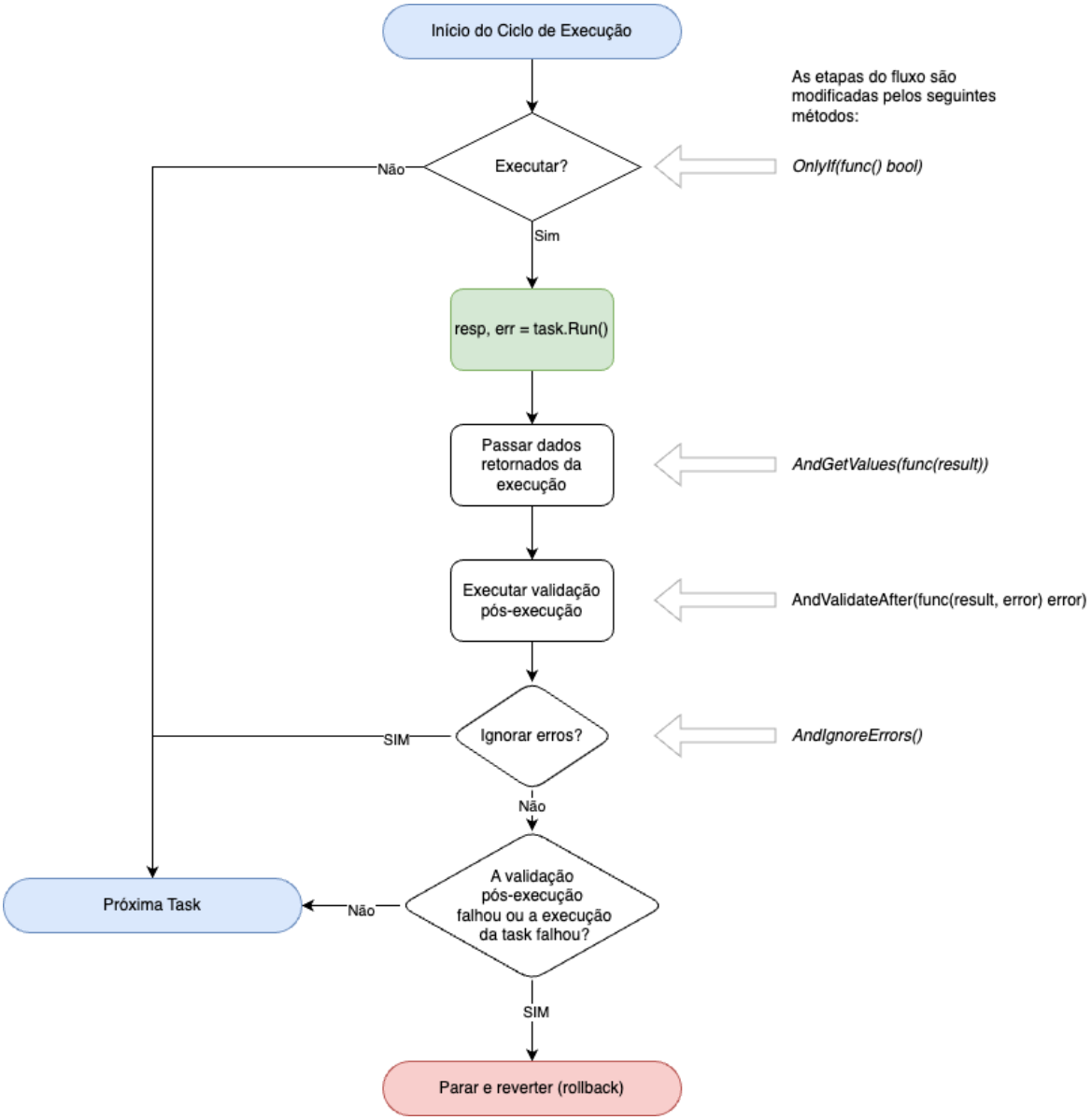


Fonte: Próprio autor.

A implementação possibilita ainda que as *Tasks* adicionadas adquiram novos comportamentos importantes para o utilizador da *Pipeline*, como condicionais para execução de uma *Task* ou modificação do tratamento de erro para uma etapa específica do processo. Este comportamento é possibilitado através da implementação de conceitos estabelecidos no *Decorator Pattern*, no qual um objeto é embrulhado em outro com funcionalidades adicionais (Gamma et al, 1994).

Desta forma, é possível criar um ciclo de execução, para cada tarefa, modificável de acordo com parâmetros fornecidos no momento da adição da *Task* à *Pipeline*. O ciclo de execução de uma *Task* dentro de uma pipeline é demonstrado no diagrama a seguir (Imagem 4). Também é possível verificar, do lado direito da imagem, quais métodos modificam cada comportamento.

Imagem 4 – Ciclo de execução de uma *Task* dentro da *Pipeline*.



Fonte: Próprio autor.

A utilização das *Pipelines* num *orchestrator* garante que o resultado final do *software* seja consistente, tendo em vista que a execução é interrompida em caso de erro e as etapas já executadas são revertidas. Além disso, permite que os orquestradores possuam um código mais simplificado à medida que dispensa a necessidade de realizar tratamentos de erros nos métodos dos orquestradores após a execução de cada *Task*. Segue abaixo o exemplo de um orquestrador utilizando a *Pipeline* (Código-fonte 1).

Código-fonte 1 – Exemplo de orquestrador utilizando um *Pipeline*.

```
type Orchestrator struct {
   mediator mediation.Mediator
}

func(o Orchestrator) Orchestrate(c context.Context,
                                 request *processor.Request) error {
   pipeline := pipeline.New(o.mediator)

   pipeline.
      AddTask(file.CreateFileCmd{
         Path:  filepath.Join(request.Name, "toth.yml"),
         Bytes: configBytes,
      }).OnlyIf(func() bool { return tothFileExists})
      AddTask(dir.CreateDirCmd{
         Path: filepath.Join(request.Name, "src"),
      }).AndIgnoreErrors()
      AddTask(gocmds.GetLibCmd{
         Pkg: "github.com/labstack/echo/v4",
      }).
      AddTask(gocmds.TidyCmd{})

   return p.Run(ctx)
}
```

Fonte: Próprio autor

Como já mencionado anteriormente, as *Tasks* são objetos especializados em realizar uma ação única, enquanto os orquestradores são responsáveis por gerenciar as ordens de execução por meio da *Pipeline* e passar os parâmetros corretos para as *Tasks*. Quando as tarefas e orquestradores desejam realizar a geração de código utilizam o módulo *representations*.

Este módulo é responsável por fornecer modelos que representam instruções da sintaxe da linguagem de programação. O módulo traz ainda as funcionalidades de converter os modelos em código *Go* ou o processo reverso, no qual um código escrito em *Go* é convertido para os objetos de representação.

Através das representações, o orquestrador consegue, por exemplo, consultar uma *struct* (equivalente a uma classe em outras linguagens de programação orientada a objetos) e de forma declarativa adicionar um novo método ou verificar se um determinado atributo está presente, bem como gerar código com estruturas e arquivos novos.

O exemplo a seguir (Código-fonte 2) demonstra a definição de uma *interface* utilizando *Representations* e logo após (Código-fonte 3) é possível observar o resultado da geração de código.

Código-fonte 2 – Exemplo de definição de geração de interface utilizando o módulo *Representations*.

```
representacao := representations.Interface{
    Name: "Calculator",
    Methods: []representations.Func{
        {
            Name: "Divide",
            Params: []representations.Object{
                {
                    Name: "a",
                    Type: representations.Type{Name: "int"},
                },
                {
                    Name: "b",
                    Type: representations.Type{Name: "int"},
                },
            },
            Return: representations.Types("int", "error"),
        },
    },
}

// Transforma representação em código GO
codigoGerado, err := representacao.Parse()
```

Fonte: Próprio autor.

Código-fonte 3 – Código gerado a partir do uso das *Representations*.

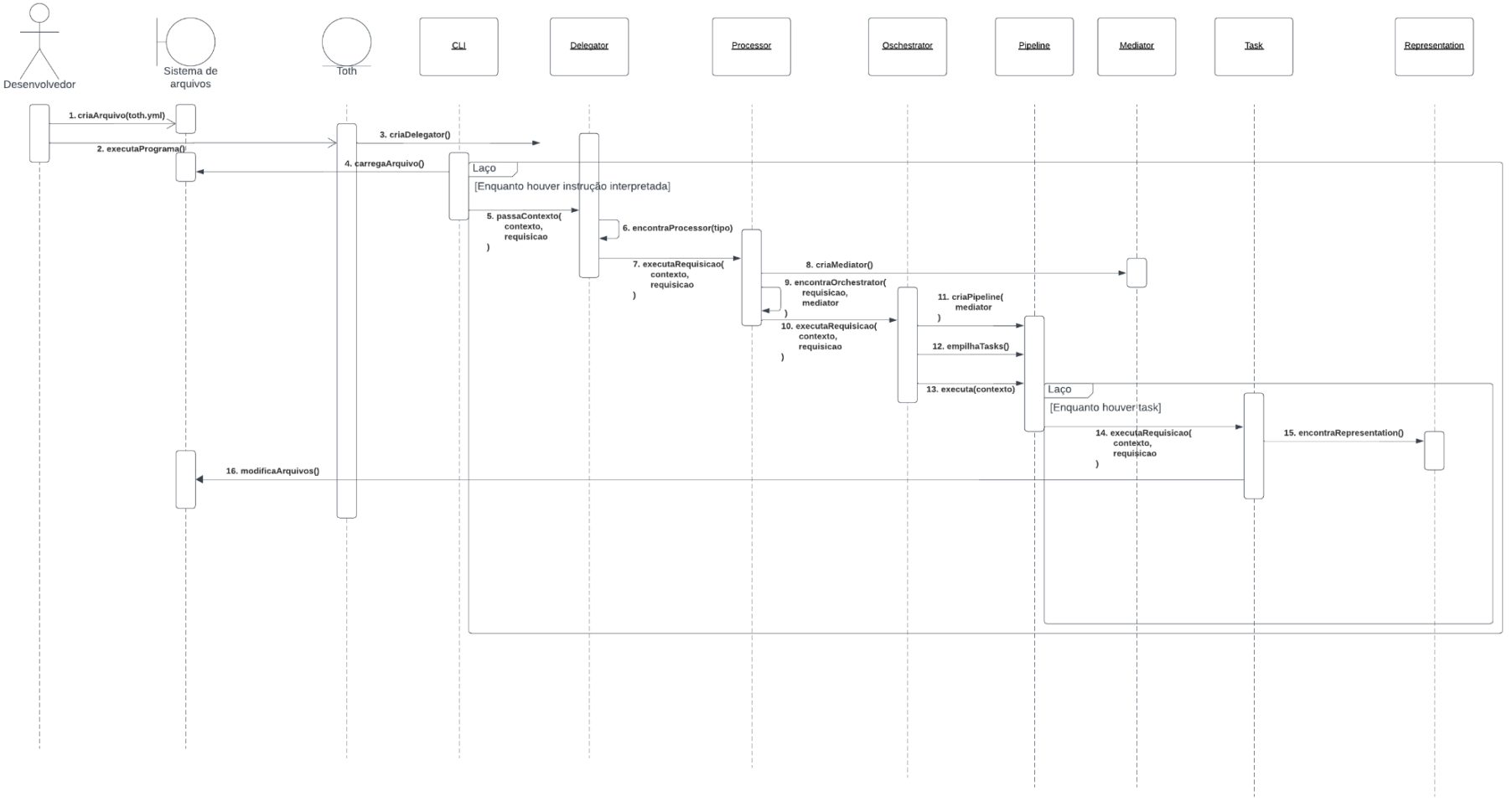
```
type Calculator interface {
   Divide(a int, b int) (int, error)
}
```

Fonte: Próprio autor.

A implementação do módulo *Representations* resolve o problema da complexidade gerada quando o desenvolvedor deseja realizar alterações em uma base de código pré-existente. Durante o processo de desenvolvimento foi levado em consideração que os arquivos previamente gerados podem ter sido modificados pelo ator e portanto não podem ser reescritos para que não haja a perda de regras de negócio importantes. Sendo assim, o decodificador presente no módulo lê os arquivos de código e gera modelos contendo o estado do código em objetos, permitindo que estes sejam alterados de forma simplificada e declarativa sem haver a perda de conteúdo.

5.3 VISÃO DE PROCESSO

Imagem 5 – Representação sequencial do funcionamento da ferramenta.



Fonte: Próprio autor.

A Imagem 5 expressa a interação entre o desenvolvedor e a ferramenta Toth por meio da funcionalidade de leitura do arquivo toth.yml, cujo propósito é representar o estado de um projeto a ser gerado pelo Toth. Como ponto de partida, o desenvolvedor deve fornecer o arquivo ao sistema de arquivos de seu computador. Após isso, a aplicação Toth deve ser executa na mesma localidade que o arquivo de configuração se encontra, acionando a interpretação do estado do projeto e desencadeando, para cada instrução interpretada, a coordenação dos objetos descritos no diagrama para processar, orquestrar, mediar e, por fim, executar um contingente de tarefas relacionadas a consulta, geração e modificação da árvore de arquivos do projeto alvo.

5.4 COMO INSTANCIAR?

No contexto do Toth, há diferentes abordagens disponíveis para instanciar projetos, oferecendo aos usuários opções que se alinham às suas preferências e necessidades específicas.

5.4.1 Yaml

Nativamente, o Toth foi desenvolvido com a capacidade de realizar a leitura de um arquivo de configuração chamado toth.yml, realizar interpretações a respeito do projeto a ser gerado e executar uma série de comandos internos para efetivamente gerar o código correspondente ao contexto. Para utilizar essa funcionalidade, basta que o arquivo de configuração esteja no diretório em que o programa será executado com a instrução de linha de comando *init* sem parâmetros adicionais. Caso haja código previamente gerado ou haja alterações de código agregadas ao projeto, a ferramenta trata de preservar essas informações, não comprometendo o desenvolvimento paralelo de usuários.

A fim de expandir o detalhamento a respeito do conteúdo do arquivo de configuração interpretado pela solução proposta, destaca-se que sua estrutura comporta desde informações básicas de nomenclatura de projeto, informações de comunicação com bases de dados até detalhes sobre modelos de dados a serem representados e ações a serem executadas sobre tais modelos (vide Código-fonte 4).

Código-fonte 4 – Exemplo de toth.yml.

```
name: myproject
type: go-hex-rest-api
dataSources:
        - name: postgres
      type: postgres
      config:
       database: $DB_NAME
       host: $DB_HOST
       password: $DB_PASSWORD
       port: $DB_PORT
       user: $DB_USER
defaultDataSource: postgres
endpoints:
 - name: person
   type: get
   target: person
domains:
 - domain: person
   attributes:
    - attribute: name
      type: string
      required: true
            datasource: postgres
    - attribute: pets
      type: pet
            list: true
      required: true
 - domain: pet
   attributes:
    - attribute: name
      type: string
```

Fonte: Próprio autor.

Diante disso, é conferida a possibilidade de atualização e de acréscimo de representações ao projeto, já que o programa verificará, a partir da configuração definida, o estado atual daquele projeto e o que deverá ser incorporado a ele,

promovendo a garantia de que o projeto esteja sincronizado com as necessidades do desenvolvedor sem prejudicar a autonomia de desenvolvimento independente à ferramenta.

5.4.2 Instruções de linha de comando

A ferramenta também foi concebida com um conjunto de instruções de linha de comando, proporcionando aos usuários a capacidade de configurar e personalizar eficientemente seus projetos. Essa abordagem concede uma liberdade significativa aos usuários, permitindo que a ferramenta seja utilizada conforme necessário, enquanto o restante do processo decisório permanece sob controle do time de desenvolvimento. Essa flexibilidade auxilia na adaptação dos diversos requisitos e preferências dos usuários, proporcionando um ambiente propício para a tomada de decisões no ciclo de desenvolvimento.

Instrução *init*:

Para iniciar um projeto utilizando a ferramenta desenvolvida nesta pesquisa, utiliza-se a instrução *init* a qual visa criar as bases fundamentais de um projeto, estabelecendo sua estrutura e configurações iniciais. Ao utilizá-la, os usuários devem fornecer o nome do projeto como primeiro parâmetro. Em seguida, a *flag --use* deve ser fornecida seguida pelo tipo de arquitetura a ser aplicada ao projeto, sendo a nativa denominada "*go-hex-rest-api*".

O comando *init* é responsável por gerar um arquivo YAML de configuração que é utilizado para futuras referências da ferramenta. Em adição, os arquivos iniciais para consolidar a *API* produzida como um servidor também são gerados neste momento, bem como a estrutura inicial de diretórios, fornecendo uma base sólida para o desenvolvimento subsequente. No suporte arquitetônico nativo da ferramenta, utiliza-se o *framework* web *Echo*[21] para facilitar o gerenciamento da comunicação *HTTP*.

[21] https://echo.labstack.com/

Instrução *add datasource*:

A funcionalidade add datasource tem por objetivo configurar o projeto para se conectar a uma base de dados, utilizando a ferramenta de mapeamento objeto-relacional (*ORM) Gorm*[22]. Os usuários devem personalizar essa integração fornecendo o apelido dado a base de dados. Em seguida, devem ser fornecidas a *flag -t* para especificar o tipo de base de dados desejado (por exemplo, *postgres*[23]) e a *flag -c* para receber um JSON de configuração da base de dados.

Instrução *add getter*:

Por sua vez, *add getter* é uma instrução que comanda a aplicação a produzir um recurso *REST* de busca de domínio. Esta instrução necessita de um nome de domínio e possui uma *flag* opcional *--by-id*; caso essa informação extra seja fornecida, a funcionalidade buscará o domínio baseado em um identificador (id); do contrário, a funcionalidade buscará todos os registros disponíveis relacionados àquele domínio.

Instrução *update*:

A funcionalidade *update* permite que o projeto seja atualizado por meio das modificações realizadas no arquivo *toth.yml*. Para obter esse resultado, basta que o desenvolvedor atualize seu arquivo e, por meio da interface de linha de comando, invoque a ferramenta Toth seguido da instrução *update*.

[22] https://gorm.io/
[23] https://www.postgresql.org/

6 AVALIAÇÃO DA SOLUÇÃO PROPOSTA

Com o objetivo de avaliar a solução proposta, foi elaborado um estudo de caso o qual será executado tanto pelo software desenvolvido neste trabalho quanto por um programador experiente, este último não utilizando a ferramenta desenvolvida. Assim, será possível comparar as abordagens com base em duas métricas: o tempo de desenvolvimento e o número de linhas de código produzidas.

6.1 ESTUDO DE CASO

Considere um gerente de uma loja de sapatos, que atualmente registra as quantidades dos produtos por meio de planilhas. Seu desejo é aprimorar esse processo através de um sistema que facilite a manipulação desses dados. Diante dessa demanda, uma *software house* foi contratada para produzir parte desse sistema. A equipe de desenvolvimento adotou a metodologia ágil *Kanban* e a primeira entrega determinada foram as funcionalidades de busca e busca por id.

O foco da análise deste estudo de caso será direcionado para a produção dessa *API*. Desse modo, a metodologia adotada para a avaliação envolverá as seguintes etapas: desenvolver uma versão da *API* conduzida por um programador experiente sem o auxílio do sistema desenvolvido, produzir uma versão da *API*, utilizando a ferramenta desenvolvida e realizar uma discussão a respeito das métricas de tempo de desenvolvimento e o número de linhas de código produzidas.

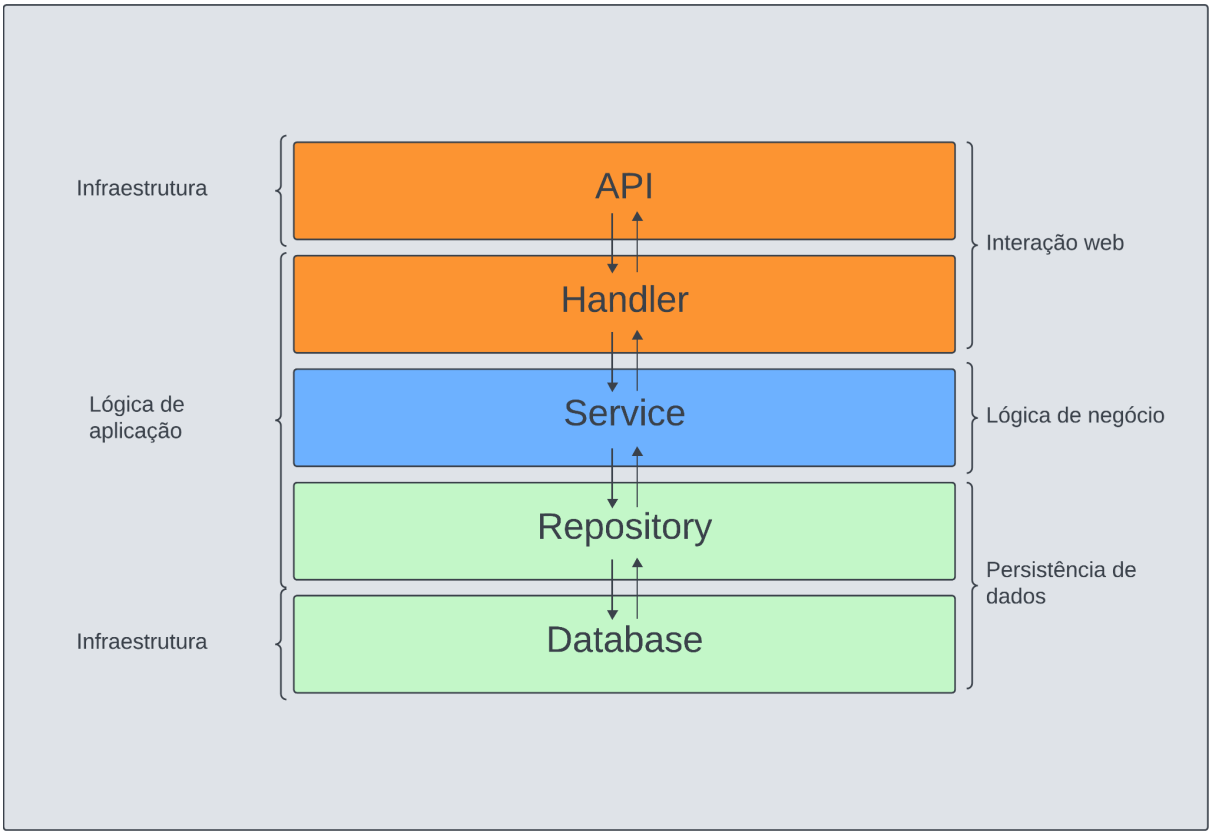
Para ambas versões, os objetivos em comum que devem ser alcançados, são:

- A configuração de um servidor *WEB* que expõe as funcionalidades da aplicação por meio de recursos *REST*.
- O estabelecimento de comunicação com uma base de dados para persistência e manipulação dos dados do estoque do estabelecimento.
- A estruturação do núcleo da aplicação, responsável por guardar as informações pertinentes ao domínio avaliado, respeitando as diretrizes da arquitetura de portas e adaptadores.
- A entrega das funcionalidades determinadas de modo funcional.

6.2 IMPLEMENTAÇÃO DE UM PROGRAMADOR EXPERIENTE

Durante o processo de implementação, foi adotada a arquitetura hexagonal para organizar as responsabilidades da aplicação. A comunicação entre as camadas - *Handler*, *Service* e *Repository* - é apresentada na Imagem 6. Essas camadas são cruciais para a manipulação de dados e a comunicação com as camadas finais, representadas pela *API* e pela *Database*.

Imagem 6 – Distribuição das camadas da aplicação.



Fonte: Próprio autor.

A camada *Handler* assume o papel de facilitar a comunicação direta com a *API*, garantindo uma interação com atores externos por meio do protocolo *HTTP*. Sua responsabilidade principal é lidar com as requisições externas e direcioná-las adequadamente para as demais camadas. Enquanto isso, a camada *Service* desempenha o papel de encarregar-se da manipulação de dados. Nela, as

operações essenciais de processamento e lógica de negócios são implementadas, garantindo uma gestão eficaz e consistente dos dados ao longo da aplicação.

A terceira camada, denominada *Repository*, é responsável por gerenciar a interação direta com a base de dados. Sua função central é fornecer uma interface que permite a persistência e recuperação de dados de maneira eficiente. Ao adotar essa abordagem, a aplicação beneficia-se da clara separação de responsabilidades entre as camadas, resultando em modularidade e facilidade de manutenção.

A fim de representar a configuração do servidor *REST*, o Código-fonte 5 expõe como o arquivo de entrada da aplicação (*main.go*) está estruturado. Inicialmente, as variáveis de ambiente são carregadas no código para que possam ser usadas nas operações subsequentes. Em seguida, as variáveis da base de dados são enviadas para o pacote responsável a fim de que esse lide com elas de modo apropriado. Por fim, o servidor é iniciado com informações advindas das variáveis de ambiente, iniciando a aplicação.

Código-fonte 5 – Arquivo *main.go.*

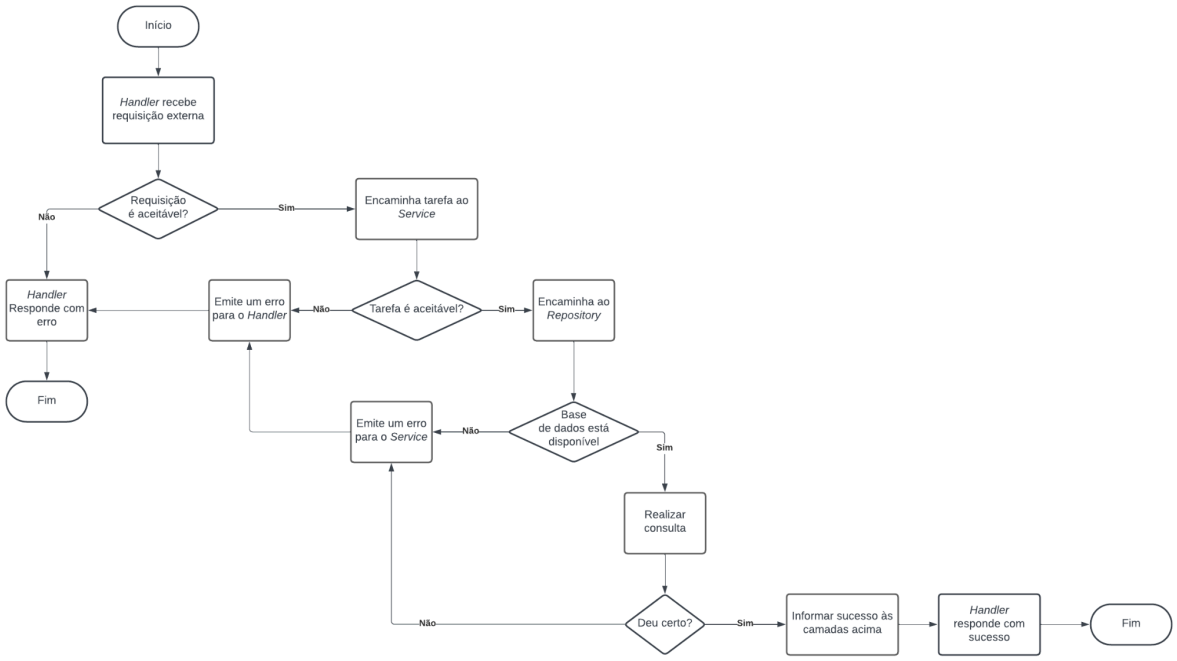
```
package main

import (
	"shopinventory/src/app/api/config"
	"shopinventory/src/app/api/endpoints/router"
	"shopinventory/src/infra/postgres"
)

func main() {
	cfg := config.Load()

	postgres.SetupCredentials(
		cfg.PGHost,
		cfg.PGPort,
		cfg.PGUser,
		cfg.PGPassword,
		cfg.PGDatabase,
	)
	router.Start(
		cfg.ServerHost,
```

```
                cfg.ServerPort,
        )
}
```

Fonte: Próprio autor.

Quando as requisições externas são recebidas pela camada *Handler*, responsável por intermediar a interação com a *API*, essas requisições são enviadas para a camada *Service*, onde ocorre o processamento e manipulação dos dados conforme as regras de negócio estabelecidas. Após a conclusão dessas operações, a camada *Service* se comunica com a camada *Repository*, que gerencia a interação com a base de dados, garantindo a persistência e recuperação adequada das informações. Este fluxo de comunicação entre as camadas cria uma estrutura modular e organizada, facilitando a manutenção e expansão do sistema. Esse fluxo está representado na Imagem 7, onde os diferentes estágios do processamento e a comunicação entre as camadas são visualmente representados.

Imagem 7 – Fluxo de processamento de uma requisição pela aplicação.



Fonte: Próprio autor.

No que tange à comunicação com a base de dados, destaca-se primariamente a escolha da tecnologia *postgres* e a utilização da biblioteca *sqlx*[24] para facilitar a comunicação entre a aplicação e a base de dados. O Código-fonte 6 é responsável por mostrar as entidades responsáveis pelo estabelecimento e cancelamento da conexão entre a base de dados e a aplicação. São estabelecidos nesse pedaço de código tanto a interface que determina o comportamento padrão de abertura e fechamento de uma conexão com a base de dados quanto a implementação aplicada para a base de dados fornecida pelas variáveis de ambiente injetadas na aplicação.

Código-fonte 6 – Arquivo *connection_manager.go.*

```
package postgres

import (
	"log"

	"github.com/jmoiron/sqlx"
	_ "github.com/lib/pq"
)

type ConnectionManager interface {
	getConnection() (*sqlx.DB, error)
	closeConnection(*sqlx.DB)
}

var _ ConnectionManager = (*PGConnectionManager)(nil)

type PGConnectionManager struct{}

func (m PGConnectionManager) getConnection() (*sqlx.DB, error) {
	return sqlx.Open("postgres", connectionURI())
}

func (m PGConnectionManager) closeConnection(conn *sqlx.DB) {
	if err := conn.Close(); err != nil {
		log.Fatal(err)
```

[24] https://github.com/jmoiron/sqlx

```
        }
}
```

Fonte: Próprio autor.

Por sua vez, representando a aplicação dos conceitos relativos à arquitetura de portas e adaptadores, os códigos fonte 7 e 8 representam, respectivamente, a interface primária dos produtos e a interface secundária dos produtos. O conteúdo das interfaces descreve como as operações de cadastro, busca, edição e remoção devem ser implementadas.

Código-fonte 7 – Arquivo *product_manager.go*

```
package primary

import "shopinventory/src/core/domain"

type ProductManager interface {
        FindProduct() ([]domain.Product, error)
        FindProductByID(string) (*domain.Product, error)
}
```

Fonte: Próprio autor.

Código-fonte 8 – Arquivo *product_loader.go*

```
package secondary

import "shopinventory/src/core/domain"

type ProductLoader interface {
        FetchProduct() ([]domain.Product, error)
        FetchProductByID(string) (*domain.Product, error)
}
```

Fonte: Próprio autor.

O desenvolvimento da aplicação sem o auxílio da ferramenta levou um tempo aproximado de duas horas e quarenta e cinco minutos e uma quantidade aproximada de trezentas setenta e duas linhas de código.

6.3 IMPLEMENTAÇÃO USANDO TOTH

No processo de construção da *API* com o apoio da ferramenta proposta, o programador responsável por essa tarefa optou por realizar o processo de construção por meio da interface de linha de comando disponibilizada. A fim de concretizar o produto seguindo as orientações determinadas durante a prescrição do estudo de caso, foi necessário fazer uso de todas as funcionalidades presentes na aplicação até o momento de produção deste trabalho.

Inicialmente, o desenvolvedor fez uso da instrução de linha de comando *init* para que o projeto fosse configurado segundo suas especificações. Para tanto, foram fornecidos os valores para o nome do projeto e para a *flag --use*, que determina qual o modelo arquitetural deve ser aplicado para o projeto em questão. Nesse contexto, os valores fornecidos foram "*shopinventory*" para o nome do projeto e "*go-hex-rest-api*", valor nativo de padrão arquitetural, para o tipo de arquitetura utilizada.

Por meio da execução da instrução supracitada, o projeto foi estruturado com uma árvore de arquivos inicial, com capacidade de iniciar um servidor *web*, como mostrado na Figura 1. Dentre os arquivos gerados, é importante destacar a responsabilidade do arquivo *toth.yml*, responsável por representar a configuração atual do projeto em relação à ferramenta, este está apresentado no Código-fonte 9.

Figura 1 – Árvore de arquivos gerada pela ferramenta proposta.

```
shopinventory
├── go.mod
├── go.sum
├── toth.yml
└── src
    ├── app
    │   └── api
    │       ├── config
```

```
│   │   └── config.go
│   ├── endpoints
│   │   └── routing
│   │       └── router.go
│   └── main.go
└── core
    └── domains
```

Fonte: Próprio autor.

Código-fonte 9 – Arquivo toth*.yml* inicial.

```
name: shopinventory
type: go-hex-rest-api
dataSources: []
defaultDataSource: ""
models: []
```

Fonte: Próprio autor.

Além disso, nesse estágio do projeto, já era possível executar o programa que configuraria e iniciaria um servidor *web*. O Código-fonte 10 expõe o algoritmo da porta de entrada da aplicação, o arquivo *main.go*, responsável por carregar a configuração do projeto, instanciar um roteador e configurá-lo a partir das configurações anteriormente carregadas.

Código-fonte 10 – Arquivo *main.go.*

```
package main

import (
        "fmt"
        "log"
        "shoinventory/src/app/api/config"
        "shoinventory/src/app/api/endpoints/routing"
)

func main() {
```

```
	cfg := config.Bind()

	r := routing.Route()
	log.Fatal(r.Start(fmt.Sprintf("%s:%d", cfg.Host, cfg.Port)))
}
```

Fonte: Próprio autor.

Em seguida, iniciou-se o processo de adição de uma conexão com uma base de dados *postgres* para a realização da persistência dos dados. Para isso, a instrução *add datasource* foi utilizada, passando, respectivamente: o nome da fonte de dados como *postgres*; o tipo da base de dados, representado pela *flag -t* como *postgres* e a configuração da base de dados, representada pela *flag -c*, com um *json* contendo as informações de conexão com a base de dados.

Como resultado dessa execução, foram adicionados um arquivo de configuração *.env*, responsável por armazenar os dados sensíveis de conexão com a base de dados do projeto, a configuração do projeto foi atualizada, um novo pacote *go* foi criado para gerir as comunicações com essa nova base de dados e um arquivo de conexão com a base de dados foi criado.

Para adicionar as informações sobre o modelo de produto a ser consumido pela aplicação, com especificações de atributos e tipos de dados, o arquivo *toth.yml* foi alterado. Após essa alteração, a instrução *add getter* seguida pelo nome do modelo adicionado à configuração foi executada, gerando uma funcionalidade que busca todos os produtos cadastrados na base de dados. Em seguida, a mesma instrução foi executada, porém com a adição da *flag --by-id*; o que gerou uma funcionalidade que busca um produto baseado no seu id.

O tempo total de produção dessa *API* com o auxílio da ferramenta Toth foi de, aproximadamente, cinco minutos, contando com um contingente de linhas de código de aproximadamente trezentas e cinquenta e quatro.

6.4 RESULTADOS E DISCUSSÃO

Ao comparar os resultados obtidos entre as duas abordagens de implementação da *API,* é necessário validar que as duas soluções cumprem os objetivos estabelecidos pelo estudo de caso, sendo ambos os produtos capazes de

performar as funcionalidades de busca de produtos e busca de produto por *id*. Porém também é possível notar diferenças consideráveis nos números relativos às métricas estabelecidas para o estudo de caso, as quais são tempo de desenvolvimento e quantidade de linhas de código.

Em relação à métrica tempo de desenvolvimento, o intervalo de tempo considerado para os fins desta análise conta desde a construção dos diretórios até a verificação de validade das funcionalidades requisitadas. Na implementação manual da aplicação, o processo levou aproximadamente duas horas e quarenta e cinco minutos, sendo executado por um programador experiente nesse tipo de produto. Por outro lado, ao utilizar a ferramenta Toth, o tempo total de produção foi reduzido para aproximadamente cinco minutos, representando uma redução de aproximadamente 97% no tempo de implementação.

Sendo assim, em termos de produtividade, a ferramenta não apenas demonstrou sua efetividade para reduzir o tempo inicial de desenvolvimento de software com a linguagem *Go*, mas também reduziu a necessidade de experiência profunda com a tecnologia, tornando seu manuseio possível por um programador com menos experiência que o programador que gerou a primeira versão da aplicação.

Sob o viés da quantidade de linhas de código, a implementação manual resultou em aproximadamente trezentas e setenta e duas linhas de código, enquanto a utilização da ferramenta Toth resultou em um contingente de linhas de código ligeiramente menor, totalizando aproximadamente trezentas e cinquenta e quatro linhas.

A diferença na quantidade de linhas de código sugere que a ferramenta Toth não apenas acelerou o desenvolvimento, mas também promoveu uma abordagem mais concisa e eficiente na construção da API. Isso é particularmente relevante, pois uma quantidade menor de código muitas vezes implica em uma base de código mais fácil de entender, manter e depurar.

Os códigos gerados não são idênticos, pois refletem abordagens distintas para a construção da API. No entanto, é importante ressaltar que a diferença não compromete a funcionalidade ou a robustez da aplicação. Ambas as implementações seguem uma arquitetura hexagonal, mas a ferramenta Toth introduziu uma estrutura mais automatizada e padronizada, facilitando a configuração inicial e a manutenção contínua.

Quanto aos requisitos estabelecidos previamente, é possível determinar que eles foram alcançados pela solução proposta, uma vez que a ferramenta é capaz de gerar e configurar uma aplicação com modelos, adição de bases de dados e funcionalidades de busca inteiras tanto por meio de instruções de linha de comando quanto por meio da leitura de um arquivo de configuração *YAML*, sendo capaz de reverter alterações quando ocorre um erro.

Por fim, vale ressaltar que a comparação realizada para avaliar a solução proposta foi estabelecida em relação a um desenvolvedor que já domina os conceitos da arquitetura hexagonal, conhece boas práticas de desenvolvimento de software e é familiar com a linguagem de programação *Go*. Sob outra perspectiva, a comparação entre o resultado obtido por um programador menos experiente tanto poderia resultar em um tempo de desenvolvimento ainda maior como conteria, potencialmente, uma quantidade maior de problemas de código (*code smells*), ressaltando o potencial da ferramenta de acelerar o processo de desenvolvimento e garantir que o empenho do desenvolvedor esteja voltado a solucionar problemas lógicos e não disposto majoritariamente a tarefas repetitivas.

7 CONSIDERAÇÕES FINAIS

Este trabalho destacou uma abordagem sobre a geração de código em linguagem Go, integrada à arquitetura de portas e adaptadores. Ao comparar as duas abordagens nos resultados, observou-se que a solução proposta não apenas acelerou o desenvolvimento, mas também se destacou pela capacidade de estruturar projetos e padronizar código. Ao permitir que os desenvolvedores dediquem mais tempo à resolução de desafios de negócios, o recurso se destaca como uma ferramenta valiosa.

Em retrospectiva aos objetivos estabelecidos no início desta pesquisa, é notável que os resultados obtidos atendem de maneira efetiva aos propósitos determinados anteriormente. O objetivo geral de desenvolver uma solução que apoiasse a geração de códigos de base extensível, respaldada por uma arquitetura de portas e adaptadores, foi plenamente alcançado com a implementação do sistema proposto. A aceleração do processo de desenvolvimento, aliada à capacidade de estruturar projetos e padronizar código, não apenas promoveu o uso de boas práticas de engenharia de software, como também evidenciou a eficácia da solução na produção de código de qualidade.

Nesse sentido, foi introduzida a ferramenta Toth, que recebe as instruções sobre a construção de um projeto de software por meio de diretrizes de linha de comando e através de um arquivo de configuração *YAML*. Por meio dos resultados obtidos na comparação da velocidade de desenvolvimento com e sem o uso do sistema desenvolvido, foi possível constatar que a utilização da Toth não somente acelera o processo de desenvolvimento, mas também colabora para a determinação da estrutura do projeto e para a padronização de código, fornecendo recursos para garantir que os desenvolvedores gastem mais tempo na solução de problemas de negócio.

Por meio da consulta ao estado da arte sobre geração de código, foi possível estabelecer uma base sólida para o desenvolvimento da Toth, incorporando as melhores práticas e tendências identificadas na literatura. Os resultados observados após o desenvolvimento da ferramenta demonstraram sua capacidade de acelerar o ciclo de desenvolvimento, proporcionando não apenas eficiência, mas também contribuindo para a qualidade do código gerado. A combinação da linguagem Go e da arquitetura de portas e adaptadores revelou-se uma escolha

estratégica, alinhando-se aos objetivos de agilidade e qualidade no desenvolvimento de software.

7.1 LIMITAÇÕES E AMEAÇA A VALIDADE

Apesar dos resultados obtidos, é importante observar que a ferramenta Toth possui algumas limitações. Em seu estágio atual, ela produz exclusivamente funcionalidades de busca e sua responsabilidade primária é a de gerar apenas o esqueleto da aplicação. Diante disso, a Toth não adiciona lógica de negócio, não realiza validação de dados ou qualquer tratamento extra necessário para o produto e também não implementa mecanismos de segurança nos produtos gerados pela ferramenta, como autenticação e autorização.

7.2 TRABALHOS FUTUROS

Além disso, é crucial reconhecer que este trabalho não representa um ponto final, mas sim um marco inicial em direção a avanços contínuos. Em seguida, apresentam-se algumas sugestões para trabalhos futuros:

- Expansão e Aperfeiçoamento da Toth: Buscar oportunidades para expandir e aprimorar a ferramenta, incorporando novos recursos e funcionalidades que possam atender a uma variedade mais ampla de necessidades no desenvolvimento de software.
- Avaliação em Ambientes de Desenvolvimento Reais: Realizar testes mais abrangentes e avaliações práticas da Toth em ambientes de desenvolvimento real, considerando diferentes tipos de projetos e cenários, para validar sua eficácia em situações do mundo real.
- Integração com Outras Ferramentas e Ecossistemas: Explorar oportunidades para integrar a Toth com outras ferramentas e ecossistemas de desenvolvimento, promovendo uma maior interoperabilidade e complementaridade no processo de desenvolvimento.

- Análise de Desempenho a Longo Prazo: Conduzir estudos de desempenho a longo prazo para avaliar a manutenibilidade e escalabilidade dos projetos desenvolvidos com a Toth, assegurando que a eficiência inicial se mantenha ao longo do ciclo de vida do software.
- Estudo de Casos e Experiência do Usuário: Coletar e analisar estudos de caso e feedback dos usuários da Toth para compreender melhor sua aplicabilidade, desafios enfrentados pelos desenvolvedores e áreas potenciais de melhoria.

REFERÊNCIAS

ANTINYAN, Vard. Evaluating essential and accidental code complexity triggers by practitioners' perception. **IEEE Software**, v. 37, n. 6, p. 86-93, 2020.

ATENCIO, Yalmar Ponce et al. Automatic generation of web applications for information systems. In: **Journal of Physics: Conference Series**. IOP Publishing, 2021. p. 012019.

BALABAN, Mira; KHITRON, Igal; MARAEE, Azzam. Accidental complexity in multilevel modeling revisited. **Software and Systems Modeling**, v. 21, n. 2, p. 517-542, 2022.

BHAVSAR, K.; SHAH, Vrutik; GOPALAN, Samir. Process life cycle framework: A conceptual model and literature study of business process re-engineering for software engineering management. **CiiT International Journal of Software Engineering and Technology**, v. 11, n. 6, p. 096-100, 2019.

CHABBI, Milind; RAMANATHAN, Murali Krishna. A study of real-world data races in Golang. In: **Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation**. 2022. p. 474-489.

COCKBURN, Alistair. Hexagonal architecture. **The Pattern: Ports and Adapters**, 2005.

CZARNECKI, Krzysztof. Generative programming: Methods, techniques, and applications tutorial abstract. In: **International Conference on Software Reuse**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. p. 351-352.

DE SOUZA, Leonardo Pereira Pinheiro; DE MORAES, Cássia Regina Bassan. Influência do clima organizacional para o compartilhamento de conhecimento tácito no desenvolvimento de software. **GESTÃO. Org**, v. 19, n. 1, p. 35-51, 2021.

DOMINGOS, Henrique Miguel Afonso. Desenvolvimento de uma Aplicação para Geração Automática de Código para o PLC de Controlo de um Shuttle Car. 2018.

DURAI, Anand Deva et al. A novel approach with an extensive case study and experiment for automatic code generation from the XMI schema Of UML models. **The Journal of Supercomputing**, p. 1-23, 2022.

ERRINGTON, J. ACK H. **Code generation in action**. 2003.

FIELDING, Roy Thomas. REST: architectural styles and the design of network-based software architectures. **Doctoral dissertation, University of California**, 2000.

GAMMA, R.; HELM R. Johnson; VLISSIDES, J; **Design Patterns - Elements of Reusable Object-Oriented Software**. Addison-Wesley. 1994.

GARLAN, David; PERRY, Dewayne E. Introduction to the special issue on software architecture. **IEEE Trans. Software Eng.**, v. 21, n. 4, p. 269-274, 1995.

GURUNG, Gagan; SHAH, Rahul; JAISWAL, Dhiraj Prasad. Software Development Life Cycle Models-A Comparative Study. **International Journal of Scientific Research in Computer Science, Engineering and Information Technology**, March, p. 30-37, 2020.

HUNING, Lars; PULVERMUELLER, Elke. Automatic Code Generation of Safety Mechanisms in Model-Driven Development. **Electronics**, v. 10, n. 24, p. 3150, 2021.

JUNIOR, Muris Lage; GODINHO FILHO, Moacir. Variations of the kanban system: Literature review and classification. **International Journal of Production Economics**, v. 125, n. 1, p. 13-21, 2010.

JURGELAITIS, Mantas et al. Solidity code generation from UML state machines in model-driven smart contract development. **IEEE Access**, v. 10, p. 33465-33481, 2022.

KLEPPMANN, M; **Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems.** O'Reilly. 2017

KUMAR, Santosh. A review on client-server based applications and research opportunity. **International Journal of Recent Scientific Research**, v. 10, n. 7, p. 33857-3386, 2019.

MARCHUK, Yaroslav; MELNYK, Bohdan; MELNYK, Nataliya. Analysis of the Speed of Execution of Business Logic in Applications Created in Different Software Environments. In: **2023 13th International Conference on Advanced Computer Information Technologies (ACIT)**. IEEE, 2023. p. 357-360.

MARS, Rawya et al. A survey on automation approaches of smart contract generation. **The Journal of Supercomputing**, p. 1-33, 2023.

MARTIN, Robert C. Design principles and design patterns. **Object Mentor**, v. 1, n. 34, p. 597, 2000.

MASSE, Mark. **REST API design rulebook: designing consistent RESTful web service interfaces**. " O'Reilly Media, Inc.", 2011.

MILICEV, Dragan. **Model-driven development with executable UML**. John Wiley & Sons, 2009.

MØLLER, Anders; VEILEBORG, Oskar Haarklou. Eliminating abstraction overhead of Java stream pipelines using ahead-of-time program optimization. **Proceedings of the ACM on Programming Languages**, v. 4, n. OOPSLA, p. 1-29, 2020.

MONTEIRO, Edwin; PEREIRA, Kelvinn; BARRETO, Raimundo. Modeling and Automatic Code Generation Tool for Teaching Concurrent and Parallel Programming by Finite State Processes. In: **International Conference on Computational Science**. Cham: Springer International Publishing, 2020. p. 593-607.

MOSTRÖ, Mathilda; RYRBERG, Sophie. How to choose a web development framework: Analyzing best practices on the adoption of web frameworks. 2022.

MÜHLBAUER, Nikolas et al. Open-source OPC UA security and scalability. In: **2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)**. IEEE, 2020. p. 262-269.

MUKHTAR, Maryam I.; GALADANCI, Bashir S. Automatic code generation from UML diagrams: the state-of-the-art. **Science World Journal**, v. 13, n. 4, p. 47-60, 2018.

NAM, Daye et al. In-IDE Generation-based Information Support with a Large Language Model. **arXiv preprint arXiv:2307.08177**, 2023.

NGUYEN, Christofer. **Priority automation engineering: Evaluating a tool for automatic code generation and configuration of PLC-Applications**. 2018.

PAOLONE, Gaetanino et al. Automatic code generation of MVC web applications. **Computers**, v. 9, n. 3, p. 56, 2020.

PINHO, Daniel; AGUIAR, Ademar; AMARAL, Vasco. What about the usability in low-code platforms? A systematic literature review. **Journal of Computer Languages**, p. 101185, 2022.

PRASANNA, Dhananjay. **Dependency injection: design patterns using spring and guice**. Simon and Schuster, 2009.

RAMÍREZ-NORIEGA, Alan et al. inDev: A software to generate an MVC architecture based on the ER model. **Computer Applications in Engineering Education**, v. 30, n. 1, p. 259-274, 2022.

RICHARDS, Mark; FORD, Neal. **Fundamentals of software architecture: an engineering approach**. O'Reilly Media, 2020.

RICHARDSON, Chris. **Microservices patterns: with examples in Java**. Simon and Schuster, 2018.

RICHARDSON, Leonard; RUBY, Sam. **RESTful web services**. " O'Reilly Media, Inc.", 2008.

ROBERT, C. **Clean Architecture: A Craftsman's Guide to Software Structure and Design (Robert C. Martin Series)**. 2019.

TANENBAUM, Andrew S. **Distributed systems principles and paradigms**. 2007.

VERNON, Vaughn. **Implementing domain-driven design**. Addison-Wesley, 2013.

XU, Frank F.; VASILESCU, Bogdan; NEUBIG, Graham. In-ide code generation from natural language: Promise and challenges. **ACM Transactions on Software Engineering and Methodology (TOSEM)**, v. 31, n. 2, p. 1-47, 2022.