UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

CENTRO TECNOLÓGICO

COLEGIADO DO CURSO DE CIÊNCIA DA COMPUTAÇÃO

Alex Santos de Oliveira

Desenvolvimento de uma extensão para o Visual Studio Code com suporte ao método FrameWeb

Vitória, ES

2026

Alex Santos de Oliveira

Desenvolvimento de uma extensão para o Visual Studio Code com suporte ao método FrameWeb

Monografia apresentada ao Curso de Ciência da Computação do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do Grau de Bacharel em Ciência da Computação.

Universidade Federal do Espírito Santo – UFES

Centro Tecnológico

Colegiado do Curso de Ciência da Computação

Orientador: Prof. Vítor E. Silva Souza

Vitória, ES

2026

Alex Santos de Oliveira
Desenvolvimento de uma extensão para o Visual Studio Code com suporte ao método FrameWeb/ Alex Santos de Oliveira. – Vitória, ES, 2026-
53 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Vítor E. Silva Souza

Monografia (G) – Universidade Federal do Espírito Santo – UFES
Centro Tecnológico
Colegiado do Curso de Ciência da Computação, 2026.

1. Engenharia de Software. 2. MDD. I. Souza, Vítor Estêvão Silva. II. Universidade Federal do Espírito Santo. IV. Desenvolvimento de uma extensão para o Visual Studio Code com suporte ao método FrameWeb

CDU 02:141:005.7

Alex Santos de Oliveira

Desenvolvimento de uma extensão para o Visual Studio Code com suporte ao método FrameWeb

Monografia apresentada ao Curso de Ciência da Computação do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do Grau de Bacharel em Ciência da Computação.

Trabalho aprovado. Vitória, ES, 02 de Março de 2026:

Prof. Vítor E. Silva Souza
Orientador

Prof. João Paulo Andrade Almeida
Universidade Federal do Espírito Santo

Matheus Lenke Coutinho
Universidade Federal do Espírito Santo

Vitória, ES
2026

Agradecimentos

Agradeço aos meus pais por me apoiarem em todas as minhas escolhas, pela educação que me deram e por terem criado as condições necessárias para que eu pudesse chegar até aqui. Por estarem presentes em todas as fases da minha vida, nos momentos de conquista e nos momentos difíceis, sempre com apoio incondicional. Tudo que sou tem muito de .

Agradeço a todos os amigos e familiares que me incentivaram ao longo da graduação, seja com palavras, conversas ou desabafos. Em especial ao meu amigo Gustavo, que acompanhou toda a minha trajetória acadêmica e teve grande influência em me motivar a cursá-la. Fico grato por nossos caminhos terem se cruzado.

Agradeço especialmente ao meu orientador Vítor Souza, que acreditou neste trabalho mesmo nos momentos mais difíceis. Sua paciência, generosidade e comprometimento foram fundamentais para que essa conclusão fosse possível.

Por fim, agradeço a mim mesmo por ter me mantido firme e resiliente diante das dificuldades e imprevistos. Não foi fácil, mas foi possível.

"O sucesso é tropeçar de fracasso em fracasso sem perder o entusiasmo."
(Winston Churchill)

Resumo

Voltado para a fase de projeto de software, o FrameWeb é um método que estende a UML para modelar arquiteturas de Sistemas de Informação Web (WISs) baseadas em *frameworks*. O FrameWeb define quatro modelos como extensões leves da UML: Modelo de Entidades, Modelo de Aplicação, Modelo de Navegação e Modelo de Persistência. O suporte computacional para este método concentrou-se em ferramentas na plataforma Eclipse, que atualmente perdeu relevância na indústria em favor de editores mais leves e modernos. Diante desse cenário, este trabalho propõe uma abordagem de modelagem textual integrada ao Visual Studio Code, unindo a expressividade do FrameWeb à flexibilidade das práticas de desenvolvimento contemporâneas.

O objetivo desta monografia é o desenvolvimento de uma Linguagem Específica de Domínio (DSL) textual e de uma extensão para o editor Visual Studio Code que ofereça suporte ao método FrameWeb, utilizando o *framework* Langium para a definição da gramática e a criação da extensão. Essa abordagem permite que o projetista se concentre na lógica e na semântica do sistema enquanto a ferramenta realiza validações semânticas do método, como a verificação de conformidade arquitetural de estereótipos, além da geração automática de diagramas.

Como resultado, apresenta-se a ferramenta FrameWeb Writer Tool (FWT), que permite a definição textual de modelos de Entidade, Aplicação, Persistência e Navegação. A FWT oferece recursos avançados de ambientes de desenvolvimento, como realce de sintaxe, validações semânticas automáticas — que garantem a conformidade com as regras de estereótipos do FrameWeb — e a geração dinâmica de diagramas visuais por meio da biblioteca Mermaid. Essa integração possibilita o uso de práticas como o controle de versão simplificado (via Git).

Conclui-se que o desenvolvimento da FWT expande a aplicabilidade do método FrameWeb, alinhando-o a um ecossistema tecnológico amplamente adotado. A solução preserva a síntese visual necessária para a interpretação arquitetural por meio dos diagramas gerados automaticamente. Perspectivas para trabalhos futuros incluem a implementação da geração automática de código-fonte a partir dos modelos textuais, visando consolidar um fluxo completo de engenharia de software dirigida por modelos.

Palavras-chaves: Engenharia de Software. MDD.

Lista de ilustrações

Sumário

1 Introdução

Nos primeiros dias da *World Wide Web* (cerca de 1990 a 1995), os sites consistiam em pouco mais do que um conjunto de arquivos de hipertexto vinculados que apresentavam informações usando texto e gráficos limitados. Com o passar do tempo, a ampliação do HTML por ferramentas de desenvolvimento (por exemplo, XML, Java) permitiu que engenheiros da Web oferecessem capacidade de computação juntamente com conteúdo informativo. Sistemas e aplicativos baseados na Web (WebApps) nasceram. Hoje, os WebApps evoluíram para ferramentas de computação sofisticadas que não apenas fornecem funções independentes ao usuário final, mas também foram integradas com bancos de dados corporativos e aplicativos empresariais (Pressman; Lowe, 2008).

Hoje, o software desempenha um papel duplo. Ele é um produto e, ao mesmo tempo, o veículo para entregar um produto. Como produto, ele fornece o potencial de computação incorporado pelo hardware de computador ou, de forma mais ampla, por uma rede de computadores que são acessíveis pelo hardware local. Quer ele resida em um telefone celular ou opere dentro de um computador mainframe, o software é um transformador de informações—produzindo, gerenciando, adquirindo, modificando, exibindo ou transmitindo informações que podem ser tão simples quanto um único bit ou tão complexas quanto uma apresentação multimídia derivada de dados adquiridos de dezenas de fontes independentes. Como veículo usado para entregar o produto, o software serve como base para o controle do computador (sistemas operacionais), a comunicação de informações (redes) e a criação e controle de outros programas (ferramentas e ambientes de software) (Pressman; Lowe, 2008).

Essas simples realidades levam a uma conclusão: o software, em todas as suas formas e em todos os seus domínios de aplicação, deve ser desenvolvido aplicando conceitos e técnicas de Engenharia de Software específicas para esse ambiente, visando garantir a qualidade dos sistemas desenvolvidos (Pressman; Lowe, 2008). Nesse contexto, o FrameWeb oferece um método sistemático baseado em modelos conceituais bem fundamentados, junto com ferramentas que automatizam certas partes do processo, facilitando a tarefa de integrar um Sistema de Informação na Web (*Web Informational System* - WIS) à Web de Dados e, assim, promovendo a adoção de dados vinculados (Souza, 2019). Este trabalho busca ampliar o método FrameWeb incorporando-o a outra ferramenta de desenvolvimento de software que será detalhada mais adiante.

1.1 Motivação e Justificativa

No início da popularização dos WISs, o desenvolvimento de sistemas baseados na Web foi feito de forma *ad hoc*, sem uma abordagem sistemática e sem procedimentos de controle e garantia de qualidade. Por isso, há uma preocupação legítima e crescente com a maneira como esses sistemas são desenvolvidos e com sua qualidade e integridade a longo prazo. Para atender tais necessidades de qualidade é de grande importância a aplicação de técnicas de engenharia (Pressman; Lowe, 2008). A Engenharia Web trata do estabelecimento e uso de princípios científicos, de engenharia e de gestão sólidos, bem como de abordagens disciplinadas e sistemáticas para o desenvolvimento, implantação e manutenção bem-sucedidos de sistemas e aplicações baseados na Web de alta qualidade (Murugesan *et al.*, 2001).

Como em qualquer processo de Engenharia de Software, o uso de ferramentas contribui para o desenvolvimento, manutenção e a qualidade do projeto (Galin, 2018). Em atividades de modelagem, ressalta-se a importância do vínculo entre os modelos e o código-fonte que representam (Pressman; Lowe, 2008). Assim, foram desenvolvidas ferramentas para apoio à construção de modelos FrameWeb e geração de código baseadas na plataforma Eclipse (Campos; Souza, 2017; Almeida; Campos; Souza, 2017). O desenvolvimento dessas ferramentas, no entanto, foi descontinuado.

Paralelamente, observa-se que o ecossistema da plataforma Eclipse tem perdido protagonismo nos últimos anos[1], especialmente quando comparado a ambientes amplamente adotados pela indústria, como o Visual Studio Code. Esse cenário impacta a manutenção e evolução de ferramentas acadêmicas desenvolvidas nesse ambiente, motivando a adoção de plataformas mais atuais, com maior comunidade ativa, atualizações frequentes e melhor integração com o fluxo de desenvolvimento contemporâneo.

Paralelamente à escolha da plataforma tecnológica, considera-se a adoção de uma linguagem textual para apoio à modelagem. A literatura em Engenharia de Software (Grönninger *et al.*, 2014; Engelen; Brand, 2010) aponta que artefatos textuais oferecem benefícios como melhor integração com ambientes de desenvolvimento, compatibilidade com sistemas de versionamento distribuído e maior potencial de automação e validação durante o processo de modelagem. Entre as principais vantagens, destacam-se:

- **Foco na lógica e formatação automática:** o posicionamento manual de elementos em modelos gráficos pode se tornar uma atividade demorada, desviando a atenção do desenvolvedor da semântica e da lógica do modelo. Em contrapartida, linguagens textuais permitem que a formatação seja tratada automaticamente por ferramentas e algoritmos padronizados, produzindo resultados consistentes e de alta qualidade.

[1] <https://www.secondtalent.com/resources/ide-statistics/>

Dessa forma, o esforço concentra-se na estrutura e no significado do modelo, e não em sua disposição visual (Grönninger *et al.*, 2014).

- **Controle de versão superior:** por utilizarem arquivos de texto puro, linguagens textuais são naturalmente compatíveis com sistemas de controle de versão como Git, Subversion e CVS. Diferentemente de modelos gráficos frequentemente armazenados em formatos XML/XMI complexos, a estrutura textual possibilita comparação eficiente entre versões (*diff*), rastreamento claro de alterações e resolução mais simples de conflitos. A mesclagem baseada em linhas permite que múltiplos desenvolvedores trabalhem simultaneamente sobre o mesmo modelo de forma mais eficaz (Grönninger *et al.*, 2014);
- **Verificações de contexto e validação antecipada:** ferramentas baseadas em linguagens textuais possibilitam a realização de verificações sintáticas e semânticas ainda durante a modelagem, antes da geração de artefatos finais. Esse suporte favorece a identificação precoce de inconsistências e erros lógicos, tornando-os mais transparentes ao usuário e contribuindo para maior qualidade do modelo. Além disso, a integração entre representações textuais e gráficas pode potencializar esses mecanismos de validação (Engelen; Brand, 2010).

Dessa forma, este trabalho tem como motivação expandir o uso do método FrameWeb (Souza, 2007) no desenvolvimento de sistemas, viabilizando seu suporte em uma plataforma amplamente adotada pela indústria, o Visual Studio Code, e explorando os benefícios associados à modelagem textual. Busca-se, assim, ampliar a aplicabilidade prática do método, alinhando-o às ferramentas e práticas mais recentes de desenvolvimento de software.

1.2 Objetivos

O objetivo deste trabalho é desenvolver uma linguagem de modelagem textual para o FrameWeb e uma extensão que dê suporte a esta linguagem no VS Code. Esta extensão deve capturar todos os conceitos estabelecidos pelo FrameWeb e ser de fácil utilização para dar suporte ao desenvolvimento de sistemas. Dessa maneira os objetivos específicos do trabalho são descritos como:

- Desenvolver uma linguagem textual que dê suporte ao FrameWeb.
- Desenvolver uma extensão para o editor de texto Visual Studio Code que utilize da linguagem desenvolvida para gerar os modelos de classes no método FrameWeb.

1.3 Método de Desenvolvimento do Trabalho

Por se tratar do desenvolvimento de uma ferramenta de suporte ao método FrameWeb, a abordagem metodológica consistiu primeiramente na leitura de artigos sobre trabalhos correlatos, principalmente dos trabalhos relacionados ao FrameWeb, bem como no estudo sobre a criação de Linguagens Específicas de Domínio (*Domain Specific Languages* – DSL) por meio de métodos de Desenvolvimento Dirigido a Modelos (*Model Driven Development* – MDD).

Após a análise do referencial teórico, as etapas seguintes seguiram características similares ao desenvolvimento de projetos de software: definição das funcionalidades a serem implementadas (levantamento de requisitos), escolha das tecnologias a serem utilizadas e desenvolvimento gradual de cada uma dessas funcionalidades em um repositório de código.

1.4 Organização da Monografia

Além desta introdução, esta monografia é composta por outros três capítulos:

- O Capítulo 2 apresenta os aspectos relativos ao conteúdo teórico relevante para o trabalho;
- O Capítulo 3 apresenta a principal contribuição do trabalho;
- O Capítulo 4 apresenta as considerações finais do trabalho;

2 Fundamentação Teórica e Tecnologias Utilizadas

Neste capítulo é apresentada a revisão bibliográfica feita. A Seção 2.1 apresenta o método FrameWeb, alvo deste trabalho. A seção 2.2 apresenta uma introdução ao Desenvolvimento Dirigido a Modelos (MDD). Por fim, a Seção 2.3 apresenta as tecnologias escolhidas para o desenvolvimento deste trabalho.

2.1 FrameWeb

FrameWeb é um método de projeto para construção de Sistemas de Informação Web (*Web Information Systems* – WISs) baseado em *frameworks*. Com a proposta inicial de auxiliar na fase de projeto do Software, o FrameWeb teve inspiração em alguns *frameworks* comuns para o desenvolvimento em Java. Para o objetivo de integrar todos os *frameworks* em uma ferramenta de apoio ao projeto, foi criada uma linguagem de modelagem própria baseada em UML para que as boas práticas de arquitetura de projetos sejam seguidas (Souza, 2007). Nas seções seguintes, os detalhes do método FrameWeb serão discutidos.

2.1.1 Arquitetura

O FrameWeb é baseado no padrão Camada de Serviço (*Service Layer Pattern*) proposta por Randy Stafford em (Fowler, 2002). Essa arquitetura separa o sistema em três camadas com funções distintas para a modelagem do sistema: uma **Camada de Apresentação**, uma **Camada de Serviço** e uma **Camada de Acesso a Dados** como se pode ver na Figura 1.

Cada camada tem seus pacotes que se comunicam para que o WIS seja implementado com cada módulo em seu lugar específico desempenhando uma única função e com fraco acoplamento entre as classes.

A Camada de Apresentação é responsável pela interface com o usuário, exibindo páginas Web, formulários, componentes visuais, etc. No pacote de Visão, todos esses recursos são encontrados, o pacote também é responsável por enviar estímulos do usuário ao pacote de Controle, este por sua vez faz uso dos serviços necessários, implementados na Camada de Lógica de Negócio, para que a ação do usuário resulte em uma alteração do estado do sistema ou recuperação de informações.

A Camada de Lógica de Negócio implementa as restrições de negócio levantadas

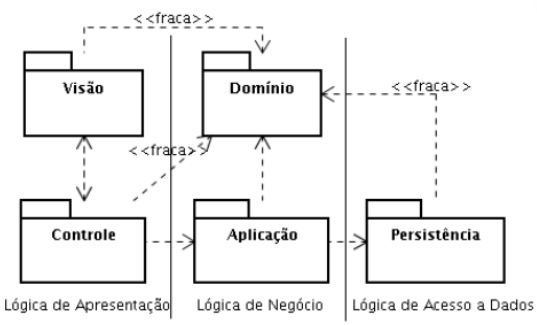

Figura 1 – Arquitetura proposta pelo FrameWeb inspirada pelo padrão *Service Layer* (Fowler, 2002)

na fase de requisitos, possui dois pacotes: Aplicação e Domínio. O pacote de Aplicação é responsável por implementar os casos de uso levantados para o sistema e, para isso, depende da Camada de Acesso a Dados para salvar ou recuperar os dados, e também do pacote de Domínio. Neste último, estão as classes relacionadas ao domínio do problema como Aluno, Professor, Departamento, etc. Essas classes são mapeadas para tabelas no banco de dados, sendo responsabilidade da Camada de Acesso a Dados realizar essas manipulações. Como mostra a Figura 1 o pacote de Aplicação depende tanto do pacote de Domínio quanto do pacote de Persistência para implementar a lógica de negócio.

A Camada de Acesso a Dados é a camada que faz a manipulação do banco de dados, e o mapeamento entre Objetos e as suas respectivas representações no banco de dados. É possível ver na figura que a Camada de Lógica de Negócios depende da Camada de Acesso a Dados, também chamada de Camada de Persistência para que os dados sejam salvos e acessados de maneira segura, sem perdas ou inconsistências.

2.1.2 Linguagem de Modelagem

O FrameWeb usa extensões leves da UML para criação de modelos de quatro tipos seguindo o padrão de arquitetura apresentado: **Modelo de Entidade**, **Modelo de Persistência**, **Modelo de Navegação** e **Modelo de Aplicação** (Souza, 2019).

O Modelo de Entidades contém as classes relacionadas diretamente ao escopo do problema. Por exemplo: Aluno, Turma, Professor. Todas as classes que modelam entidades ficam dentro do pacote Domínio na camada de lógica de negócio. Apesar da função de persistência ser da camada de acesso a dados é na camada de domínio que os mapeamentos para as tabelas do banco de dados são feitos.

O Modelo de Persistência contém as classes que fazem acesso a dados e a persistência no banco de dados. As funções de acesso a dados são todas manipuladas nessa camada, seguindo as restrições de mapeamento indicadas na camada de domínio. Nessa camada está implementado o padrão Data Access Object (DAO) (Alur; Malks; Crupi, 2001) ou seja, são as classes DAO da camada de Persistência que fazem a manipulação dos dados,

permitindo mais uma camada de abstração e separação das responsabilidades no WIS.

O Modelo de Navegação representa páginas, formulários e classes controladoras que formam a Camada de Apresentação. Por meio desses diagramas também é possível ver o fluxo de interação do usuário com páginas e as controladoras. Por exemplo ao executar login/logout, o Modelo de Navegação consegue exibir quais são as páginas os dados requeridos e as classes controladoras responsáveis por desempenhar esses papéis, além das páginas que são exibidas ao usuário nesse fluxo.

O Modelo de Aplicação é onde estão as classes de serviço, que se comunicam tanto com as classes do Modelo de Navegação, quanto com o Modelo de Persistência. Essas classes são responsáveis pela lógica de negócio (funcionalidades) do sistema, acessando a camada de dados e devolvendo as respostas para as classes controladoras na camada de Apresentação.

2.1.3 Síntese da evolução do FrameWeb

O FrameWeb foi inicialmente desenvolvido para dar suporte a 3 *frameworks* de desenvolvimento Java. Esses *frameworks* posteriormente deram suporte à definição de categorias de *frameworks* suportadas pelo FrameWeb (Souza, 2019). As categorias suportadas foram definidas como:

1. Controlador Frontal (*Front Controller*), como Struts: *frameworks* desse tipo implementam uma versão ligeiramente modificada do padrão Model View Controller (MVC), adaptado para a Web;

2. Injeção de Dependências (*Dependency Injection*) *frameworks* como Spring: *frameworks* desse tipo são usados quando classes dependem de objetos de outras classes e utlizam interfaces para comunicação ao invés de suas implementações. Quando uma classe é criada, todas as dependências são injetas e satisfeitas automaticamente;

3. Mapeamento Objeto/Relacional (*Object/Relational Mapping*) *frameworks* como Hibernate: *frameworks* desse tipo fazem um mapeamento automático e transparente de classes para para tabelas relacionais de um Sistema Gerenciador de Bancos de Dados (SGBDs) usando meta dados que descrevem esse mapeamento entre as classes e os SGBDs.

O FrameWeb incorpora os conceitos desses *frameworks* com design de modelos. E dessa maneira alcança duas contribuições principais descritas nas seções 2.1.1 e 2.1.2:

1. A definição de uma arquitetura básica para melhor integração entre as categorias de *frameworks* descritas acima;

2. Uma extensão leve da UML para a construção de 4 diferentes modelos de design que capturam os conceitos usados pelos *frameworks* para os modelos.

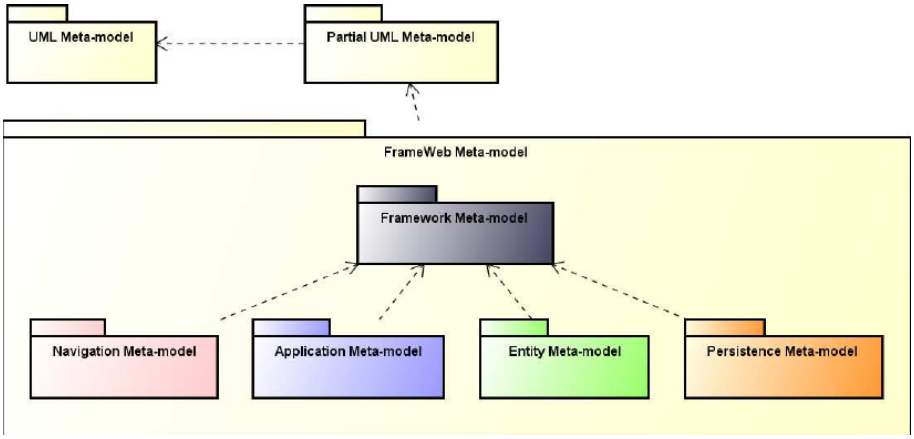

Figura 2 – Meta modelo que define a linguagem do FrameWeb (Souza, 2016)

Posteriormente, um metamodelo denominado FW-15 (Souza, 2016) foi criado usando técnicas de Desenvolvimento Orientado a Modelos (MDD – *Model-Driven Development*) que será discuto nesse texto. Isso permitiu a criação de ferramentas como o FrameWeb Editor (Almeida; Campos; Souza, 2017) e o FrameWeb Code Generator (Campos; Souza, 2017). Uma visão geral do metamodelo FW-15 é exibida na Figura 2.

Por meio do metamodelo FW-15 foi possível formalizar a linguagem do método FrameWeb e assim garantir a semântica da linguagem e possibilitar melhorias futuras (Souza, 2016). Para que seja uma linguagem de conhecimento geral dos desenvolvedores o Meta Modelo foi inspirado em extensões leves da UML. Por isso a linguagem depende do *Partial Meta Model* UML que contém as partes da UML que são usados pelo FrameWeb. Como demonstrado na figura, o *FrameWeb meta-model* é então dividido em 5 componentes, um para cada componente do método FrameWeb e um para o componente *Framework Meta-model* que habilita a definição de regras que são específicas aos *frameworks* escolhidos para o desenvolvimento do WIS. Dessa forma é possível modelar os componentes básicos para a arquitetura.

Com a linguagem definida, foi possível criar ferramentas baseadas nessa linguagem. O FrameWeb Editor foi criado e habilita a criação de diagramas baseados na arquitetura do método FrameWeb (Campos; Souza, 2017).

Outra melhoria implementada é o FrameWeb Code Generator (Almeida; Campos; Souza, 2017), essa ferramenta captura as restrições e definições geradas no FrameWebEditor e para cada elemento que representa um artefato de código, um *template* é usado para para aquele artefato específico em um *framework* de escolha do usuário, e preenche o *template* com os dados do modelo como classes controladoras, atributos, métodos, etc.

Além disso, uma funcionalidade que foi adicionada ao FrameWeb devido ao grande uso em sistemas Web são as funções de autenticação e autorização baseado em papéis dentro do sistema. Para dar suporte à essa funcionalidade o meta modelo foi ajustado para estender a sintaxe da linguagem, as modificações também foram feitas no FrameWeb Code

Generator e no FrameWeb Editor. Agora é possível definir autenticação e autorização no modelo arquitetural usando uma linguagem genérica e gerando código para o *framework* de sua escolha (Prado; Souza, 2018).

Em um trabalho posterior, houve a migração do suporte ao método FrameWeb para a plataforma *Visual Paradigm* com o desenvolvimento de um *plug-in* (Silva, 2023), substituindo o editor anteriormente na plataforma Eclipse, o qual apresentava limitações de confiabilidade, como o desaparecimento de associações e dificuldades na organização dos diagramas. A adoção de uma ferramenta de apoio ao desenvolvimento de software especializada em UML teve como objetivo oferecer um ambiente de modelagem mais estável e independente da IDE Eclipse, além de possibilitar a criação de modelos mais completos e eliminar a exigência de um projeto Java Web prévio para iniciar a modelagem. Essa iniciativa representou um avanço importante para a qualidade e padronização da arquitetura de Sistemas de Informação Web (SIWs) desenvolvidos com base em frameworks (Silva, 2023).

Entre as principais contribuições desse trabalho destacam-se o suporte aos Modelos de Entidade, Aplicação e Persistência, bem como a integração de extensões UML — como estereótipos e restrições — diretamente à interface do Visual Paradigm por meio de menus de contexto. Ademais, houve a introdução de um gerador de código baseado no motor de *templates Apache FreeMarker*, responsável por transformar modelos UML em esqueletos de código compatíveis com diferentes frameworks, como *JButler* e *Spring Boot*. A possibilidade de criação de *templates* personalizados amplia a flexibilidade da solução e contribui para o aumento da produtividade, ao aproximar a documentação arquitetural da implementação e garantir que o código inicial esteja alinhado aos padrões definidos pelo método FrameWeb (Silva, 2023).

2.2 Desenvolvimento Orientado a Modelos (MDD)

MDD é uma abordagem de Engenharia de Software que consiste na aplicação de modelos e tecnologias de modelagem para aumentar o nível de abstração com que os desenvolvedores criam software, com ambos os objetivos de simplificar e formalizar (assim é possível alguma automatização) as várias atividades que compreendem o ciclo de vida de um software. MDD impõe uma estrutura e um vocabulário comum, então esses artefatos são úteis para seus propósitos em um particular estágio do ciclo de vida do software, para a necessidade subjacente de ligar com artefatos relacionados e servir como um meio de comunicação entre os participantes do projeto (Hailpern; Tarr, 2006).

A literatura apresenta diversas vantagens no uso do MDD do ponto de vista da Engenharia Web (Kleppe; Warmer; Bast, 2003; Deursen; Klint, 1998; Mernik; Heering; Sloane, 2005; Trask; Roman, 2006):

- **Produtividade:** porque ocorre um aproveitamento melhor do tempo, já que o tempo é gasto na produção dos modelos de alto nível, deixando as tarefas repetidas serem implementadas por meio de transformações;
- **Portabilidade:** os modelos em níveis mais altos de abstração podem gerar código para diferentes tecnologias alvo, reduzindo tempo de desenvolvimento;
- **Interoperabilidade:** um subconjunto de um modelo pode sofrer transformações e assim ser usado em diferentes plataformas. Isso torna o ambiente heterogêneo, porém mantém a estrutura e semântica original intactas. Assim, a funcionalidade global do modelo é mantida;
- **Manutenção:** as alterações podem ser feitas diretamente no modelo uma vez que também fazem parte do software. Isso facilita tarefas de manutenção já que pode-se trabalhar diretamente com níveis mais altos de abstração;
- **Comunicação:** ao nível de abstração dos modelos facilita o entendimento do software mesmo quando as partes interessadas têm visões diferentes. Assim, a comunicação e a tomada de decisão são facilitadas;
- **Otimização:** assim como compiladores, as ferramentas de geração de código a partir de modelos podem aplicar otimizações no código gerado e diminuir a incidência de erros;
- **Corretude:** erros conceituais são facilmente identificados pois a identificação ocorre nos níveis mais altos de abstração do modelo e além disso os geradores não produzem erros acidentais.

Apesar de ter boas intenções, os autores apontam uma série de problemas com o MDD (Hailpern; Tarr, 2006; Ambler, 2003; Thomas, 2004):

- **Redundância**: há múltiplos representações de artefatos inerentes ao processo de desenvolvimento de software, representando diferentes visualizações de diferentes níveis de abstração sobre o mesmo conceito;
- **Rond-Trip Problem (Problema de ida e volta):** o round-trip problem acontece quando as relações entre os modelos são importantes, em especial quando há transformações ocorrendo de níveis mais altos de abstração para níveis mais baixos. A pior forma em que o problema de round-trip geralmente ocorre é quando ocorrem mudanças em artefatos em um nível mais baixo de abstração, como código, pois inferir uma semântica para os níveis mais altos é muito mais difícil do que gerar níveis mais baixos de abstração a partir de níveis mais altos. O principal problema é garantir a consistência mútua entre os modelos de níveis diferentes de abstração;

- **Aprendizado ou complexidade:** as relações entre múltiplos tipos de modelos, e potencialmente, diferentes formalismos de modelos, sugerem que será muito difícil para qualquer pessoa envolvida no projeto entender os impactos que uma proposta de mudança irá causar em todos os artefatos relacionados. Ou seja, é necessário mais conhecimentos dos desenvolvedores para lidarem com vários modelos em formalismos diferentes ou em níveis de abstração de diferentes, mas que tratam do mesmo domínio. Além disso, precisam ter experiência nas ferramentas utilizadas, o que requer treinamento dedicado e maior tempo de aprendizado;

- **Rigidez:** como a maior parte da geração de código se dá por ferramentas de geração de código e não sofrem atuação por parte dos desenvolvedores, os softwares produzidos são mais rígidos;

- **Desempenho:** apesar de haver técnicas de otimização nos geradores de código eles podem gerar códigos além do necessário e não muito compactos, o que pode ocasionar em perdas de desempenho se comparado à codificação manual.

O MDD pode ser usado na prática para o desenvolvimento de Linguagens Específicas de Domínio (*Domain Specific Languages* – DSL). Uma DSL provê ganhos substanciais em expressividade e facilidade de uso comparada com linguagens de propósito geral. Um modelo é escrito em uma linguagem bem definida. Uma transformação descreve como um modelo em uma linguagem de origem pode ser transformado em um modelo em uma linguagem alvo e assim formalizar uma linguagem (Mernik; Heering; Sloane, 2005; Embley; Liddle; Pastor, 2011). No entanto, para que a linguagem seja definida alguns requisitos devem ser seguidos: ter uma sintaxe abstrata e uma sintaxe concreta e uma semântica (Kleppe; Warmer; Bast, 2003).

O método FrameWeb propõe uma DSL gráfica que estende a UML com estereótipos específicos para modelagem de aplicações web baseadas em frameworks, conforme apresentado na Seção 2.1.3. Essa abordagem de modelagem foi desenvolvida utilizando métodos de MDD.

2.3 Tecnologias escolhidas

Esta seção descreve as tecnologias utilizadas para a implementação na prática das propostas deste trabalho.

2.3.1 Langium

Langium[1] é uma ferramenta de código aberto para definição de linguagens, com suporte de primeira classe ao Language Server Protocol (LSP)[2], escrita em TypeScript e executada no Node.js. O LSP é um protocolo desenvolvido pela Microsoft que permite a IDEs e editores de código (como Visual Studio Code, Vim, Emacs, entre outros) obterem funcionalidades de linguagem — como autocompletar, navegação para definições, validação de sintaxe e refatoração — a partir de um servidor de linguagem, agilizando o desenvolvimento de suporte a novas linguagens sem necessidade de reimplementação para cada editor.

Com o Langium é possível fazer um analisador sintático de linguagem. Linguagens de programação, assim como DSLs, não podem ser sintaticamente analisadas usando simplesmente expressões regulares. Ao invés disso é necessário uma estratégia mais sofisticada de analisador sintático. Para fazer isso o Langium provê uma representação em alto nível da linguagem livre de contexto sendo desenvolvida de forma similar ao EBNF.

Baseado na gramática, o Langium é capaz de construir um analisador sintático que transforma a string de entrada em uma representação de um modelo semântico. Esse modelo captura a estrutura essencial da linguagem em desenvolvimento. O modelo semântico é gerado como interfaces em TypeScript. Quando um programa na linguagem desenvolvida é analisado, uma Árvore de Sintaxe Abstrata (AST) será automaticamente gerada usando as interfaces.

Essa tecnologia permite a criação de DSL's no VS Code, Eclipse Theia, aplicações Web e muito mais.

2.3.2 Visual Studio Code

Visual Studio Code[3], também conhecido como VS Code, é um leve porém podereso editor de código fonte que está disponível para Linux, Windows e macOS. Por padrão vem com suporte para JavaScript, TypeScript e Node.js e possui um vasto ecossistema de extensões para outras linguagens (como C++, C#, Java, Python, PHP, GO, .NET).

As extensões disponíveis para o VS Code permitem aos desenvolvedores algumas funcionalidades úteis como destacar sintaxe, formatação de código, refatoração de código, execução e depuração, controle de versão, entre outras funcionalidades úteis.

A flexibilidade e extensibilidade do VS Code será usada para a criação de uma extensão que dê suporte ao método FrameWeb.

[1] <https://langium.org/>
[2] <https://microsoft.github.io/language-server-protocol/>
[3] VS Code

2.3.3 Tecnologias de Desenvolvimento e execução

TypeScript[4] é um superconjunto tipado de JavaScript desenvolvido pela Microsoft. A linguagem adiciona tipagem estática, interfaces, classes e módulos ao JavaScript, possibilitando a construção de componentes mais robustos e facilitando a manutenção de código em projetos de maior escala. O código TypeScript é compilado para JavaScript padrão, podendo ser executado em qualquer ambiente que suporte JavaScript, como Node.js[5] e navegadores web. A linguagem TypeScript foi utilizada para o desenvolvimento das ferramentas apresentadas nesta monografia.

Node.js é um ambiente de execução para o JavaScript gratuito, de código aberto e cross-platform que permite aos desenvolvedores criarem servidores, aplicações Web, aplicativos de linha de comando e scripts. Ele é criado para construir aplicações de rede escaláveis e é assíncrono. Seu ambiente de execução é dirigido a eventos, possibilitando que múltiplas conexões sejam criadas paralelamente. Nesse trabalho o Node.js será o ambiente de execução das ferramentas da extensão.

[4] TypeScript
[5] Node.js

3 Uma linguagem textual para o FrameWeb

Neste capítulo é apresentada a DSL desenvolvida para dar suporte ao método FrameWeb, bem como a extensão *FrameWeb Writer Tool* (FWT) que foi criada para o Visual Studio Code, na qual a linguagem pode ser utilizada com recursos típicos de ambientes de desenvolvimento, como validação de sintaxe e geração automática de diagramas a partir da modelagem criada.

A extensão está disponível para download em <https://github.com/alekswheeler/frameweb-writer-tool> e pode ser instalada no *Visual Studio Code*: (1) via interface (*Install from VSIX*) ou (2) via terminal com o comando `code --install -extension frameweb-writer-tool.vsix`. A ferramenta identifica arquivos `.fwt` e, desde que o código esteja livre de erros sintáticos, produz diagramas de classe de forma automatizada ao salvar o documento.

O restante do capítulo é dividido da seguinte forma: a Seção 3.1 apresenta a gramática da linguagem, a Seção 3.2 descreve a funcionalidade de geração automática de diagramas e, por fim, a Seção 3.3 ilustra como a linguagem atende o método FrameWeb.

3.1 Gramática

Esta seção apresenta os elementos da gramática da linguagem desenvolvida para suporte ao método FrameWeb. Arquivos com extensão `.fwt` (*FrameWeb Writer Tool*) são utilizados para definição dos modelos do sistema a ser representado. A linguagem é suficientemente genérica para abranger tanto aspectos de modelagem UML pura quanto as extensões específicas do método FrameWeb, uma vez que este foi concebido como uma extensão da UML. A definição completa da gramática em EBNF, implementada em Langium, pode ser consultada no Apêndice A.

A apresentação dos elementos da linguagem segue a seguinte estrutura: primeiramente são apresentados os conceitos de modelagem UML, seguidos das extensões que habilitam o suporte ao método FrameWeb.

3.1.1 Classes

Classes constituem o elemento central da linguagem, sendo utilizadas para modelar entidades de domínio, serviços, controladores e demais componentes da arquitetura de software. Em **FWT**, classes são definidas usando a estrutura apresentada na Listagem 3.1.

A Figura 3 exemplifica a definição de duas classes (`Paciente` e `Medico`) com

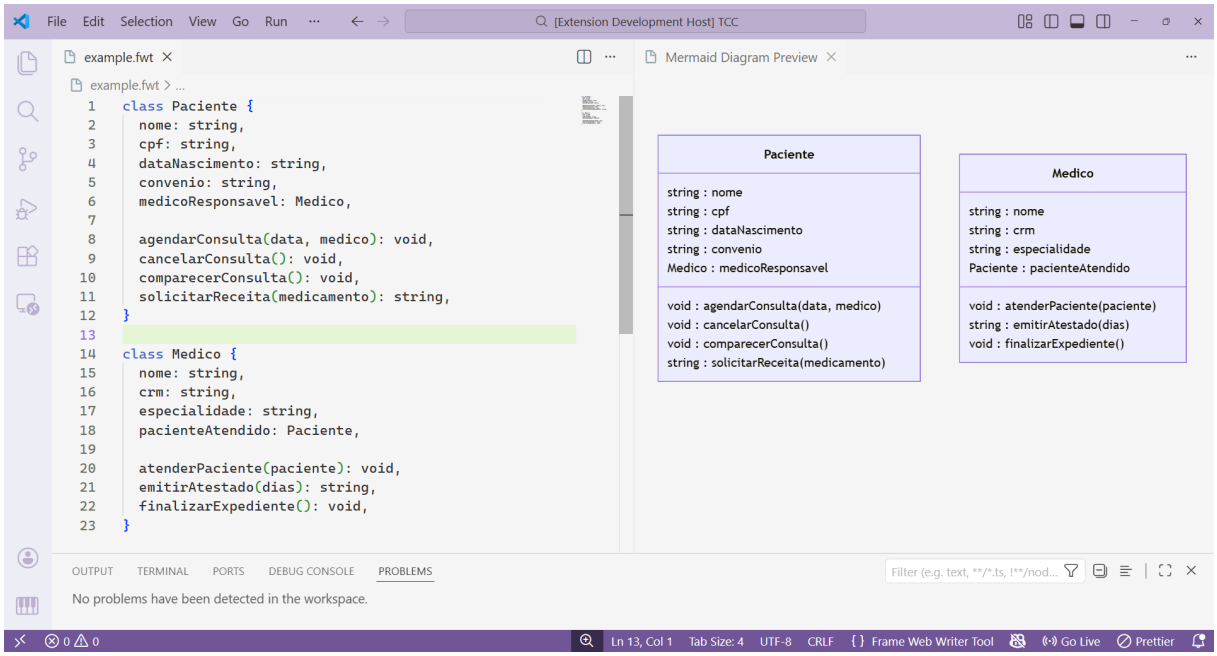

Figura 3 – Definição de classe em FWT.

atributos e métodos tipados, acompanhadas de suas representações gráficas. Note que, diferentemente dos atributos e métodos que requerem especificação de tipo (inclusive tipos customizados como `Medico` e `Paciente`), os parâmetros de métodos são declarados apenas por seus identificadores. Além disso, cada definição de atributo ou método deve ser seguida por vírgula. Por fim, é importante observar que tipo e nome de atributo e método na geração do diagrama pelo Mermaid são feitos de forma invertida (i.e., `tipo : nome` ao invés de `nome : tipo`) por conta de uma limitação do Mermaid.

Listagem 3.1 – Definição de classes em FWT

```
class NomeDaClasse {
    nomeAtributo: tipo  {restricao},
    nomeAtributo: tipo {restricao=valor},
    nomeMetodo(): tipo,
    nomeMetodo(parametros): tipo,
}
```

A FWT não define um conjunto fixo de tipos primitivos estáticos a serem utilizados pelo modelador. Assim como na UML, é possível declarar tipos personalizados, sejam eles tipos complexos ou referências a outras classes. A ferramenta oferece suporte a esse mecanismo de duas formas. A primeira consiste no uso de atributos complexos, em que o tipo do atributo é uma outra classe do modelo, estabelecendo uma referência cruzada. A segunda forma é por meio da definição explícita dos tipos utilizados na modelagem, realizada a partir de um arquivo de configuração do projeto nomeado como `fwt.config.json`. Como cada projeto pode representar um domínio distinto e possuir diferentes tecnologias alvo, os tipos podem variar conforme o contexto. A partir desse arquivo, o modelador especifica quais tipos são válidos, e a extensão passa a validar seu uso na linguagem.

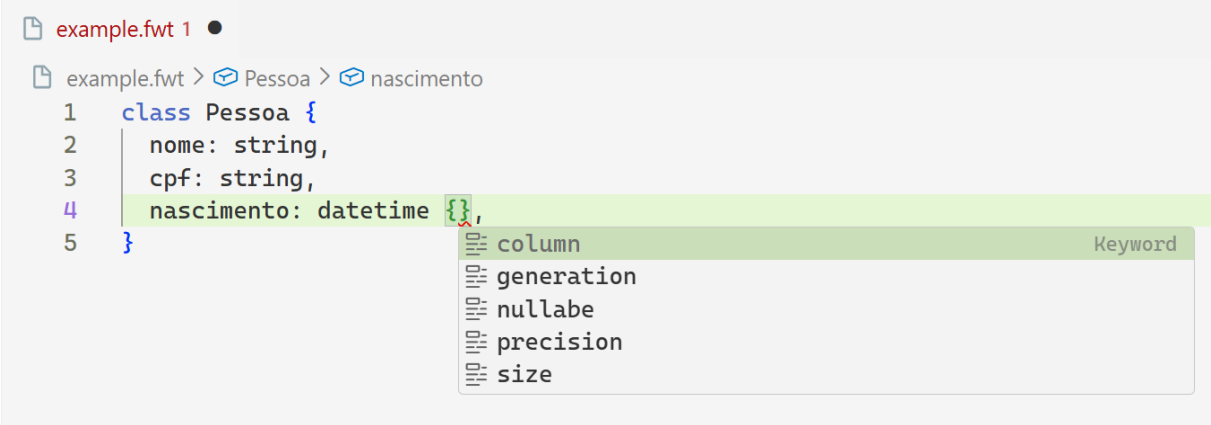

Figura 4 – Restrições sugeridas pela FWT.

Esse mecanismo contribui tanto para a flexibilidade quanto para a qualidade da modelagem, evitando ambiguidades comuns, como variações de escrita (por exemplo, string e String), e permitindo a detecção de tipos não definidos durante o processo de desenvolvimento. Caso não seja definido um arquivo de configuração, o uso de tipos na modelagem é permitido sem que haja validações sobre a declaração de tipos, exceto em referências cruzadas onde a classe deve existir para que seja referenciada como atributo. Veja a estrutura do arquivo de configuração na Listagem 3.2.

Listagem 3.2 – Arquivo de configuração para FWT

```
{
  "types": ["tipo1", "tipo2"]
}
```

Pelo fato de o FrameWeb utilizar *frameworks* responsáveis pela gestão do banco de dados, as restrições de atributos são amplamente empregadas para que a criação do esquema seja realizada de forma automática e aderente aos requisitos da aplicação. Dessa forma, a FWT também compreende as restrições de atributos que podem ser utilizadas na modelagem do sistema. Entre as restrições atualmente suportadas pela linguagem, destacam-se: *precision, generation, not null, null, size e column*. Adicionalmente, é possível empregar restrições que não estejam previamente listadas, desde que sigam o padrão de chave-valor ou apenas chave como exemplificado na Listagem 3.1. As figuras 4 e 5 apresentam as opções que a linguagem sugere para a restrição *precision*.

3.1.2 Interfaces

As interfaces representam um mecanismo fundamental de abstração na modelagem de sistemas orientados a objetos, sendo utilizadas para definir contratos que especificam um conjunto de operações sem impor detalhes de implementação. Diferentemente das classes, as interfaces não definem atributos, mas apenas métodos, que devem ser implementados pelas classes que as realizam. Dessa forma, as interfaces favorecem o desacoplamento

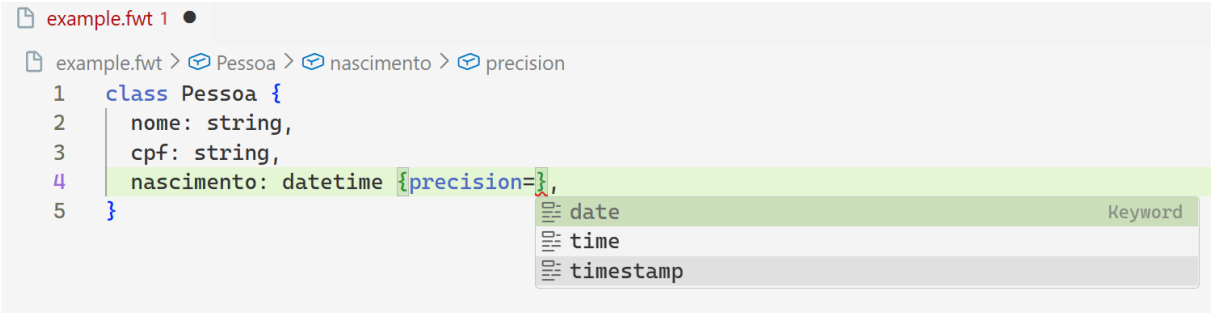

Figura 5 – Valores possíveis para *precision*.

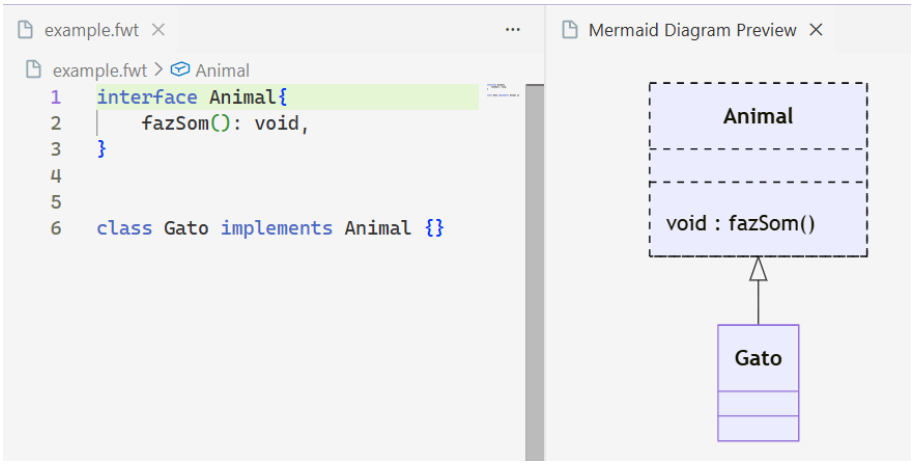

Figura 6 – Interfaces em FWT.

entre componentes, promovendo maior flexibilidade, reutilização e manutenibilidade do sistema. Interfaces são definidas como exemplificado na Listagem 3.3. A Figura 6 ilustra como classes e interfaces são definidas em **FWT**, onde interfaces são convencionalmente representadas com bordas tracejadas.

Listagem 3.3 – Definição de classe em FWT

```
interface NomeDaInterface {
    nomeMetodo(): tipo,
    nomeMetodo(parametros): tipo,
}
```

3.1.3 Herança e Implementação

A FWT oferece dois mecanismos para estabelecer relações entre tipos: a herança, por meio da palavra-chave *extends*, e a implementação de interfaces, por meio da palavra-chave *implements*. Esses mecanismos permitem o reuso e a definição de contratos que as classes devem cumprir, promovendo uma organização mais modular e flexível dos sistemas.

A herança com *extends* possibilita que uma classe derive de outra classe existente, herdando seus atributos e comportamentos. O mecanismo de herança com *extends* permite que uma classe herde de apenas uma outra classe, assim como uma interface pode herdar

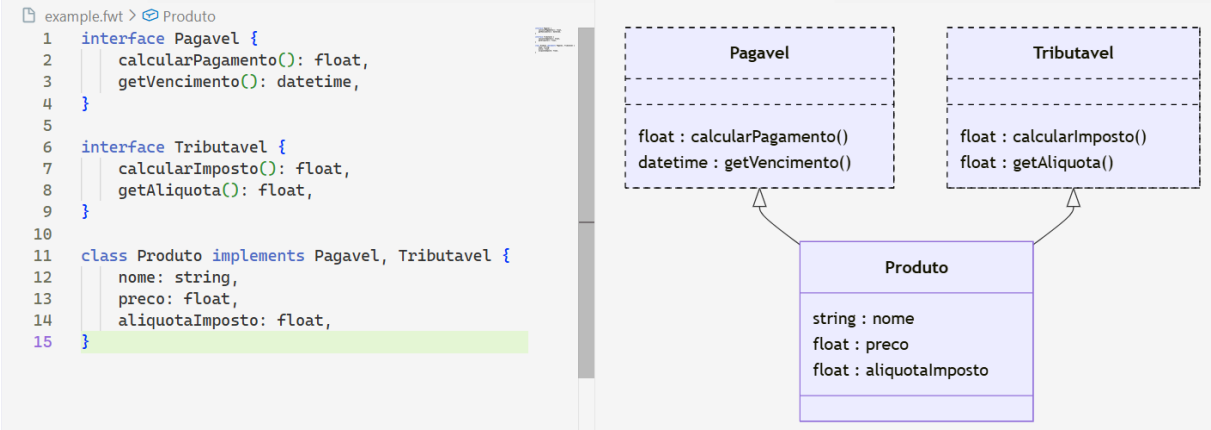

Figura 7 – Exemplo de Interface em FWT.

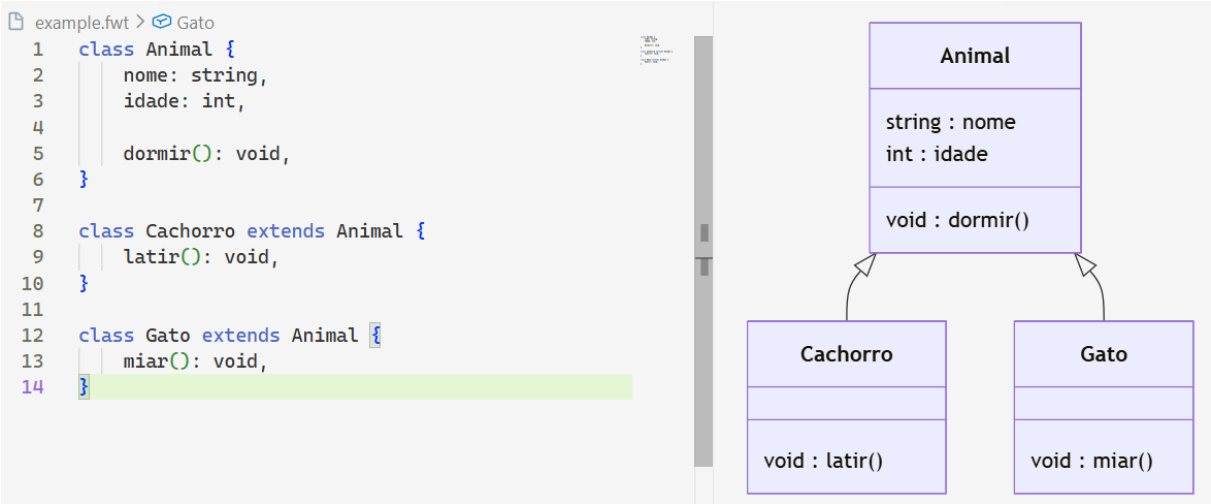

Figura 8 – Exemplo de Herança em FWT.

de apenas uma outra interface. Como exemplo, considere uma classe *Animal* que declara atributos como *nome* e *idade* e métodos como *dormir*; a classe *Cachorro* pode estendê-la com *extends Animal*, herdando seus membros e adicionando funcionalidades específicas como o método *latir*, enquanto a classe *Gato* também pode estender *Animal* e adicionar seu próprio comportamento com o método *miar*.

A implementação com *implements* estabelece que uma classe assume o compromisso de fornecer as funcionalidades declaradas em uma ou mais interfaces. Diferentemente da herança, uma classe pode implementar múltiplas interfaces simultaneamente, devendo implementar todos os métodos por elas definidos. Interfaces atuam como contratos que desacoplam a definição do comportamento da sua implementação concreta. Por exemplo, as interfaces *Pagavel* e *Tributavel* declaram, respectivamente, os métodos *calcularPagamento* e *calcularImposto*; a classe *Produto* pode implementar ambas com *implements Pagavel, Tributavel*, comprometendo-se a fornecer as implementações concretas dos métodos exigidos por cada interface, além de declarar seus próprios atributos como *nome*, *preco* e *aliquotaImposto*.

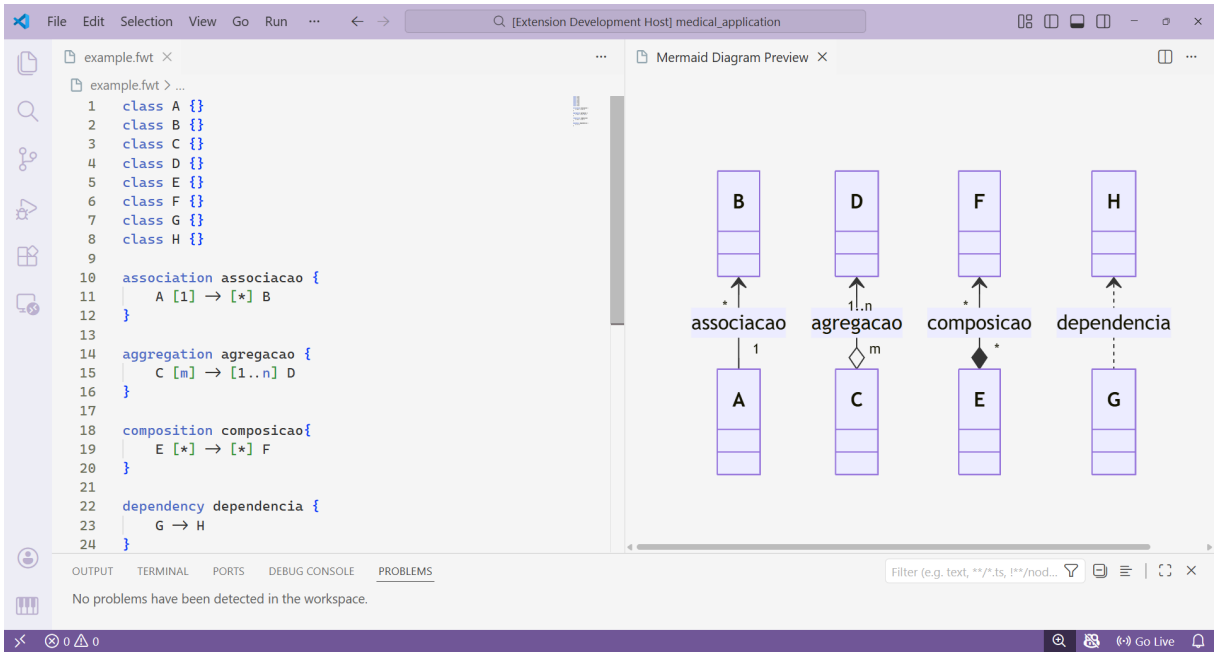

Figura 9 – Relacionamentos entre classe em FWT.

As Figuras 7 e 8 ilustram graficamente essas relações. Esses mecanismos, quando utilizados adequadamente, contribuem para a construção de sistemas mais coesos, com menor acoplamento e maior potencial de reuso.

3.1.4 Relacionamento entre classes

Relações ou Associações são uma parte importante da modelagem UML que permite reuso e generalizações e adicionam muita flexibilidade e expressividade semântica para a modelagem. Como a proposta da linguagem é ser uma ferramenta UML e também de apoio ao FrameWeb as relações são definidas na linguagem.

A estrutura de UML de associação na linguagem é definida como na Listagem 3.4.

Listagem 3.4 – Relacionamento entre classes

```
1 association {
2     ClasseA [multiplicidadeA] -> [multiplicidadeB] ClasseB
3 }
```

A Figura 9 exemplifica como os relacionamentos de classe são definidos em FWT. Outros tipos de associação podem ser definidos como Agregação e Composição. Utilizando a mesma estrutura apresentada.

As associações em **FWT** possuem obrigatoriamente navegabilidade (direção e sentido), podendo ainda especificar multiplicidade e nome identificador. A Listagem 3.5 apresenta todas essas estruturas sendo utilizadas na linguagem. No entanto, a dependência difere de associações e composições por representar um uso transitório entre classes, sem vínculo estrutural permanente. Por isso, não possui multiplicidade.

Listagem 3.5 – Multiplicidade em relacionamentos

```
1     aggregation subareas {
2         Area [0..1] -> [*] Area
3     }
```

Na Listagem 3.5 é apresentado um relacionamento recursivo, no qual a classe *Area* possui uma associação com ela mesma. Esse relacionamento representa a hierarquia de subáreas, em que uma área pode possuir nenhuma ou várias subáreas, enquanto uma subárea pertence ou é referenciada por, no máximo, uma área.

3.1.5 Definição de pacotes

Pacotes são usados em UML para delimitar *namespaces*, agrupar funcionalidades ou fazer um agrupamento lógico. É um recurso muito importante para o projeto de sistemas pois permite organizar seus módulos e fazer reuso de suas classes.

Um pacote pode ser declarado de acordo com a seguinte sintaxe descrita na Listagem 3.6.

Listagem 3.6 – Pacotes em FWT

```
1 package NomeDoPacote {
2     // Conteudo do pacote
3 }
```

Um pacote pode conter classes, interfaces e associações. Além disso, os pacotes são importantes para a atribuição dos estereótipos do FrameWeb que podem ser adicionados às classes. Uma vez que o estereótipo do pacote é definido, os possíveis estereótipos de classe do tipo do pacote ficam disponíveis para serem utilizados.

3.1.6 Importação de pacotes

Para melhor organização do projeto, é possível criar a modelagem em diversos arquivos e realizar importações (veja a Listagem 3.7) como é feito nas linguagem de programação. Apenas referenciamos o arquivo que contém a definição da classe e quais as classes são importadas daquele arquivo.

Listagem 3.7 – Importação de classes

```
1 import Classe1, Classe2 from "caminho/arquivo.fwt";
```

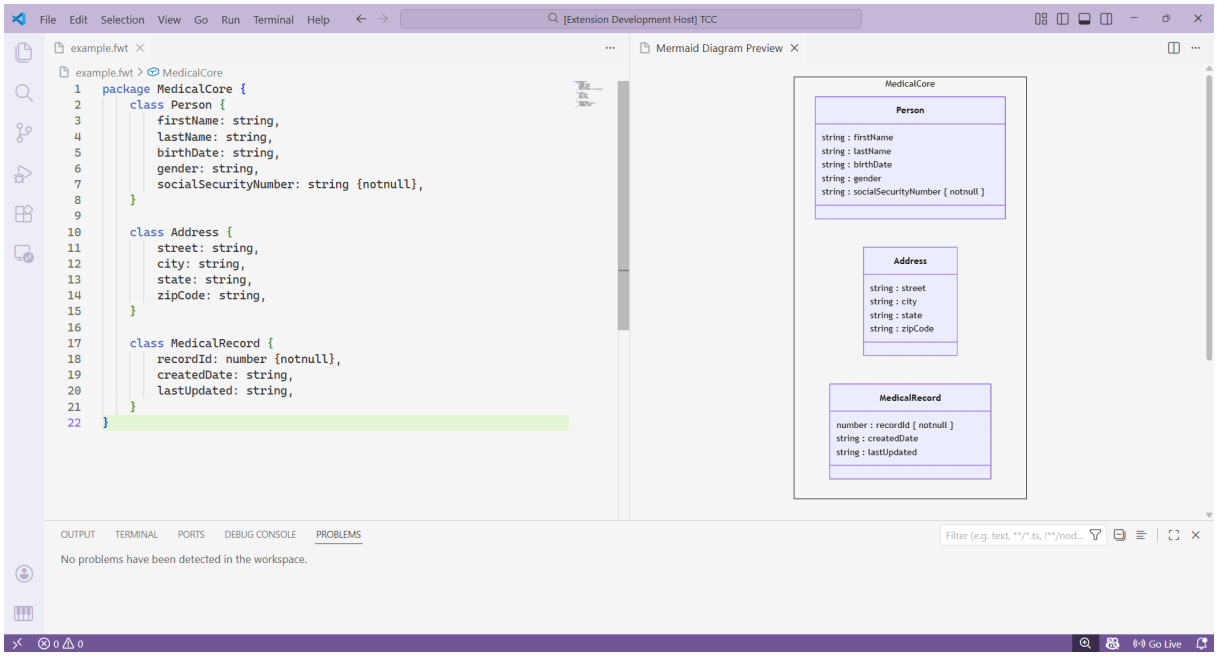

Figura 10 – Editor de código **FWT** (esquerda) e diagrama de classes gerado automaticamente (direita) no Visual Studio Code.

3.2 Geração de diagramas

Apesar das vantagens das linguagens textuais para automação e versionamento, o processo puramente textual pode impactar na fluidez da atividade de modelagem. Para trazer recursos familiares a essa atividade, a **FWT** oferece a geração automática de diagramas a partir da linguagem proposta, fornecendo um suporte visual.

Com a extensão instalada e ativa a **FWT** observa eventos com a API do Visual Studio Code e gera diagramas com um comando natural no processo de desenvolvimento: salvar o arquivo. Ao clicar no ícone de salvamento ou utilizando o atalho CTRL+S a **FWT** automaticamente processa o código criado, faz as validações e abre uma *webview*, normalmente no lado direito com o diagrama criado para o arquivo em foco no editor de texto, como ilustrado na Figura 10.

O processo de geração de diagramas utiliza o modelo semântico (AST anotada) fornecido pelo Langium em forma de interfaces TypeScript. A **FWT** percorre os nós dessa estrutura por meio de um padrão de visitação, mapeando cada elemento da linguagem para sua representação equivalente em código Mermaid. Dessa forma, classes, interfaces, atributos, métodos e relacionamentos definidos em **FWT** são traduzidos sistematicamente para a sintaxe de diagramas de classe do Mermaid.

Para a visualização gráfica dos modelos, a extensão emprega uma *webview*, recurso do Visual Studio Code que permite a renderização de conteúdos baseados em HTML, CSS e JavaScript. Nessa *webview*, a biblioteca Mermaid[1] é responsável pela geração dos

[1] Mermaid

diagramas a partir das descrições textuais produzidas pela linguagem, enquanto a biblioteca Panzoom[2] fornece mecanismos de navegação e interação, como zoom e movimentação, contribuindo para uma experiência mais fluida na interação dos diagramas.

3.3 Suporte ao FrameWeb

Na Seção 2.1 discutimos sobre o FrameWeb e as camadas sugeridas de *design*. Nessa seção será demonstrado como a DSL criada atende às especificações do método a partir de exemplos.

3.3.1 Extensão da UML

O FrameWeb define extensões leves (*lightweight extensions*) ao meta-modelo da UML para representar componentes típicos da plataforma Web e das categorias de *frameworks* propostas (Souza, 2007), para representar essas estruturas a **FWT** define marcações de estereótipos e restrições em classes e atributos para representar as extensões do FrameWeb. A Listagem 3.8 apresenta um pseudocódigo que exemplifica a estrutura típica de modelagem seguindo o método FrameWeb.

Listagem 3.8 – Pseudocódigo de estereótipos e restrições em FWT

```
1
2 @packageStereotype
3 package namePackage {
4     @classStereotype
5     class ClassName{
6         @attributeStereotype
7         attributeName: attributeType {constraints},
8         methodName(params): methodType,
9     }
10 }
```

3.3.2 Modelo de Navegação

Para essa demonstração iremos utilizar o modelo de navegação detalhado de um Sistema Web, exibido na Figura 11. O modelo navegação contém o fluxo de interação do usuário com a interface do sistema e as classes de controle e serviço para prover as funcionalidades do sistema. Para isso são representados formulários e páginas e também as possibilidades de navegação dentro desse módulo do sistema. A Listagem 3.9 contém a representação desse diagrama utilizando os elementos de linguagem criados. E a Figura 12 contém o diagrama gerado.

O estereótipo de classes e pacotes é feito similarmente aos *Annotations* em Java utilizando o caractere '@'. O estereótipo de pacote *view* é utilizado para o modelo de

[2] Panzoom

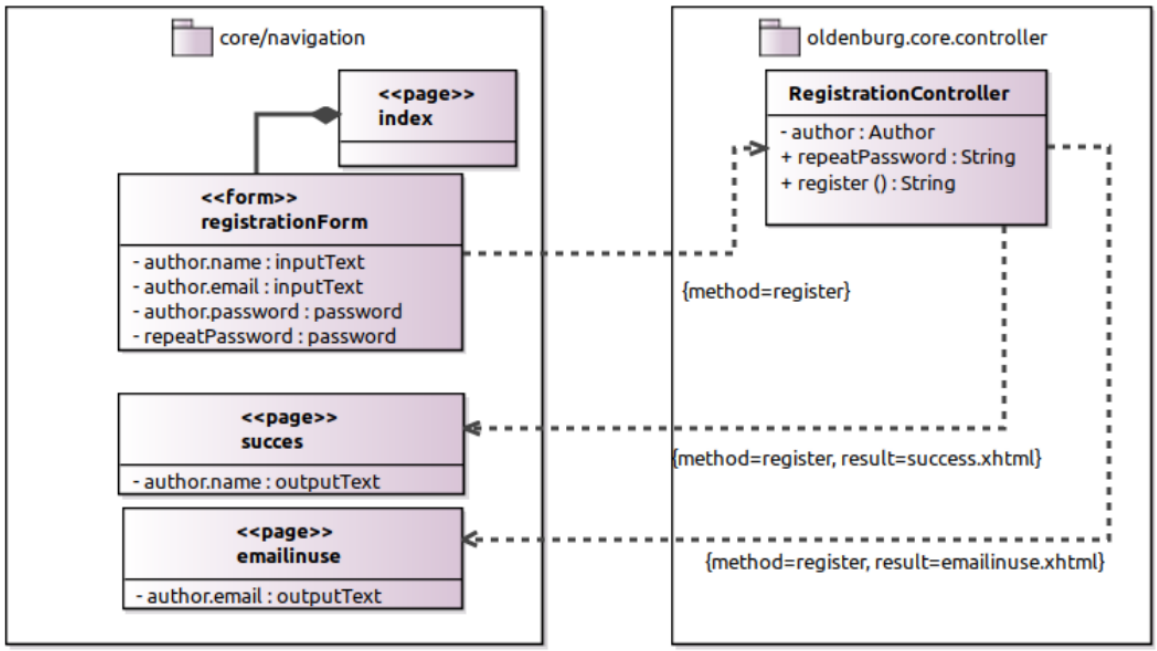

Figura 11 – Modelo de Navegação de um WIS referência (Hoppe; Souza, 2023)

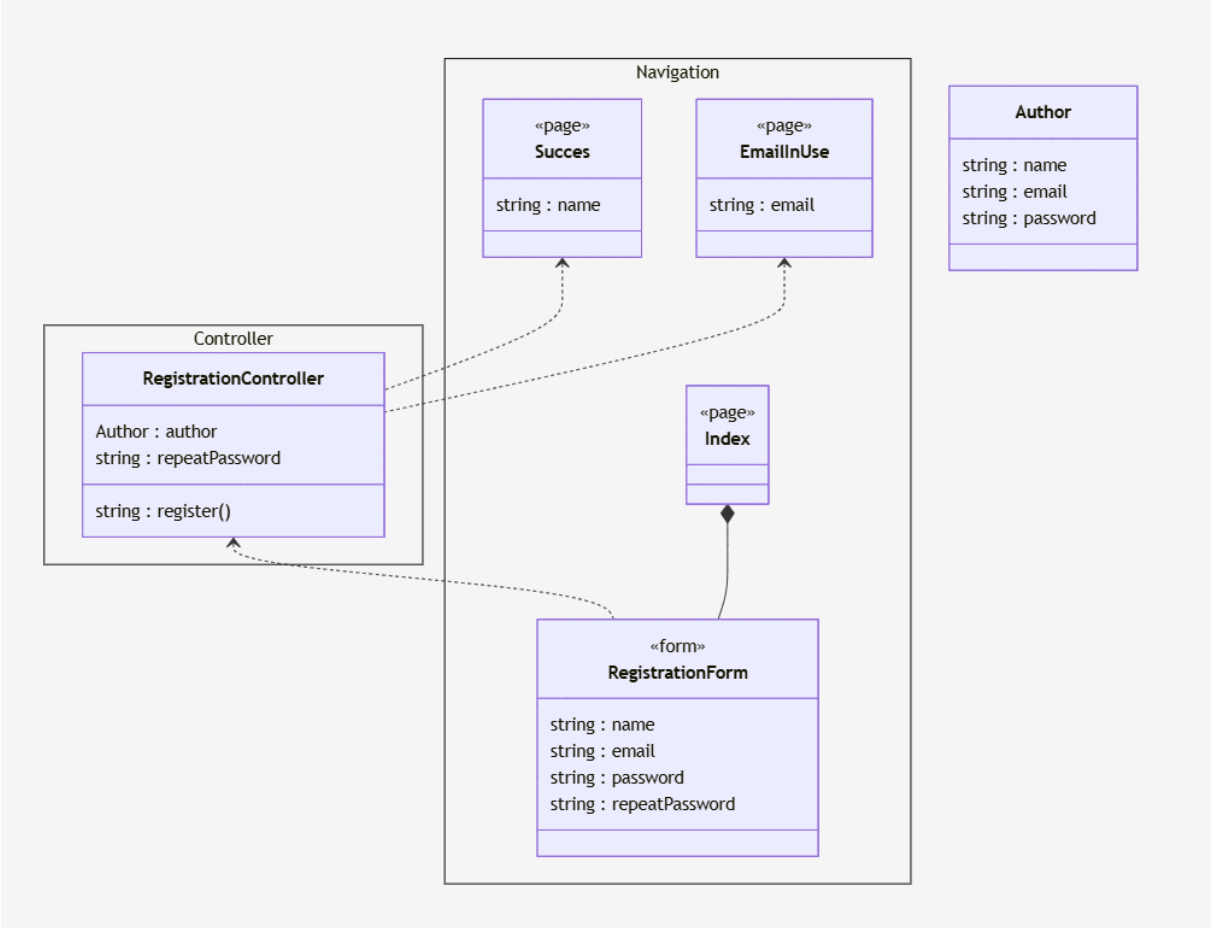

Figura 12 – Modelo de navegação em FWT.

Listagem 3.9 – Modelo de navegação em FWT.

```
@view
package Navigation {

  @page
  class Index{ }

  @form
  class RegistrationForm {
    name: string,
    email: string,
    password: string,
    repeatPassword: string,
  }

  dependency {
    RegistrationForm -> RegistrationController
  }

  @page
  class Success { name: string, }

  @page class EmailInUse { email: string, }

  composition {
    RegistrationForm -> Index
  }
}

@controller
package Controller {

  dependency {
    RegistrationController -> Success
    RegistrationController -> EmailInUse
  }

  class RegistrationController {
    author: Author,
    repeatPassword: string,
    register(): string,
  }
}

class Author {
  name: string,
  email: string,
  password: string,
}
```

navegação. Para atribuir estereótipos, o usuário utiliza o recurso de auto-complete que apresenta os tipos de estereótipos definidos no método FrameWeb (Souza, 2007). A extensão adota uma abordagem em duas camadas: o auto-complete diferencia estereótipos por tipo de elemento (pacote, classe ou atributo), e validações semânticas verificam restrições arquiteturais mais específicas.

3.3.3 Modelo de Entidades

O modelo de entidades é um diagrama de classes UML que representa as entidades do domínio do problema, e o seu mapeamento para as tabelas do banco de dados relacional.

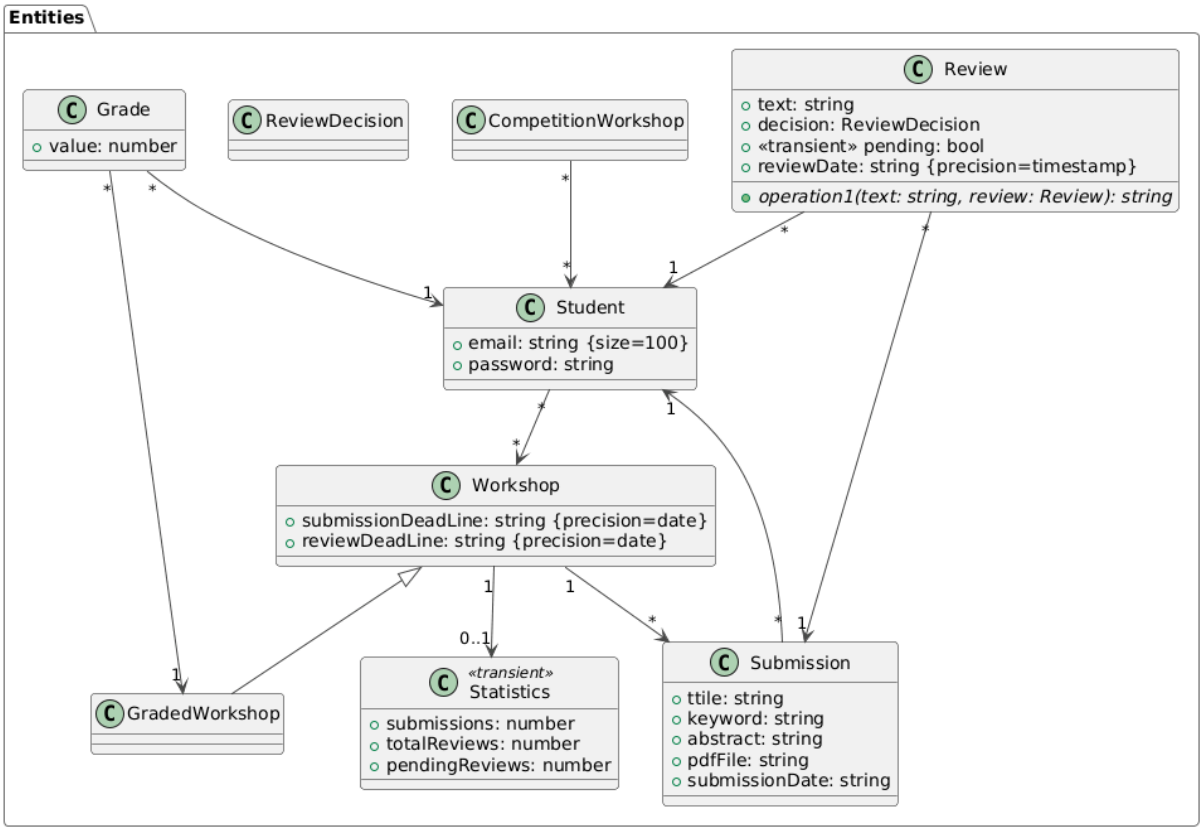

Figura 13 – Modelo de Entidades do WIS *Oldenburg* (Silva, 2023).

Os mapeamentos de persistência são meta-dados das classes de entidades que permitem aos *frameworks* ORM realizar a persistência dos objetos em memória num banco de dados relacional. Tais mapeamentos são feitos por meio de extensões leves da UML como estereótipos e restrições (Souza, 2007). Na DSL criada esse comportamento é natural ao estender as construções criadas para modelagem UML permitindo a adição das etiquetas de meta-dados do FrameWeb.

Para a exemplificação desse modelo utilizaremos a Figura 13 que contém o modelo de Entidades de uma aplicação web. A classe *Student* contém restrições de atributo que são utilizadas em bancos de dados como *column* e *size*. Esses elementos podem ser observados na linguagem ao utilizar a mesma sintaxe para representar as restrições como visto na Seção 3.1.1. A Listagem 3.10 contém o código em **FWT** para a representação da Figura 13. A Figura 14 contém o diagrama gerado usando **FWT**.

Listagem 3.10 – Modelo de Entidades de um WIS.

```
1 @domain
2 package MyPackage {
3
4   class Grade {
5     value: number,
6   }
7
8   class Student {
```

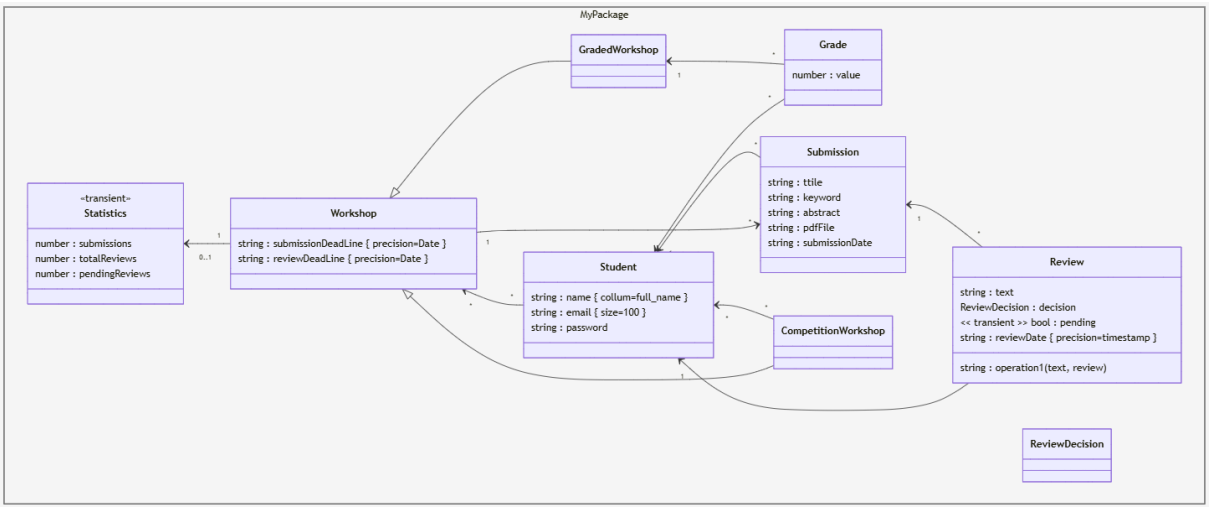

Figura 14 – Modelo de entidades usando FWT.

```
9     name: string {column=full_name},
10    email: string {size=100},
11    password: string,
12  }
13
14  class ReviewDecision {}
15
16  class Review {
17    text: string,
18    decision: ReviewDecision,
19    @transient pending: bool,
20    reviewDate: string {precision=timestamp},
21    operation1(text, review): string,
22  }
23
24  class Workshop {
25    submissionDeadLine: string {precision=date},
26    reviewDeadLine: string {precision=date},
27  }
28
29  class GradedWorkshop extends Workshop { }
30
31  class CompetitionWorkshop extends Workshop { }
32
33  @transient
34  class Statistics {
35    submissions: number,
36    totalReviews: number,
37    pendingReviews: number,
38  }
39
40  class Submission {
41    title: string,
42    keyword: string,
43    abstract: string,
44    pdfFile: string,
45    submissionDate: string,
46  }
```

```
47
48   association {
49     Student [*] -> [*] Workshop
50     Grade [*] -> [1] GradedWorkshop
51     Review [*] -> [1] Submission
52     Workshop [1] -> [*] Submission
53   }
54
55   association {
56     CompetitionWorkshop [*] -> [*] Student
57     Submission [*] -> Student
58     Workshop [1] -> [0..1] Statistics
59     Review [*] -> [1] Student
60     Grade [*] -> [1] Student
61   }
62 }
```

Além disso, outras restrições podem ser adicionadas às classes para representarem interações com o banco de dados como Persistente (*persistent*), Transiente (*transient*) ou Mapeada (*mapped*):

- **Persistent** («*persistent*»): indica que a classe (ou atributo) é persistente. Os objetos desta classe serão salvos em tabelas de um banco de dados relacional pelo *framework* ORM. É o valor padrão (*default*) para classes no Modelo de Entidades, o que significa que se nenhum estereótipo for especificado, a classe é considerada persistente (Souza, 2007);
- **Transient** («*transient*»): indica que a classe (ou atributo) é não persistente. Se aplicado a uma classe, significa que ela não é uma entidade persistente. Se aplicado a um atributo, significa que esse atributo não será armazenado no banco de dados, mesmo que a classe seja persistente (Souza, 2007);
- **Mapped** («*mapped*»): é um tipo especial de classe não persistente. Uma classe estereotipada como *mapped* não é uma entidade persistente em si. Suas propriedades (atributos e associações) e mapeamentos de persistência serão herdados pelas subclasses que a estendem. Exemplo: classes utilitárias que definem atributos comuns, como identidade (id) e versionamento (version), são declaradas como mapeadas, permitindo que suas subclasses herdem tanto os atributos quanto os mapeamentos de persistência definidos (Souza, 2007).

Na Figura 15 a classe *ObjetoPersistente* exemplifica como o *mapped* pode ser utilizado para a criação de atributos universais, ou seja, todas as subclasses de *ObjetoPersistente* mapeada herdam seus atributos de persistência, porém a classe *ObjetoPersistente* não é persistida no banco de dados. A Listagem 3.11 contém o código para esse diagrama.

Em resumo, enquanto um objeto *persistent* é diretamente salvo no banco e um

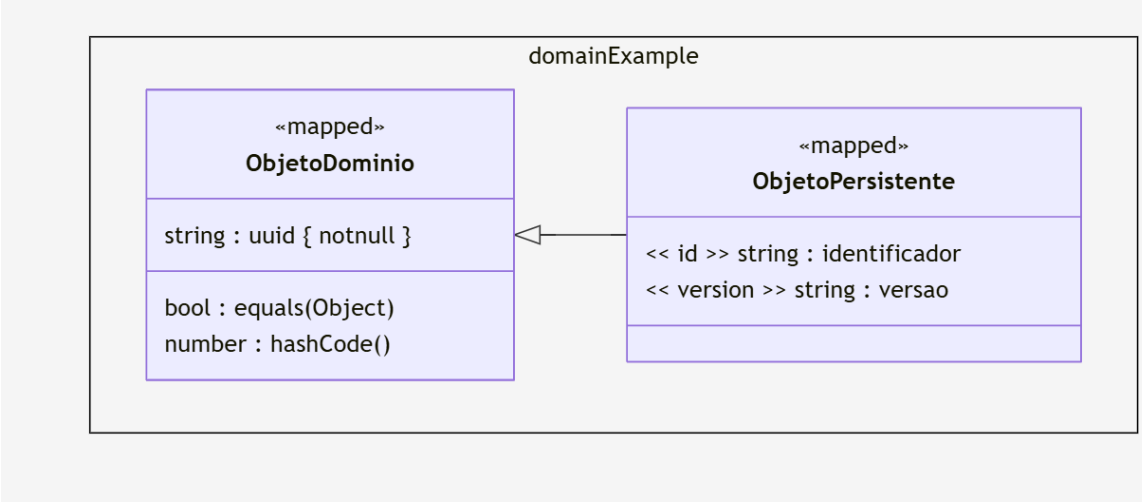

Figura 15 – Diagrama gerado pela extensão utilizando mapped.

Listagem 3.11 – Utilizando estereótipo mapped em FWT.

```
1  @domain
2  package domainExample {
3
4    @mapped
5    class ObjetoDominio {
6      uuid: string {notnull},
7      equals(Object): bool,
8      hashCode(): number,
9    }
10
11   @mapped
12   class ObjetoPersistente extends ObjetoDominio{
13     @id
14     string: identificador,
15
16     @version
17     string: versao,
18   }
19 }
```

objeto *transient* não é, um objeto *mapped* serve como uma *"classe base de persistência"* que empresta seus mapeamentos e propriedades para classes persistentes filhas.

3.3.4 Modelo de Aplicação e Modelo de Persistência

O modelo de persistência é um diagrama de classes da UML que representa as classes responsáveis pela persistência das instâncias das classes de domínio. Para o modelo de persistência o FrameWeb indica a utilização do padrão de projeto DAO, este pacote não possui extensões da UML, porém recebe um estereótipo próprio na **FWT**. Veja a Listagem 3.12.

Ao adicionar esse estereótipo no pacote, a **FWT** fará validações específicas do mesmo, além de permitir o uso de estereótipo de classes correspondente, indica ao modelador que as suas classes concretas devem herdar de uma interface DAO. Conforme apresentado

Listagem 3.12 – Pacote de persistencia do Oldenburg (Silva, 2023).

```
@persistence
package oldenburg.core.persistence_ {

   interface GradeRepositoryDAO {}
   interface ReviewRepositoryDAO {}
   interface StudentRepositoryDAO {}
   interface SubmissionRepositoryDAO {}
   interface WorkshopRepositoryDAO {}

   class GradeRepository implements GradeRepositoryDAO {}
   class ReviewRepository implements ReviewRepositoryDAO {}
   class StudentRepository implements StudentRepositoryDAO{}
   class SubmissionRepository implements SubmissionRepositoryDAO {}
   class WorkshopRepository implements WorkshopRepositoryDAO {}
}
```

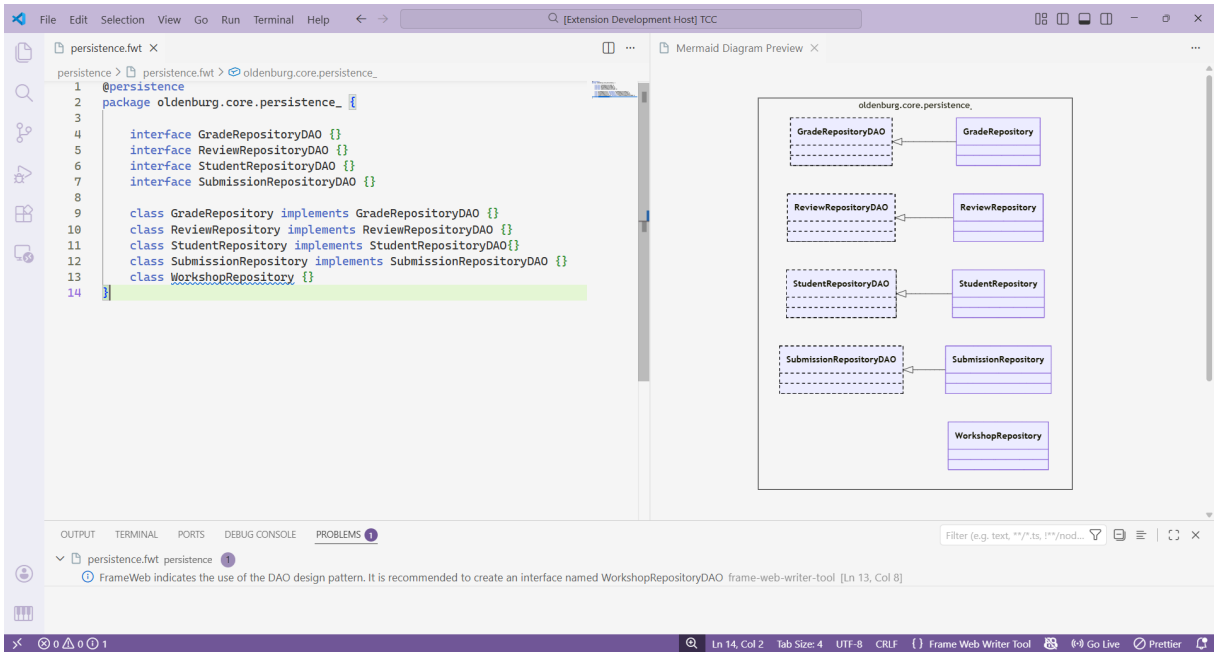

Figura 16 – Modelo de persistência em FWT.

na Figura 16, a ferramenta sugere ao modelador a criação de interfaces DAO para as classes concretas.

Já o modelo de Aplicação representa as classes de serviço responsáveis pela implementação dos casos de uso. Essas classes tornam explícitas as dependências com os modelos de Entidades e de Persistência. Assim como o modelo de Persistência, o modelo de Aplicação não possui extensões próprias da UML; entretanto, como é sugerido no design a separação por meio de interfaces, a **FWT** também propõe um estereótipo específico para esse modelo. A Listagem 3.13 apresenta um exemplo do modelo de serviço do sistema *Oldenburg* (Silva, 2023).

Semelhante ao modelo de persistência, a **FWT** sugere ao modelador a criação de interfaces para cada uma das classes de serviço. A Figura 17 ilustra esse mecanismo em ação.

Listagem 3.13 – Modelo de aplicação do sistema Oldenburg (Silva, 2023) em FWT.

```
@service
Package oldenburg.core.application_ {
  interface WorkshopService {}
  interface ReviewService {}
  interface StudentService {}
  interface SubmissionService {}
  interface GradeService {}

  class WorkshopServiceImpl implements WorkshopService {}
  class ReviewServiceImpl implements ReviewService {}
  class StudentServiceImpl implements StudentService {}
  class SubmissionServiceImpl implements SubmissionService {}
  class GradeServiceImpl implements GradeService {}
}
```

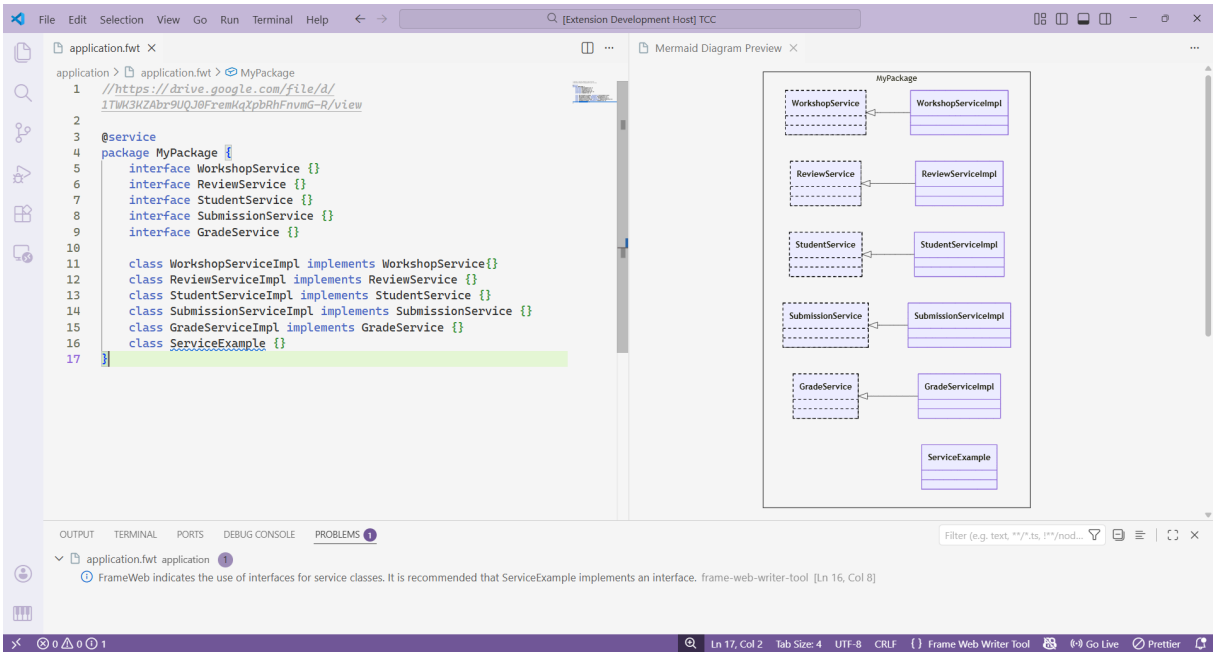

Figura 17 – Pacote de serviço em FWT.

3.4 Validações

Como o Langium é profundamente integrado ao LSP[3], diversas facilidades são disponibilizadas no desenvolvimento de DSLs. Essa integração possibilita funcionalidades comumente encontradas em linguagens de programação, como *code completion*, validações, resolução de referências entre arquivos, entre outras. Dessa forma, ao utilizar essa ferramenta, as validações sintáticas são realizadas automaticamente de acordo com a gramática definida e, adicionalmente, algumas validações semânticas também são tratadas pela própria ferramenta.

Nesta seção, são apresentadas as validações sintáticas e semânticas relevantes para a modelagem de sistemas Web utilizando o método FrameWeb, que podem ser implementadas a partir da linguagem definida. O foco principal está nas validações semânticas associadas às

[3] <https://microsoft.github.io/language-server-protocol/>

regras do método FrameWeb, embora também sejam abordadas validações mais genéricas, aplicáveis a processos de modelagem em geral.

Alguns exemplos de validações incluem verificações de classes não declaradas em Atributos Complexos, erros de sintaxe, além de validações específicas do FrameWeb, como a verificação dos tipos de classes permitidos em determinados pacotes.

As validações do FrameWeb contemplam a estrutura proposta pelo método, que divide a modelagem em diferentes modelos e pacotes, restringindo quais tipos de elementos podem ser declarados na linguagem e em quais contextos eles são permitidos.

3.4.1 Validação de Estereótipos

Uma das principais validações diz respeito ao uso dos estereótipos de classe e de pacote. Por exemplo, não é permitido declarar uma classe com estereótipo *page* dentro de um pacote do tipo *persistence*, pois isso configura uma violação semântica das regras do método. De forma análoga, estereótipos do Modelo de Entidades não podem ser utilizados em pacotes do tipo *view*.

Como ilustrado na Figura 18, a extensão realiza a validação dos estereótipos de classe aceitos de acordo com o estereótipo do pacote em que estão inseridos, garantindo a conformidade com a estrutura proposta pelo FrameWeb. A gramática diferencia estereótipos apenas por tipo de elemento (pacote, classe ou atributo), o que implica que o auto-complete apresenta todos os estereótipos do tipo correspondente sem considerar o contexto arquitetural. As validações arquiteturais mais granulares, como a compatibilidade entre estereótipos de pacote e classe, são delegadas para a fase de validação semântica.

Outra validação importante é que não é possível declarar estereótipos de classe sem que o pacote ao qual pertencem possua um estereótipo definido. Ou seja, uma classe estereotipada só pode ser declarada dentro de um pacote que também esteja corretamente estereotipado, reforçando a separação entre os diferentes modelos da arquitetura. Na Figura 19 é possível visualizar essa restrição.

Como discutido na Seção 3.3.4, o FrameWeb recomenda o uso de interfaces e respectivas implementações nos pacotes de classes de serviço e de persistência. Essa característica já foi apresentada anteriormente, porém é relevante mencioná-la também no contexto das validações da linguagem, uma vez que sua verificação é realizada durante a fase de validação semântica.

Nesse processo, os nós de classe da Árvore de Sintaxe Abstrata (AST) são analisados e, caso uma classe pertencente a esses pacotes estereotipados não implemente nenhuma interface, a linguagem emite uma dica sugerindo ao modelador que especifique a interface correspondente. Essa validação não impede a continuidade do processo de modelagem, ou seja, não gera erros que bloqueiem a geração dos diagramas, como ilustrado na Figura 16.

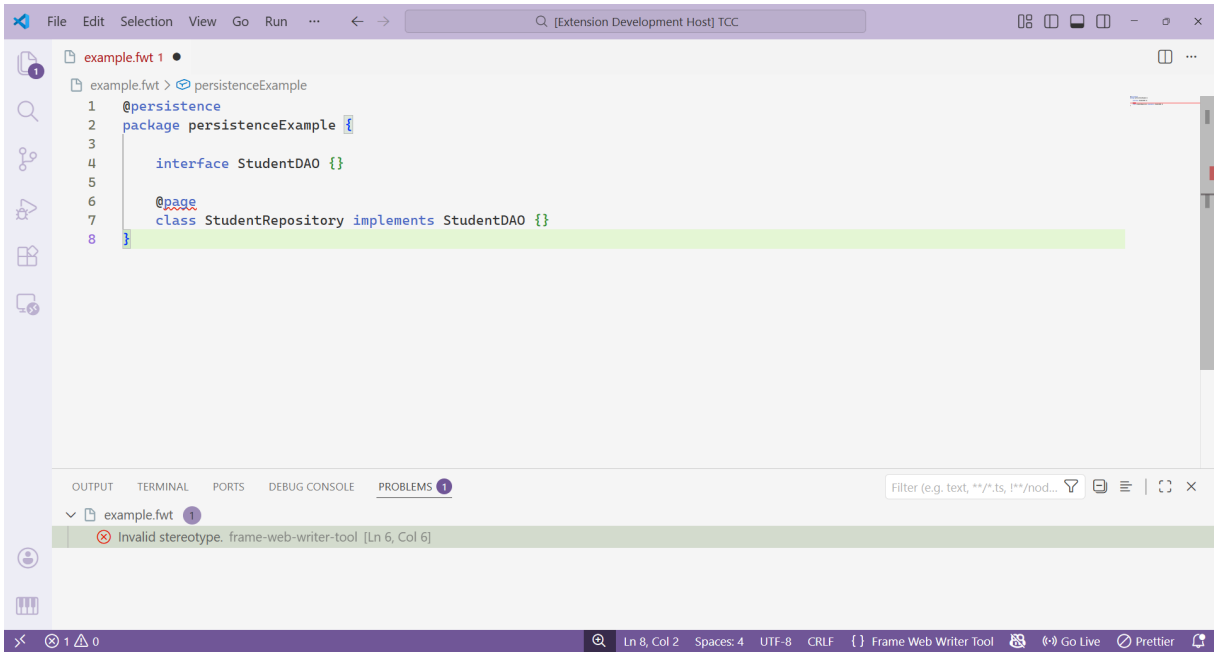

Figura 18 – Validação de estereótipo de classe.

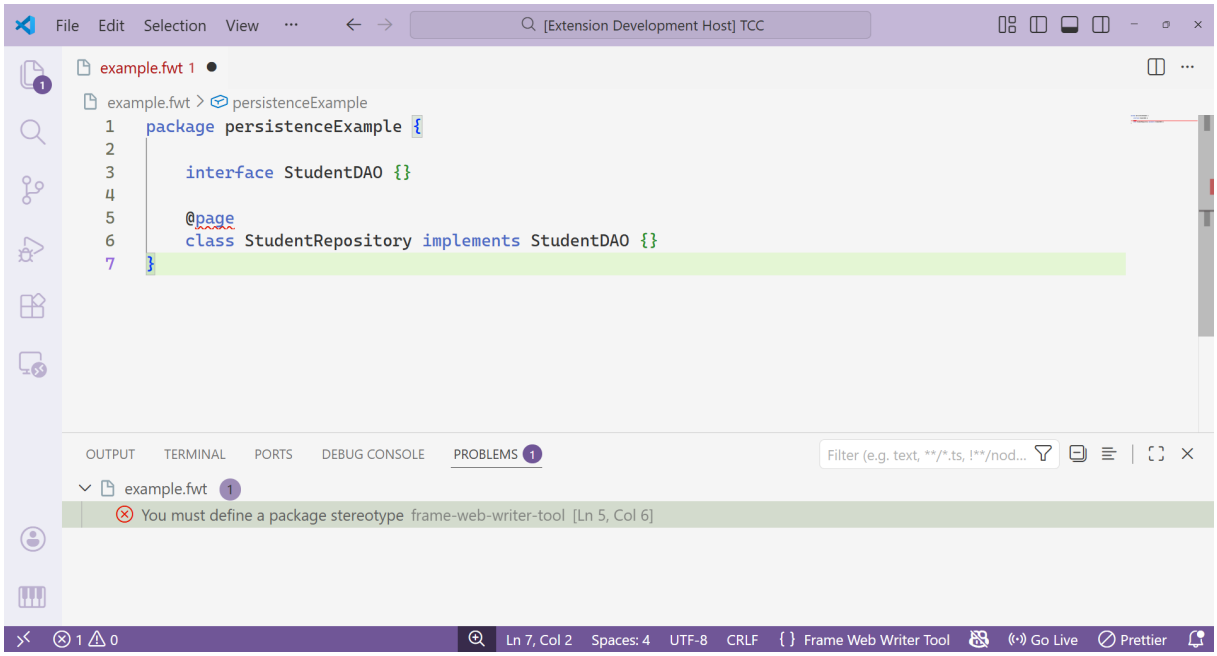

Figura 19 – Validação de estereótipo de classe em pacotes sem estereótipo.

3.4.2 Validação de tipos

No contexto das validações semânticas da linguagem, a verificação dos tipos é realizada continuamente durante o processo de modelagem. Quando existem tipos definidos no arquivo de configuração, esse artefato é visitado durante a fase de inicialização da extensão, por meio de um serviço customizado que o Langium permite criar, o qual registra os tipos definidos pelo modelador. Esse serviço pode ser atualizado caso o arquivo de configuração seja modificado, porém a visitação direta desse artefato não ocorre a todo momento. Durante a fase de validação, a linguagem consulta esse serviço para comparar os tipos declarados nos atributos com os tipos previamente registrados. Na ausência do arquivo de configuração, não há validação de tipos primitivos, uma vez que a linguagem não impõe um conjunto fixo de tipos básicos, restando apenas a validação de tipos complexos, definidos por meio de referências a outras classes na modelagem.

Para viabilizar suporte a *code completion* e checagem automática de existência das classes, todos os tipos de atributos foram modelados como referências cruzadas na gramática. Contudo, no caso de tipos definidos via arquivo de configuração, essa referência permanece vazia na AST, o que semanticamente não representa um erro, mas sim um tipo primitivo definido externamente. Dessa forma, foi necessária uma customização no processo de resolução de referências cruzadas para garantir simultaneamente flexibilidade ao permitir a definição de tipos próprios pelo usuário e rigor semântico ao manter as verificações e resoluções de referência para atributos de tipos complexos.

3.4.3 Validação de referência a classes

Além das validações semânticas implementadas especificamente para a linguagem, o *framework* Langium também realiza validações automáticas por meio de seus mecanismos internos. Um exemplo disso ocorre no contexto de tipos complexos, situação em que uma classe é referenciada como tipo de atributo de outra classe, ou em relacionamentos, nos quais as classes são referenciadas nos nós da relação. Nesses casos, a resolução das referências é realizada por meio de referências cruzadas (*cross-references*), mecanismo pelo qual a ferramenta verifica a existência do elemento referenciado e estabelece a ligação correspondente na AST. Essa validação é fundamental para garantir a consistência semântica do modelo, uma vez que impede referências a elementos inexistentes.

Ressalta-se, entretanto, que tal validação só é possível quando as referências cruzadas são devidamente especificadas na gramática da DSL, evidenciando que, embora a ferramenta ofereça suporte automático, sua efetivação depende da correta modelagem da linguagem. A Figura 20 ilustra esse mecanismo da ferramenta.

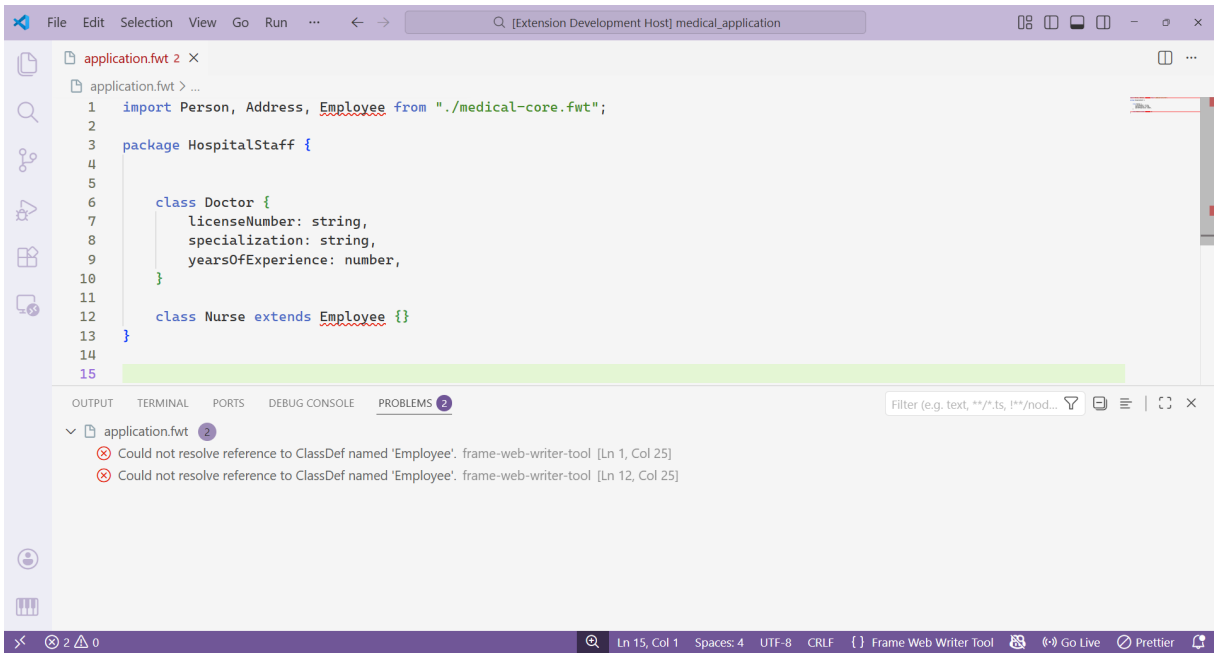

Figura 20 – Verificação de referências em FWT.

4 Conclusão

A ausência de ferramentas textuais de apoio ao método FrameWeb motivou o desenvolvimento deste trabalho, com o objetivo de ampliar as possibilidades de uso do método para além da modelagem exclusivamente visual. Embora já existissem iniciativas de suporte por meio de ferramentas baseadas em diagramas, ainda não havia uma solução voltada à modelagem textual integrada a um ambiente moderno de desenvolvimento.

Como resultado, este Trabalho de Conclusão de Curso teve como principal produto a extensão FrameWeb Writer Tool (FWT), desenvolvido com o uso do *Langium* e disponibilizado como extensão do Visual Studio Code, permitindo a modelagem textual alinhada ao método FrameWeb e integrada ao ecossistema de desenvolvimento contemporâneo.

A solução proposta integra validações semânticas do método FrameWeb e a geração de diagramas visuais a partir dos modelos criados. Contribuindo em expandir o uso do método FrameWeb a outras ferramentas de desenvolvimento.

Na sequência, discutem-se as contribuições deste trabalho, as dificuldades e limitações durante o desenvolvimento. Por fim, são propostas perspectivas para trabalhos futuros, visando a continuidade e o aprimoramento da solução apresentada.

4.1 Considerações Finais

Foi criada uma DSL denominda FrameWeb Writer Tool (**FWT**), a gramática da FWT tem como propósito capturar os aspectos de modelagem de sistemas utilizando o método FrameWeb. Vale ressaltar que devido ao modo como o FrameWeb foi criado (uma extensão leve da UML) a FWT permite modelar utilizando apenas UML puro enquanto cumpre o seu propósito de auxiliar a modelagem utilizando FrameWeb.

Além disso, a extensão FrameWeb Writer Tool (**FWT**) para o Visual Studio Code foi desenvolvida para fornecer suporte à DSL apresentada. A extensão interpreta e valida a gramática, incluindo validações semânticas específicas do FrameWeb, como a conformidade entre estereótipos de classes e pacotes, e gera automaticamente os diagramas correspondentes aos modelos definidos. Dessa forma, o projetista obtém tanto as vantagens de ferramentas textuais — como versionamento, refatoração e busca textual — quanto o benefício da visualização gráfica, que sintetiza informações complexas e facilita a compreensão da arquitetura do sistema.

4.2 Limitações e dificuldades

Uma das principais dificuldades encontradas durante o desenvolvimento deste trabalho esteve relacionada ao ambiente de desenvolvimento no sistema operacional Linux. A etapa de empacotamento da extensão não funcionava corretamente nesse ambiente, mesmo com a configuração padrão do Langium, o que resultou em um impacto significativo no cronograma do projeto. A solução foi a substituição da ferramenta de empacotamento por uma alternativa mais robusta e estável, com funcionamento consistente nos ambientes Windows, macOS e Linux.

O principal desafio em relação ao Langium foi a necessidade de compreender a fundo a arquitetura da ferramenta. Foi preciso estudar como seus componentes se organizam, interagem e qual o papel de cada um, além de explorar as possibilidades de extensão e customização. Apesar da documentação ser satisfatória, a falta de familiaridade inicial exigiu um estudo aprofundado do ciclo de vida e funcionamento da ferramenta, o que levou a um aprendizado detalhado de cada um de seus aspectos.

Outro desafio relevante esteve relacionado à necessidade de aprofundamento técnico no ecossistema de extensões do Visual Studio Code, especialmente na arquitetura baseada em um *Extension Host* (processo responsável por executar as extensões) e na comunicação via *Language Server Protocol* (LSP), protocolo utilizado para viabilizar recursos inteligentes de linguagem, como validação e autocompletar.

Os conhecimentos adquiridos durante a graduação foram essenciais para essa etapa, especialmente em função das disciplinas de Lógica, que forneceram base formal para a definição de regras e validações semânticas, e das disciplinas de Linguagens Formais e Compiladores, que contribuíram com os conceitos de gramáticas formais, *parsers*, máquinas de estados e analisadores sintáticos, e das disciplinas como Compiladores, Linguagens de Programação, Engenharia de Software, Orientação a Objetos e Estruturas de Dados, que contribuíram diretamente para a implementação da solução.

Em relação às limitações da solução proposta, observa-se que a geração automática dos diagramas visuais ainda apresenta restrições, como a sobreposição de elementos em situações específicas, especialmente quando uma classe referencia a si própria. Adicionalmente, a renderização padrão do Mermaid exibe o tipo do atributo antes do nome (`tipo: nome`), em desacordo com a convenção UML que posiciona o tipo após o nome (`nome: tipo`).

Além das limitações técnicas da ferramenta, algumas funcionalidades específicas do método FrameWeb não foram implementadas. Para o Modelo de Navegação o FrameWeb utiliza associações de dependência com restrições específicas para definir o fluxo de interação entre componentes da camada de apresentação. Quando uma dependência parte de uma página ou formulário em direção a uma classe de ação, ela representa, respectivamente, o

acionamento de um link ou a submissão de dados. Dependências que partem de classes de ação indicam os possíveis resultados da execução. O método utiliza restrições como `method=nome`, `result=nome` e `resultType=nome` para detalhar esse comportamento, além de `inMethod` e `outMethod` para coordenar encadeamento de ações (*chaining*). A **FWT** não implementa essas restrições específicas de navegação.

No Modelo de Entidades o FrameWeb define estereótipos para estratégias de mapeamento objeto-relacional (ORM) em relacionamentos de herança: «*single-table*» mapeia toda a hierarquia para uma única tabela; «*join*» cria uma tabela para cada classe e utiliza junções; e «union» cria uma tabela para cada classe concreta, unindo resultados via *SQL UNION*. Esses estereótipos não são suportados pela **FWT**.

4.3 Trabalhos Futuros

A partir do desenvolvimento deste projeto, identificam-se oportunidades de evolução que podem contribuir para que a **FWT** se consolide como uma solução textual para o método. Dessa forma, apresentam-se a seguir possíveis direcionamentos para trabalhos futuros:

- **Geração automática de código:** o Langium, por se tratar de uma ferramenta de engenharia de linguagens, oferece suporte nativo para a implementação de geradores de código, permitindo percorrer a árvore sintática abstrata (AST) e produzir artefatos em uma linguagem alvo. Dessa forma, os modelos definidos na **FWT** poderiam ser transformados, por exemplo, em código-fonte em linguagens como Java, gerando automaticamente classes, interfaces e estruturas de relacionamento a partir da modelagem. Essa funcionalidade é relevante pois aproxima a ferramenta de um fluxo completo de desenvolvimento dirigido por modelos, reduzindo esforço manual, aumentando a consistência entre modelo e implementação e alinhando a **FWT** a abordagens já presentes em trabalhos relacionados.
- **Publicação da FWT na loja oficial do Visual Studio Code:** atualmente, a ferramenta é distribuída por meio de um pacote no formato VSIX, sendo necessária a instalação manual pelo usuário. A disponibilização na loja permitiria a instalação direta pela própria interface do editor, além de oferecer suporte a atualizações automáticas, reduzindo o atrito de uso e melhorando a experiência do usuário. A publicação pode ser realizada por meio da criação de uma conta na plataforma Azure e do uso das ferramentas de empacotamento e submissão disponibilizadas pela própria Microsoft.
- **Implementação de funcionalidades faltantes do método FrameWeb:** atualmente, a **FWT** oferece suporte aos principais elementos dos quatro modelos do

método, porém algumas funcionalidades específicas ainda não foram implementadas. No Modelo de Navegação, isso inclui o suporte às restrições de dependência (`{method}`, `{result}`, `{resultType}`, `{inMethod}` e `{outMethod}`) que detalham o fluxo de interação entre componentes no Modelo de Navegação. No Modelo de Entidades, compreende a implementação dos estereótipos de estratégias de herança ORM (`«single-table»`, `«join»` e `«union»`), que orientam diferentes abordagens de mapeamento de hierarquias de classes para o banco de dados. A implementação dessas funcionalidades ampliaria significativamente a cobertura completa do método, permitindo a modelagem de sistemas mais complexos que demandam controle refinado de navegação e estratégias avançadas de persistência.

- **Migração para biblioteca de diagramação alternativa:** atualmente, a geração de diagramas é realizada por meio da biblioteca Mermaid, que apresenta algumas limitações que impactam a qualidade visual dos modelos. Entre essas limitações, destaca-se a sobreposição de elementos em situações específicas, como quando uma classe referencia a si própria, e a renderização padrão que exibe o tipo do atributo antes do nome (`tipo: nome`), em desacordo com a convenção UML que posiciona o tipo após o nome (`nome: tipo`). A migração para uma biblioteca de diagramação mais flexível, ou uma solução customizada baseada em SVG, permitiria maior controle sobre o posicionamento de elementos, a formatação de atributos e a personalização visual dos diagramas, melhorando significativamente a conformidade com os padrões UML e a experiência do usuário.

Referências

ALMEIDA, N. V. de; CAMPOS, S. L.; SOUZA, V. E. S. A model-driven approach for code generation for web-based information systems built with frameworks. *Proc. of the 23rd Brazilian Symposium on Multimedia and the Web (WebMedia 2017)*, 2017. Citado 2 vezes nas páginas 11 e 17.

ALUR, D.; MALKS, D.; CRUPI, J. Core j2ee patterns: Best practices and design strategies. 2001. Available on: <https://api.semanticscholar.org/CorpusID:109456385>. Citado na página 15.

AMBLER, S. Agile model driven development is good enough. *IEEE Software*, v. 20, n. 5, p. 71–73, 2003. Citado na página 19.

CAMPOS, S. L.; SOUZA, V. E. S. Frameweb editor: Uma ferramenta case para suporte ao método frameweb. *Anais do 16o Workshop de Ferramentas e Aplicações, 23o Simpósio Brasileiro de Sistemas Multimedia e Web (WFA/WebMedia 2017)*, 2017. Citado 2 vezes nas páginas 11 e 17.

DEURSEN, A. V.; KLINT, P. Little languages: little maintenance? *Journal of Software Maintenance: Research and Practice*, v. 10, n. 2, p. 75–92, 1998. Available on: <https://onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291096-908X%28199803/04%2910%3A2%3C75%3A%3AAID-SMR168%3E3.0.CO%3B2-5>. Citado na página 18.

EMBLEY, D. W.; LIDDLE, S. W.; PASTOR, O. Conceptual-model programming: a manifesto. *Handbook of Conceptual Modeling: Theory, Practice, and Research Challenges*, Springer, p. 3–16, 2011. Citado na página 20.

ENGELEN, L.; BRAND, M. van den. Integrating textual and graphical modelling languages. *Electronic Notes in Theoretical Computer Science*, Elsevier, v. 253, n. 7, p. 105–120, 2010. Citado 2 vezes nas páginas 11 e 12.

FOWLER, M. Patterns of enterprise application architecture. *Addison-Wesley, 1 edn*, 2002. Citado 3 vezes nas páginas 7, 14 e 15.

GALIN, D. Case tools and ides – impact on software quality. In: _____. *Software Quality: Concepts and Practice*. [S.l.: s.n.], 2018. p. 544–560. Citado na página 11.

GRÖNNINGER, H. *et al.* Textbased modeling. *arXiv preprint arXiv:1409.6623*, 2014. Citado 2 vezes nas páginas 11 e 12.

HAILPERN, B.; TARR, P. Model-driven development: The good, the bad, and the ugly. *IBM SYSTEMS JOURNAL*, 2006. Citado 2 vezes nas páginas 18 e 19.

HOPPE, P. H. B.; SOUZA, V. E. a. S. Support for single page application frameworks on frameweb. In: *Proceedings of the 29th Brazilian Symposium on Multimedia and the Web*. New York, NY, USA: Association for Computing Machinery, 2023. (WebMedia '23), p. 260–268. ISBN 9798400709081. Available on: <https://doi.org/10.1145/3617023.3617059>. Citado 2 vezes nas páginas 7 e 32.

KLEPPE, A. G.; WARMER, J. B.; BAST, W. *MDA explained: the model driven architecture: practice and promise*. [S.l.]: Addison-Wesley Professional, 2003. Citado 2 vezes nas páginas 18 e 20.

MERNIK, M.; HEERING, J.; SLOANE, A. M. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, ACM New York, NY, USA, v. 37, n. 4, p. 316–344, 2005. Citado 2 vezes nas páginas 18 e 20.

MURUGESAN, S. *et al.* Web engineering: A new discipline for development of web-based systems. *Web engineering: Managing diversity and complexity of Web application development*, Springer, p. 3–13, 2001. Citado na página 11.

PRADO, R. C. do; SOUZA, V. E. S. Securing frameweb: Supporting role-based access control in a framework-based design method for web engineering. *24th Brazilian Symposium on Multimedia and the Web, Salvador, BA, Brazil, ACM, 2018*, 2018. Citado na página 18.

PRESSMAN, R.; LOWE, D. *Web Engineering: A practioner's approach*. [S.l.]: McGraw-Hill, Inc., 2008. Citado 2 vezes nas páginas 10 e 11.

SILVA, I. S. e. Desenvolvimento de plugin para o visual paradigm com suporte ao método frameweb. *UFES*, 2023. Citado 5 vezes nas páginas 7, 18, 34, 38 e 39.

SOUZA, B. F. M. Evolução do método frameweb para o projeto de sistemas de informação web utilizando uma abordagem dirigida a modelos. *UFES*, 2016. Citado 2 vezes nas páginas 7 e 17.

SOUZA, V. E. S. Frameweb: um método baseado em frameworks para o projeto de sistemas de informação web. *UFES*, 2007. Citado 6 vezes nas páginas 12, 14, 31, 33, 34 e 36.

SOUZA, V. E. S. The frameweb approach to web engineering: Past, present and future. *UFES*, 2019. Citado 3 vezes nas páginas 10, 15 e 16.

THOMAS, D. Mda: revenge of the modelers or uml utopia? *IEEE Software*, v. 21, n. 3, p. 15–17, 2004. Citado na página 19.

TRASK, B.; ROMAN, A. Using domain specific modeling in developing software defined radio components and applications. In: *ECOOP Workshop on Domain-Specific Program Development (DSPD), Nantes, France*. [S.l.: s.n.], 2006. Citado na página 18.

Apêndices

APÊNDICE A – Gramática

Este apendice apresenta a sintaxe concreta da linguagem usando a gramática do Langium[1], similar à EBNF.

Listagem A.1 – Código fonte da gramática da Linguagem

```
grammar FrameWebWriterTool

entry Program:
    (imports+=FileImport)*
    (stmts+=Stmt)*
    ;

Stmt: packageDeclaration=PackageDeclaration | model=Model;

FileImport:
    'import' (imports+=ImportSpec (',' personImports+=ImportSpec)*) 'from' file=
        STRING ';';

ImportSpec:
    className=[ClassDef:ID] ('as' alias=ID)?;

PackageType returns string:
    'controller'    |
    'service'       |
    'domain'        |
    'view'          |
    'persistence';

PackageDeclaration:
    ('@' pType=PackageType)?
    'package' name=QualifiedName '{'
        (classes+=ClassDef | relations+=RelationDefinition | interfaces+=
            Interface)*
    '}';

ClassStereotype returns string:
    'transient' | 'mapped'| 'persistent'
    | 'page' | 'form' | 'binary'
;

AttributeStereotype returns string:
    'transient' | 'id' |'persistent' | 'version' | 'lob' | 'embedded';

Model:
    (Page | ClassDef | RelationDefinition | Interface);

Page:
    'page' name=ID;

```

[1] Gramática do langium

```
43 ClassDef:
44     ('@' stereotype=ClassStereotype)?
45     'class' name=ID
46     ('extends' superClass=[ClassDef:ID])?
47     ('implements' interfaces+=[Interface:ID] (',' interfaces+=[Interface:ID])*)?
48     '{' attributes+=Attribute* methods+=Method* '}' ;
49
50 Interface:
51     'interface' name=ID ('extends' implements=[ClassDef: ID] )? '{' methods+=
           Method* '}' ;
52
53 Attribute:
54     ('@' stereotype=AttributeStereotype)?
55     name=ID ':'
56     type=TypeRef ( '{' constraints+=AttributeConstraint (',' constraints+=
           AttributeConstraint)* '}')? ',';
57
58 Method:
59     name=ID  '(' ( parameters+=ID (',' parameters+=ID)* )? ')'
60     ':'
61     type=TypeRef ',';
62
63 TypeRef:
64     typeName=[ClassDef:QualifiedName];
65
66 RelationDefinition:
67     relationType=RelationType name=ID? '{'
68         block=RelationBlock?
69     '}';
70
71 RelationBlock:
72     (relations+=Relation)+;
73
74 Relation:
75   from=CustomType
76     cardinalityFrom=Cardinality?
77     ('->' | '--')
78     cardinalityTo=Cardinality?
79     to=CustomType
80     ;
81
82 Cardinality:
83     (('[' start=(INT|'*'|'n'|'m') '..' end=(INT|'*'|'n'|'m') ']') | ('[' self=(
           INT|'*'|'n'|'m') ']'))
84     ;
85
86 RelationType:
87     associationType=('association' | 'aggregation' | 'composition' | 'dependency'
           );
88
89 CustomType:
90     type=[ClassDef : QualifiedName];
91
92 AttributeConstraint:
93     name = 'size' '=' value=INT
94     | name = 'precision' '=' value = ('date' | 'time' | 'timestamp')
95     | name = 'generation' '=' value=('auto'|'identity'|'none'|'sequence')
```

```
96      | name='nullabe'
97      | name = 'column' '=' value=ID
98      | name=ID (operator=('='|'>='|'<='|'>'|'<'|'!=') value=ID)?
99      ;
100
101 QualifiedName returns string:
102     ID ('.' ID)*;
103
104 hidden terminal WS: /\s+/;
105 terminal ID: /[_a-zA-Z][\w_]*/;
106 terminal INT returns string: /[0-9]+/;
107 terminal STRING: /"(\\.|[^"\\])*"|'(\\.|[^'\\])*'/;
108 hidden terminal ML_COMMENT: /\/\*[\s\S]*?\*\//;
109 hidden terminal SL_COMMENT: /\/\/[^\n\r]*/;
```