Documentação de Projeto

para o sistema

Swagger to SDK

Versão 1.8

Projeto de sistema elaborado pelo aluno Arthur do Nascimento Sita Gomes e apresentado ao curso de **Engenharia de Software** da **PUC Minas** como parte do Trabalho de Conclusão de Curso (TCC) sob orientação de conteúdo dos professores Cleiton Silva Tavares, Danilo de Quadros Maia Filho, Leonardo Vilela Cardoso e Raphael Ramos Dias Costa, orientação de TCC II do professor (a ser definido no próximo semestre).

24/09/2025

Tabela de Conteúdo

Histórico de Revisões

Nome	Data	Razões para Mudança	Versão
Arthur do Nascimento Sita Gomes	24/09/2025	• Criação das seções: Introdução, Descrição de Atores, Modelos de Usuários, Diagrama de Casos de Uso e História de Usuários.	1.0
Arthur do Nascimento Sita Gomes	22/10/2025	• Corrigir numeração e títulos na tabela de conteúdos. • Deixar palavras em itálico no modelo de usuários	1.1
Arthur do Nascimento Sita Gomes	29/10/2025	• Modificar diagrama de casos de uso; • Modificar diagrama de sequência de sistema	1.2

Arthur do Nascimento Sita Gomes	31/10/2025	• Modificar diagrama de classes	1.3
Arthur do Nascimento Sita Gomes	01/11/2025	• Criar diagrama de sequência • Criar diagrama de comunicação	1.4
Arthur do Nascimento Sita Gomes	02/11/2025	• Criar diagrama de pacotes e seção de arquitetura • Criar diagrama de estados • Criar diagrama de componentes • Criar diagrama de implantação • Modificar índice	1.5
Arthur do Nascimento Sita Gomes	04/11/2025	• Criar glossário da aplicação • Criar modelo de dados • Modificar índice • Modificar diagrama de classes	1.6
Arthur do Nascimento Sita Gomes	16/11/2025	• Corrigir componentes de comunicação • Corrigir diagrama de casos de uso	1.7
Arthur do Nascimento Sita Gomes	17/11/2025	• Criar casos de teste de aceitação • Criar Cronograma e Processo de Implementação • Criar Kanban com *sprints* no *Github Projects*	1.8

1. Introdução

Este documento agrega: 1) a elaboração e revisão de modelos de domínio e 2) modelos de projeto para o sistema *Swagger* to *SDK*. O *Swagger* to *SDK* é uma extensão para o *Visual Studio Code* (VScode), com o intuito de ajudar os desenvolvedores a automatizar a geração de clientes de acesso a serviços baseados no padrão de Transferência de Estado Representacional (*REST*, do inglês *Representational State Transfer*) para aplicações de interface com o usuário. A referência principal para a descrição geral do problema, domínio e requisitos do sistema é o documento de especificação que descreve a visão de domínio do sistema. Tal especificação acompanha este documento. Anexo a este documento também se encontra o Glossário.

2. Modelos de Usuário e Requisitos

Esta seção apresenta os modelos de usuário e os requisitos do sistema. A apresentação é composta pelas seguintes divisões: descrição dos atores (Seção 2.1); descrição dos modelos de usuários (Seção 2.2); descrição dos casos de uso e histórias do usuário (Seção 2.3); e descrição diagrama de sequência do sistema e contrato de operações (Seção 2.4).

2.1 Descrição de Atores

O ator identificado nesta etapa é uma pessoa desenvolvedora que trabalha com o desenvolvimento de aplicações *front-end*, que integra aplicações *client* a serviços *REST* via *Swagger/OpenAPI* no *VSCode*. Por estes motivos ela se enquadra na primeira categoria de desenvolvedores *front-end.* Suas dores concentram-se no excesso de *boilerplates*, inconsistências entre contrato e implementação, erros de tipagem detectados tardiamente, documentação desatualizada e baixa padronização entre projetos.

2.2 Modelos de Usuários

Nesta seção são apresentados os modelos de usuários. O modelo foi construído usando a estratégia de personas, no qual são especificados: a biografia, os objetivos, as tarefas, e as dores. A Figura 1 é apresentada a primeira persona que representa os desenvolvedores *front-end*. A Figura 2 temos a persona que representa os desenvolvedores *full-stack*.

Figura 1. Persona que representa um desenvolvedor *front-end.*

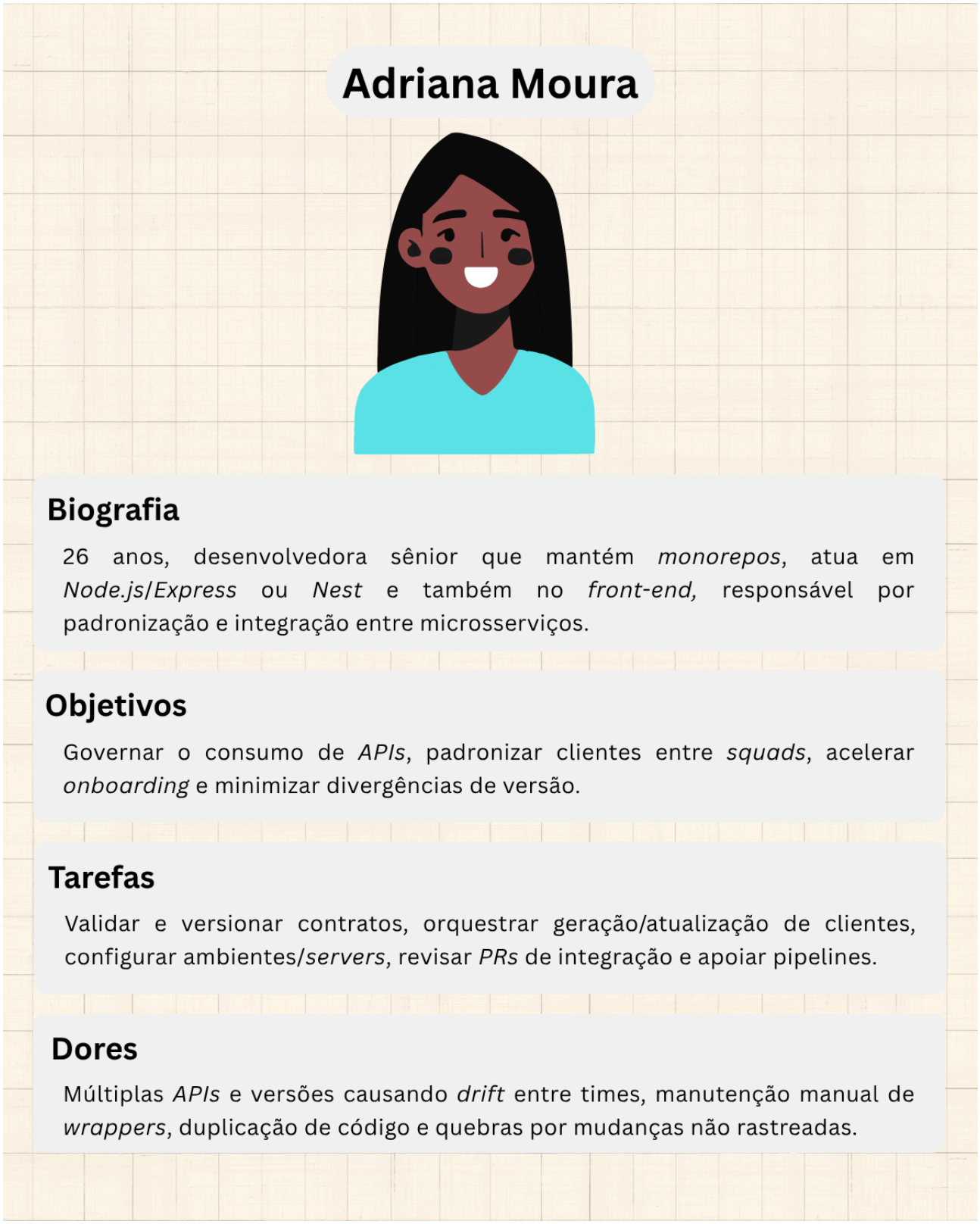

Figura 2. Persona que representa um desenvolvedor *full-stack*.

2.3 Modelo de Casos de Uso e Histórias de Usuários

Esta seção descreve os casos de uso e as histórias de usuário que contemplam as possíveis ações dos usuários da extensão, apresentados respectivamente nas seções 2.3.1 e 2.3.2. Para descrever os casos de uso, foi criado o Diagrama de Caso de Uso, o qual é responsável por descrever as ações que um usuário pode realizar na extensão. As histórias de usuário têm o objetivo de apresentar as necessidades do usuário em relação à extensão de uma maneira clara e concisa para facilitar o entendimento das funcionalidades.

2.3.1 Diagrama de Casos de Uso

Na Figura 3 é apresentado o diagrama de casos de uso do sistema proposto, ele representa as ações dos usuários durante a utilização do sistema. Podemos observar no diagrama que existe apenas um ator que chamamos desenvolvedor, este ator visa suprir os seus problemas de *boilerplate* com a extensão. Para fins de organização, os casos de uso estão numerados de 1 a 11 e incluem as letras UC (UC, do inglês, *Use Case*). Estes números não significam ordem de prioridade e/ou importância.

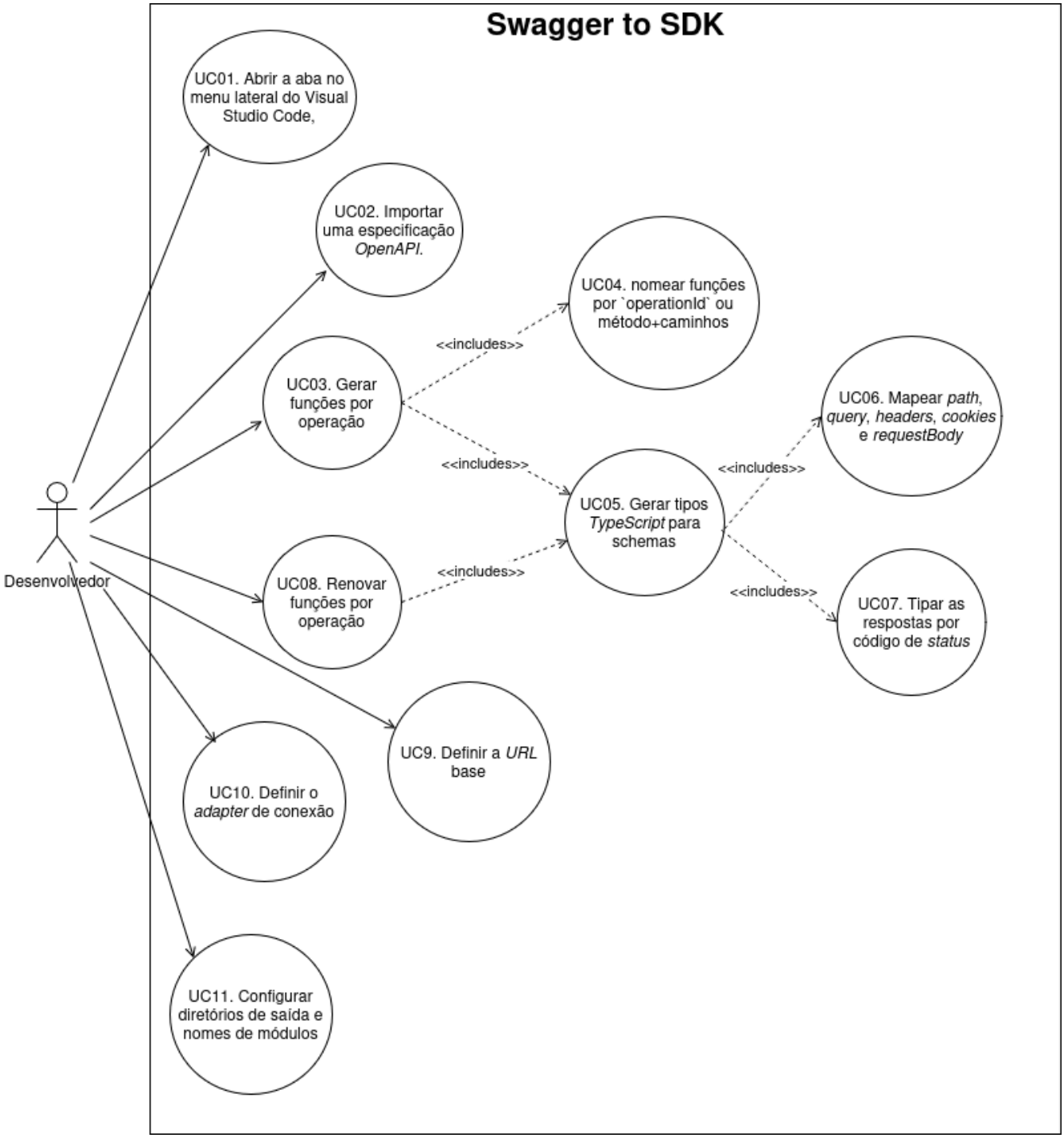

Figura 3. Diagrama de Caso de Uso do sistema Swagger to SDK

2.3.2 Histórias de Usuários

As histórias de usuário (US, do inglês *User Stories*) são uma forma de representar os requisitos do usuário. Trata-se de uma descrição simples, que informa quem realizará uma determinada ação, qual

será essa ação, e a razão que ela é realizada. A seguir, são descritas as histórias de usuários levantadas para a plataforma, identificadas pela sigla US e uma numeração.

US01. Como pessoa desenvolvedora, desejo abrir a aba no menu lateral do *Visual Studio Code*, para trazer informações pertinentes à extensão (relativo ao UC01);

US02. Como pessoa desenvolvedora, desejo importar uma especificação *OpenAPI* através de um arquivo local ou *URL*, para iniciar a geração do cliente sem escrever integração manual (relativo ao UC02 e UC10);

US03. Como pessoa desenvolvedora, desejo gerar funções por operação com assinatura tipada completa, para consumir a *API* com segurança e autocomplete (relativo ao UC03);

US04. Como pessoa desenvolvedora, desejo nomear funções por `*operationId*` ou método+caminho quando ausente, para manter rastreabilidade entre contrato e código (relativo ao UC03 e UC04);

US05. Como pessoa desenvolvedora, desejo gerar tipos *TypeScript* para *schemas*, parâmetros e respostas, para ter uma tipagem estática e reduzir erros (relativo ao UC03, UC08 e UC05);

US06. Como pessoa desenvolvedora, desejo mapear *path*, *query*, *headers*, *cookies* e *requestBody*, para ter previsibilidade e validação das chamadas (relativo ao UC03, UC05, UC06 e UC08);

US07. Como pessoa desenvolvedora, desejo tipar as respostas por código de *status*, para tratar fluxos felizes e de erro corretamente (relativo ao UC03, UC05, UC07 e UC08);

US08. Como pessoa desenvolvedora, desejo renovar os dados da *OpenAPI* por meio de um botão na interface, para poder reiniciar a extensão, caso exista algum dado inconsistente com o estado atual da aplicação (relativo ao UC08).

US09. Como pessoa desenvolvedora, desejo definir a *URL* base, para configurar o ambiente de forma consistente (relativo ao UC09);

US10. Como pessoa desenvolvedora, desejo definir o adaptador de conexão, para padronizar todas as requisições (relativo ao UC10);-

US11. Como pessoa desenvolvedora, desejo configurar diretórios de saída e nomes de módulos, para organizar o código gerado conforme o projeto (relativo ao UC11);

2.4 Diagrama de Sequência do Sistema e Contrato de Operações

Nesta subseção, são apresentados os diagramas de sequência do sistema (DSS) com seus respectivos

contratos de operações, sendo baseados nos casos de uso apresentados na Seção 2.3.1. Esses diagramas têm como finalidade descrever os fluxos de interação que ocorrem entre os usuários e o sistema desenvolvido.

A Figura 4 apresenta o DSS01 relacionado ao caso de uso UC01. Nesse fluxo, o usuário interage com o botão do sistema que, por sua vez, abre as configurações disponíveis da extensão. Além disso, a Tabela 1 faz referência ao mesmo caso de uso trazendo a normalização do contrato de operação do diagrama de sequência de sistema.

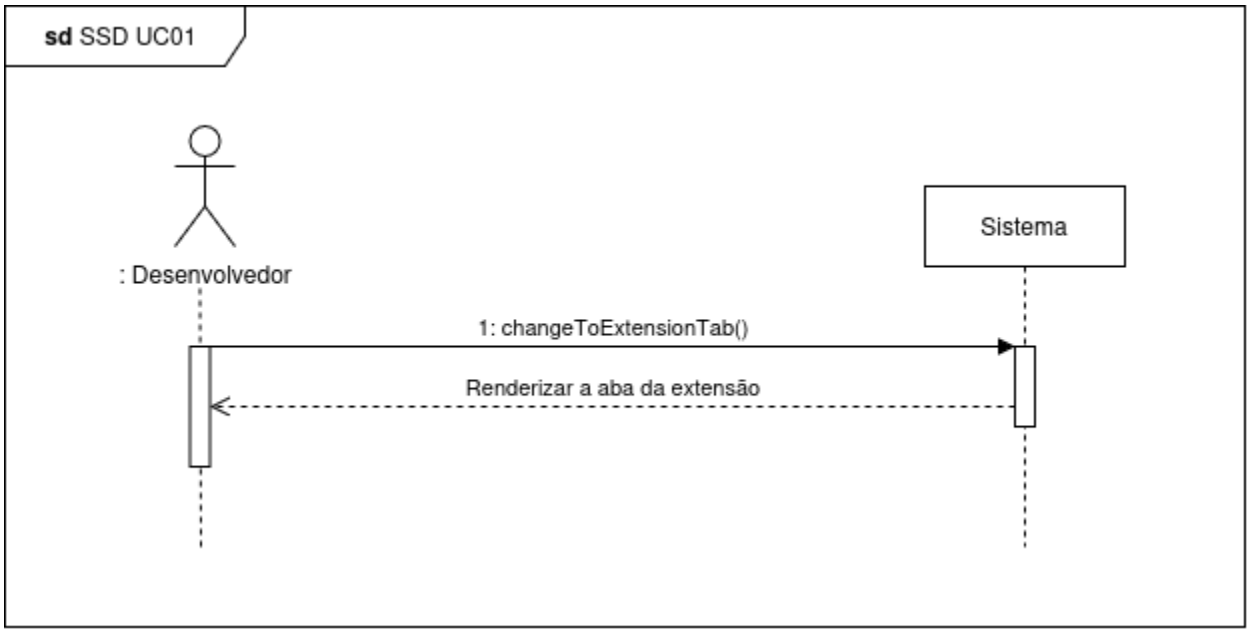

Figura 4. Diagrama de sequência de sistema do caso de uso UC01.

Contrato	Mudar a aba da extensão
Operação	changeToExtensionTab()
Referências cruzadas	• UC01: abrir aba da extensão
Pré-condições	O ator não deve estar na aba da extensão
Pós-condições	A aba da extensão deve estar inicializada e em foco

Tabela 1. Contrato de operação para mudar a aba da extensão.

A Figura 5 apresenta o DSS02 relacionado ao caso de uso UC02. Nesse fluxo, o usuário insere o caminho local para o seu arquivo *OpenAPI* ou a *URL* publica do *Swagger*. Além disso, a Tabela 2 faz referência ao mesmo caso de uso trazendo a normalização do contrato de operação do diagrama de sequência de sistema.

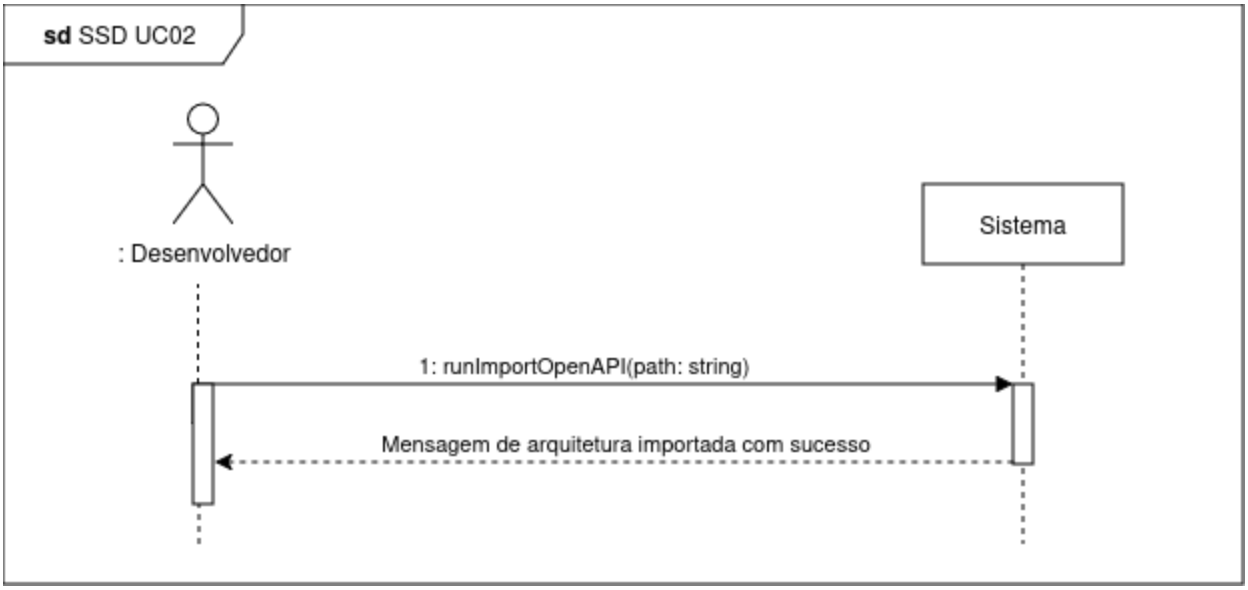

Figura 5. Diagrama de sequência de sistema do caso de uso UC02.

Contrato	Importar especificação da *OpenAPI*
Operação	runImportOpenAPI(path : string)
Referências cruzadas	• UC02: Importar especificações *OpenAPI*
Pré-condições	A aba da extensão deve estar aberta
Pós-condições	Botão de importar funções e gerar tipos habilitado

Tabela 2. Contrato de operação para importar especificação da *OpenAPI*.

A Figura 6 apresenta o DSS03 contempla os caso de usos UC03, UC04, UC05, UC06 e UC07. Nesse fluxo, o usuário cria as funções que conectam aos *endpoints* com as tipagens, *schemas* e nomenclatura baseada no *operationId*. Além disso, o contrato de operação desse diagrama é detalhado pela Tabela 3.

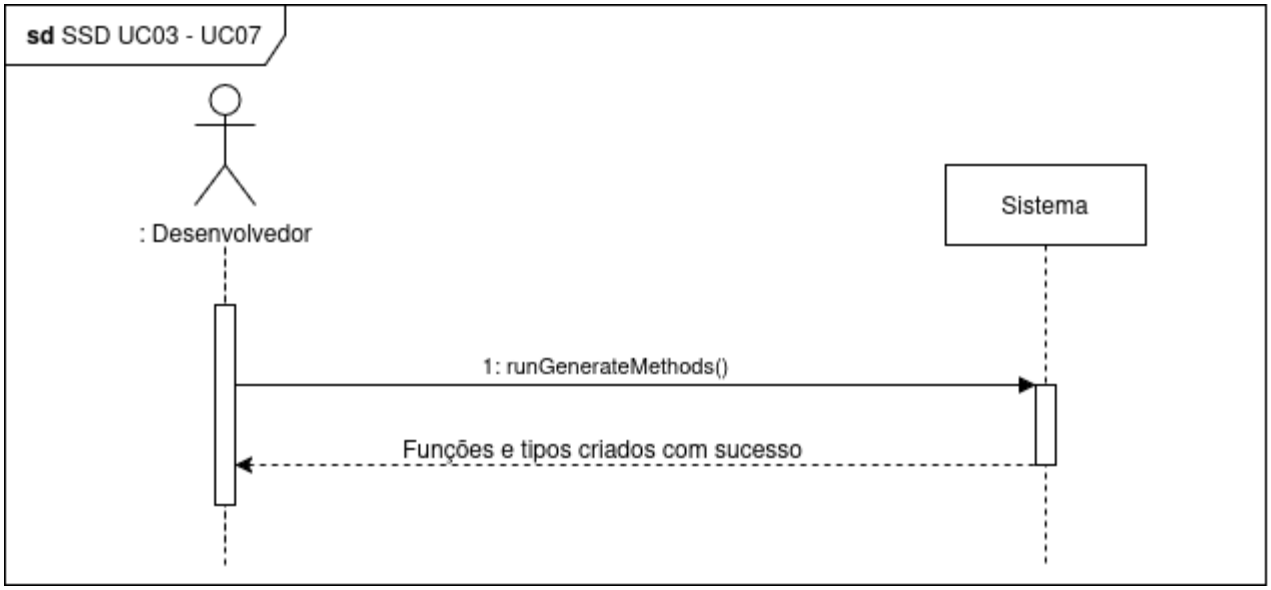

Figura 6. Diagrama de sequência de sistema do caso de uso UC03, UC04, UC05, UC06 e UC07.

Contrato	Criar funções que conectam aos *endpoints* e suas tipagens
Operação	runGenerateMethods()
Referências cruzadas	• UC02: Importar especificações *OpenAPI* • UC03: Gerar funções que conectam aos *endpoints* • UC05: Criar tipos para *schemas* e respostas • UC06: Mapear os *path*, *query*, *headers*, *cookies* e *requestBody* • UC07*:* Tipar respostas por *status*.
Pré-condições	Aba da extensão deve estar aberta e importação da especificação *OpenAPI* feita com sucesso.
Pós-condições	Arquivo com funções que conectam os *endpoints* criada com sucesso.

Tabela 3. Contrato de operação para criação das funções que conectam aos *endpoints* e sias tipagens.

A Figura 7 apresenta o DSS04 contempla os caso de usos UC05, UC06 e UC07. Nesse fluxo, o usuário cria as tipagens das funções que conectam aos *endpoints*. Além disso, o contrato de operação desse diagrama é detalhado pela Tabela 4.

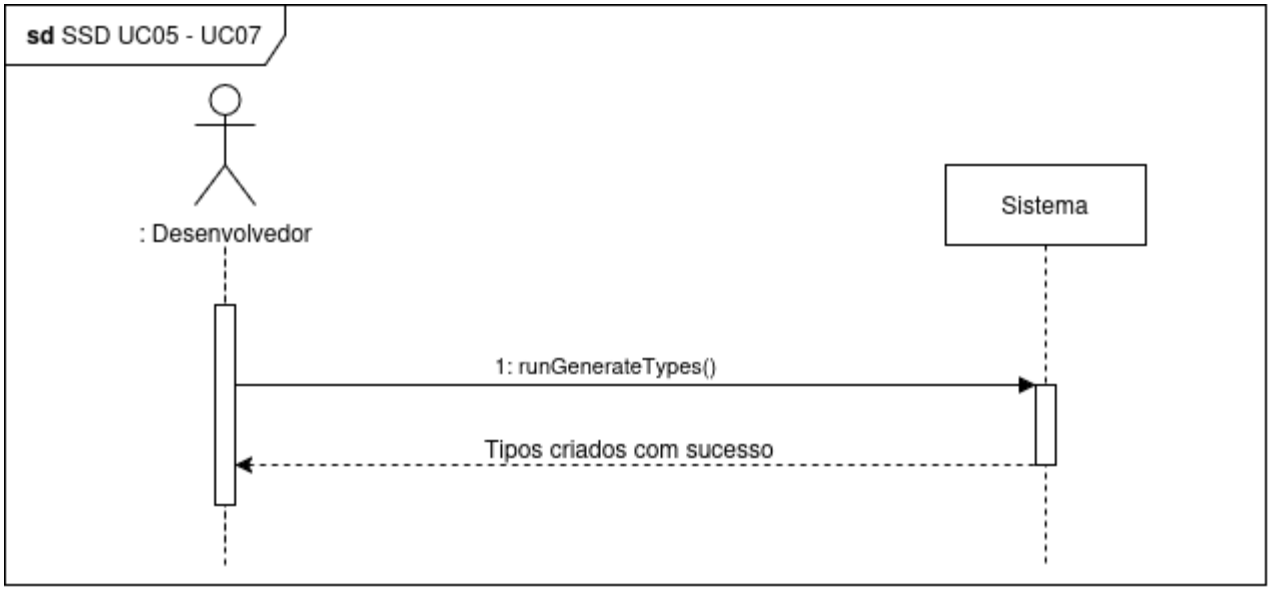

Figura 7. Diagrama de sequência de sistema do caso de uso UC05, UC06 e UC07.

Contrato	Criar funções que conectam aos *endpoints* e suas tipagens
Operação	runGenerateTypes()
Referências cruzadas	• UC05: Criar tipos para *schemas* e respostas • UC06: Mapear os *path*, *query*, *headers*, *cookies* e *requestBody* • UC07*:* Tipar respostas por *status*.
Pré-condições	Aba da extensão deve estar aberta e importação da especificação *OpenAPI* feita com sucesso.
Pós-condições	Arquivo com funções que conectam os *endpoints* criada com sucesso.

Tabela 4. Contrato de operação para criação das funções que conectam aos *endpoints* e sias tipagens.

A Figura 8 apresenta o DSS05 relacionado ao caso de uso UC02, UC03, UC04, UC05, UC06, UC07 e UC08. Nesse fluxo, o usuário atualiza a implementação das funções que conectam aos *endpoints* e as suas tipagens com a documentação *OpenAPI* mais recente. Além disso, o contrato de operação desse diagrama é detalhado pela Tabelas 5.

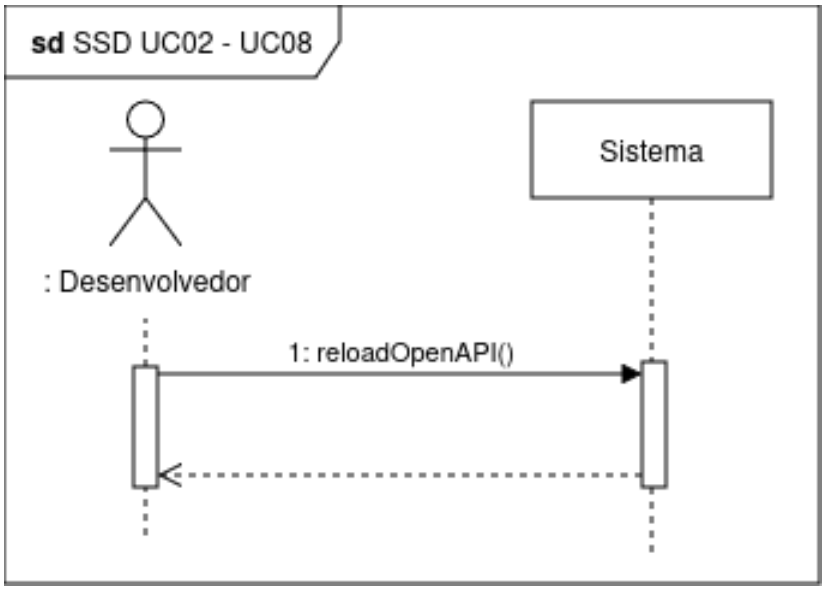

Figura 8. Diagrama de sequência de sistema do caso de uso UC02, UC03, UC04, UC05, UC06, UC07 e UC08.

Contrato	Atualiza as funções que conectam aos *endpoints* e as suas tipagens
Operação	reloadOpenAPI()
Referências cruzadas	• UC02: Importar especificações *OpenAPI* • UC03: Criar funções que conectam aos *endpoints* • UC05: Criar tipos para *schemas* e respostas • UC06: Mapear os *path*, *query*, *headers*, *cookies* e *requestBody* • UC07*:* Tipar respostas por *status*. • UC08: Renovar as funções que conectam aos *endpoints*
Pré-condições	Aba da extensão deve estar aberta e importação da especificação *OpenAPI* feita com sucesso.
Pós-condições	Arquivo com tipos e *schemas* das funções que conectam os *endpoints* atualizada com sucesso.

Tabela 5. Contrato de operação para renovar os tipos e *schemas* das funções que conectam aos *endpoints*.

A Figura 9 apresenta o DSS06 relacionado ao caso de uso UC09. Nesse fluxo, o usuário defini uma *URL* base para utilizar nos *endpoints*. Além disso, a Tabela 6 faz referência ao mesmo caso de uso trazendo a normalização do contrato de operação do diagrama de sequência de sistema.

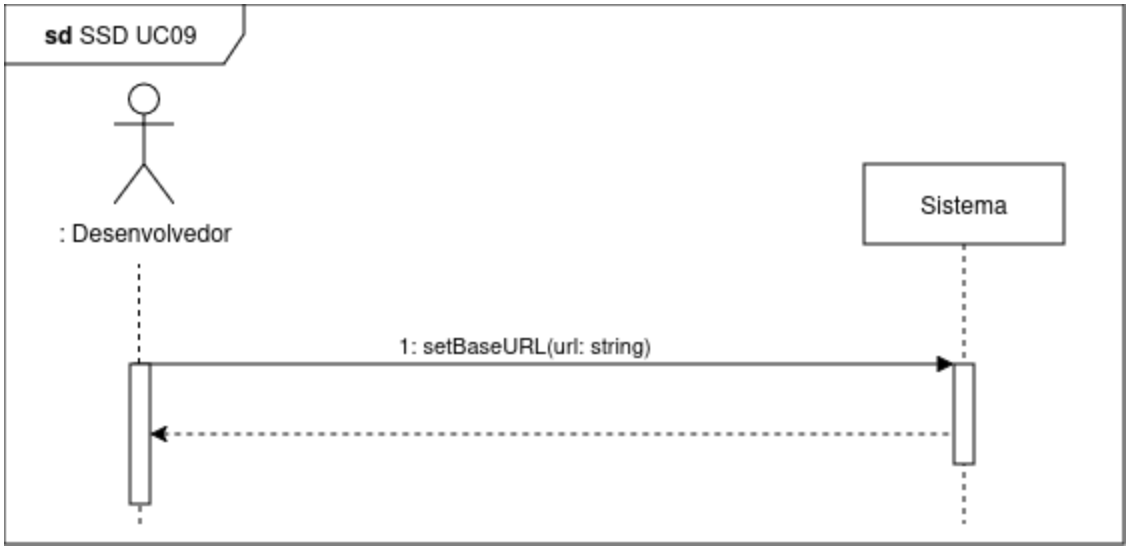

Figura 9. Diagrama de sequência de sistema do caso de uso UC09.

Contrato	Definir a *URL* base para as chamadas externas
Operação	setBaseUrl(url : string)
Referências cruzadas	• UC09: Definir *URL* base
Pré-condições	Aba da extensão deve estar aberta
Pós-condições	*URL* base é salva com sucesso

Tabela 6. Contrato de operação para definir a *URL* base para as chamadas externas.

A Figura 10 apresenta o DSS07 relacionado ao caso de uso UC10. Nesse fluxo, o usuário defini um *adapter* para padronizar as chamadas externas. Além disso, a Tabela 7 e 8 fazem referência ao mesmo caso de uso trazendo a normalização do contrato de operação do diagrama de sequência de sistema.

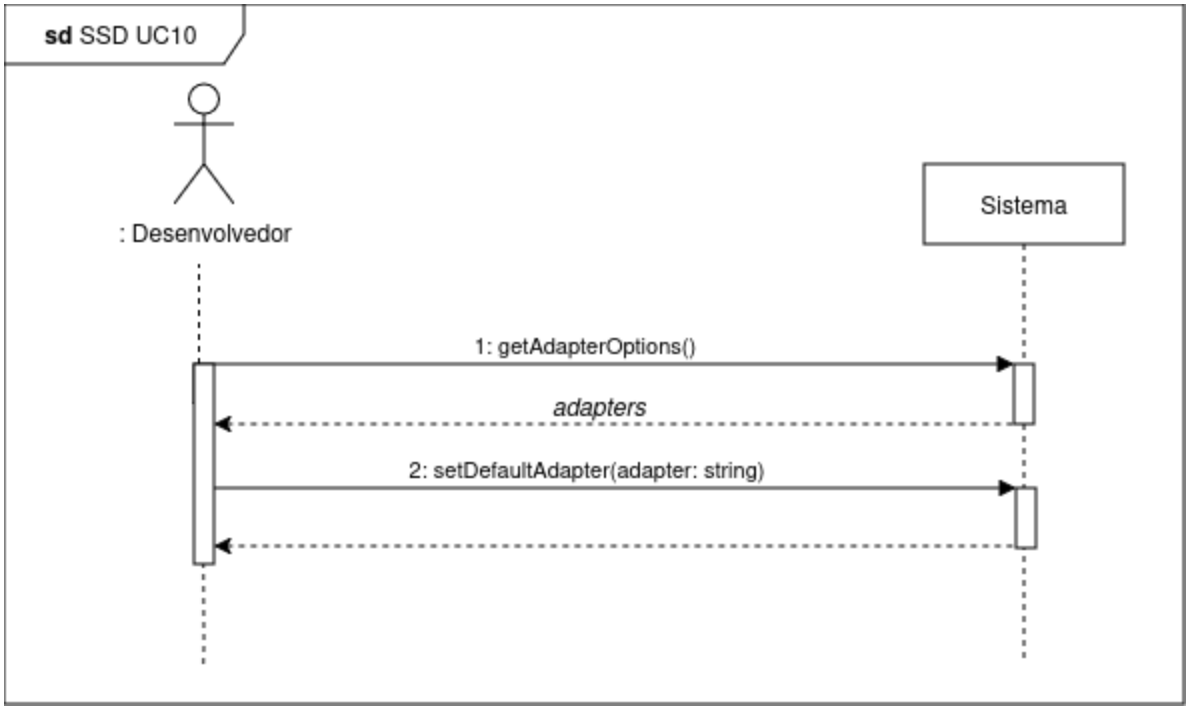

Figura 10. Diagrama de sequência de sistema do caso de uso UC10.

Contrato	Buscar os *adapters* disponiveis para uso
Operação	getAdapterOptions()
Referências cruzadas	• UC10: Definir *adapter* padrão
Pré-condições	Aba da extensão deve estar aberta
Pós-condições	Lista de *adapters* renderizada para o usuário

Tabela 7. Contrato de operação para buscar os *adapter* disponíveis para uso.

Contrato	Definir a *adapter* base para padronizar todas as requisições
Operação	setDefaultAdapter(adapter : string)
Referências cruzadas	• UC10: Definir *adapter* padrão
Pré-condições	Aba da extensão deve estar aberta
Pós-condições	*Adapter* padrão configurado com sucesso

Tabela 8. Contrato de operação para definir a *adapter* base para padronizar todas as requisições.

A Figura 11 apresenta o DSS08 relacionado ao caso de uso UC11. Nesse fluxo, o usuário defini a saída padrão para as funções e tipagens criadas pelo sistema. Além disso, a Tabela 9 faz referência ao mesmo caso de uso trazendo a normalização do contrato de operação do diagrama de sequência de sistema.

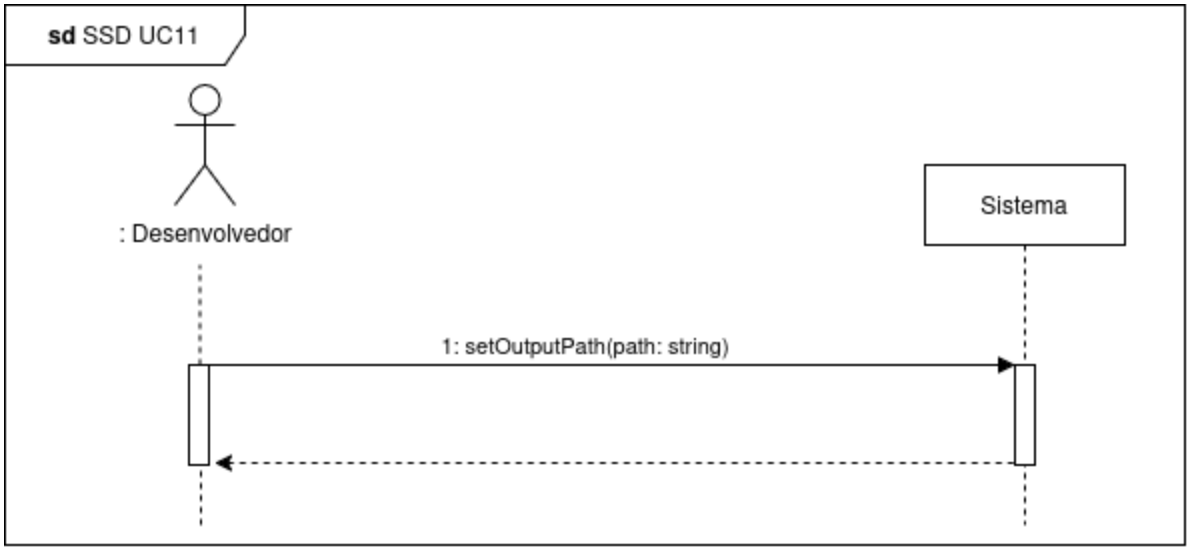

Figura 11. Diagrama de sequência de sistema do caso de uso UC11.

Contrato	Definir saída padrão para os tipos e funções
Operação	setOutputPath(path: string)
Referências cruzadas	• UC03: Criar funções que conectam aos *endpoints* • UC05: Criar tipos para *schemas* e respostas • UC11: Definir saída padrão
Pré-condições	Aba da extensão deve estar aberta
Pós-condições	Saída padrão configurada com sucesso

Tabela 9. Contrato de operação para definir saída padrão para os tipos e funções.

3. Modelos de Projeto

Nesta seção são apresentados os modelos de projeto por meio de diagramas em Linguagem de Modelagem Unificada (UML, do inglês *Unified Modeling Language*). A seção é composta pelos Diagramas de Classes, apresentados na Seção 3.1, os Diagramas de Sequência, apresentados na Seção 3.2, os Diagramas de Comunicação na Seção 3.3, a Arquitetura apresentada na Seção 3.4, os Diagramas de Estado, apresentados na Seção 3.5 e os Diagramas de Componentes e Implantação na Seção 3.6.

3.1 Diagrama de Classes

Nesta seção, é apresentado o diagrama de classes que modela objetos e suas relações e seus respectivos relacionamentos utilizados para tratar as informações da plataforma. O propósito desta seção consiste em apresentar todos os modelos de dados implementados, com o intuito de fornecer

uma visão geral da arquitetura das classes que compõem o sistema e assegurar o funcionamento correto da plataforma.

A Figura 12 apresenta o pacote de *parsers*, o qual contém as classes *PathParser, OpenAPILoader, SchemaParser, OpenAPI, RawOpenAPI, Info, PathItem, Schema, SchemaProperties, Operation, OperationResponse, OperationRequestBody, OperationSchema* e *Parameter,* responsáveis pelo armazenamento e tratamento dos dados da *OpenAPI* que serão utilizados pelas classes de *generators* para criar as funções e suas tipagens.

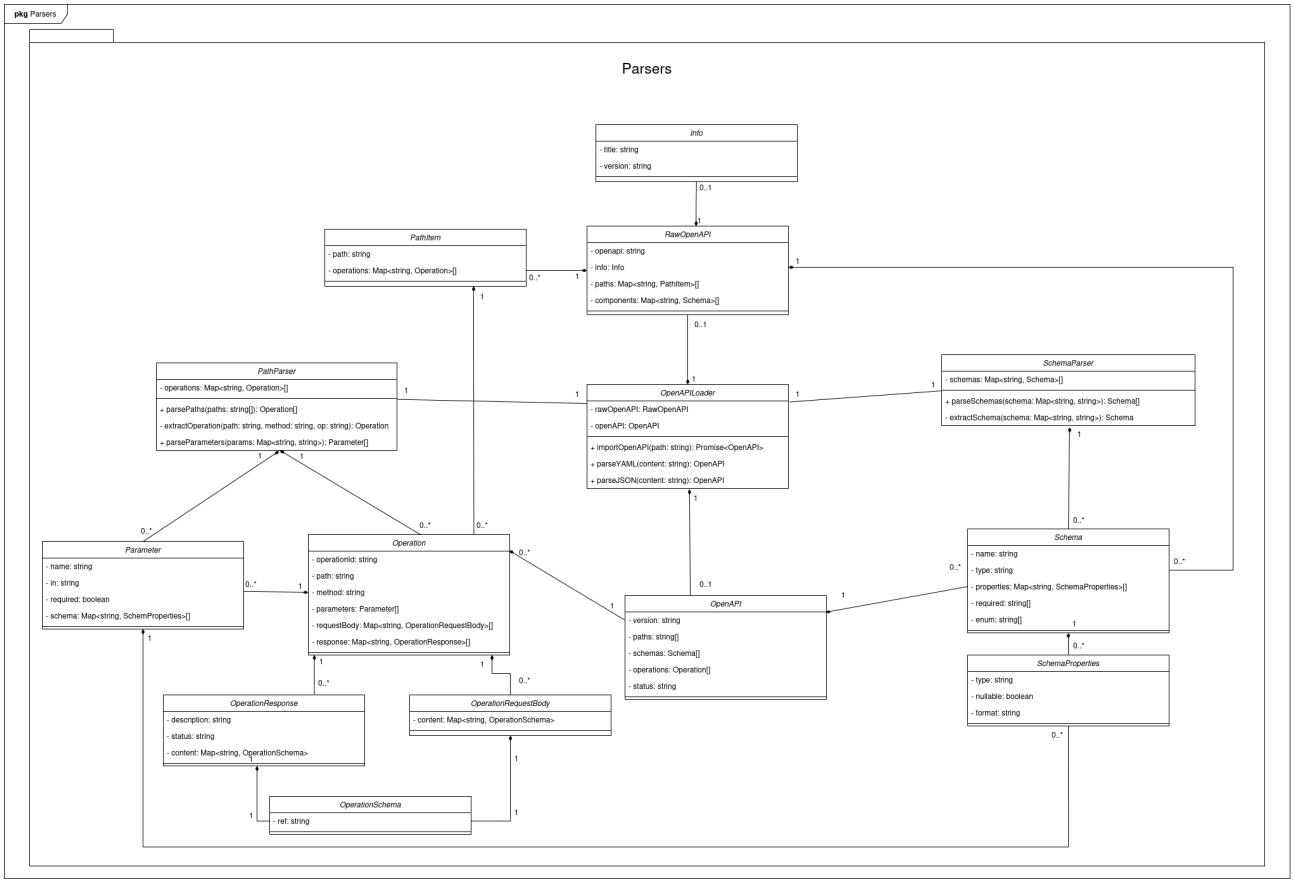

Figura 12. Diagrama de Classe de *parsers* do sistema.

A Figura 13 apresenta o pacote de *generators*, o qual contém as classes *SDKGenerator* e *TypeGenerator* responsáveis por gerar as funções que conectam aos *endpoints* e a tipagem dos *schemas, headers, body, param*.

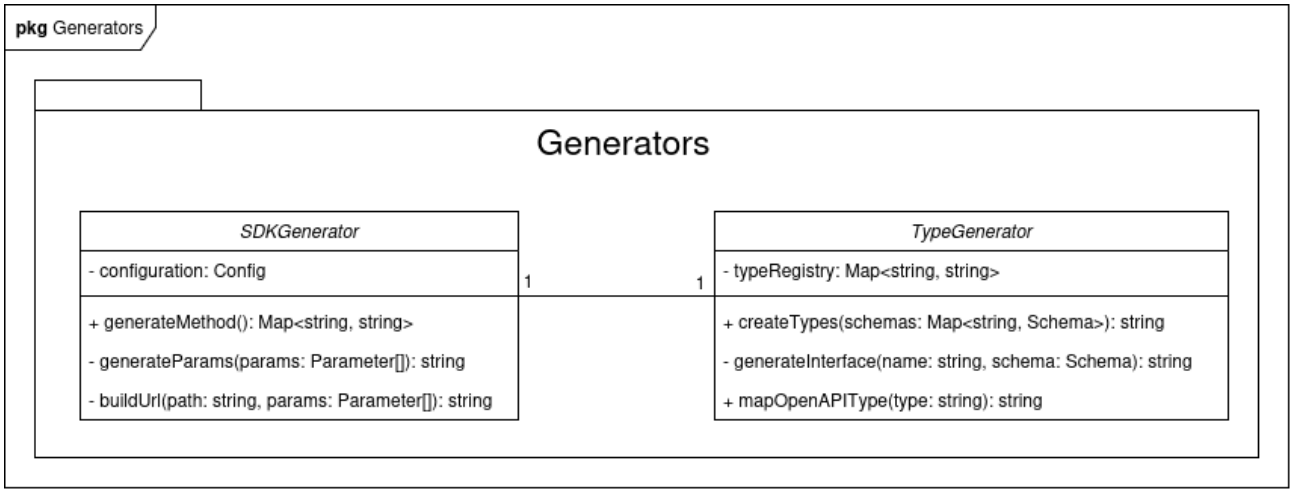

Figura 13. Diagrama de Classe de *generators* do sistema.

A Figura 14 apresenta o pacote de *config*, o qual contém as classes *ConfigManager* e *Config* responsáveis por toda a gestão das configurações do sistema.

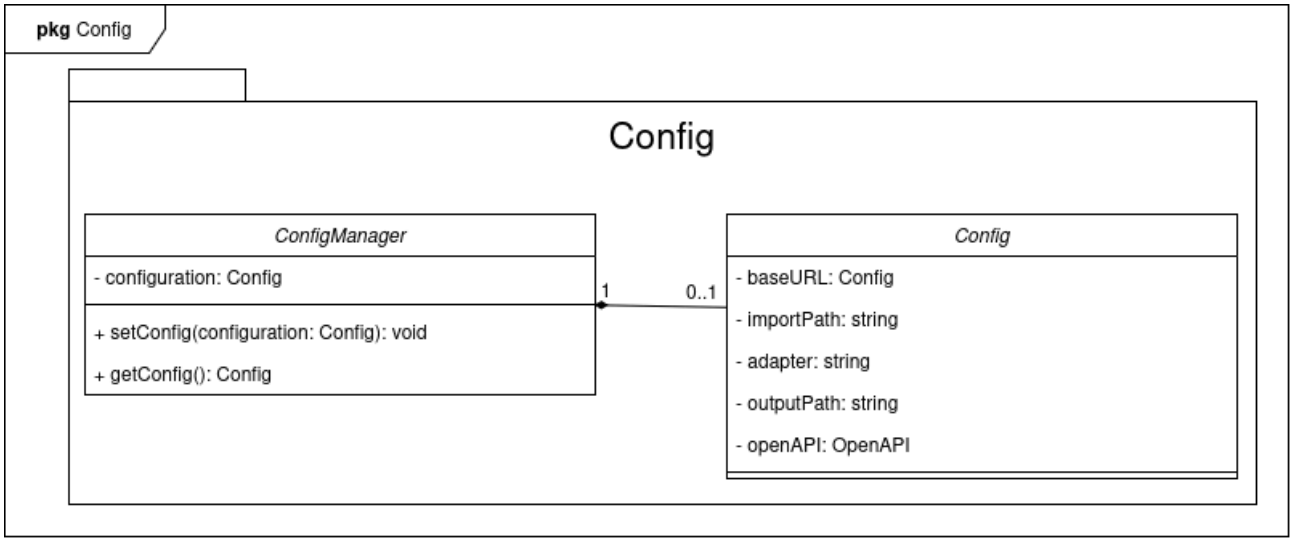

Figura 14. Diagrama de Classe de *config* do sistema.

A Figura 15 apresenta o pacote de *UI,* o qual contém a classe *ExtensionController* responsável por controlar toda interação da à interface com o usuário , esse pacote atua como uma camada anticorrupção para integração com o *Visual Studio Code*. A classe *ExtensionController* refere-se a camada de comunicação externa.

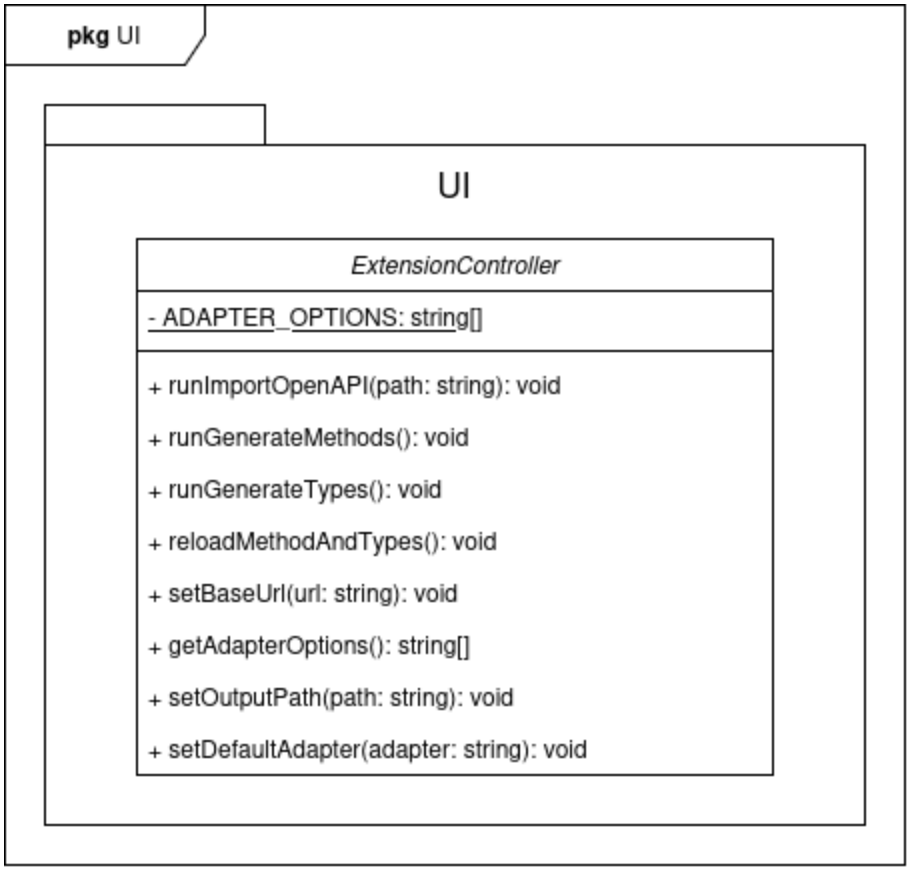

Figura 15. Diagrama de Classe de *UI* do sistema.

3.2 Diagramas de Sequência

Nesta seção, são apresentados diagramas de sequência referentes aos casos de uso do sistema implementado. Esses diagramas demonstram o fluxo entre os componentes para cada cenário de uso, dessa forma, eles apresentam como funcionam os fluxos de chamadas entre as entidades que participam de uma interação. As entidades desses diagramas foram apresentadas no diagrama de classes das Figura 12, Figura 13, Figura 14 e Figura 15.

A Figura 16 representa o diagrama de sequência responsável pelo fluxo de abrir a aba da extensão. Este fluxo contempla a interação do usuário com o *Visual Studio Code* para visualização dos dados. Este diagrama faz referência ao caso de uso UC01.

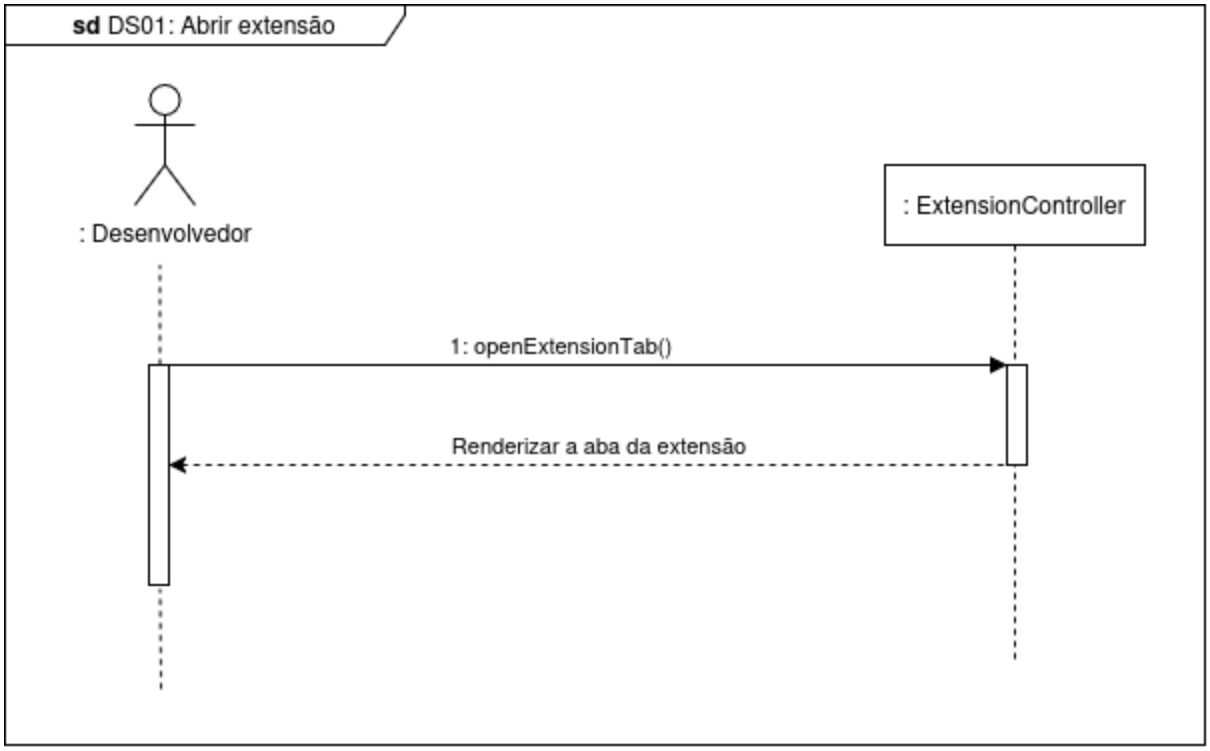

Figura 16. Diagrama de Sequência do caso de uso UC01.

A Figura 17 representa o diagrama de sequência responsável pelo fluxo de importação da documentação da *OpenAPI*. Neste fluxo, o usuário Desenvolvedor digita o caminho para a documentação *OpenAPI*, podendo ser uma *URL* ou caminho local, a partir deste, são extraídos os dados dos *schemas, operations,* e *parameters.* Este diagrama faz referência ao caso de uso UC02.

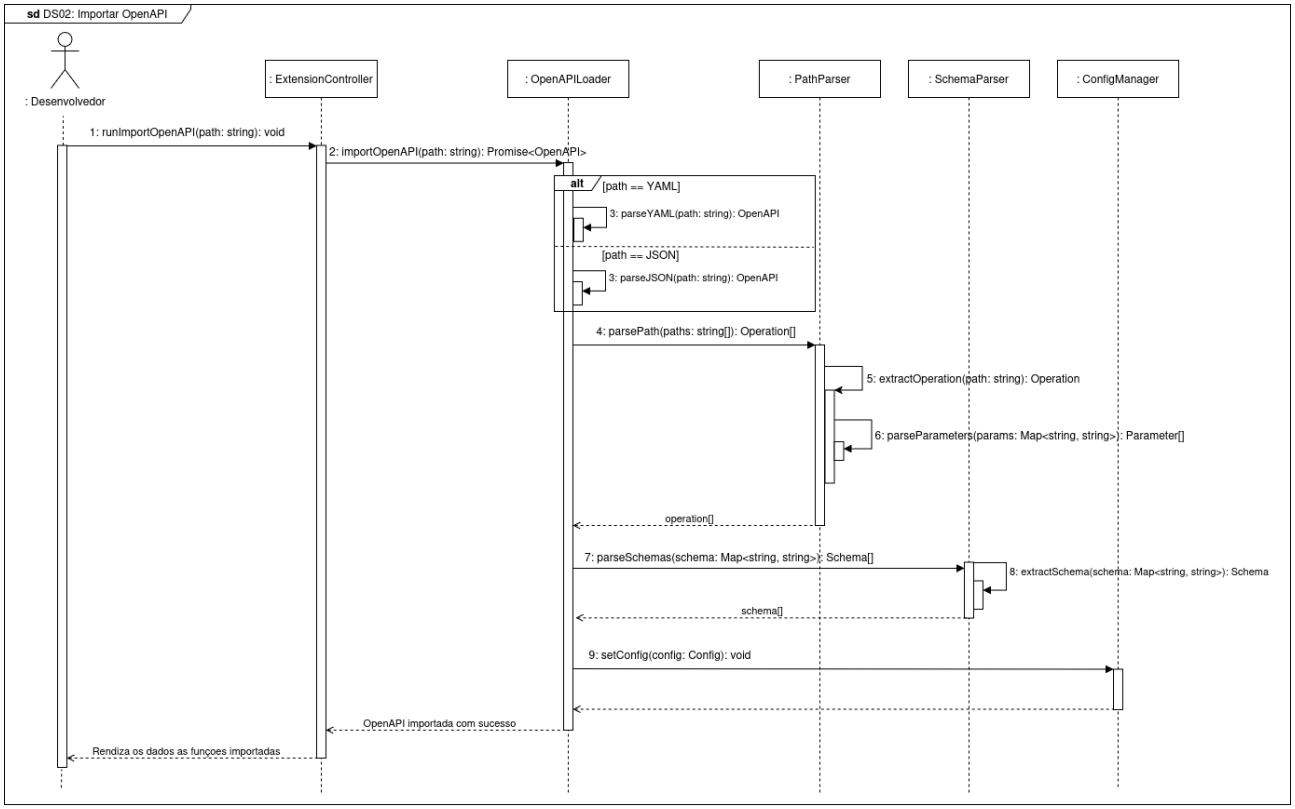

Figura 17. Diagrama de Sequência do caso de uso UC02.

A Figura 18 representa o diagrama de sequência responsável pelo fluxo de criação das funções e tipos que conectam aos *endpoints*. Neste fluxo, o usuário Desenvolvedor utiliza a documentação da OpenAPI importada previamente, para gerar as funções que conectam aos endpoints e as suas tipagens. Este diagrama faz referência ao caso de uso UC03, UC04, UC05, UC06 e UC07.

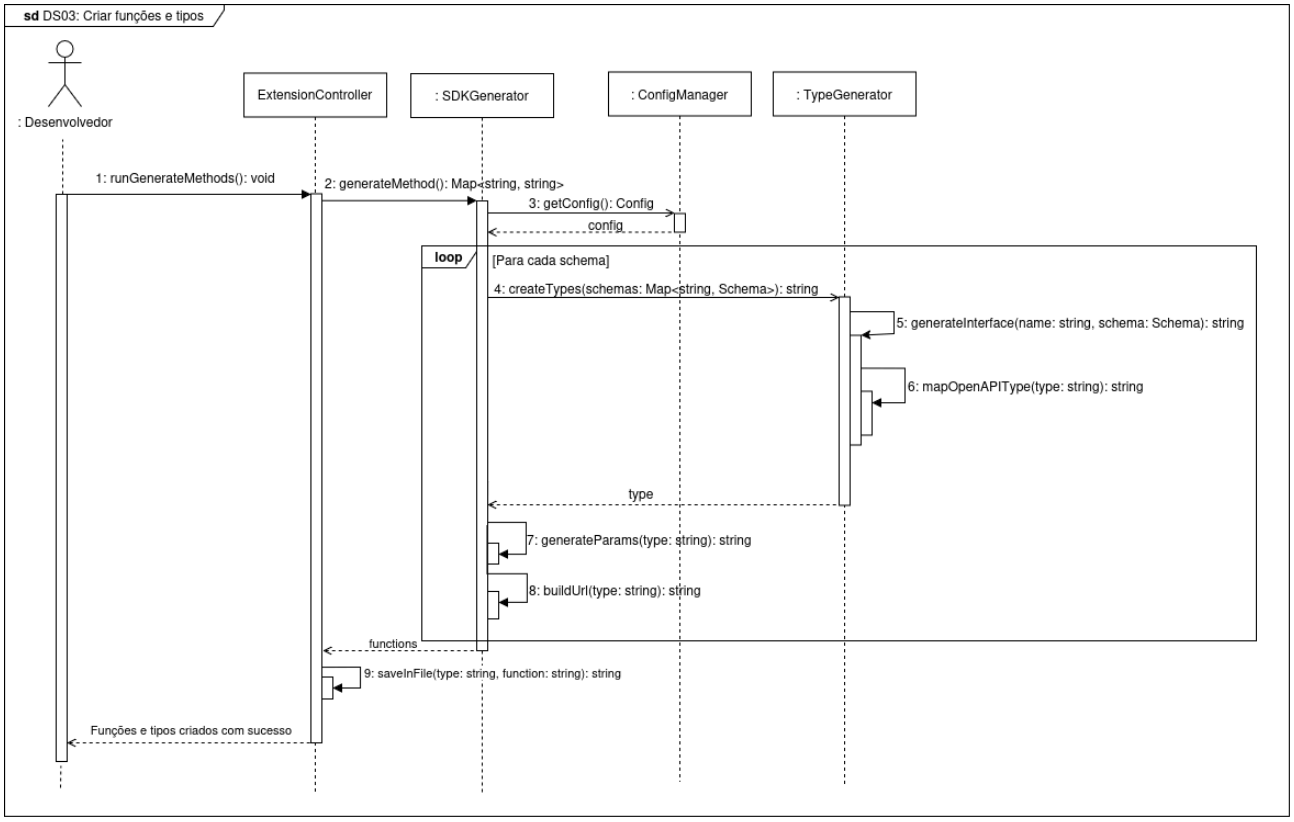

Figura 18. Diagrama de Sequência do caso de uso UC03, UC04, UC05, UC06 e UC07.

A Figura 19 representa o diagrama de sequência responsável pelo fluxo de criação da tipagem para as funções que conectam aos *endpoints*. Neste fluxo, o usuário Desenvolvedor utiliza a versão mais recente da *OpenAPI* salva nas configurações, para gerar as tipagens. Este diagrama faz referência ao caso de uso UC05, UC06 e UC07

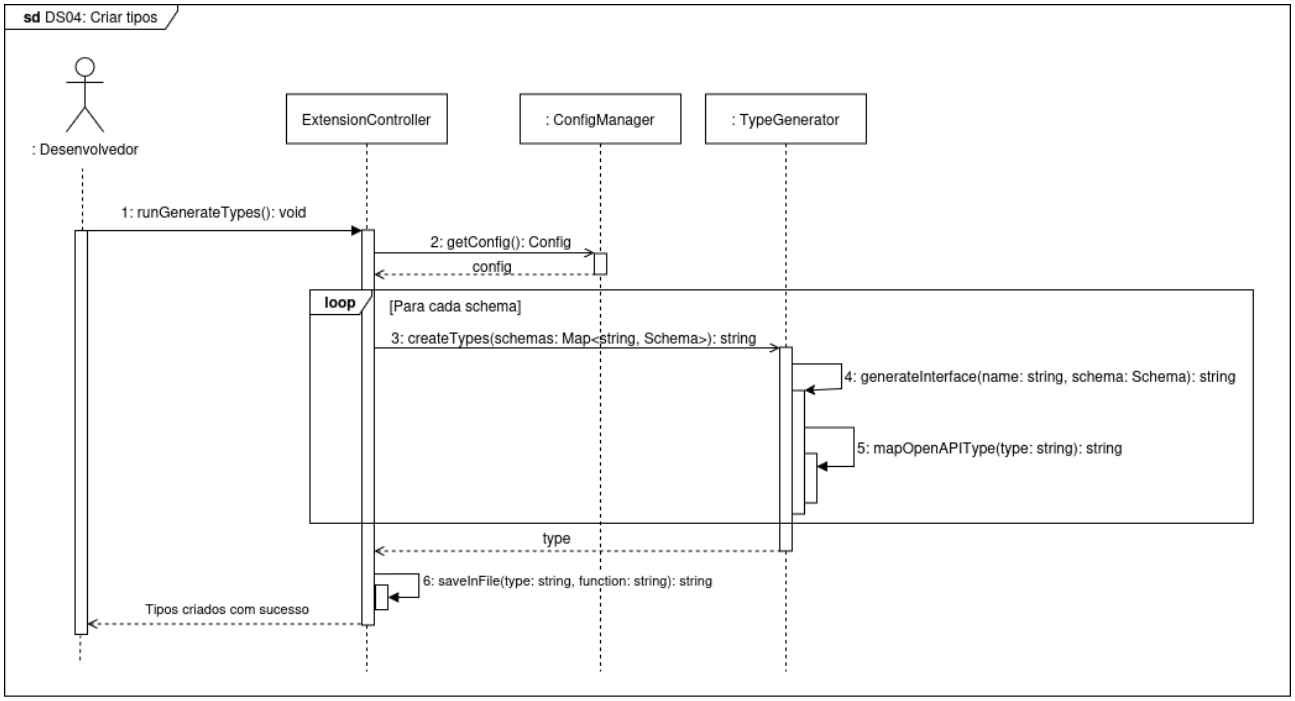

Figura 19. Diagrama de Sequência do caso de uso UC05, UC06 e UC07.

A Figura 20 representa o diagrama de sequência responsável pelo fluxo de atualização da implementação das funções e tipos que conectam aos *endpoints*. Neste fluxo, o usuário Desenvolvedor recupera a versão mais atualizada da *OpenAPI*, para gerar novamente as funções que conectam aos endpoints e as suas tipagens. Este diagrama faz referência ao caso de uso UC02, UC03, UC04, UC05, UC06, UC07 e UC08.

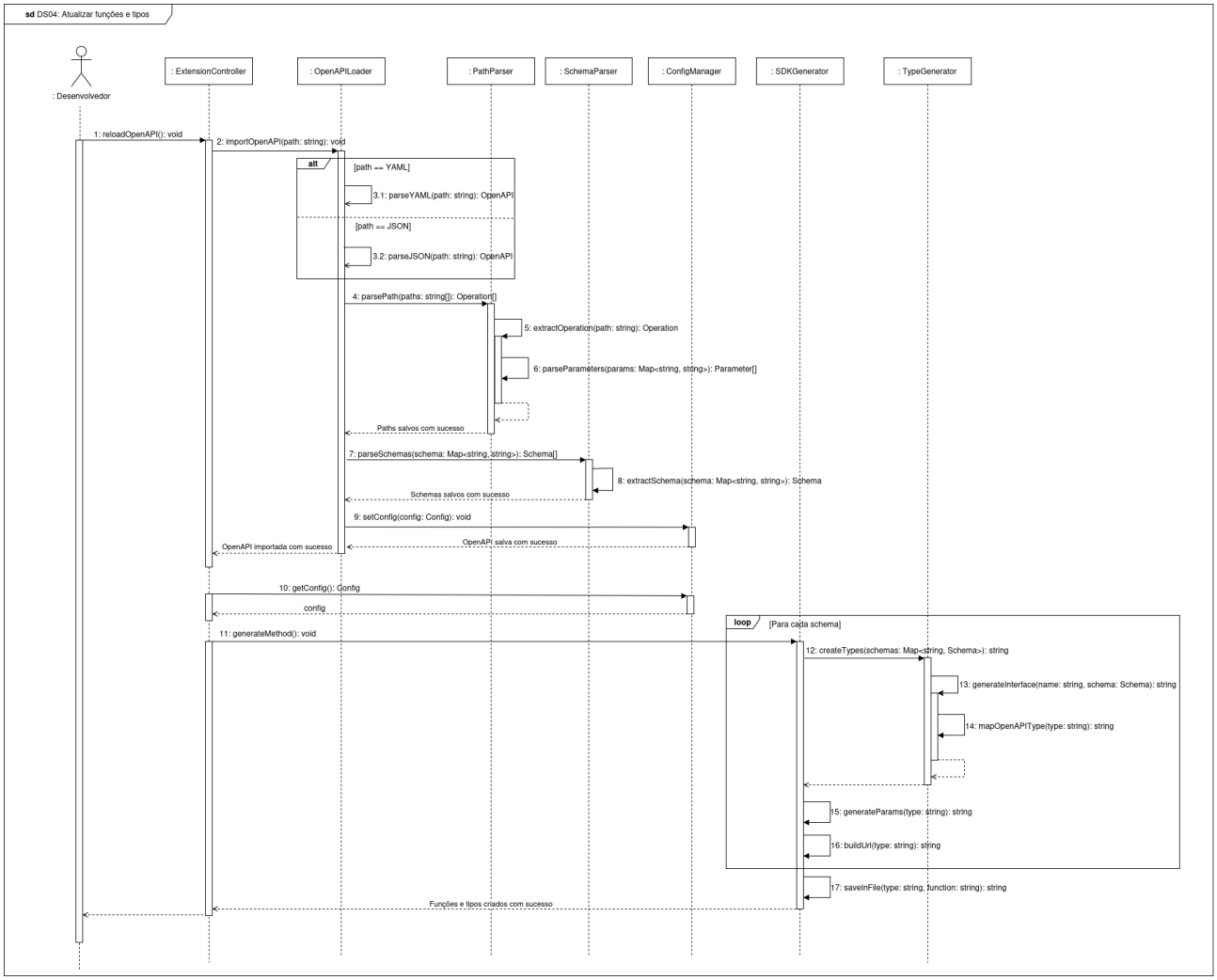

Figura 20. Diagrama de Sequência do caso de uso UC02, UC03, UC04, UC05, UC06, UC07 e UC08.

A Figura 21 representa o diagrama de sequência responsável pelo fluxo de salvar a *URL* base da extensão. Neste fluxo, o usuário Desenvolvedor pode salvar a *URL base* nas configurações do sistema. Este diagrama faz referência ao caso de uso UC09.

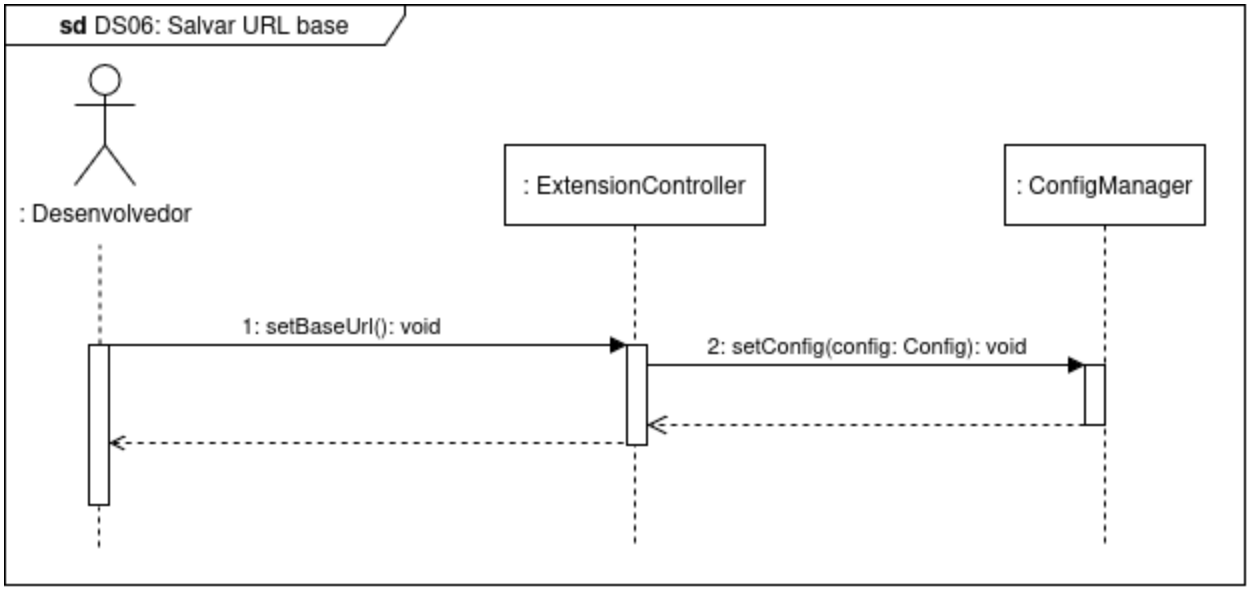

Figura 21. Diagrama de Sequência do caso de uso UC09.

A Figura 22 representa o diagrama de sequência responsável pelo fluxo de definir o *adapter* padrão da extensão. Neste fluxo, o usuário Desenvolvedor escolhe entre os *adapters* disponíveis para uso e salvar o *adapter* padrão nas configurações do sistema. Este diagrama faz referência ao caso de uso UC10.

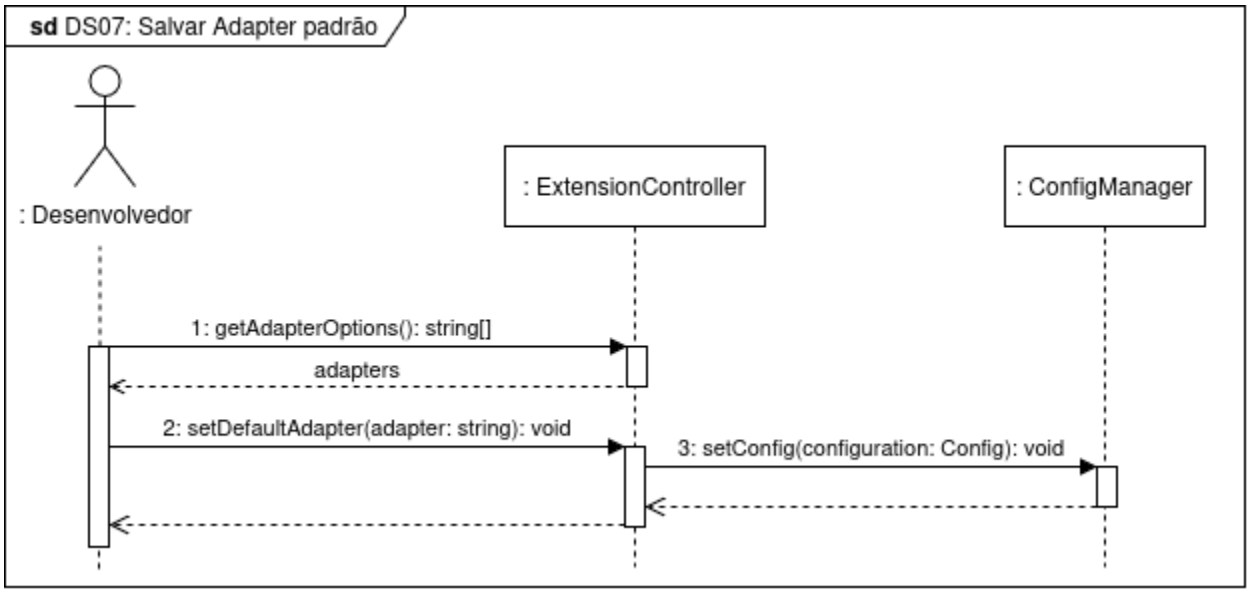

Figura 22. Diagrama de Sequência do caso de uso UC10.

A Figura 23 representa o diagrama de sequência responsável pelo fluxo de definir o caminho de sáida das funções e sua tipagens. Neste fluxo, o usuário Desenvolvedor escolhe o diretorio de saída padrão para todas as funções e suas tipagens criadas pelo sistema. Este diagrama faz referência ao caso de uso UC11.

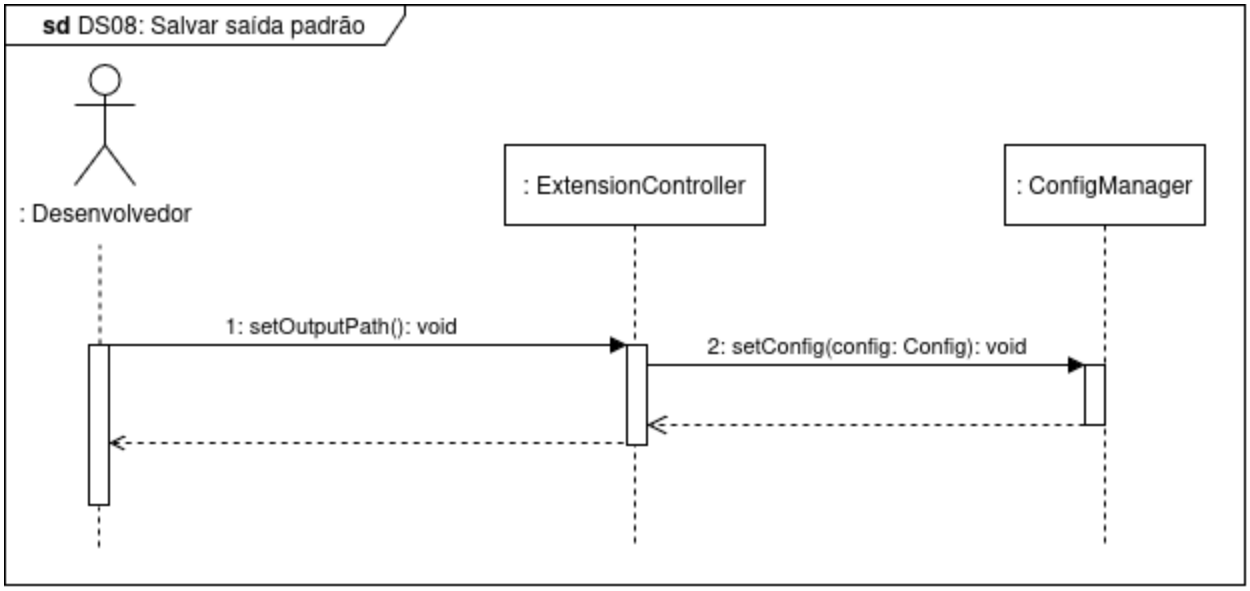

Figura 23. Diagrama de Sequência do caso de uso UC11.

3.3 Diagramas de Comunicação

Nesta seção, são apresentados diagramas de comunicação que modelam as trocas de mensagens dos casos de uso do sistema implementado. Esses diagramas representam as interações entre componentes do sistema para realizar uma determinada funcionalidade. Dessa forma, os principais fluxos da aplicação são representados como meio de documentar as comunicações que os compõem.

A Figura 24 representa o diagrama de comunicação responsável pelo fluxo de abrir a aba da extensão. Nesse fluxo, o usuário Desenvolvedor realiza uma comunicação com o *Visual Studio Code*, para poder abrir a aba da extensão. Este diagrama faz referência ao caso de uso UC01.

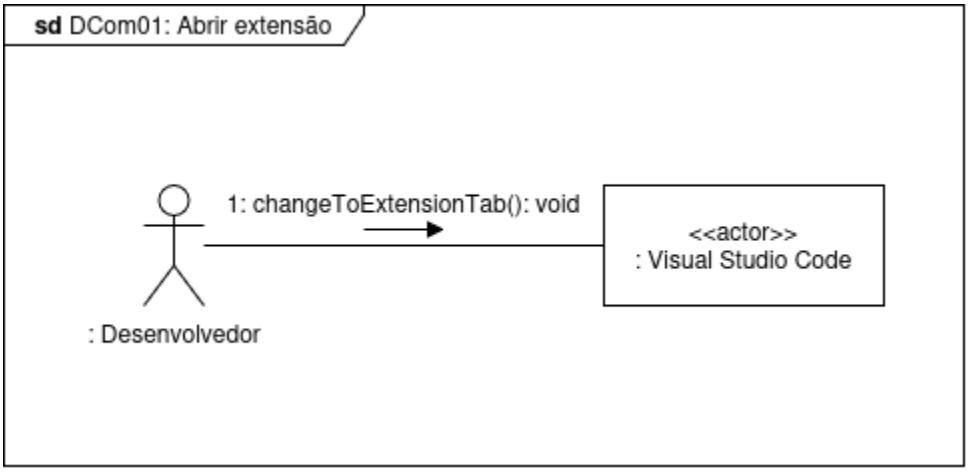

Figura 24. Diagrama de Comunicação do caso de uso UC01.

A Figura 25 representa o diagrama de comunicação responsável pelo fluxo de importação da documentação da *OpenAPI*. Nesse fluxo, o usuário Desenvolvedor realiza uma comunicação com os pacotes de *ui, parsers* e *config*, que possuem as classes *ExtensionController*, *OpenAPILoader, PathParser, SchemaParser* e *ConfigManager*, aonde o *ExtensionController* comunica o *OpenAPILoader* para importar a documentação *OpenAPI*, após baixar a documentação comunica o *PathParser* para extrair e formatar os dados dos *paths* e *parameters,* que em seguida comunica o *SchemaParser* para extrair e formatar os dados do *schema*, e concluindo comunica com o *ConfigManager* para salvar os dados formatados do *OpenAPI*. Este diagrama faz referência ao caso de uso UC02.

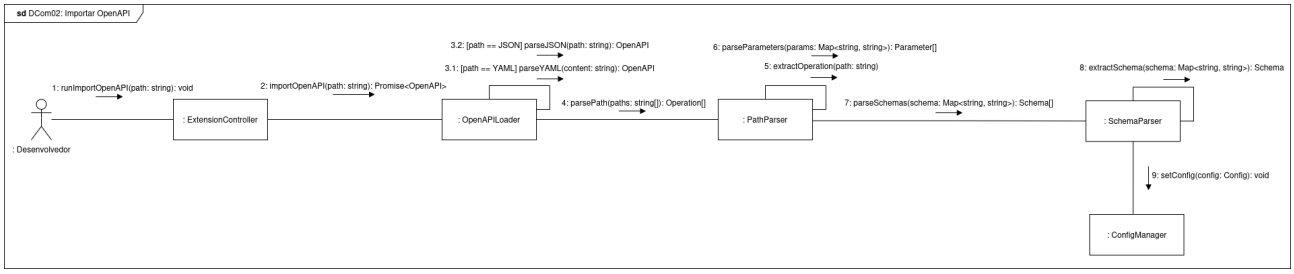

Figura 25. Diagrama de Comunicação do caso de uso UC02.

A Figura 26 representa o diagrama de comunicação responsável pelo fluxo de criação das funções e tipos que conectam aos *endpoints*. Nesse fluxo, o usuário Desenvolvedor realiza uma comunicação com os pacotes de *ui, generators* e *config*, que possuem as classes *ExtensionController, SDKGenerator, TypeGenerator* e *ConfigManager*, aonde o *ExtensionController* comunica com o *SDKGenerator* para gerar as funções que conectam aos *endpoints*, mas primeiro, comunica com o *ConfigManager* para buscar as configurações mais recentes da extensão, logo após comunica o *TypeGenerator* para gerar a tipagem das funções, devolvendo para o *SDKGenerator*, e assim, conclui salvando as funções no diretório apontado pela *config*. Este diagrama faz referência ao caso de uso UC03, UC04, UC05, UC06 e UC07.

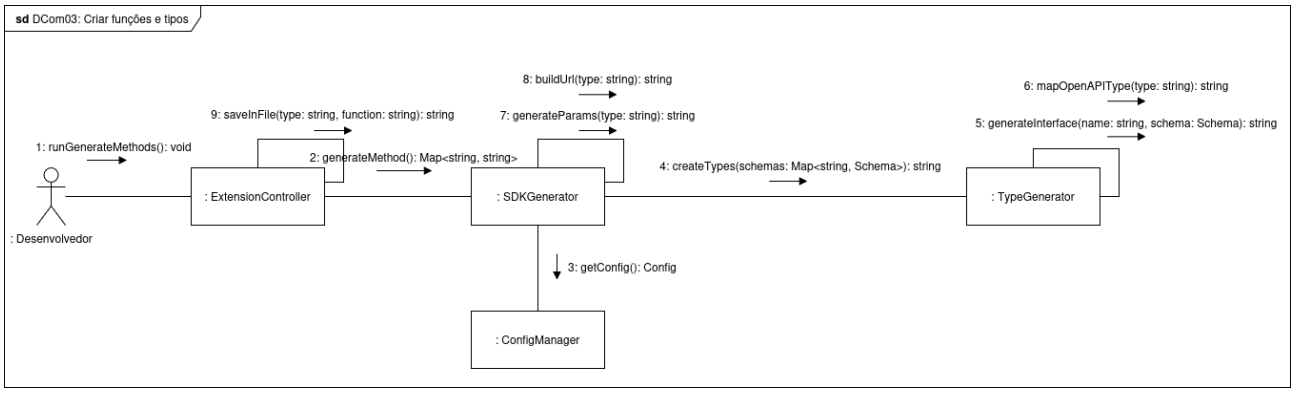

Figura 26. Diagrama de Comunicação do caso de uso UC03, UC04, UC05, UC06 e UC07.

A Figura 27 representa o diagrama de comunicação responsável pelo fluxo de criação da tipagem para as funções que conectam aos *endpoints*. Nesse fluxo, o usuário Desenvolvedor realiza uma comunicação com os pacotes de *ui, generators* e *config*, que possuem as classes *ExtensionController, TypeGenerator* e *ConfigManager* aonde o *ExtensionController* comunica com o *ConfigManager* para buscar as configurações mais recentes, logo em seguida o *TypeGenerator* é comunicado para gerar a tipagem com base nas configurações da *OpenAPI*. Este diagrama faz referência ao caso de uso UC05, UC06 e UC07.

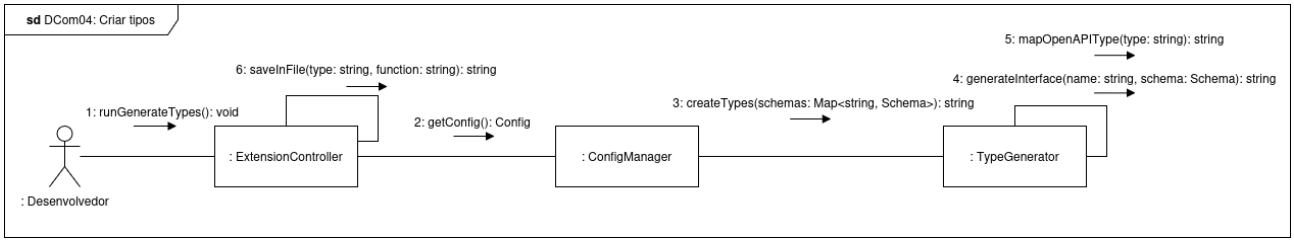

Figura 27. Diagrama de Comunicação do caso de uso UC05, UC06 e UC07.

A Figura 28 representa o diagrama de comunicação responsável pelo fluxo de atualização da implementação das funções e tipos que conectam aos *endpoints*. Nesse fluxo, o usuário Desenvolvedor realiza uma comunicação com os pacotes de *ui*, *parsers, generators* e *config*, que possuem as classes *ExtensionController, OpenAPILoader, PathParser, SchemaParser*, *SDKGenerator, TypeGenerator* e *ConfigManager* para atualizar a implementação das funções que conectam aos *endpoints* e as suas tipagens, com os dados mais recentes da documentação externa *OpenAPI*. Este diagrama faz referência ao caso de uso UC04, UC05, UC06 e UC07.

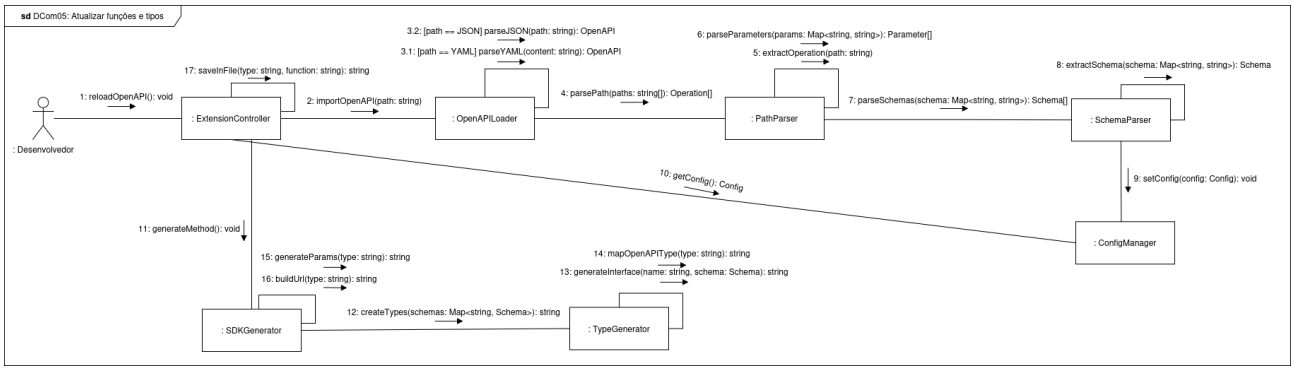

Figura 28. Diagrama de Comunicação do caso de uso UC04, UC05, UC06 e UC07.

A Figura 29 representa o diagrama de comunicação responsável pelo fluxo de salvar a *URL* base da extensão. Neste fluxo, o usuário Desenvolvedor realiza uma comunicação com os pacotes de *ui* e *config*, que possuem as classes *ExtensionController* e *ConfigManager* para poder salvar a *URL base.* Este diagrama faz referência ao caso de uso UC09.

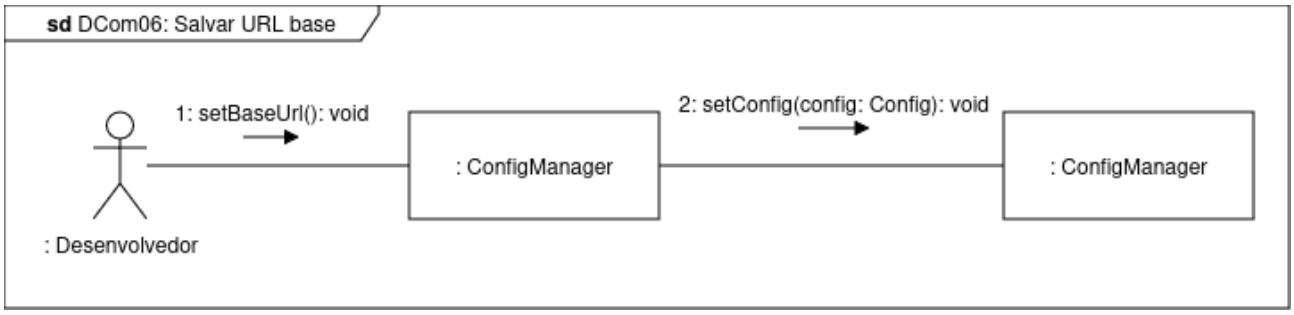

Figura 29. Diagrama de Comunicação do caso de uso UC09.

A Figura 30 representa o diagrama de comunicação responsável pelo fluxo de salvar o *adapter* padrão da extensão. Neste fluxo, o usuário Desenvolvedor realiza uma comunicação com os pacotes de *ui* e *config*, que

possuem as classes *ExtensionController* e *ConfigManager* para poder salvar o *adapter* padrão, com base nos *adapters* disponíveis para uso. Este diagrama faz referência ao caso de uso UC10.

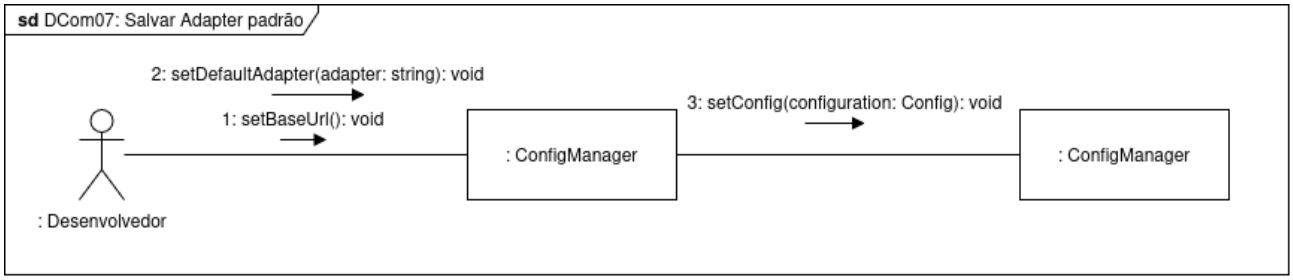

Figura 30. Diagrama de Comunicação do caso de uso UC10.

A Figura 31 representa o diagrama de comunicação responsável pelo fluxo de salvar o caminho de saída padrão da extensão. Neste fluxo, o usuário Desenvolvedor realiza uma comunicação com os pacotes de *ui* e *config*, que possuem as classes *ExtensionController* e *ConfigManager* para poder salvar o diretório de saída padrão das funções e tipagens criadas pelo sistema. Este diagrama faz referência ao caso de uso UC11.

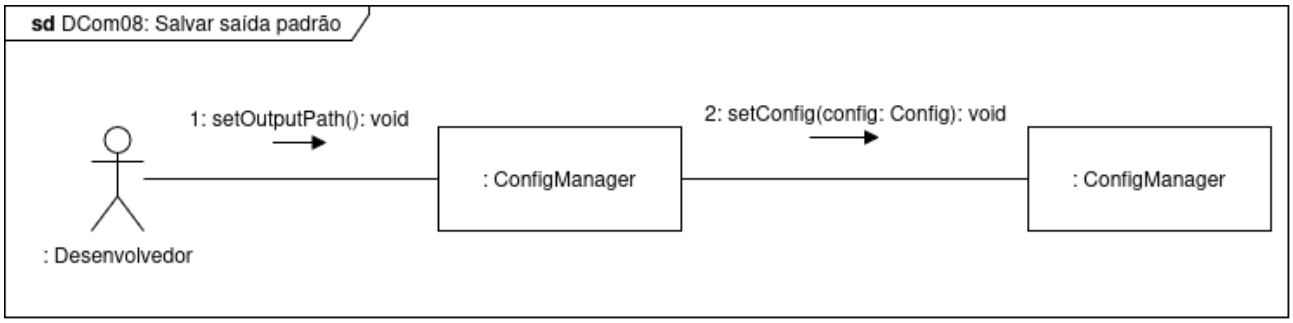

Figura 31. Diagrama de Comunicação do caso de uso UC11.

3.4 Arquitetura

Nesta seção, é apresentada a modelagem da arquitetura lógica do sistema. Essa modelagem foi feita por meio do diagrama de pacotes, o qual descreve a estrutura organizacional do sistema, mostrando como os elementos do sistema estão organizados e hierarquizados em pacotes e como esses pacotes se relacionam entre si.

A Figura 32 representa o diagrama de pacotes da plataforma, temos o pacote *src* o qual serve como pacote pai para todos os outros pacotes do sistema, em seguida temos os pacotes *Services, UI* e *Utils,* o pacote *Services* é um dos pacotes principais, pois contem as classes de serviço que importam, formatam e geram as funções e tipos do sistema, sendo assim uma parte crucial, o pacote *UI* contem as classes responsáveis pela interação com a interface do usuário e por último, o pacote *Utils* que contem funções auxiliares que serão utilizadas para momentos como formatação de data para padrão internacional.

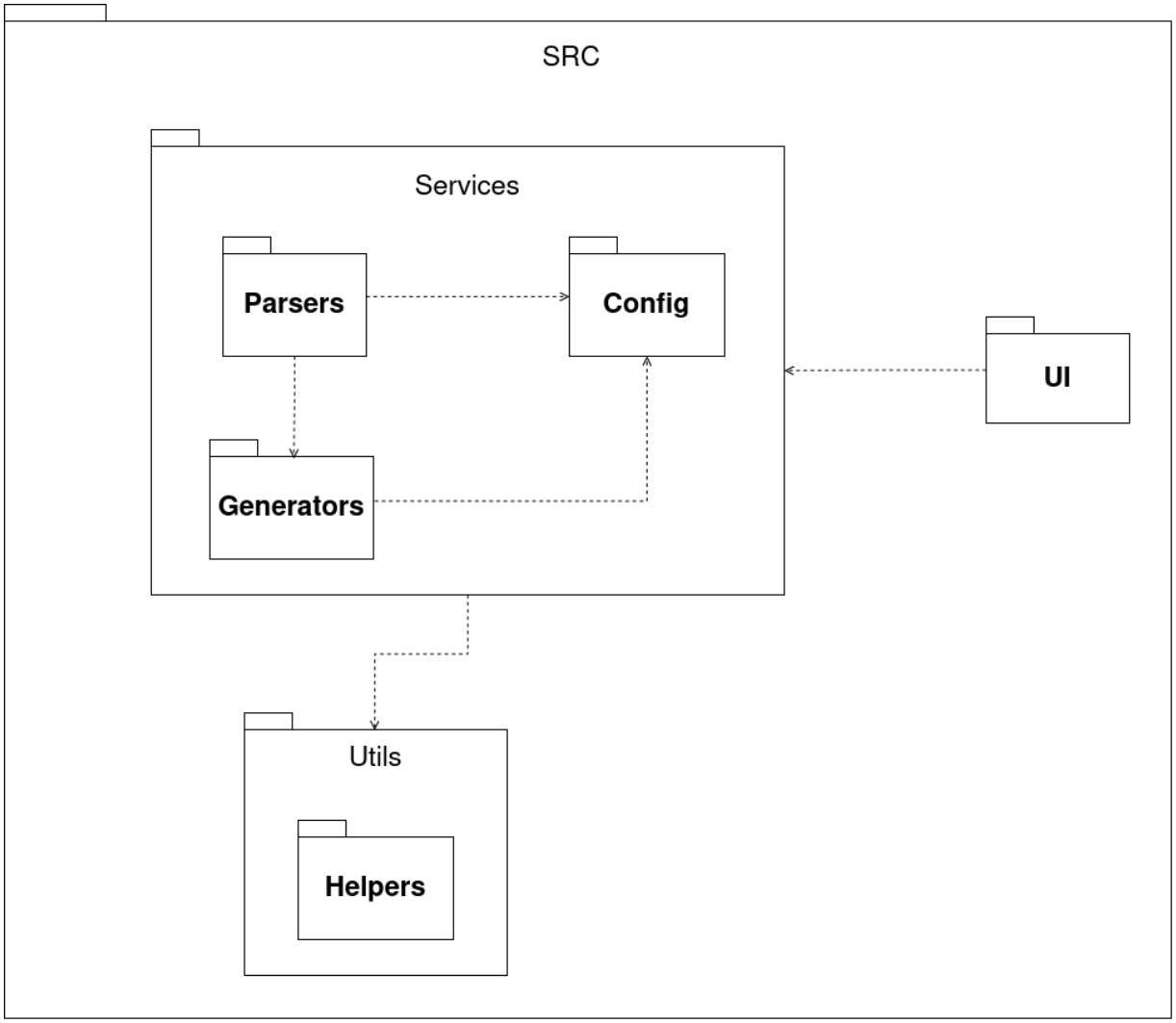

Figura 32. Diagrama de pacotes do sistema Swagger to SDK.

3.5 Diagramas de Estados

Nesta seção, são apresentados os diagramas de estados do sistema implementado. Esses diagramas modelam os possíveis estados que um objeto de uma determinada entidade pode estar, e especificam as transições entre eles.

A Figura 33 apresenta o diagrama de estados do objeto da entidade *OpenAPI*. Nesse diagrama, é apresentado que o objeto se inicia com o *status* "NOT_INITIALIZED". Assim que a importação é finalizada e bem sucedida, o seu *status* e alterado para "COMPLETED". Caso ocorra algum erro durante a importação, o *status* passa a ser "FAILED". Nos dois últimos casos, esse é o estado final que o objeto pode assumir, não havendo nenhuma transição possível uma vez que ele chega em um desses estados. Esse diagrama se relaciona com o caso de uso UC02.

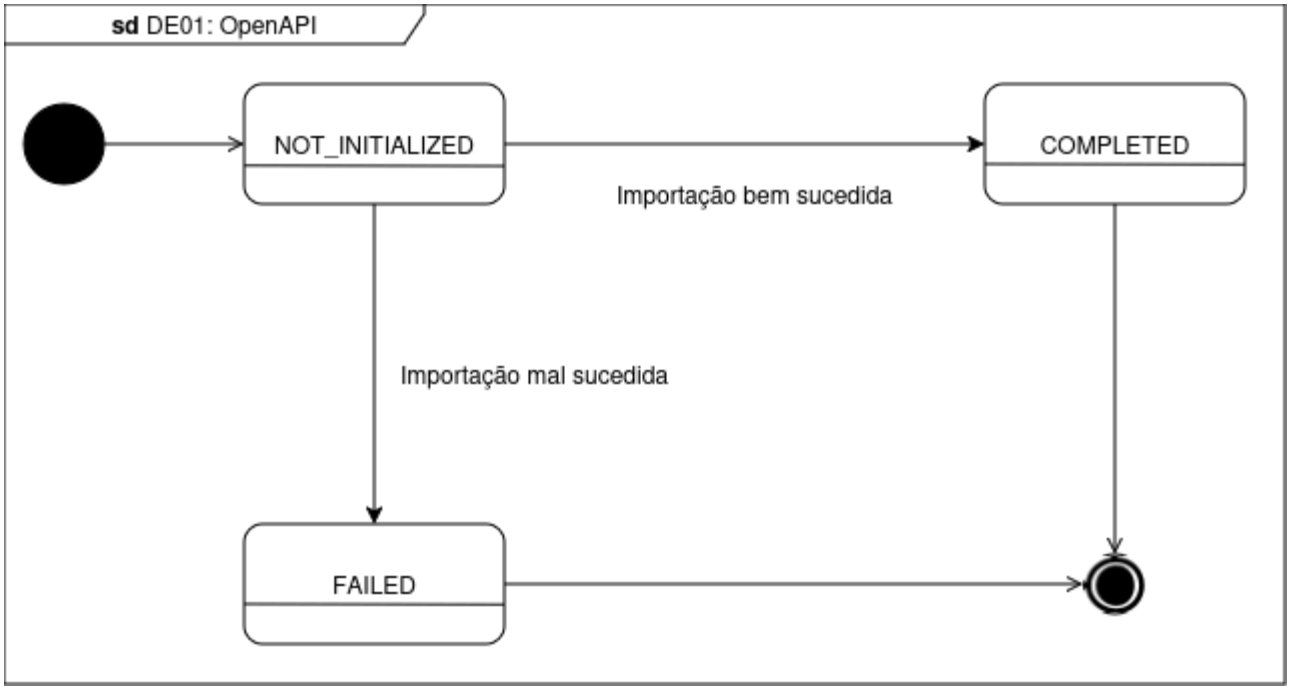

Figura 33. Diagrama de Estado do caso de uso UC02.

3.6 Diagrama de Componentes e Implantação.

Nesta seção são apresentados os diagramas de componentes e implantação da plataforma. O diagrama de componentes modela os principais componentes do sistema, apontando suas composições e dependências, bem como as interfaces que eles consomem e que eles proveem. Enquanto isso, o diagrama de implantação representa os recursos físicos e de software necessários para o sistema ser implantado adequadamente.

A Figura 34 é apresentado o diagrama de componentes do sistema, o qual contém um componente principal que é a propria aplicação Swagger to SDK, que possue uma comunicação com um unico componente o *Visual Studio Code,* por meio de uma interface. Também podemos ter noção dos pacotes internos do sistema, como o pacote *UI* que é responsavel por toda a interação do usuário com o sistema interno, e os pacotes secundários *Parsers, Generators* e *Config* que se comunicam entre sí. Por fim temos o *FileSystem*, os quais são requidos pelo componente *Visual Studio Code,* sendo uma dependencia que será utilizada pelo Swagger to SDK.

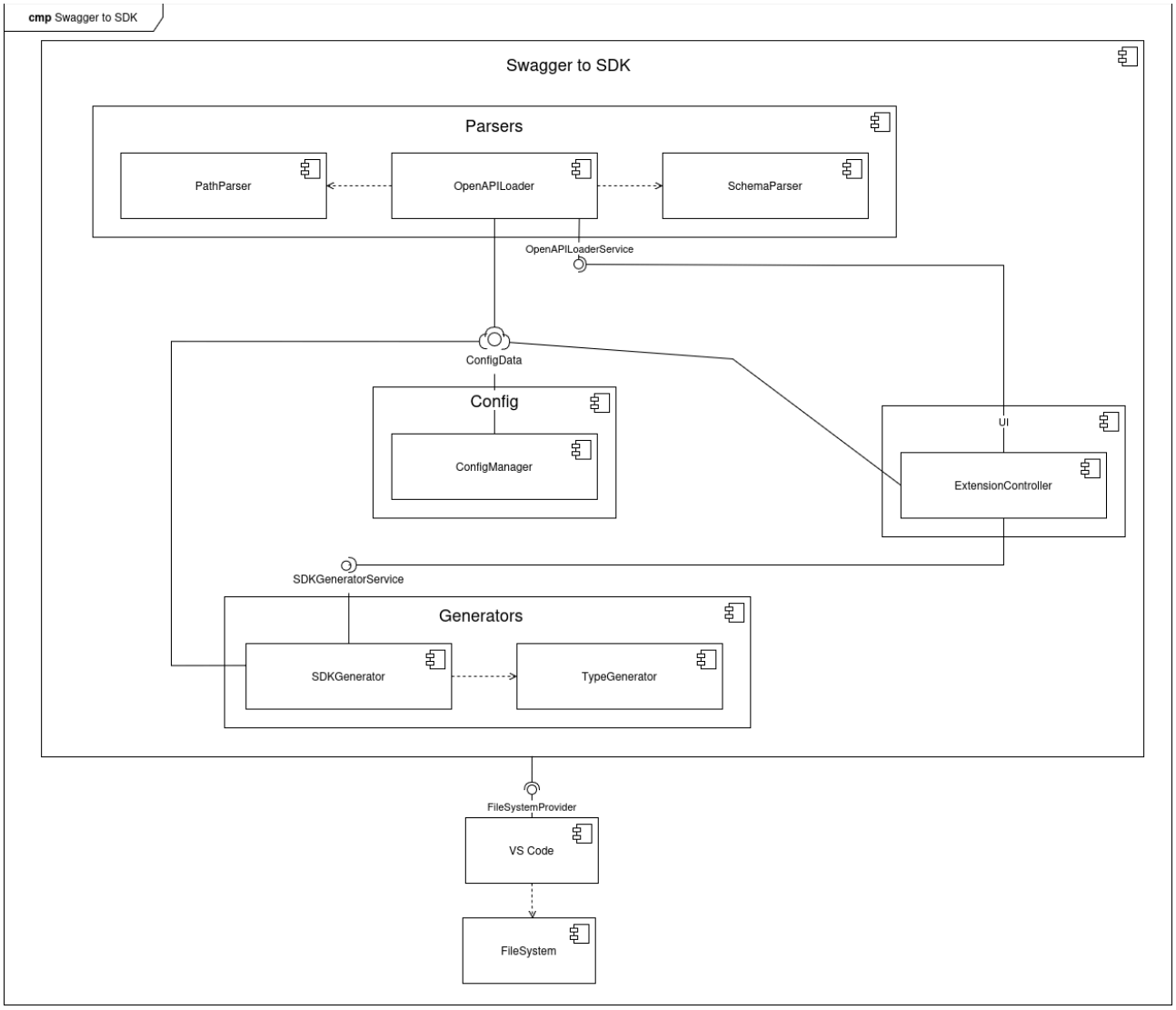

Figura 34. Diagrama de Componentes do sistema *Swagger to SDK*.

A Figura 35 apresenta o diagrama de implantação do sistema, pelo sistema Swagger to SDK se tratar de uma extensão para o *Visual Studio Code,* ele é executado na própria maquina do usuário, no contexto de extensões de *IDE*. Por isso, no diagrama de implantação é apresentado o *Visual Studio Code* como o ambiente de execução da aplicação. Mesmo sendo um ambiente de execução, o *Visual Studio Code* possui dependências de outros componentes executados na própria máquina do usuário e estão diretamente vinculadas ao sistema operacional deste, o sistema de arquivos.

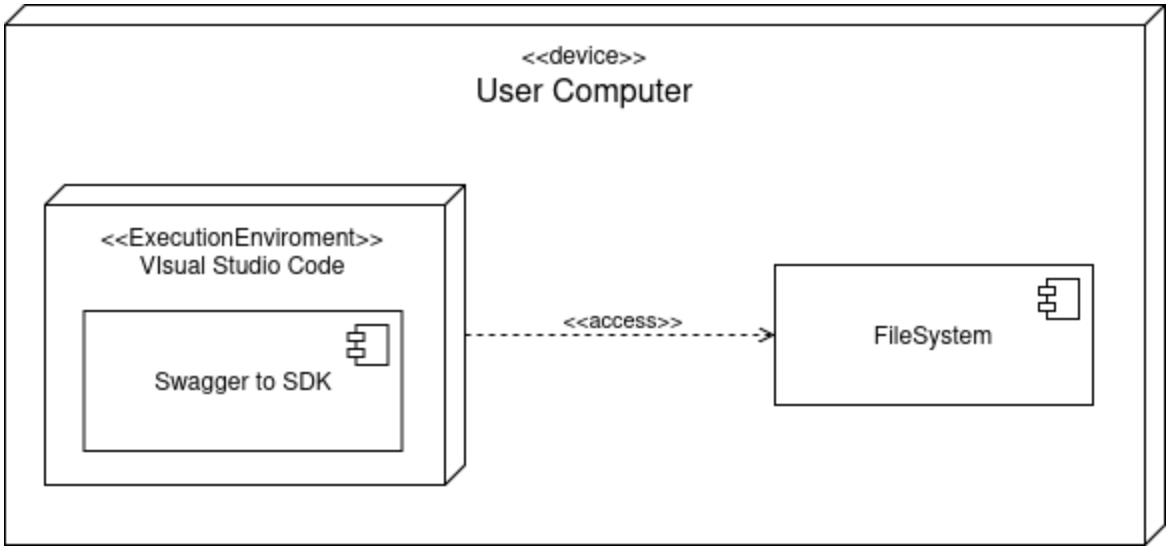

Figura 35. Diagrama de Implantação do sistema *Swagger to SDK*.

4. Projeto de Interface com Usuário

Nesta seção, são apresentados os protótipos das interfaces do sistema. Para criação destas interfaces foi utilizado o *software Figma*, o qual disponibiliza a criação de interfaces com base em componentes prontos, feitos por instituições, como neste caso, ou pela própria comunidade. Nesse projeto foi utilizado o modelo *Visual Studio Code Toolkit*, fornecido pela *Microsoft*. As estruturas utilizadas têm como base os componentes do *VSCode* e estas foram utilizadas na criação das interfaces. Porém, os componentes que não são nativos dos *VSCode* foram criados exclusivamente para o projeto apresentado neste documento pelos próprios autores.

A Figura 36 apresenta as opções de configuração da extensão *Swagger to SDK* na região chamada “Barra de Atividades” no canto lateral esquerdo. Esse conteúdo é dividido em duas partes, a primeira é referente as configurações da documentação *OpenAPI* e carregamento das funções que conectam aos *endpoints* e a sua tipagem. Já a segunda é referente as configurações de *url base*, o *adapter* que será utilizado pelas funções ao conectar com os *endpoints* e o caminho de saída customizado escolhido pelo usuário, caso este campo não seja preenchido, será utilizado o caminho padrão.

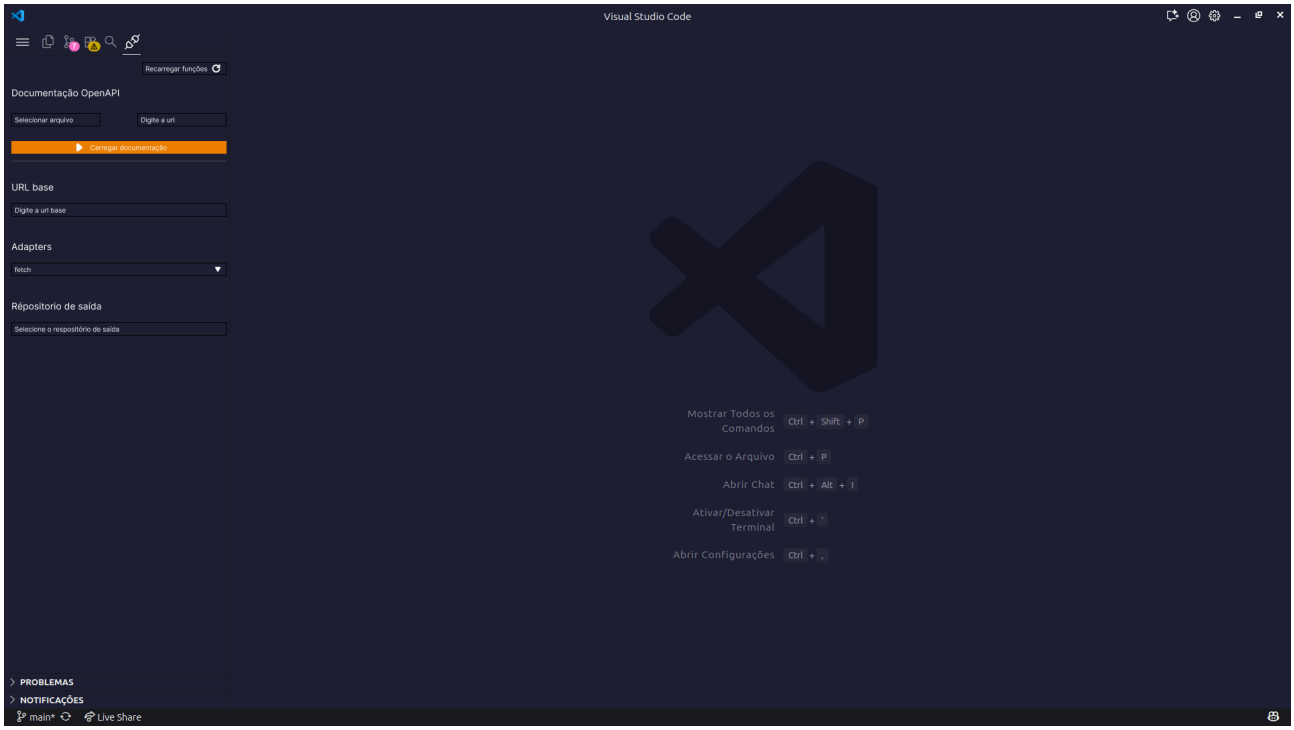

Figura 36. Tela com conteúdo da extensão na barra lateral.

5. Glossário e Modelos de Dados

Esta seção tem o objetivo de apresentar e detalhar o glossário e modelo de dados do sistema, fornecendo uma compreensão mais abrangente sobre os termos técnicos utilizados e a estrutura de organização dos dados dentro da plataforma. Assim, a Tabela 10 especifica os atributos de configuração do sistema, este campo serve como parâmetro para como o sistema deve criar os outros dados.

Atributo	**Formato**	**Descrição**
BaseUrl	*String*	É o valor inserido pelo usuário na tela da extensão que define a *URL* que será usada nas chamadas dos *endpoints*.
Adapter	*String*	É o valor inserido pelo usuário na tela da extensão que define qual *adapter* será utilizado para realizar as chamadas *https* dos *endpoints*.
OutputPath	*String*	É o valor inserido pelo usuário na tela da extensão que define o caminho do diretório de saída padrão para as funções e as suas tipagens criadas pelo sistema.

ImportPath	*String*	É o valor inserido pelo usuário na tela da extensão que define o caminho do arquivo ou *URL* de importação da documentação.
OpenAPI	*OpenAPI*	Instância do objeto *OpenAPI* contendo a especificação completa importada e processada.
version	*String*	Versão da especificação *OpenAPI* utilizada pela importação. Indica o padrão de estruturação da API.
paths	*Array<String>*	Lista de *endpoints* disponíveis na API.
schemas	*Array<Schema>*	Lista de *schemas* reutilizáveis extraídos da especificação *OpenAPI*.
name	*String*	Nome identificador único do *schema*, *parameter* ou *properties*.
type	*String*	Tipo primitivo do dado conforme especificação *OpenAPI*.
properties	*SchemaProperties*	Objeto mapeando nomes de propriedades para suas definições de tipo em um *schema*.
nullable	*Boolean*	Índica se o valor do campo pode ser nulo.
format	*String*	Formato específico do tipo primitivo, fornecendo semântica adicional.
required	*Array<String>*	Lista de nomes de propriedades obrigatórias em um *schema.*
enum	*Array<String>*	Lista de valores permitidos para o campo, criando um tipo enumerado.
operations	*Array<Operation>*	Lista de todas as *operations HTTP* extraídas da especificação *OpenAPI*.
operationId	*String*	Identificador único de uma *operation HTTP* na especificação *OpenAPI*. Utilizado como nome

		da função *TypeScript* gerada. Se ausente, o sistema gera um nome baseado no método *HTTP* e *path*.
path	*String*	Caminho relativo do *endpoint* na *API* conforme especificação *OpenAPI*. Pode conter parâmetros dinâmicos entre chaves.
method	*String*	Método *HTTP* da operação, os valores possíveis são: *get, post, put, patch, delete, head e options.*
parameters	*Array<Paremter>*	Lista de parâmetros aceitos pela operação *HTTP*. Cada objeto define um parâmetro com sua localização, tipo e obrigatoriedade.
in	*String*	Localização do parâmetro na requisição *HTTP*, os valores possiveis são: *path, query, header* e *cookie.*
requestBody	*Array<OperationRequestBody>*	Lista de definições do corpo da requisição *HTTP*. Especifica os tipos *MIME* aceitos e seus *schemas* correspondentes
content	*Map<string, OperationSchema>*	Mapeamento de tipos *MIME* para suas definições de *schema* e define os formatos de dados aceitos/retornados.
application/json	*Map<string, string>*	Definição específica para conteúdo no formato *JSON*, possui geralmente uma referência ($ref) para um *schema*
$ref	*String*	Referência a um *schema* definido em outra parte da especificação *OpenAPI*.
response	*Array<OperationResponse>*	Lista de respostas possíveis da operação *HTTP*, cada objeto define uma resposta para um código de *status* específico.
description	*String*	Descrição textual em linguagem natural do elemento, utilizada

		como comentário *JSDoc* no código gerado.
status	*String*	Estado da importação da *OpenAPI.*

Tabela 10. Glossário de dados da aplicação.

Neste projeto não é necessário o uso de banco de dados, isto que suas entidades são criadas, utilizadas e deletadas ainda em tempo de execução e não são persistidas. Mesmo assim, com o intuito de ilustração e documentação, o modelo de dados apresentado neste tópico ilustra um exemplo da estrutura de dados de forma simplificada utilizando *Javascript Object Notation* (JSON).

A Figura 37 ilustra o modelo de dados do sistema em formato *JSON*, os valores apresentados na Figura 37 são apresentados no glossário de dados da aplicação.

```
{
  "baseURL": "https://api.exemplo.com",
  "adapter": "axios",
  "outputPath": "./src/api",
  "importPath": "./openapi.json",
  "openAPI": {
    "version": "3.0.0",
    "paths": ["/api/v1/users", "/api/v1/users/{id}"],
    "schemas": [
      {
        "name": "User",
        "type": "object",
        "properties": {
          "id": {
            "type": "string",
            "nullable": false,
            "format": "string"
          },
          "name": {
            "type": "string",
            "nullable": false,
            "format": "string"
          }
        },
        "required": ["id", "name"],
        "enum": ["USER", "ADMIN"]
      }
    ],
    "operations": [
      {
        "operationId": "getUser",
        "path": "/api/v1/users",
        "method": "GET",
        "parameters": [
          {
            "name": "id",
            "in": "query",
            "required": false,
            "schema": [
              {
                "type": "string",
                "nullable": false,
                "format": "string"
              }
            ]
          }
        ],
        "requestBody": [
          {
            "content": {
              "application/json": {
                "$ref": "#/components/schemas/User"
              }
            }
          }
        ],
        "response": [
          {
            "description": "User created",
            "status": "201",
            "content": {
              "application/json": {
                "$ref": "#/components/schemas/User"
              }
            }
          }
        ]
      }
    ],
    "status": "COMPLETED"
  }
}
```

Figura 37. Exemplo de estado da aplicação em formato *JSON*.

6. Casos de Teste

Nesta seção, são apresentados os testes de aceitação e integração realizados conforme as necessidades explicitadas para o desenvolvimento do projeto. Para definição dos testes, são consideradas as necessidades criadas no documento de visão, as pré-condições realizadas pelas funcionalidades, as entradas como as ações e comportamentos executados para observar o caso de teste e a saída/estado como os resultados esperados de acordo com cada entrada

6.1 Teste de aceitação

Nesta seção, é apresentada a Tabela 11, a qual propõe o teste de aceitação para as necessidades apresentadas no documento de visão. Essa tabela foi criada a partir das funcionalidades de cada necessidade apresentada, podendo se testar cada cenário proposto em cada funcionalidade da aplicação.

Necessidade	**Pré-condição**	**Entrada**	**Saida/Estado**
Permitir dentro do *VSCode* importar, validar e gerar clientes via interface gráfica integrada ao editor.	Aba da extensão no *VSCode* deve estar aberta.	Inserir a *url* de uma documentação *OpenAPI.*	Exibir status de importação bem sucedida.
		Inserir o caminho para a documentação *OpenAPI.*	
		Clicar em carregar documentação para gerar as funções e tipos.	Deve criar arquivos de saída *api.ts* e *types.ts* com código das funções e tipagens.
Reduzir o esforço repetitivo de integração *REST*.	Deve ter importado a documentação *OpenAPI*	Clicar em renovar funções.	Deve renovar o código das funções conforme as mudanças na documentação.
		Clicar em renovar os tipos.	Deve renovar o código dos tipos conforme as mudanças na documentação.
Oferecer tipagem estática de parâmetros.	Deve ter importado a documentação *OpenAPI*	Ao clicar em renovar os tipos ou carregar documentação.	Deve gerar no arquivo *types.ts* a tipagem dos *headers, path, queries, cookies,*

			requestBody e tipos de respostas (200, 201, 400).
Permitir organização opinativa, porém configurável.	Deve ter importado a documentação *OpenAPI*	Ao mudar o *adapter* padrão no *select* de configurações.	Deve carregar a documentação ou renovar as funções para gerar o novo código.
		Ao mudar a *url* base no *input* de configurações.	Deve carregar a documentação ou renovar as funções e tipos para gerar o novo código.
		Ao mudar o caminho padrão de saída dos arquivos no *input* de configurações.	
Garantir compatibilidade de execução em navegadores.	Aba da extensão no *VSCode* deve estar aberta.	Ao carregar a extensão, ter o *adapter* padrão já selecionado *fetch*.	Deve carregar a documentação com o *adapter* padrão selecionado.
Expor *feedbacks* compreensíveis de erros.	Aba da extensão no *VSCode* deve estar aberta.	Ao clicar em carregar a documentação da *OpenAPI.*	Deve mostrar uma mensagem de erro ao tentar carregar.

Tabela 11. Teste de aceitação da aplicação.

7. Cronograma e Processo de Implementação

Nesta seção, é apresentado o cronograma e processo de implementação adotado para a construção do projeto. O cronograma de desenvolvimento deste trabalho está dividido em duas fases: documentação do projeto e desenvolvimento de código da plataforma. A fase de documentação do projeto compreende as entregas do projeto no período de 17 de agosto de 2025 até 14 de dezembro de 2025, enquanto a fase de desenvolvimento de código ocorre no período de 15 de fevereiro de 2026 até 17 de maio de 2026.

A Figura 38 apresenta o cronograma geral do projeto criado no GitHub por meio de *milestones*, os quais indicam os ciclos de entrega do cronograma e agrupam as *issues*. São apresentados 6 *milestones* relacionadas as entregas da documentação do projeto, a primeira *milestone* “TCC 1 - Entrega A1”, contempla a entrega do artefato Memorial Descritivo da Graduação. Na segunda *milestone*, “TCC 1 - Entrega A2”, foram entregues os artefatos Documento de Visão e Estudo de Viabilidade. Já na terceira, quarta e quinta *milestones*, “TCC 1 - Entrega A3”, “TCC 1 - Entrega A4” e “TCC 1 - Entrega A5”, respectivamente, foram entregues seções do artefato Documento do

Projeto. As *issues* associadas as *milestones* foram divididas para facilitar na identificação dos problemas específicos que precisam ser resolvidos para a entrega.

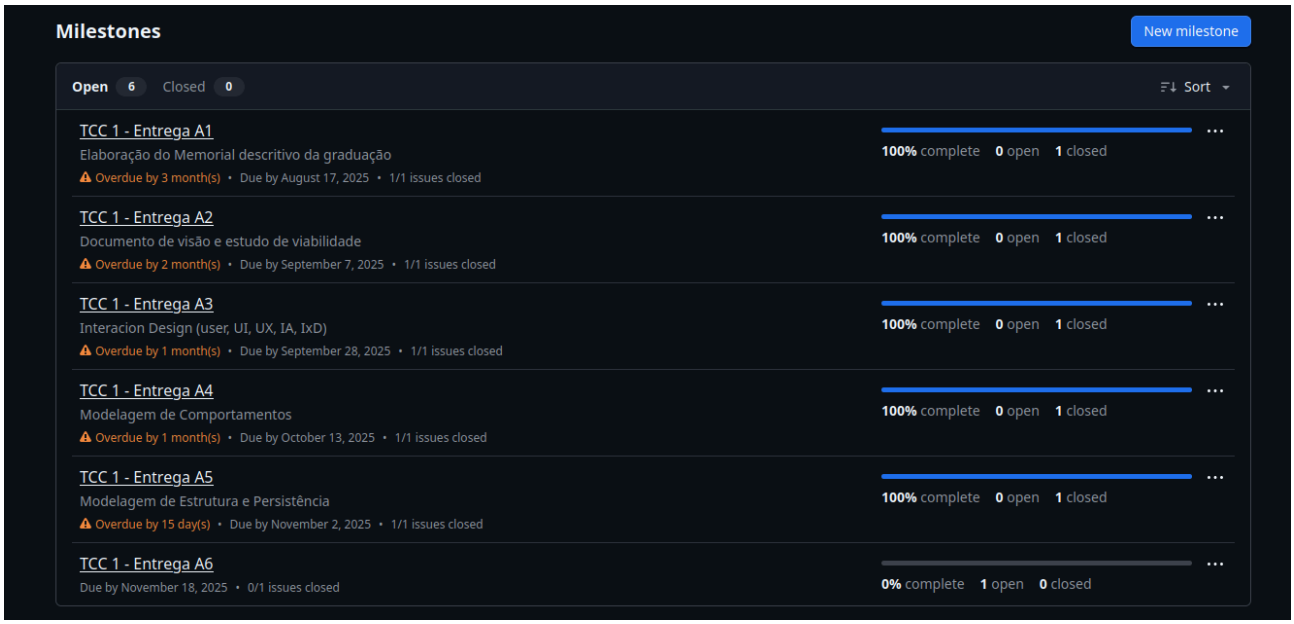

Figura 38. *Milestones* das entregas de documentação do projeto.

Na Figura 39, apresenta-se o conjunto de 8 *milestones*, correspondente a cada *sprint* na fase de desenvolvimento da plataforma. Para o processo de desenvolvimento, foram criadas *issues* que correspondem a cada entrega a ser feita de código necessário para a *sprint* correspondente. Já na Tabela 12 foram alocadas as atividades em cada ciclo, priorizadas por ordem de criticidade. As entregas da *sprint 7* e *sprint 8* aparecem vazias na tabela, pois essas etapas são focadas totalmente em testes, documentação e criação da apresentação para a banca avaliadora.

Figura 39. *Milestones* das entregas de desenvolvimento do código.

Tarefa por entrega	Fevereiro		Março		Abril		Maio	
	Sprint 1	Sprint 2	Sprint 3	Sprint 4	Sprint 5	Sprint 6	Sprint 7	Sprint 8
Setup	X							
UC01	X							
UC02		X						
UC03			X					
UC04			X					
UC05				X				
UC06				X				
UC07				X				
UC08						X		
UC09					X			
UC10					X			
UC11					X			

Tabela 12. Cronograma de entrega das *sprints.*

A primeira *sprint* é dedicada a criação do setup do projeto e entrega do caso de uso UC01. Na segunda *sprint* será desenvolvido o caso de uso UC02. Na terceira *sprint* serão desenvolvidos os casos de uso UC03 e UC04. Durante a quarta *sprint* serão desenvolvidos os casos de uso UC05, UC06 e UC07. A quinta *sprint* tera a entrega dos casos de uso UC09, UC10 e UC11. Já na sexta *sprint* será desenvolvido o caso de uso UC08. A sétima e oitava *sprint* serão focadas para criação da *suite* de testes de aceitação e também na apresentação dos resultados para a banca avaliadora. Vale ressaltar que as entregas da *sprint* 2, 3 e 4 são destinadas às entregas críticas do sistema, pois se tratam das funcionalidades essências do sistema.

Além disso, para manter a organização foi utilizado a estratégia de gerenciamento de trabalho chamada *Kanban*, recurso oferecido pelo *Github Projects,* como mostra a Figura 40. Esse *Kanban* possui as colunas *Backlog* aonde são armazenadas as todas as tarefas que não entraram na *sprint*, a coluna *Ready*, responsável pelas tarefas a serem entregues naquela *sprint,* a coluna *In Progress* responsável pelas tarefas sendo desenvolvidas na *sprint,* e por último a coluna *Done* que possui as tarefas concluídas.

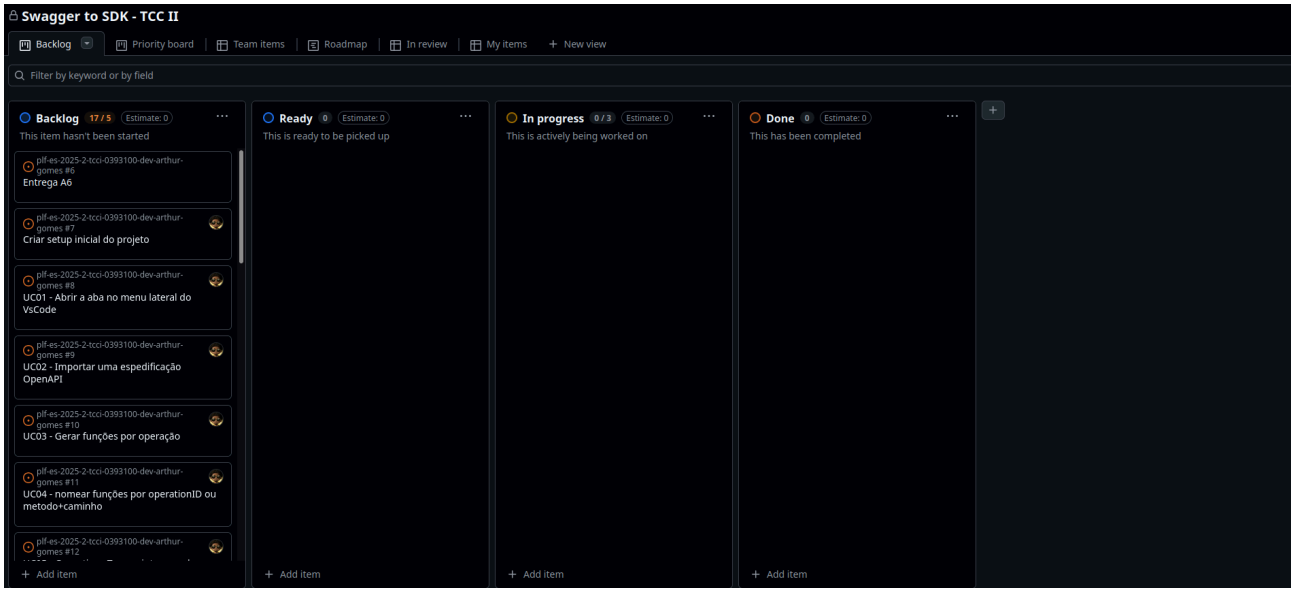

Figura 40. *Kanban de divisão de tarefas de implementação do projeto*

No processo de desenvolvimento, adotou-se o Git para versionamento do código desenvolvido e para armazenamento do código versionado, o GitHub funciona como repositório para salvar o desenvolvimento da extensão.