UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

CENTRO TECNOLÓGICO

COLEGIADO DO CURSO DE CIÊNCIA DA COMPUTAÇÃO

Igor Sunderhus e Silva

Desenvolvimento de plugin para o Visual Paradigm com suporte ao método FrameWeb

Vitória, ES

2023

Igor Sunderhus e Silva

Desenvolvimento de plugin para o Visual Paradigm 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. Dr. Vítor Estêvão Silva Souza

Vitória, ES

2023

Igor Sunderhus e Silva

Desenvolvimento de plugin para o Visual Paradigm 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, 16 de novembro de 2023:

Prof. Dr. Vítor Estêvão Silva Souza
Orientador

Prof. André Georghton Cardoso Pacheco
Universidade Federal do Espírito Santo

Pedro Henrique Brunoro Hoppe
Universidade Federal do Espírito Santo

Vitória, ES
2023

Dedico esse trabalho aos meus pais, que me deram educação e apoio durante toda a minha vida, seja pessoal, profissional ou acadêmica e a minha noiva que me ajudou a superar todas as noites em claro e me apoiou durante todo o desenvolvimento desse trabalho.

Agradecimentos

Ao meu orientador, Vítor E. Souza, pelos ensinamentos durante as aulas que incentivaram a escolher o assunto. E principalmente pela paciência e dedicação durante as revisões deste trabalho.

A todos os professores do curso de Ciência da Computação da UFES, que tanto me ensinaram durante esses anos, mostrando a importância de se ter uma boa base teórica e prática para se tornar um bom profissional.

Aos meus colegas de classe, com quem pude compartilhar conhecimentos, de Engenharia de Software, essenciais para o desenvolvimento desse trabalho.

Aos meus colegas de trabalho, que me cobriram durante as minhas ausências para me dedicar a esse trabalho.

"Uma máquina consegue fazer o trabalho de 50 homens ordinários. Nenhuma máquina consegue fazer o trabalho de um homem extraordinário". Elbert Hubbard – escritor.

Resumo

A expansão da Internet como meio de comunicação e seus sistemas de informação fez necessária a adaptação de técnicas de Engenharia de *Software* para esse ambiente, a chamada Engenharia *Web*. Neste contexto, foi proposto *FrameWeb*, um método para o projeto de sistemas de informação *Web* que utilizam *frameworks*. O método define uma linguagem de modelagem baseada na UML, indicando a criação de alguns modelos específicos e a utilização de esteriótipos e restrições, com o objetivo de documentação e geração de código destes sistemas *Web*.

Este trabalho visa estender a ferramenta de modelagem UML *Visual Paradigm*, por meio de um *plugin* que ofereça suporte ao método *FrameWeb*, facilitando a construção de parte de seus modelos e proporcionando geração básica de código a partir dos modelos construídos. Os resultados deste trabalho abrem caminho para o desenvolvimento de um editor que atenda completamente o método *FrameWeb*.

Palavras-chaves: Engenharia *Web*, *FrameWeb*, *Visual Paradigm*, *plugin*, ferramenta CASE, geração de código, *templates*.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

UML	Unified Modeling Language (Linguagem de Modelagem Unificada)
SIW	Sistema de Informação Web
WWW	World Wide Web (Rede de Alcance Mundial)
IDE	Integrated Development Environment (Ambiente de Desenvolvimento Integrado)
API	Application Programming Interface (Interface de Programação de Aplicação)
TTM	Time to Market (Tempo de Mercado)
CASE	Computer-Aided Software Engineering (Engenharia de Software Assistida por Computador)
MVC	Model-View-Controller (Modelo-Visão-Controlador)
ORM	Object-Relational Mapping (Mapeamento Objeto-Relacional)
AOP	Aspect-Oriented Programming (Programação Orientada a Aspectos)
JSF	JavaServer Faces
SQL	Structured Query Language (Linguagem de Consulta Estruturada)
LDAP	Lightweight Directory Access Protocol (Protocolo de Acesso a Diretório Leve)
DAO	Data Access Object (Objeto de Acesso a Dados)
FTL	FreeMarker Template Language
LOB	Large Object (Objeto Grande)

Sumário

1 Introdução

Com a popularização da *World Wide Web* (WWW), a Internet se estabeleceu para comunicação de massa, abrindo possibilidades diversas que vão desde a simples publicação de conteúdo até a realização de transações comerciais. Nesse contexto, a *Web* se tornou um ambiente favorável ao desenvolvimento de Sistemas de Informação, conhecidos como Sistemas de Informação *Web* (SIWs), que utilizam essa plataforma para sua execução. Como resultado, o mercado de desenvolvimento de *software* passou a considerar o desenvolvimento de SIWs como uma atividade comum, o que gerou a necessidade de aplicar conceitos e técnicas de Engenharia de *Software* específicas para esse ambiente, visando garantir a qualidade dos sistemas desenvolvidos. Esse contexto de integração entre Engenharia de *Software* e *Web* é conhecido como Engenharia *Web* (PRESSMAN; LOWE, 2009).

Neste contexto, o método *FrameWeb* (SOUZA, 2007) define uma arquitetura padrão para facilitar a integração com *frameworks* muito utilizados para o desenvolvimento nessa plataforma, além de propor um conjunto de modelos baseados na UML que trazem para o projeto arquitetural do sistema conceitos inerentes a estes *frameworks*. Para apoiar o método, foi desenvolvido um *plugin* para a plataforma Eclipse que permitia a usuários de *FrameWeb* (por exemplo, alunos de graduação e pós-graduação na disciplina de Desenvolvimento *Web*) criar modelos segundo a linguagem *FrameWeb* e até mesmo gerar código-fonte a partir dos mesmos.

Enquanto a ferramenta foi um bom primeiro passo, foram encontrados vários problemas e limitações, muitos deles ligados ao fato de terem sido desenvolvidas dentro do ambiente Eclipse como, por exemplo, (i) desaparecimento das associações em alguns modelos, reduzindo drasticamente a confiabilidade do editor como ferramenta de documentação; (ii) dificuldade para manter os diagramas organizados, com pequenas movimentações de elementos no quadro acarretando uma completa desorganização dos diagramas; (iii) excesso de opções nas tabelas de configuração de elementos *FrameWeb*, sendo que a maioria das opções não são utilizadas ou possuem valor padrão contrário às necessidades mais comuns; (iv) dependência da IDE Eclipse e de um projeto Java *Web* para inicializar um projeto *FrameWeb*, o que acaba tornando o processo pouco prático para aqueles que não desejam utilizar a plataforma Java (SILVA, 2021).

Iniciou-se, então, um esforço para experimentar novas plataformas para as ferramentas de apoio ao *FrameWeb*, avaliando seu uso e comparando vantagens e desvantagens das diferentes plataformas. Considerando que uma ferramenta similar foi tema de um projeto iniciado em nosso grupo de pesquisa (SALES, 2018) e hoje é mantida por egressos

também deste grupo,[1] foi decidida a migração das ferramentas do *FrameWeb* para a mesma plataforma utilizada por ela, a saber, um *plugin* para o editor UML *Visual Paradigm*,[2] e seu desenvolvimento foi iniciado pelo orientador deste trabalho, que implementou parcialmente o suporte da ferramenta para um dos modelos propostos pelo *FrameWeb*.

Este trabalho tem como objetivo dar continuidade ao desenvolvimento do *plugin FrameWeb* para o *Visual Paradigm*, adicionando suporte completo a outros modelos propostos por *FrameWeb*, bem como permitindo a geração de código-fonte com base nestes modelos. Está previsto o desenvolvimento das seguintes funcionalidades:

- Complementação do modelo de Entidade;
- Geração de código-fonte baseado em *templates*;
- Disponibilização de *templates* básicos;
- Permitir que o usuário crie seus próprios *templates*;
- Disponibilizar a API com as informações fornecidas pelo *plugin*, permitindo que o usuário crie seus próprios *templates* utilizando esses dados;
- Desenvolvimento do modelo de Aplicação e Persistência.

Apesar do escopo do trabalho não incluir toda a linguagem *FrameWeb* (notoriamente, não inclui suporte ao Modelo de Navegação), seus resultados abrem caminho para o desenvolvimento de um editor que atenda completamente o método *FrameWeb* em trabalhos futuros.

1.1 Motivação e Justificativa

Trabalhando constantemente com o desenvolvimento de SIWs, percebeu-se que mesmo que os projetos sejam muito parecidos, sempre é realizada a mesma atividade de modelagem, e que muitas vezes não é formalizada. Em uma mesma organização, observamos cada área utilizar as suas próprias convenções e isso dificulta a comunicação, e assim a troca de informações e conhecimento.

Gerar uma ferramenta que consiga validar e padronizar essa arquitetura de forma que seja possível a geração de código-fonte, além de ser uma ferramenta que possa ser utilizada por qualquer pessoa que tenha conhecimento em UML, é um grande passo para a melhoria da qualidade dos projetos de SIWs. Neste quesito, o *plugin* para a plataforma do Eclipse, se mostrou um bom primeiro passo para a criação de uma ferramenta para auxiliar

[1] <https://github.com/OntoUML/ontouml-vp-plugin>
[2] <https://www.visual-paradigm.com/>

no desenvolvimento de SIWs que optem por adotar o *FrameWeb*. Na nova plataforma, a ferramenta *Visual Paradigm*, própria para modelagem UML, será possível criar modelos mais ricos e sem os problemas encontrados na plataforma Eclipse.

1.2 Objetivos

O objetivo principal deste trabalho é dar prosseguimento ao desenvolvimento do *plugin* do *FrameWeb* para o Visual Paradigm, permitindo que um desenvolvedor possa modelar e gerar código-fonte de um SIW utilizando o método *FrameWeb* de acordo com *templates* pré-definidos ou criados pelo próprio usuário.

São objetivos específicos:

Modelar Entidades

Complementar o modelo de Entidades iniciado pelo orientador do trabalho, utilizando como guia as tarefas documentadas pelo próprio orientador, que permanece como *stakeholder* no projeto, e as discussões realizadas no repositório do projeto no *GitHub*. São adicionados os estereótipos e restrições necessárias para que o modelo esteja completo e válido, segundo o método *FrameWeb*.

Gerar código-fonte

O modelo deve incluir todas as informações das classes necessárias para a geração de código, como as próprias classes, atributos e relações, além de estereótipos e restrições. O *plugin* então deve extrair essas informações e deixá-las em um formato comum utilizado pelos *templates* para a geração de código-fonte.

Criar templates

Como prova de conceito, são gerados alguns *templates* base para desenvolvimento com: (i) *JButler* (SOUZA, 2023), utilitário que fornece classes base que permitem o desenvolvimento de SIWs utilizando a pilha de tecnologias padrão do *Jakarta EE*; e (ii) *Spring Boot* (BOOT, 2023), um *framework* para desenvolvimento de aplicações em Java alternativo ao padrão *Jakarta EE*. Esses dois *templates* são suficientes para mostrar a capacidade do *plugin* de gerar código-fonte. Além disso, os mesmos são disponibilizados para que o usuário possa utilizar como base para a criação de seus próprios *templates*.

Permitir que o usuário crie seus próprios templates

Dado que nem todos utilizam os mesmos *frameworks*, deve ser possível selecionar um *template* próprio, i.e., uma pasta contendo os arquivos necessários para a geração de

código-fonte. O usuário pode criar seus próprios *templates* e disponibilizar para outros usuários, ou ainda utilizar os *templates* disponibilizados por outros usuários. Para que o *plugin* consiga ler os arquivos corretamente, uma API deve indicar quais serão os nomes de arquivos, nomes de atributos no mesmo e assim por diante.

Desenvolver modelo de Aplicação e Persistência

Desenvolver também os modelos de Aplicação e Persistência, de forma análoga ao desenvolvimento do modelo de Entidades. São adicionados os estereótipos e restrições necessárias para que o modelo seja válido com o método *FrameWeb*. Ao término do desenvolvimento do modelo, o *plugin* pode gerar código-fonte baseado no mesmo, utilizando novamente as mesmas técnicas utilizadas no modelo de Entidades.

1.3 Método de Desenvolvimento do Trabalho

Para alcançar os objetivos descritos anteriormente, foram realizadas as tarefas descritas na Tabela 1.

Atividade	**Descrição**
1	Estudo do Método *FrameWeb* (SOUZA, 2020) e do *plugin FrameWeb* para o *Visual Paradigm* (em desenvolvimento), além de experimentação do mesmo no contexto da disciplina de Programação *Web* ministrada pelo orientador.
2	Adição das extensões UML para o Modelo de Entidades para as quais o *plugin* ainda não provia suporte, adicionando as restrições e relacionamentos restantes.
3	Criação de uma forma de representação das informações dos modelos de classe para que tudo que seja necessário à geração de um esqueleto de código esteja disponível ao *plugin*.
4	Criação de *templates* básicos para geração de código.
5	Criação das extensões para o Modelo de Aplicação e Persistência com suas restrições e relacionamentos.
6	Elaboração e apresentação do texto da monografia

Tabela 1 – Atividades do trabalho

1.4 Organização da Monografia

Esta monografia foi particionada em 4 diferentes partes, incluindo este capítulo de Introdução. O Capítulo 2 traz informações que dizem respeito ao referencial teórico utilizado como base para a elaboração deste trabalho. O Capítulo 3 demonstra a contribuição deste trabalho, apresentando o *plugin* desenvolvido assim como sua utilização, demonstrando as extensões UML, os menus adicionados e a configuração de geração de

código-fonte. O Capítulo 4 apresenta as conclusões deste trabalho, apresentando as principais contribuições, limitações, lições aprendidas durante o desenvolvimento do trabalho, dificuldades enfrentadas e perspectivas de trabalhos futuros.

2 Fundamentação Teórica e Tecnologias Utilizadas

Neste capítulo apresentamos os conceitos e tecnologias utilizadas neste trabalho. A Seção 2.1 define os conceitos da Engenharia *Web*, na Seção 2.2 apresentamos quais *frameworks* são comumente utilizados no desenvolvimento de Sistemas de Informação *Web* (SIWs), a Seção 2.3 descreve o que são ferramentas CASE e seus usos, a Seção 2.4 apresenta o método *FrameWeb* e na Seção 2.5 apresentamos o motor de *templates* que utilizamos para geração de código, o *Apache FreeMarker*.

2.1 Engenharia Web

Ao expandir o uso de sistemas para a *Web*, encontramos desafios que não existem em sistemas *desktop* tradicionais. Sistemas agora são hospedados em servidores remotos ao usuário, com recursos compartilhados e imprevisibilidade de quantidade de acessos simultâneos. Isso gera um desafio quanto a dimensionamento, concorrência, segurança, disponibilidade, aparência e tempo para mercado (TTM), que deu origem a uma área de pesquisa chamada Engenharia *Web* (PRESSMAN; LOWE, 2009).

Nela é proposto um conjunto de atividades, parte de um *framework* genérico, ilustradas na Figura 1 e descritas abaixo:

- **Comunicação**: interação e colaboração com o cliente para entender o problema e definir requisitos;
- **Planejamento**: criação de um plano incremental que descreve as tarefas a serem feitas, riscos possíveis e estimativas de tempo e custo;
- **Modelagem**: criação de modelos que descrevem o sistema e suas funcionalidades, como diagramas de caso de uso, de classe, de sequência, etc.;
- **Construção**: implementação do sistema e testes;
- **Implantação**: entrega de um incremento ao cliente e avaliação do mesmo.

Antes de podermos desenvolver o sistema, passamos por várias atividades importantes para entender o problema e definir requisitos. Para isso, temos a fase de levantamento de requisitos junto ao cliente e partes interessadas, o que pode ser feito de várias formas, como entrevistas, questionários, *workshops*, etc., gerando assim o Documento de Definição

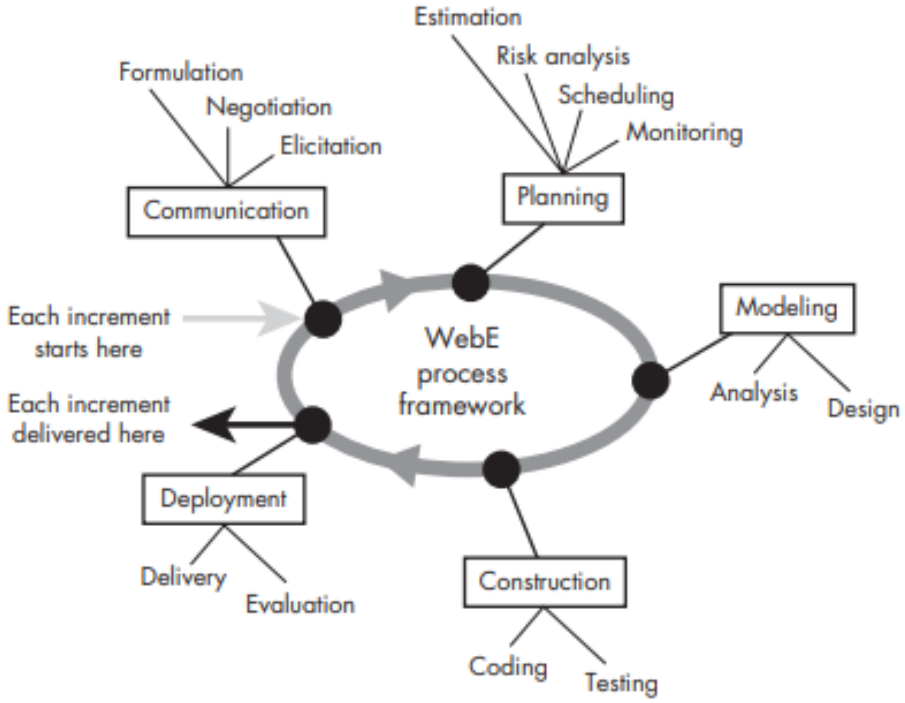

Figura 1 – Processo do *framework* de Engenharia *Web* (PRESSMAN; LOWE, 2009). Nele descrevemos as atividades que devem ser feitas para desenvolver um SIW.

de Requisitos (FALBO, 2023). Nessa fase que definimos as funcionalidades e restrições que o sistema deverá atender.

Após a fase de levantamento de requisitos, temos a fase de modelagem, em que criamos modelos que descrevem o sistema e suas funcionalidades. Esses modelos podem ser feitos de várias formas, como diagramas de caso de uso, de classe, de sequência, etc. Esses modelos são importantes para entender o sistema e suas funcionalidades, e também para auxiliar no projeto e construção do sistema. Essa fase se preocupa com o que será feito, ao invés de como. Ao fim dessa fase, temos o Documento de Especificação de Requisitos (FALBO, 2023).

Em posse dos requisitos e modelos gerados na fase anterior, podemos entrar na fase do Projeto do Sistema. Aqui temos como objetivo definir como será feito, agora considerando as limitações de *hardware* e *software* que existem, invés de supor que a tecnologia é perfeita como na fase de levantamento. Também nesta fase, definimos qual a arquitetura e quais *frameworks* (c.f. Seção 2.2) serão utilizados (FALBO, 2018).

De posse do projeto dos elementos da arquitetura, agora de fato começa a codificação do sistema, utilizando os *frameworks* necessários ao desenvolvimento. Durante todo o desenvolvimento, testes devem ser feitos para garantir que o sistema está funcionando corretamente. Caso o sistema se prove satisfatório, ele será então disponibilizado em um

servidor *Web* para todas as partes interessadas.

2.2 Frameworks

No desenvolvimento diário de SIWs, utilizamos extensivamente de *frameworks* para auxiliar e acelerar o desenvolvimento. Nesta seção, serão elencados algumas das categorias *frameworks* mais utilizadas no desenvolvimento de SIWs.

Ao desenvolver um SIW, encontramos várias atividades consideradas repetitivas e abstratas o suficiente que podem ser encapsuladas em componentes prontos para serem utilizados. Tais componentes são chamados *frameworks*, disponibilizando um esqueleto suficiente para que o desenvolvedor possa se preocupar primariamente com a lógica do negócio ao invés da infraestrutura do sistema.

As principais categorias de *frameworks* utilizados no desenvolvimento de SIW, segundo Souza (2007), são:

- MVC (Controladores Frontais);
- Decoradores;
- Mapeamento Objeto/Relacional (ORM);
- Injeção de Dependência (Inversão de Controle);
- Programação Orientada a Aspectos (AOP);
- Autenticação e Autorização.

2.2.1 MVC (Controladores Frontais)

Uma das arquiteturas para SIW mais difundidas é a MVC, ou *Model-View-Controller* (GAMMA et al., 1994), desenvolvida pelo Centro de Pesquisas da Xerox de Palo Alto (Xerox PARC) para a linguagem Smalltalk (REENSKAUG, 1979; SOUZA, 2007). Ela é composta por três componentes: *Model*, *View* e *Controller*, conforme ilustra a Figura 2.

O *Model* é responsável pela lógica de negócio, contendo as regras, os objetos e as operações que o sistema deve realizar. Também é responsável por acesso a fontes externas de dados como banco de dados, arquivos, etc.

A *View* é responsável pela estruturação e exibição dos dados ao usuário, nela os dados do Modelo são exibidos e todas as ações do usuário com a *interface* são tratadas. Em um SIW, a *View* é responsável por gerar o conteúdo HTML, CSS e *JavaScript* que será exibido no navegador do usuário.

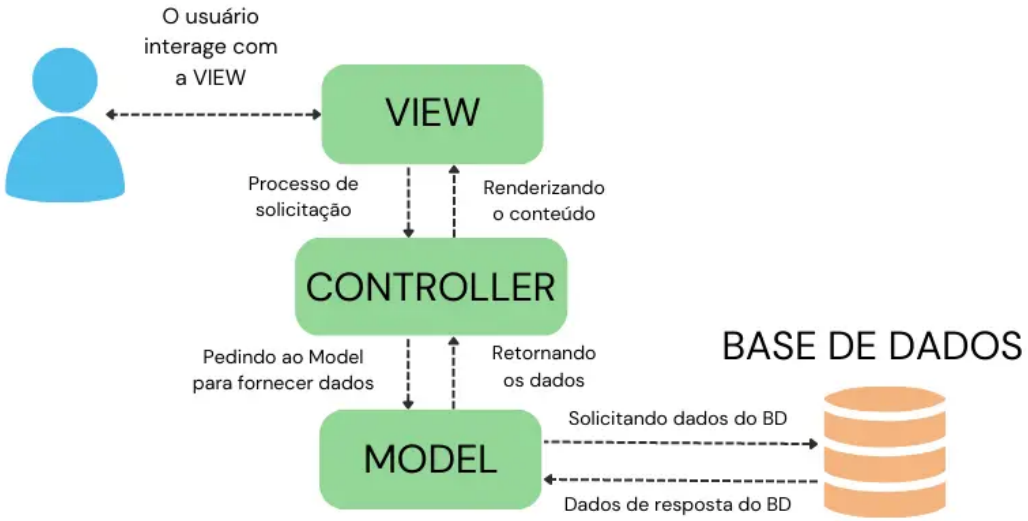

Figura 2 – Exemplificação de uma arquitetura MVC (Usando Python, 2023). As informações são passadas do usuário através da *View* para o *Controller*, que atualiza o *Model*. A *View* é atualizada com as informações do *Model*.

O *Controller* é responsável pela coordenação entre o *Model* e a *View*. As requisições do usuário feitas na *View* são monitoradas pelo *Controller* para que o mesmo consiga repassar os dados para o *Model*. Da mesma forma, o *Controller* recupera as informações do *Model* para atualizar a *View*.

Como exemplos de *frameworks* MVC, temos na plataforma C# o *ASP.NET MVC*, já na plataforma Java temos o *Spring MVC* e o JSF (*Jakarta Server Faces*).

2.2.2 Decoradores

A identidade visual de uma aplicação é muito importante e enquanto o conteúdo de uma página é dinâmico, as aplicações tendem a ter estruturas semi estáticas, como cabeçalhos, rodapés, menus, etc. Para facilitar a criação dessas estruturas, são utilizados decoradores, componentes que encapsulam a estrutura e permitem que o desenvolvedor se preocupe apenas com o conteúdo da página. Vale ressaltar que em alguns *frameworks* esses componentes são denominados *templates*, não confundindo assim como o conceito de Decoradores em Python, por exemplo.

Decoradores são *templates* que ditam a estrutura básica de uma página, mas reservam espaços para alguns tipos de conteúdo. A Figura 3 exemplifica um decorador utilizando JSF. Nele, incluímos um cabeçalho, um rodapé e o conteúdo da página. Podemos definir estilos e scripts no *template* e então todas as páginas podem utilizar desses estilos e scripts sem a necessidade de reescrever o código (TEAM, 2023).

Figura 3 – Exemplificação de um Decorador (MARCHIONI, 2022). O decorador define a estrutura básica da página, com cabeçalho e rodapé, enquanto o conteúdo é definido na página index.xhtml.

2.2.3 Mapeamento Objeto/Relacional (ORM)

Parte importante de um SIW é a persistência de dados, para isso utilizamos de bancos de dados relacionais em sua maioria (para bancos não relacionais outras técnicas de mapeamento podem ser utilizadas). Para facilitar a comunicação entre o banco de dados e o sistema, utilizamos dos chamados *frameworks* ORM (*Object/Relational Mapping*), para facilitar a comunicação dos diferentes paradigmas.

Em aplicações que utilizam o paradigma orientado a objetos, temos o domínio representado por classes, *interfaces*, todas interligadas através de vários relacionamentos. Já em bancos de dados relacionais, temos tabelas, colunas e linhas, onde as tabelas são relacionadas através de chaves estrangeiras. Além disso, a linguagem de consulta utilizada em bancos de dados relacionais é SQL, enquanto em sistemas orientados a objetos utilizamos de linguagens de programação como Java, C#, etc.

Para evitar que seja necessário criação manual de código para fazer a comunicação entre o banco de dados e o sistema, utilizamos de *frameworks* que utilizam os meta dados

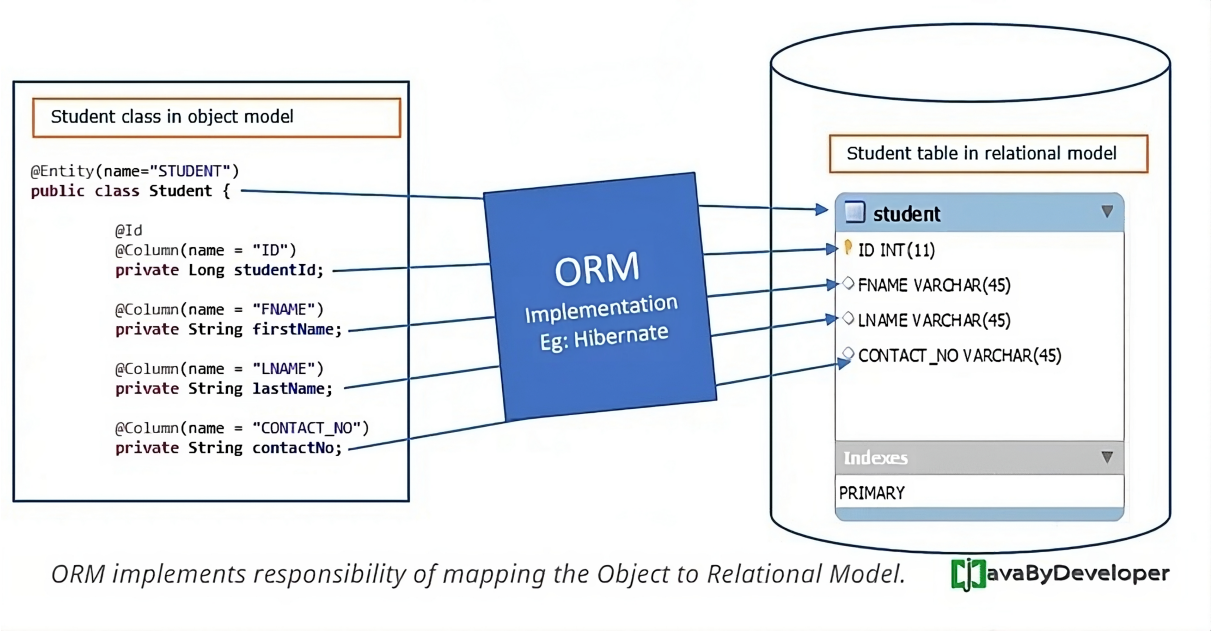

Figura 4 – Exemplificação de um Mapeamento Objeto/Relacional em Java utilizando de JPA (VARMA, 2023). Uma classe Student é mapeada para uma tabela student, onde os atributos são mapeados para as colunas baseado nas anotações.

de um objeto para traduzi-lo para uma linha de uma tabela e vice-versa.

Esses mapeamentos definem qual classe representa qual tabela, quais atributos representam quais colunas e quais relacionamentos existem entre as classes. Além disso, esses *frameworks* também permitem que o desenvolvedor utilize de linguagens de programação para fazer as consultas ao banco de dados, ao invés de utilizar SQL diretamente, disponibilizando assim uma camada de abstração entre o sistema e o banco de dados, através de métodos para ações comuns como recuperação, inserção, atualização e remoção de linhas da tabela.

Na Figura 4 temos a exemplificação de um mapeamento de uma classe Student para uma tabela student, utilizando Java e JPA (*Java Persistence API*). Nela conseguimos separar, por exemplo, o nome da coluna do nome da propriedade.

2.2.4 Injeção de Dependência (Inversão de Controle)

Quando um SIW evolui suas funcionalidades, a quantidade de classes e interdependências aumenta consideravelmente. A Injeção de Dependência, ou Inversão de Controle, pretende remover/diminuir o acoplamento de uma classe com a implementação de suas dependências. Para isso movemos a responsabilidade de instanciar as dependências para uma classe externa, no caso um *framework* de injeção de dependência.

Um exemplo seria de uma classe de serviço que necessita de acesso a um repositório, sendo que queremos abstrair os detalhes da implementação do repositório, estando interessados apenas na *interface* que ele provê. Para isso, ao invés de inicializarmos uma

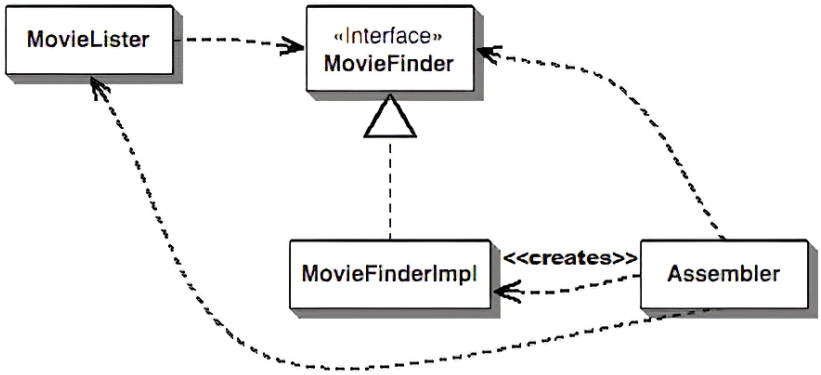

Figura 5 – Exemplificação de Injeção de Dependência (FOWLER, 2004). Ao invés do serviço instanciar uma implementação específica do repositório, ele requisita uma implementação da *interface* do repositório a algum *framework* de injeção de dependência.

implementação específica do repositório no serviço, requisitamos uma implementação da *interface* do repositório a algum *framework* de injeção de dependência. A Figura 5 exemplifica esse processo (FOWLER, 2004).

2.2.5 Programação Orientada a Aspectos (AOP)

Em sistemas desenvolvidos no paradigma da Orientação a Objetos, há muitos trechos de código que podem ser considerados repetitivos em vários métodos do sistema como, por exemplo, gerenciamento de transação, logs, segurança, etc. Na Figura 6, podemos observar esses trechos de código, conhecidos como *cross-cutting concerns*, pois são atividades executadas em vários módulos do sistema, não pertencendo a apenas um deles.

A Programação Orientada a Aspectos (AOP) visa aumentar a modularidade por meio da separação de *cross-cutting concerns*. Podemos separar esses trechos em aspectos implementados uma única vez e então entrelaçá-los (*weaving*) com os módulos do sistema que necessitam deles. Esse entrelaçamento pode ser feito por *frameworks* ou compiladores, que fazem a transformação do código-fonte para incluir os aspectos. O desenvolvedor, portanto, precisa apenas informar os pontos de código (*pointcuts*) onde isso deve acontecer (SOUZA, 2007).

A Figura 7 exemplifica como separar a responsabilidade do controle de transação do método de negócio para o *framework* AOP. O desenvolvedor apenas informa que o método de negócio necessita de controle de transação e o *framework* AOP se encarrega de adicionar o código necessário para isso.

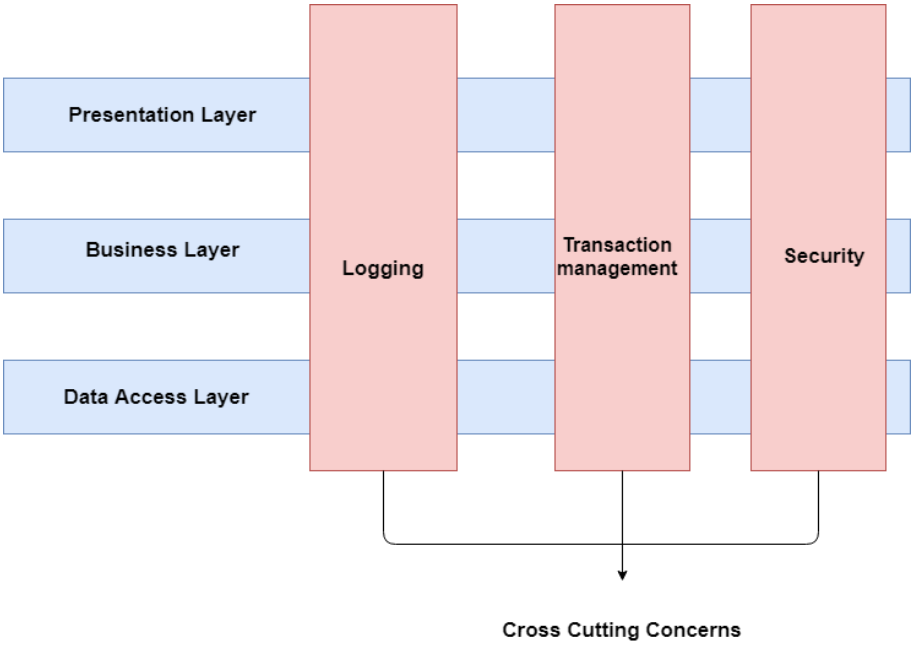

Figura 6 – Demonstração de *cross-cutting concerns* (FADATARE, 2023). *Logging*, controle de transação e segurança são atividades que podem ser consideradas *cross-cutting concerns*, não estando contidos em apenas um módulo do sistema.

2.2.6 Autenticação e Autorização

Em SIWs, autenticação e autorização são dois conceitos importantes. Autenticação é o processo de verificar se um usuário é quem ele diz ser, enquanto Autorização é o processo de verificar se um usuário tem permissão para acessar um recurso.

Em SIWs, a Autenticação é comumente feita por meio de um formulário de *login*, em que o usuário informa suas credenciais e o sistema verifica se as credenciais estão corretas. Já a Autorização é feita por um controle de acesso, em que o sistema verifica se o usuário tem permissão para acessar um recurso. Essa permissão pode ser concedida a grupos de usuários ou a usuários específicos. Podemos ter diferentes níveis de permissão, como leitura, escrita, etc., de modo que um usuário possa ter permissão de leitura em um recurso, mas não de escrita.

Existem diferentes formas de fazer a Autenticação e Autorização, como, por exemplo, por meio de um banco de dados, de um servidor LDAP, dentre outros. Temos o conceito de *allowlisting* e *denylisting*,[1] sendo que com *allowlisting* todos os recursos são bloqueados inicialmente e os recursos que o usuário tem permissão são liberados, enquanto com *denylisting* todos são liberados por padrão e apenas os que o usuário não tem permissão são bloqueados.

A Figura 8 exemplifica o processo em uma arquitetura *Web* utilizando o *framework* de autenticação *Keycloak*.

[1] Originalmente se usavam os termos *whiltelisting* e *blacklisting*, porém eles foram substituídos neste texto por termos mais inclusivos e neutros.

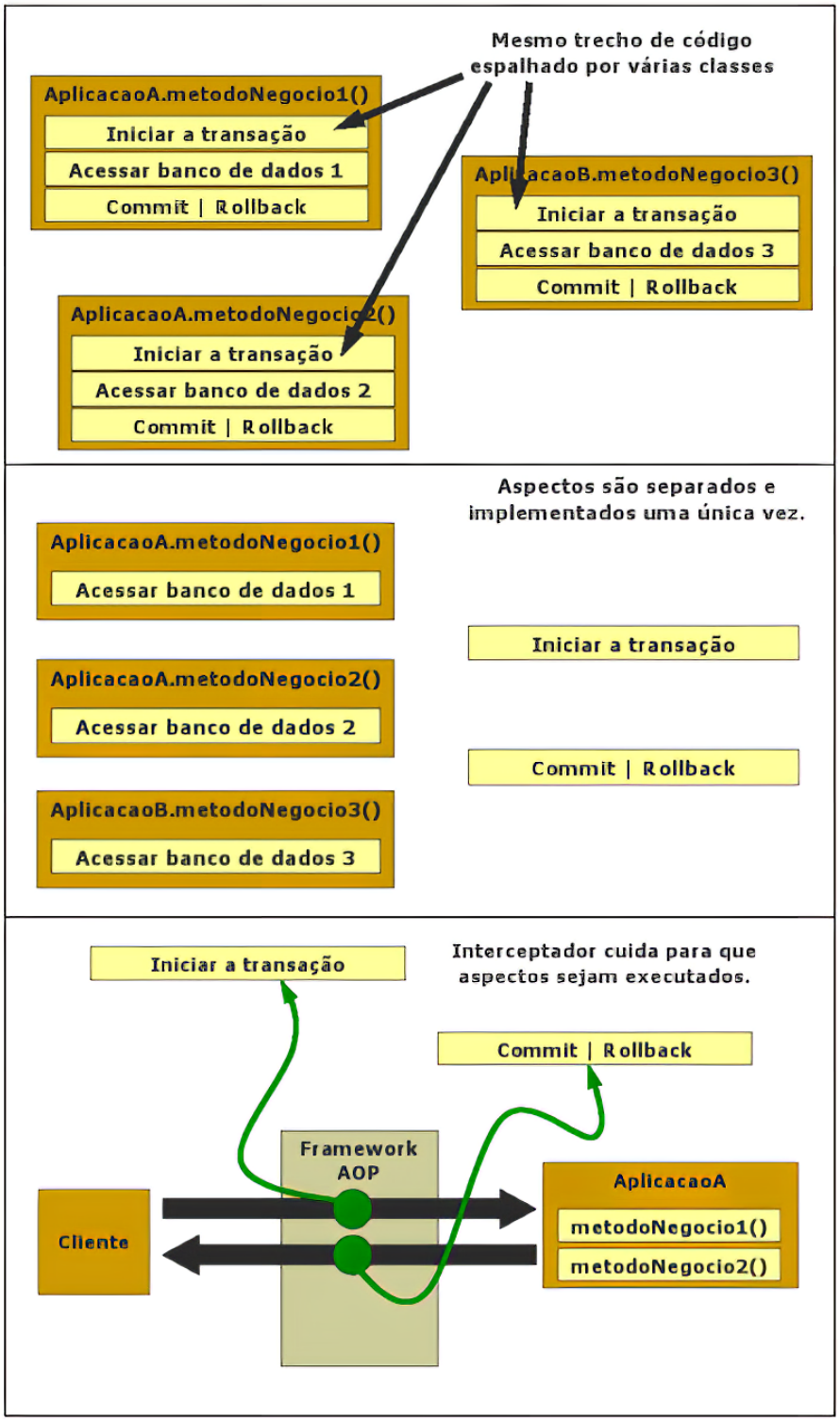

Figura 7 – Funcionamento de um Framework AOP (SOUZA, 2007). Ao invés de definir chamar métodos de controle de transação manualmente em cada um dos métodos de negócio, o desenvolvedor apenas informa que o método de negócio necessita de controle de transação (em alguns *frameworks* esse controle é dado, mas em outros é necessário a implementação) e o *framework* AOP se encarrega de adicionar o código necessário para isso.

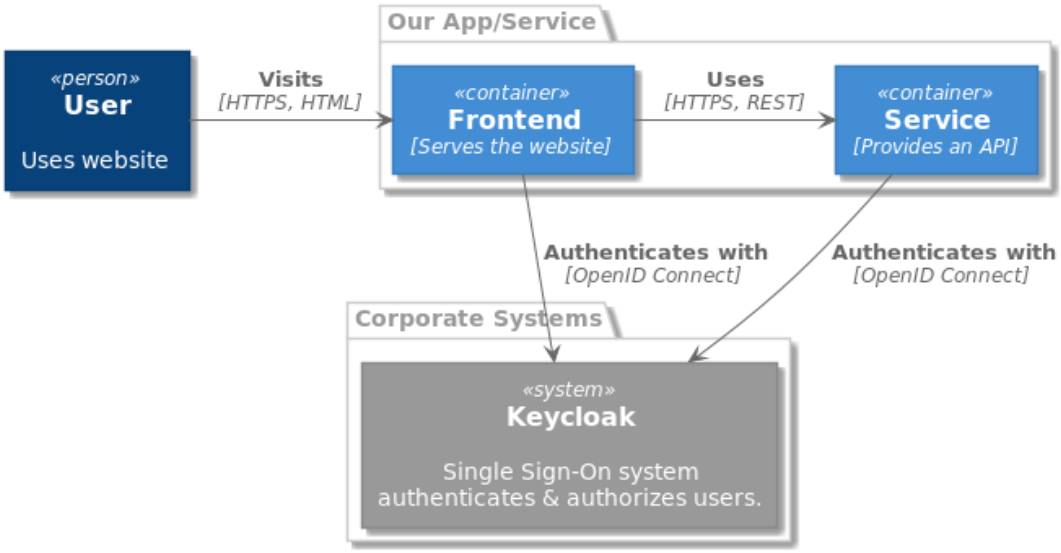

Figura 8 – Usuário autenticando em um sistema utilizando do framework *Keycloak* (WEENEN, 2018). O usuário informa suas credenciais no formulário de *login* e o sistema verifica se as credenciais estão corretas. Caso estejam, o usuário é autenticado e o sistema verifica se o usuário tem permissão para acessar o recurso. Caso tenha, o usuário é autorizado e o recurso é exibido.

2.3 Ferramentas CASE

Ferramentas CASE (*Computer Aided Software Engineering*) são ferramentas que auxiliam no desenvolvimento de *software*, automatizando tarefas repetitivas e abstratas. Essas ferramentas podem ser utilizadas para auxiliar em várias atividades do desenvolvimento de *software*.

Shiangjen (2023) divide Ferramentas CASE em três categorias, conforme demonstrado visualmente na Figura 9:

- *Upper CASE*: ferramentas utilizadas para auxiliar nas fases iniciais, como levantamento de requisitos, modelagem, projeto, etc.;
- *Lower CASE*: ferramentas utilizadas para codificação, testes, manutenção, etc.;
- *Integrated CASE*: produtos completos que auxiliam em todas as fases do desenvolvimento de *software*.

No dia a dia, várias ferramentas CASE são utilizadas, como IDEs (*Integrated Development Environment*), editores de texto, editores de diagramas, controle de versão de código, até mesmo compiladores e analisadores estáticos podem ser considerados ferramentas CASE.

Neste trabalho, estenderemos a Ferramenta CASE *Visual Paradigm* por meio do seu suporte a *plugins* para auxiliar na modelagem de SIWs utilizando do método *FrameWeb*. A

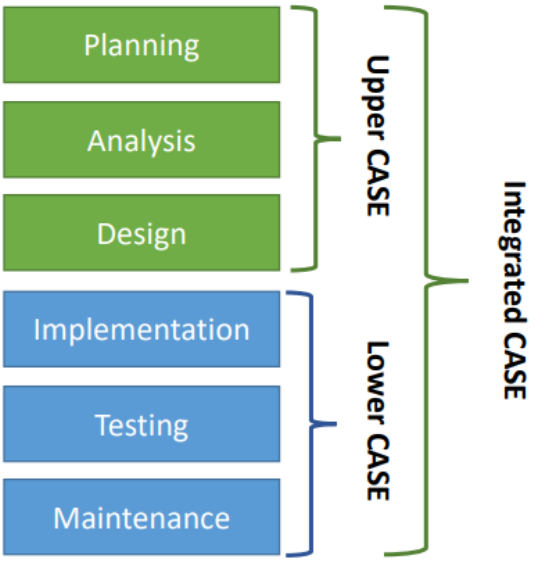

Figura 9 – Categorias de Ferramenta CASE (SHIANGJEN, 2023).

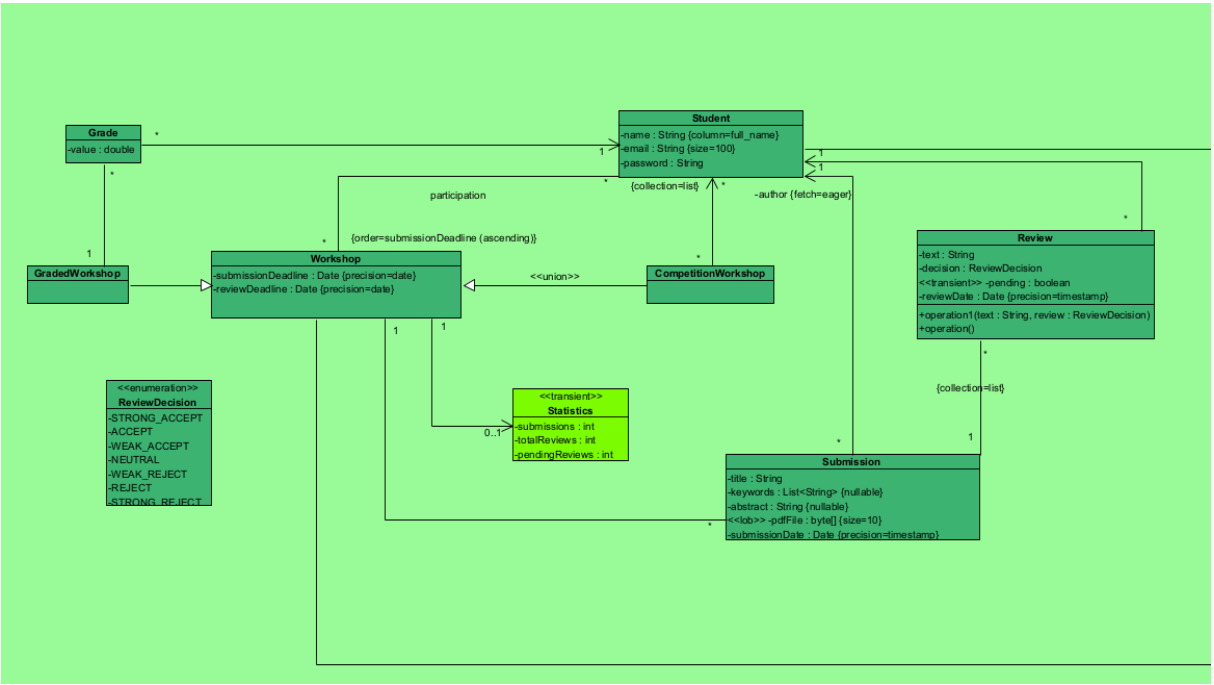

Figura 10 – *Visual Paradigm*. Na imagem temos um diagrama de classes para o modelo de entidades do método *FrameWeb* e o *plugin* desenvolvido neste trabalho.

Figura 10 exemplifica a ferramenta *Visual Paradigm*, onde temos um diagrama de classes para o modelo de entidades do método *FrameWeb* e o *plugin* desenvolvido neste trabalho.

2.4 FrameWeb

O *FrameWeb* (SOUZA, 2020) é um método de desenvolvimento de Sistemas de Informação *Web* (SIW). Nele são definidas extensões da UML de modo a representar

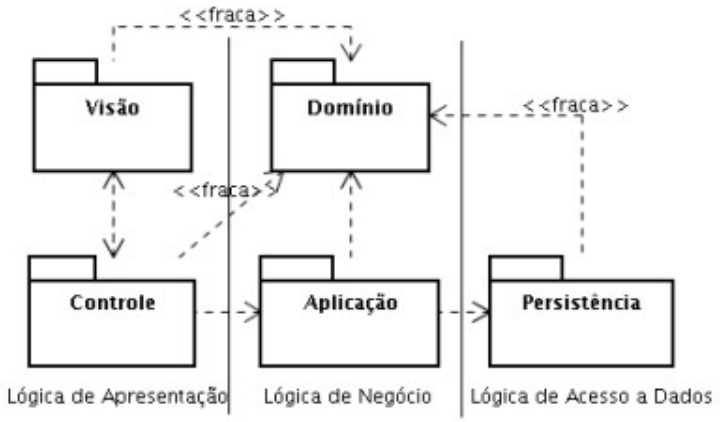

Figura 11 – Arquitetura do *FrameWeb* (SOUZA, 2007). O *FrameWeb* é baseado no padrão arquitetural Camada de Serviço, dividindo as lógicas de Apresentação, Negócio e Acesso a Dados em diferentes camadas.

componentes comuns ao desenvolvimento *Web*. São definidos quatro categorias de diagramas, chamados de Modelos, a saber: Entidades; Persistência; Aplicação; e Navegação. Baseando-se no padrão arquitetural, Camada de Serviço (FOWLER, 2002), esses modelos são separados em três camadas, conforme ilustra a Figura 11.

A Camada de Apresentação, representada em modelos de Navegação, contém o pacote de Visão e Controle. O Pacote de Visão contém todo o conteúdo voltado para a exibição em navegadores, estruturas HTML, folhas de estilo (CSS), *scripts* de lado de cliente (JavaScript ou similares), além do conteúdo multimídia. O Pacote de Controle contém as classes responsáveis por receber as requisições do usuário e encaminhar para a camada de negócio.

A segunda é chamada Camada de Negócio, representada nos modelos de Entidades e Aplicação. O Modelo de Entidades contém o domínio do problema, ilustrando todas as entidades e seus mapeamentos para o banco de dados. O Modelo de Aplicação contém as classes responsáveis por implementar as regras de negócio do sistema (classes de serviço), representando também sua comunicação com as camadas de apresentação e acesso a dados.

Por fim temos a Camada de Acesso a Dados, representada nos modelos de Persistência. Geralmente utiliza-se nesta camada o padrão *Data Access Object* (DAO) (ALUR; CRUPI; MALKS, 2003) para fazer a persistência das informações com o banco de dados.

Todas as camadas utilizam das entidades para fazer a comunicação entre si, sendo que a camada de apresentação utiliza das entidades para exibir as informações ao usuário, a camada de negócios utiliza as mesmas para implementar as regras de negócio e a de persistência gerencia a persistência das informações no banco de dados.

O pacote de Controle utiliza o pacote de Aplicação para ter acesso a todas as regras de negócio. O pacote de Aplicação utiliza o pacote de Persistência para ter acesso

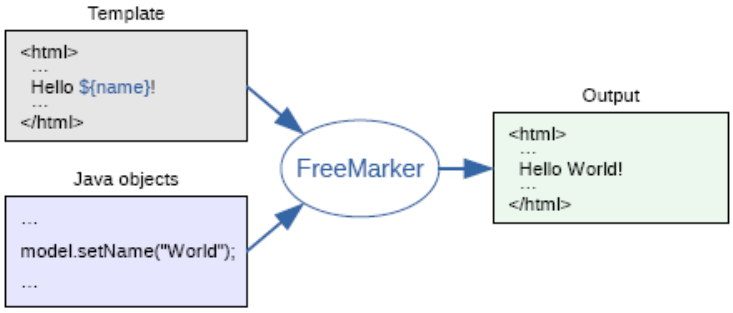

Figura 12 – Funcionamento de um *template Apache FreeMarker* (FOUNDATION, 2023). Um programa gera um modelo de dados como uma propriedade *name*, no modelo, utilizando a FTL, inserimos essa propriedade dentro de uma página HTML que cumprimenta o usuário. O motor então gera o HTML substituindo a propriedade *name* pelo valor passado no modelo de dados.

ao banco de dados onde as Entidades serão persistidas.

2.5 *Templates Apache FreeMarker*

Apache FreeMarker é um motor de *templates* para a plataforma Java, que permite a criação de *templates* para geração de texto (HTML, código-fonte, e-mails, etc.). Os *templates* são escritos na linguagem criada pela Apache chamada *FreeMarker Template Language* (FTL), uma linguagem simples e especializada. O *template* espera receber um modelo de dados a partir de uma linguagem mais completa, como Java, e então gera o texto segundo o *template*.

Na Figura 12 temos uma exemplificação de como funciona o *FreeMarker*. Um programa gera um modelo de dados como uma propriedade *name*, no modelo, utilizando a FTL, utilizamos essa propriedade dentro de uma página HTML que cumprimenta o usuário. O motor então gera o HTML substituindo a propriedade *name* pelo valor passado no modelo de dados (FOUNDATION, 2023).

3 Funcionalidades do *Plugin*

Este capítulo se destina a demonstrar as funcionalidades expostas pelo *plugin* desenvolvido neste trabalho. A Seção 3.1 mostra as extensões do *FrameWeb* adicionadas ao *Visual Paradigm* e a Seção 3.2 mostra a *interface* de geração de código do *plugin*. O código-fonte do *plugin* está disponível em <https://github.com/nemo-ufes/frameweb-vp-plugin>, juntamente com as instruções de instalação e contribuição.

3.1 Extensões FrameWeb adicionadas

Como um dos objetivos deste trabalho (cf. Seção 1.2) temos a extensão do *Visual Paradigm*, facilitando a inclusão dos estereótipos e restrições previstos na linguagem *FrameWeb*, para classes, atributos e associações. Os Modelos de Entidade, Persistência e Aplicação tiveram essas extensões adicionadas.

O *plugin* funciona definindo estereótipos que, ao serem aplicados a um determinado nível, restringem quais estereótipos podem ser aplicados no próximo nível. Por exemplo, ao aplicar o estereótipo *«Entity Package»* em um pacote, por meio do menu visualizado na Figura 13, o estereótipo *«Persistent Class»* fica disponível para ser aplicado em suas classes, sendo o valor padrão.

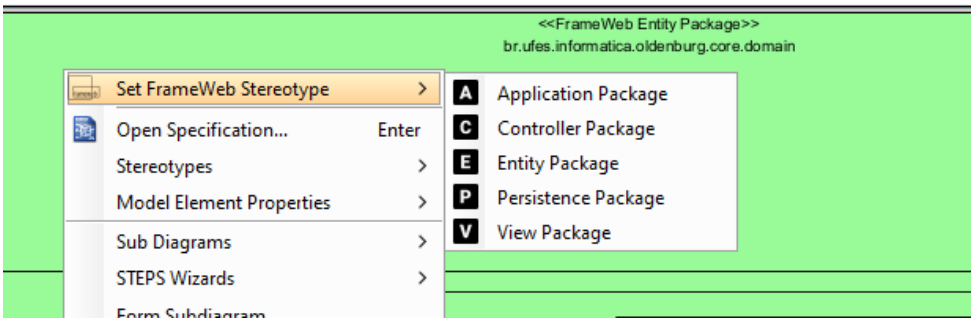

Figura 13 – Menu de contexto para aplicação de estereótipos em pacotes.

Nas próximas seções exploramos os novos menus de contexto e as novas opções de estereótipos e restrições incluídas no *plugin*. Utilizamos como base o SIW *Oldenburg*, utilizado por Souza (2020) para ilustrar a linguagem *FrameWeb*. O exemplo foi expandido para fins de ilustração das funcionalidades do *plugin* desenvolvido.

3.1.1 Modelo de Entidades

Este é o modelo mais completo e complexo, pois é o que possui mais estereótipos e restrições. A Figura 14 mostra o Modelo de Entidade do SIW *Oldenburg*.

No mesmo conseguimos visualizar algumas das características da linguagem de modelagem *FrameWeb*, tendo a aplicação dos estereótipos e restrições nos pacotes, classes, atributos e associações. Para criar um pacote de entidades começamos atribuindo o estereótipo *«Entity Package»*. As classes adicionadas e sem estereótipo explícito são então consideradas persistentes, atributos em classes persistentes também são por padrão persistentes. Quando não queremos persistir uma classe ou atributo, podemos aplicar o estereótipo *«Transient»*. Na imagem temos os estereótipos e restrições da linguagem *FrameWeb* aplicados em alguns elementos do modelo, como tamanho de atributo, anulabilidade de atributo, precisão de datas, cardinalidade de associação, propagação de alterações em associações, categoria de generalização, etc. Essas extensões têm atalhos introduzidos pelo *plugin* que também são demonstradas.

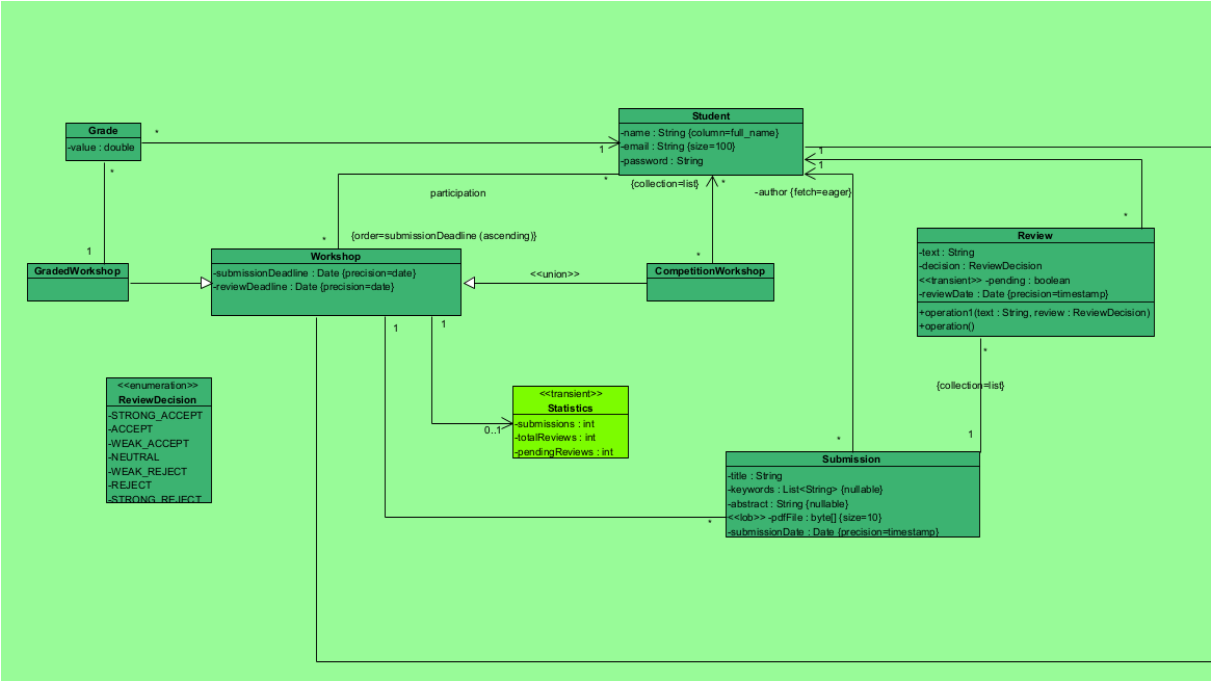

Figura 14 – Modelo de Entidades do exemplo do SIW *Oldenburg*.

Em uma classe do Modelo de Entidades conseguimos adicionar os estereótipos definidos pelo *FrameWeb*. As figuras 15, 16 e 17 mostram os menus de contextos para aplicação de estereótipos em classes, atributos e generalizações. Os estereótipos são disjuntos, ou seja, não é possível aplicar mais de um estereótipo em um mesmo elemento.

Nas figuras a seguir podemos ver os menus de contexto para aplicação de restrições em atributos e associações. A Figura 18 mostra o menu de contexto para aplicação de restrições em atributos e a Figura 19 mostra a entrada de dados para aplicação de restrições em atributos com parâmetros.

A Figura 20 mostra o menu de contexto para aplicação de restrições em associações. Algumas das restrições são disjuntas e outras não, ou seja, é possível aplicar mais de uma restrição em um mesmo elemento. As restrições de atributos de mesmo nome são disjuntas,

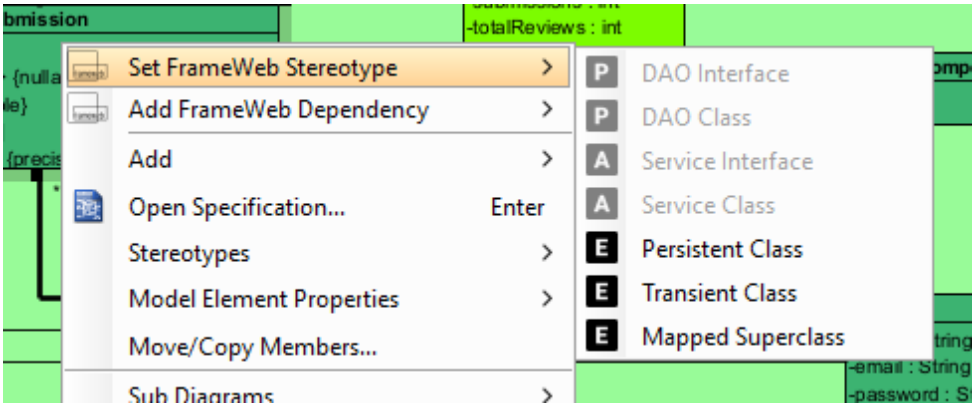

Figura 15 – Menu de contexto para aplicação de estereótipos em classes do Modelo de Entidades.

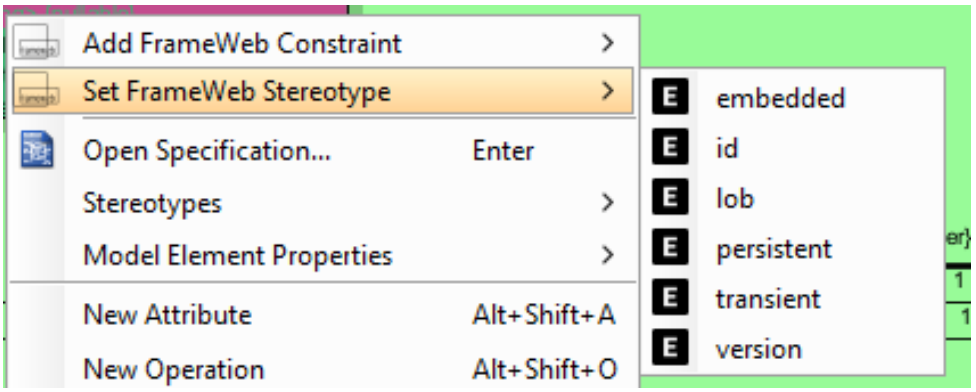

Figura 16 – Menu de contexto para aplicação de estereótipos em atributos do Modelo de Entidades.

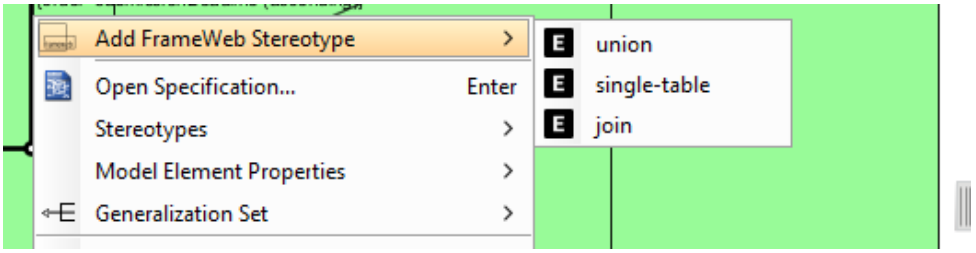

Figura 17 – Menu de contexto para aplicação de estereótipos em generalizações do Modelo de Entidades.

não sendo possível, por exemplo, dizer que um atributo é anulável e não anulável. Já é possível, no entanto, dizer que um campo com precisão date é também anulável.

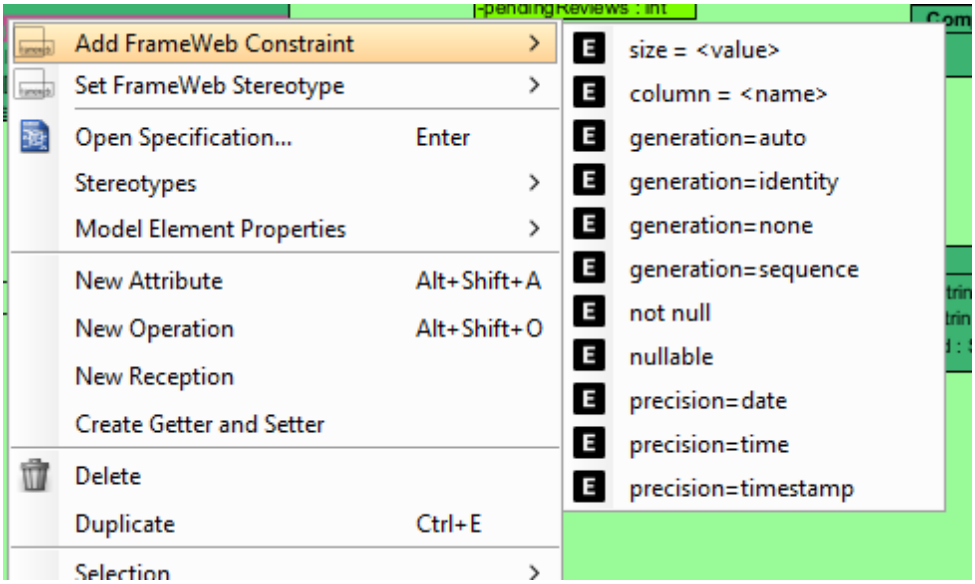

Figura 18 – Menu de contexto para aplicação de restrições em atributos do Modelo de Entidades.

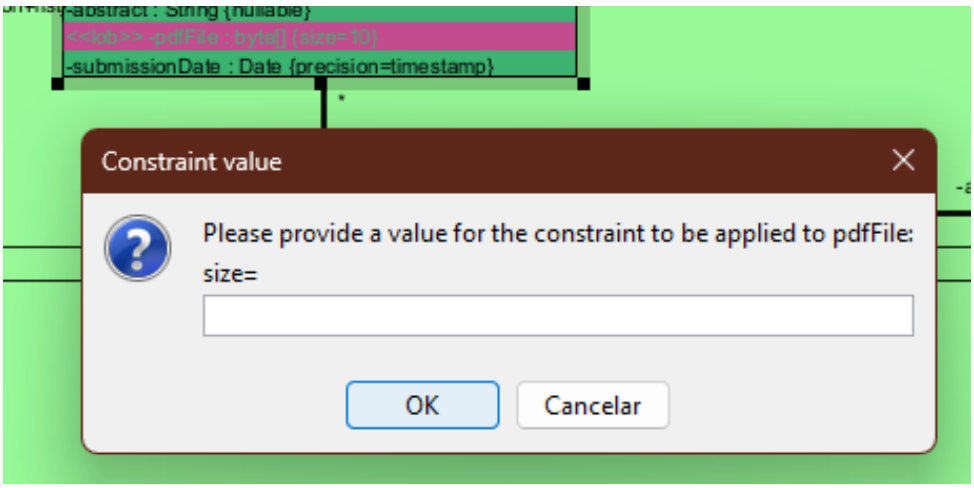

Figura 19 – Menu de contexto para aplicação de restrições em atributos com parâmetros no Modelo de Entidades.

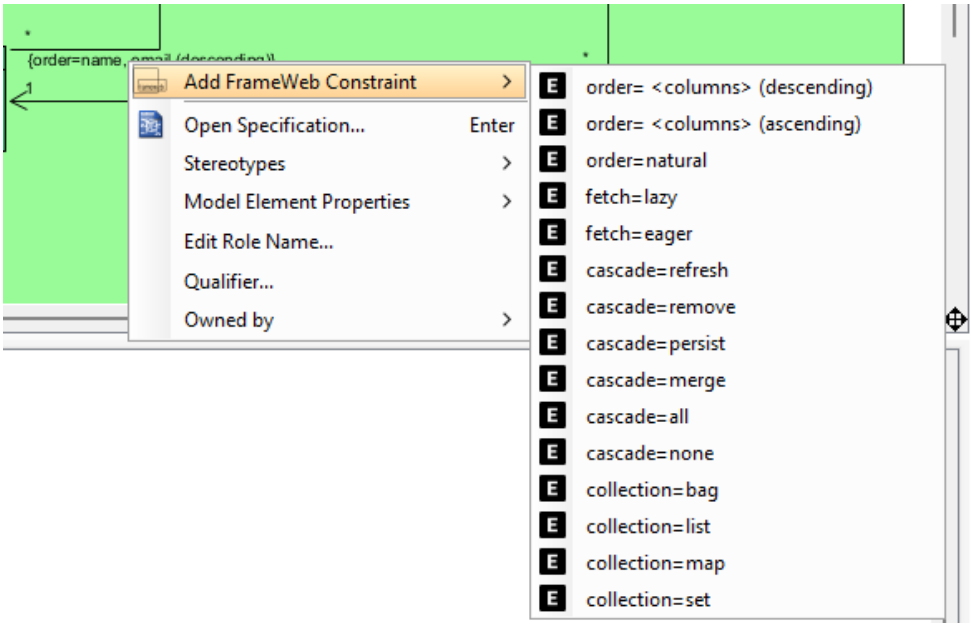

Figura 20 – Menu de contexto para aplicação de restrições em associações do Modelo de Entidades.

3.1.2 Modelo de Persistência

Já o Modelo de Persistência é mais simples, nele não temos extensões UML, apenas separamos em dois estereótipos, classes concretas e *interfaces*. Dessa forma podemos adicionar classes concretas para diferentes *ORM's*, por exemplo, de modo que abstraímos a implementação de persistência do Modelo de Entidades, mencionado na seção anterior, e o de aplicação, tema da próxima seção. A Figura 21 mostra o Modelo de Persistência do SIW *Oldenburg* e após os atalhos de aplicação.

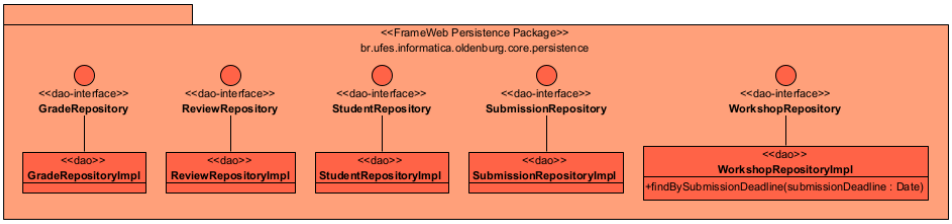

Figura 21 – Modelo de Persistência do exemplo do SIW *Oldenburg*.

Já a Figura 22 mostra o menu de contexto para aplicação de estereótipos em classes de persistência. Os estereótipos são disjuntos, ou seja, não é possível aplicar mais de um estereótipo em um mesmo elemento.

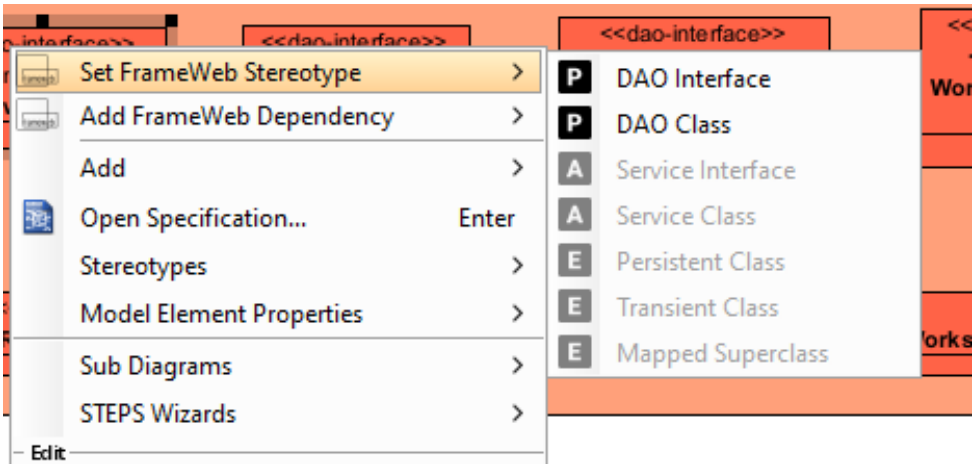

Figura 22 – Menu de contexto para aplicação de estereótipos em classes do Modelo de Persistência.

3.1.3 Modelo de Aplicação

No Modelo de Aplicação, também não temos extensões UML, apenas separamos em dois estereótipos, classes concretas e *interfaces*, assim como na Persistência. Permitindo a abstração das diferentes implementações para aproveitar os *frameworks* de injeção de dependência nas classes de Navegação, não tratadas neste trabalho. A Figura 23 mostra o modelo de aplicação do SIW *Oldenburg* e após os atalhos de introduzidos pelo *plugin*.

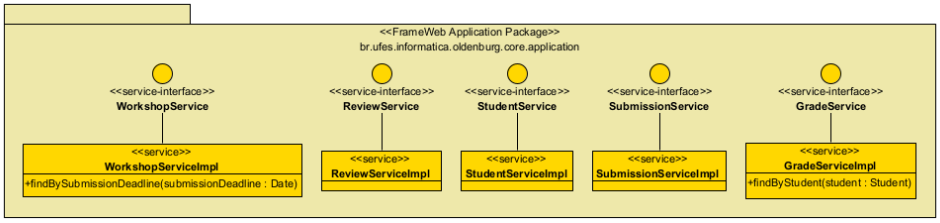

Figura 23 – Modelo de Aplicação do exemplo do SIW *Oldenburg*

Já a Figura 24 mostra o menu de contexto para aplicação de estereótipos em classes de aplicação. Os estereótipos são disjuntos, ou seja, não é possível aplicar mais de um estereótipo em um mesmo elemento.

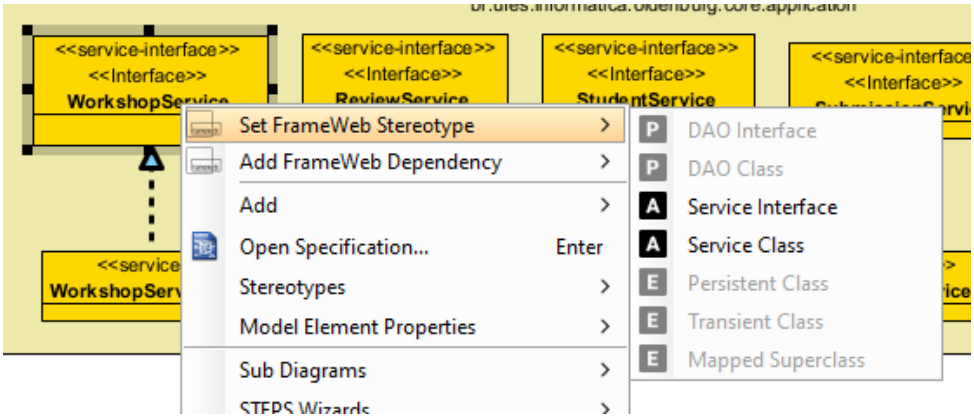

Figura 24 – Menu de contexto para aplicação de estereótipos em classes de aplicação

3.2 Geração de Código Fonte

Para os modelos com extensões adicionadas, também foi criada uma API para geração de código-fonte utilizando o *Apache Freemarker*. O Apêndice A descreve as informações disponibilizadas pelo plugin para geração de código e exemplos de códigos gerados. Nas próximas seções demonstraremos a geração de código para o SIW *Oldenburg*.

3.2.1 Seleção de *template*

Ao acessar as opções do plugin temos a opção de escolher um conjunto de *templates* para geração de código. A Figura 25 mostra a tela de seleção de *templates*. O plugin já vem com um conjunto de *templates* para geração de código para o SIW *Oldenburg*, para *JButler* e *Spring*, mas é possível adicionar novos *templates* para outros projetos *FrameWeb*.

3.2.2 Importação de novos *templates*

Para adicionar novos *templates*, temos uma tela onde é possível inserir nome, descrição, os caminhos de entrada do *template* e saída do código e os nomes definidos para

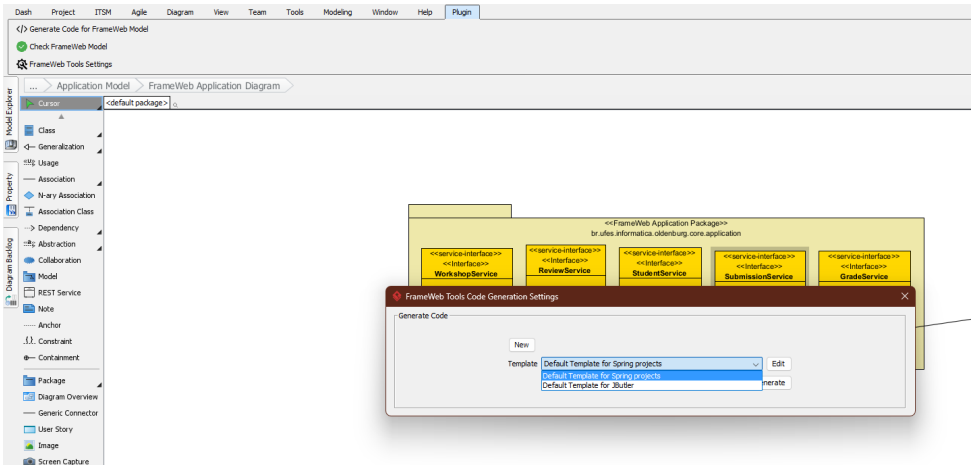

Figura 25 – Tela de seleção de *templates* para geração de código

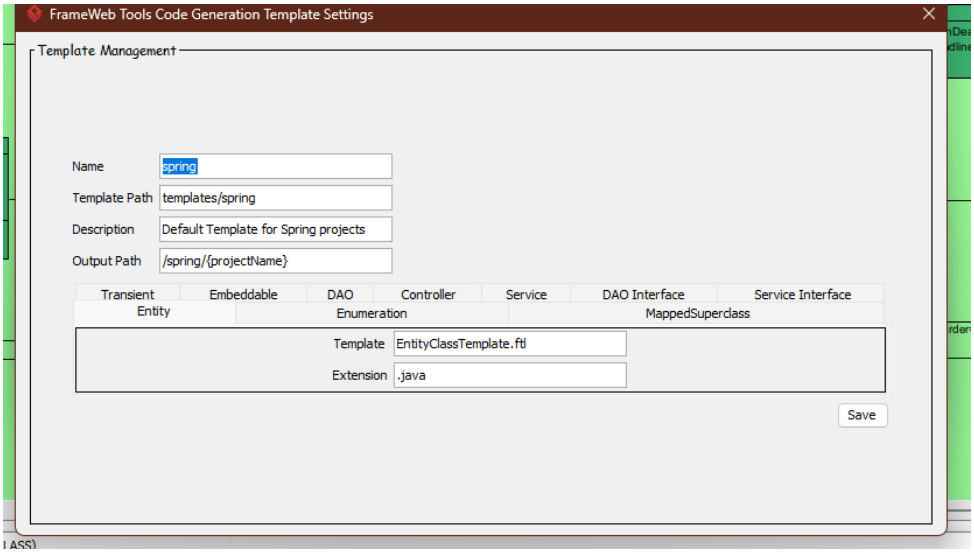

Figura 26 – Tela de importação de *templates* para geração de código

os *templates* e a extensão do arquivo de saída, para cada estereótipo do *FrameWeb*. A Figura 26 mostra a tela de importação de *templates*. Os campos de edição contém valores referência para um *template*.

Ao preencher os campos e clicar em *Save* é validada a existência de todos os campos e dos arquivos físicos, que deverão estar na pasta indicada em *Template Path* (arquivos auxiliares que não os especificados podem ter qualquer organização), que serão então importados. Uma entrada com nome dado é adicionada às configurações de geração de código do *plugin* e os arquivos são adicionados. Vale ressaltar que o *plugin* não acessa os arquivos originais e sim a cópia que está na pasta de configurações do *plugin*. Para fazer alterações, é necessário alterar os arquivos originais e importar novamente.

Se o *template* estiver configurado corretamente, é possível gerar o código-fonte ao clicar no botão *Generate*, visível na Figura 25, o código então será gerado na pasta indicada no campo *Output Path*. Se o caminho tiver a marcação *{projectName}*, será

adicionada uma pasta a mais com o nome do projeto que está tendo o código gerado. É preciso cuidado, pois se o caminho de saída já existir, todos os arquivos de mesmo nome terão seus conteúdos substituídos.

Vale ressaltar que a importação apenas valida a existência dos arquivos e a sintaxe dos mesmos, se os elementos modelados não tiverem todas as informações esperadas pelo modelo, exceções podem ocorrer durante a geração de código, caso em que as classes com problemas não terão seus códigos gerados corretamente.

4 Conclusão

O desenvolvimento do *plugin* do *FrameWeb* para o *Visual Paradigm* se mostrou uma tarefa complexa dada a falta de documentação deste último, mas com bons resultados quanto à adição das extensões UML e geração de *stubs* (esqueleto de código) para os modelos implementados. Nas próximas seções serão apresentadas as principais contribuições, limitações, lições aprendidas durante o desenvolvimento do trabalho, dificuldades enfrentadas e perspectivas de trabalhos futuros.

4.1 Considerações Finais

A utilização da Engenharia *Web* e ferramentas de modelagem UML como o *Visual Paradigm* se tornam cada vez mais essenciais para o desenvolvimento de SIW de qualidade e com maior produtividade. A utilização de *frameworks* se tornou muito comum e o *FrameWeb*, e consequentemente o *plugin* desenvolvido, auxiliam a produtividade da equipe de desenvolvimento e análise, criando uma ferramenta capaz de ajudar na documentação e na geração de código (utilizando das definições padrões de *frameworks*).

Este trabalho apresentou um *plugin* capaz de estender o *Visual Paradigm* para auxiliar na modelagem de Sistemas de Informação *Web* (SIWs) utilizando o *FrameWeb*, tendo como produto uma ferramenta capaz de modelar e gerar código para os modelos de Entidade, Persistência e Aplicação do *FrameWeb*.

Infelizmente, o fato do *Visual Paradigm* ser uma ferramenta proprietária de código fechado e a falta de documentação de como suas representações visuais são expostas nas *interfaces* da sua *API* de integração dificultaram o desenvolvimento do *plugin*. Foram necessárias várias horas de tentativa e erro para descobrir onde se encontravam informações relativamente simples como, por exemplo, as generalizações, associações e restrições de uma classe, etc.

A utilização da linguagem de *templates Apache FreeMarker* se mostrou uma boa escolha para a geração de código, pois a mesma é simples e possui uma boa documentação. A utilização de um *template engine* se mostrou acertada, pois permitiu que fosse definida uma *API* da representação UML que pode ser utilizada para várias linguagens diferentes, bastando apenas que a linguagem seja orientada a objetos e a criação de um novo conjunto de *templates* para a linguagem desejada.

Dito isso, as principais contribuições deste trabalho podem ser resumidas da seguinte forma:

- **Extensões UML:** o trabalho conseguiu adicionar as extensões UML do *FrameWeb* para os modelos indicados na Seção 1.2 para o *Visual Paradigm*, permitindo que analistas comuniquem intenção e funcionalidades com mais facilidade para os desenvolvedores;
- **Integração de Métodos de Desenvolvimento:** o *plugin* adiciona atalhos para as extensões UML do *FrameWeb* ao *Visual Paradigm*, permitindo que analistas tenham mais facilidade para modelar os sistemas comunicando intenção e funcionalidade;
- **Melhoria na Produtividade:** o *plugin* auxilia na integração da documentação com a geração de código, permitindo que os analistas possam delinear o sistema e agilizando as atividades dos desenvolvedores para que se concentrem em aspectos mais críticos do projeto;
- **Padronização e Qualidade:** o uso do método *FrameWeb* promove a padronização e a qualidade do código gerado, visto que o código inicial é gerado a partir de modelos UML e não diretamente pelos desenvolvedores, definindo *API's* e padrões de projeto.
- **Nova *build* do *Visual Paradigm*:** Durante o desenvolvimento foi encontrada uma falha no sistema ao tentar alterar a apresentação de uma *interface*. Foi então aberto um chamado no suporte do *Visual Paradigm* e a falha foi corrigida na nova *build* (20231040bs ou superior) do sistema.

Por fim, podemos relacionar os objetivos listados na Seção 1.2 com os resultados obtidos neste trabalho, como apresentado na Tabela 2.

Objetivo	**Situação (Seção / Apêndice)**
Modelar Entidades	Completo (3.1.1)
Gerar código-fonte	Completo (3.2)
Criar *templates*	Completo (3.2 e A)
Permitir que o usuário crie seus próprios templates	Completo (3.2 e A)
Desenvolver modelo de Aplicação e Persistência	Completo (3.1.2 e 3.1.3)

Tabela 2 – Relação de objetivos e resultados obtidos.

4.2 Trabalhos Futuros

Com base no trabalho realizado até o momento, existem várias perspectivas para trabalhos futuros que podem aprimorar e expandir o *plugin* do método *FrameWeb* para o *Visual Paradigm*:

- **Ampliação de Recursos:** adição do Modelo de Navegação, não incluído no escopo deste trabalho, ao *plugin*, permitindo a modelagem de telas e fluxos de navegação de modo mais simples e intuitiva;

- **Melhorias na Usabilidade:** alteração nas telas de parametrização de restrições e *templates* para serem mais funcionais e não demandem conhecimento prévio do usuário. Além da possibilidade de importação de *interfaces* já existentes em outros diagramas através do *plugin* e não manualmente;
- **Avaliação Empírica:** realização de estudos empíricos para avaliar o impacto do uso do *plugin* no processo de desenvolvimento *Web*, incluindo medições de produtividade e qualidade do código gerado.

Referências

ALUR, D.; CRUPI, J.; MALKS, D. *Core J2EE Patterns: Best Practices and Design Strategies*. 2nd. ed. [S.l.]: Prentice Hall / Sun Microsystems Press, 2003. Citado na página 27.

BOOT, S. *Spring Boot*. 2023. Official Website. Disponível em: <https://spring.io/projects/spring-boot>. Citado na página 13.

FADATARE, R. *What is a Cross-Cutting Concern*. 2023. <https://www.javaguides.net/2019/05/understanding-spring-aop-concepts-and-terminology-with-example.html>. Accessed on 1 July 2023. Citado na página 22.

FALBO, R. de A. Projeto de sistemas de software. In: . [s.n.], 2018. Accessed on 11th July 2023. Disponível em: <http://www.inf.ufes.br/~vitorsouza/falbo/Notas_Aula_Projeto_Sistemas_Falbo_2018.pdf>. Citado na página 17.

FALBO, R. de A. Engenharia de requisitos. In: . [s.n.], 2023. Accessed on 27th June 2023. Disponível em: <http://www.inf.ufes.br/~vitorsouza/falbo/Notas_Aula_Engenharia_Requisitos_Falbo_2017.pdf>. Citado na página 17.

FOUNDATION, T. A. S. *Apache FreeMarker™*. 2023. Disponível em: <https://freemarker.apache.org/>. Citado na página 28.

FOWLER, M. *Patterns of Enterprise Application Architecture*. 1. ed. [S.l.]: Addison-Wesley, 2002. ISBN 9780321127426. Citado na página 27.

FOWLER, M. Inversion of control containers and the dependency injection pattern. *martinfowler.com*, 2004. Disponível em: <https://www.martinfowler.com/articles/injection.html>. Citado 2 vezes nas páginas 21 e 22.

GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1. ed. Addison-Wesley Professional, 1994. ISBN 0201633612. Disponível em: <http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt_at_ep_dpi_1>. Citado na página 18.

MARCHIONI, F. *Facelets Tutorial: Using Templates*. 2022. <https://www.mastertheboss.com/java-ee/facelets/facelets-tutorial-using-templates/>. Accessed on July 4, 2023. Citado na página 20.

PRESSMAN, R. S.; LOWE, D. *Web Engineering: A Practitioner's Approach*. 1st. ed. New York, NY: McGraw-Hill, 2009. Copyright © 2009 by The McGraw-Hill Companies, Inc. All rights reserved. ISBN 978-0-07-352329-3. Citado 3 vezes nas páginas 11, 16 e 17.

REENSKAUG, T. *THING-MODEL-VIEW-EDITOR, an Example from a planning system*. [S.l.], 1979. Citado na página 18.

SALES, M. V. P. *Um plug-in para a ferramenta Visual Paradigm para edição de modelos OntoUML*. [S.l.], 2018. Citado na página 11.

SHIANGJEN, K. *CASE Tools*. 2023. PDF document. Computer Science, School of Information and Communication Technology, University of Phayao. Disponível em: <https://www.ict.up.ac.th/kanokwatt/se/Chapter6_CASE_Tools.pdf>. Citado 2 vezes nas páginas 25 e 26.

SILVA, L. R. M. *Aplicação do método FrameWeb no desenvolvimento de um sistema de informação com ASP.NET Core*. Vitória, ES, Brasil, 2021. Citado na página 11.

SOUZA, V. E. S. *FrameWeb: um método baseado em frameworks para o Projeto de Sistemas de Informação Web*. 124 p. Dissertação (Mestrado) — Universidade Federal do Espírito Santo, Centro Tecnológico, 2007. Dissertação de Mestrado. Citado 5 vezes nas páginas 11, 18, 23, 24 e 27.

SOUZA, V. E. S. The frameweb approach to web engineering: Past, present and future. In: ALMEIDA, J. P. A.; GUIZZARDI, G. (Ed.). *Engineering Ontologies and Ontologies for Engineering*. 1. ed. Vitória, ES, Brazil: NEMO, 2020. cap. 8, p. 100–124. ISBN 9781393963035. Disponível em: <http://purl.org/nemo/celebratingfalbo>. Citado 3 vezes nas páginas 14, 27 e 29.

SOUZA, V. E. S. *JButler: Serving you EJB utility code on a platter*. 2023. Git repository. Disponível em: <https://gitlab.labes.inf.ufes.br/labes/jbutler>. Citado na página 13.

TEAM, J. S. F. *Jakarta Server Faces 3.0*. 2023. Website. Version 3.0, September 23, 2020, Final. Accessed on 27th June 2023. Disponível em: <https://jakarta.ee/specifications/faces/3.0/jakarta-faces-3.0.html>. Citado na página 19.

Usando Python. *O que é MVC? Entenda arquitetura de padrão MVC*. 2023. Website. Accessed on 27th June 2023. Disponível em: <https://www.usandopy.com/pt/artigo/o-que-e-mvc-entenda-arquitetura-de-padrao-mvc/>. Citado na página 19.

VARMA, S. *What is ORM, how does it work?* 2023. Website. By Satish Varma, June 15, 2023. Disponível em: <https://javabydeveloper.com/orm-object-relational-mapping/>. Citado na página 21.

WEENEN, J. van. *Securing Spring Microservices with Keycloak - Part 2*. 2018. <https://blog.jdriven.com/2018/10/securing-spring-microservices-with-keycloak-part-2/>. Accessed on 1 July 2023. Citado na página 25.

Apêndices

APÊNDICE A – API de geração de código

Este apêndice contém os modelos que alimentam a API de geração de código do *FrameWeb*. O modelo de dados básico está descrito na Tabela 3, composto por representações de classes, atributos, associações, métodos, generalizações e realizações. Cada um dos modelos possui uma tabela de conteúdo com as propriedades disponíveis para cada modelo, um exemplo de modelo e o código gerado. *Templates* auxiliares utilizados não serão apresentados aqui, mas podem ser encontrados no repositório do *plugin*.

Atributo	**Descrição**
pack	Pacote, contém uma propriedade chamada name apenas
clazz	Classe descrita pela Tabela 4
path	Caminho do sistema para o arquivo, gerado ao quebrar o nome do pacote por ponto
methods	Métodos descritos pela Tabela 5
attributes	Atributos descritos pela Tabela 6
associations	Associações descritas pela Tabela 7

Tabela 3 – Propriedades disponibilizadas para a API de geração de código.

Atributo	**Descrição**
name	Nome.
generalization	Classe herdada, se existir. Se não existir, é uma *string* vazia
realizations	*Interfaces* implementadas pela classe. Se não existir, é uma lista vazia

Tabela 4 – Dados básicos para classes.

Atributo	**Descrição**
name	Nome
type	Retorno do método. Se nenhum retorno estiver definido, o retorno padrão é definido pelo modelo
visibility	Visibilidade do método. Se nenhuma visibilidade estiver definida, a visibilidade padrão é definida pelo modelo. A visibilidade pode ser: *public*, *protected*, *private*, *package*
parameters	Parâmetros descritos pela Tabela 8

Tabela 5 – Dados básicos para métodos.

Atributo	**Descrição**
name	Nome
type	Tipo
visibility	Visibilidade. Se nenhuma visibilidade estiver definida, a visibilidade padrão é definida pelo modelo. A visibilidade pode ser: *public*, *protected*, *private*, *package*

Tabela 6 – Dados básicos para atributos de uma classe.

Atributo	**Descrição**
sourceTypeName	Tipo da origem
targetTypeName	Tipo do destino
sourceName	Nome da propriedade para a origem. Se o nome da propriedade não estiver definido, a propriedade *sourceTypeName* em *camelCase* é usado
targetName	Nome da propriedade para o destino. Se o nome da propriedade não estiver definido, a propriedade *targetTypeName* em *camelCase* é usado

Tabela 7 – Dados básicos para associações entre duas classes.

Atributo	**Descrição**
name	Nome
type	Tipo

Tabela 8 – Dados básicos para parâmetros de métodos.

A.1 Modelo de Entidade

O modelo de dados para o Modelo de Entidades é o mais complexo entre os três apresentados, necessitando de uma maior quantidade de dados para a geração de código.

A.1.1 Dados de uma Classe do Modelo de Entidade

Os dados passados para a API quando é gerada uma classe do Modelo de Entidade é composto pelos dados básicos da Tabela 3 com algumas extensões às suas propriedades: *clazz* é estendida conforme a Tabela 9, *attributes* é estendida como descrito na Tabela 10 e *associations* é expandida como apresentado na Tabela 11.

Atributo	**Descrição**
inheritanceStereotype	Esteriótipo de herança. Possíveis valores: *join*, *single-table*, *union*

Tabela 9 – Dados específicos para classes que são entidades.

Atributo	Descrição
notNull	Se não permite valores nulos
isTransient	Se é transitório
size	O tamanho do atributo
dateTimePrecision	Precisão de data. Valores possíveis: *date*, *time*, *timestamp*
embedded	Se é incorporado
lob	Se é um *LOB*
id	Se é o identificador
version	Se é a versão
column	O nome da coluna física

Tabela 10 – Dados específicos para atributos de entidades.

A.1.2 Exemplo de *Template* para o Modelo de Entidades

O *template* exemplo para a geração de código de uma classe do Modelo de Entidade, utilizando do *framework Spring* é apresentado na Listagem A.1.

Listagem A.1 – Exemplo d e*template* para uma classe do Modelo de Entidades.

```
1   <#ftl strip_whitespace=true>
2   <#import "AttributeTemplate.ftl" as attr>
3   <#import "AttributeGetterSetterTemplate.ftl" as attrgs>
4   <#import "AssociationTemplate.ftl" as assoc>
5   <#import "AssociationGetterSetterTemplate.ftl" as assocgs>
6   <#import "MethodTemplate.ftl" as method>
7   <#import "ClassGeneralizationTemplate.ftl" as generalization>
8   <#import "ClassRealizationTemplate.ftl" as realizations>
9   <#import "EntityGeneralizationStereotype.ftl" as generalization_stereotype>
10  package ${pack.name};
11
12
13  import java.util.*;
14  import java.math.*;
15  import javax.persistence.*;
16  import javax.validation.constraints.*;
17
18  <#--Start of class-->
19  /** TODO: generated by FrameWeb. Should be documented. */
20  @Entity
21  <@generalization_stereotype.generate_generalization_stereotype stereotype=
        clazz.inheritanceStereotype/>
22  public class ${clazz.name} <@generalization.generate_generalization
        generalization=clazz.generalization defaultGeneralization=''/> <
        @realizations.generate_realization realizations=clazz.realizations
        defaultRealization=''/> {
23  /** Serialization id. */
24  private static final long serialVersionUID = 1L;
25
26  <#-- ASSOCIATIONS -->
27  <@assoc.generate_associations associations=associations className=clazz.name
        />
28
```

Atributo	**Descrição**
sourceToTargetCardinality	Cardinalidade da relação de origem para destino. Valores possíveis: *OneToOne*, *OneToMany*, *ManyToOne*
targetToSourceCardinality	Cardinalidade da relação de destino para origem. Valores possíveis: *OneToOne*, *OneToMany*, *ManyToOne*
sourceTransient	Se a origem é transitória
targetTransient	Se o destino é transitório
targetCollection	Tipo de coleção do destino. Valores possíveis: *bag*, *list*, *map*, *set*
sourceCollection	Tipo de coleção da origem. Valores possíveis: *bag*, *list*, *map*, *set*
targetFetch	Tipo de busca do destino. Valores possíveis: *eager*, *lazy*
sourceFetch	Tipo de busca da origem. Valores possíveis: *eager*, *lazy*
targetCascade	Tipo de cascata do destino. Valores possíveis: *none*, *all*, *merge*, *persist*, *refresh*, *remove*
sourceCascade	Tipo de cascata da origem. Valores possíveis: *none*, *all*, *merge*, *persist*, *refresh*, *remove*
targetOrder	Tipo de ordem do destino. Valores possíveis: natural, (lista separada por vírgulas de nomes de propriedades)
sourceOrder	Tipo de ordem da origem. Valores possíveis: natural, (lista separada por vírgulas de nomes de propriedades)

Tabela 11 – Dados específicos para associações entre entidades.

```
29  <#-- ATTRIBUTES -->
30  <@attr.generate_attributes attributes=attributes/>
31
32
33  <#-- METHODS -->
34  <@method.generate_methods methods=methods/>
35
36  // region Boilerplate Code
37  <#-- GETTERS AND SETTERS ASSOCIATIONS -->
38  <@assocgs.generate_getter_setter_associations associations=associations
        className=clazz.name/>
39
40  <#-- GETTERS AND SETTERS ATTRIBUTES -->
41  <@attrgs.generate_getter_setter_attributes attributes=attributes/>
42  // endregion
43
44
45  <#--end of class-->
46  }
```

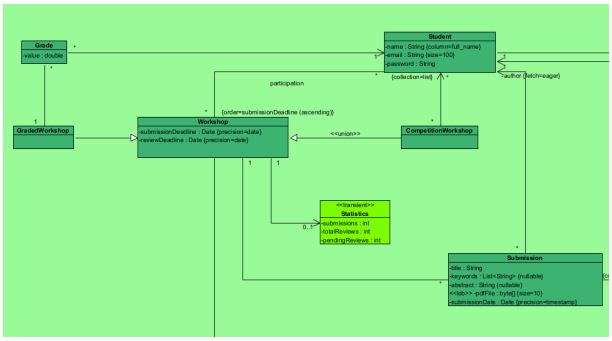

Figura 27 – Classe *Workshop* modelada no editor.

A.1.3 Código Gerado para uma Classe do Modelo de Entidades

Utilizando o *template* apresentado na Listagem A.1, geramos o código apresentado na Listagem A.2 para a classe *Workshop* do SIW *Oldenburg*, apresentada na Figura 27. A classe em questão possui várias associações e restrições, demonstrando a capacidade de geração de código do *plugin*.

Listagem A.2 – Código gerado para a classe *Workshop* com base num *template*.

```
1   package br.ufes.informatica.oldenburg.core.domain;
2
3   import java.util.*;
4   import java.math.*;
5   import javax.persistence.*;
6   import javax.validation.constraints.*;
7
8   /** TODO: generated by FrameWeb. Should be documented. */
9   @Entity
10  @Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
11  public class Workshop extends PersistentObjectSupport {
12      /** Serialization id. */
13      private static final long serialVersionUID = 1L;
14
15      /** Workshop to Statistics */
16      @Transient
17      private Statistics statistics;
18
19      /** Workshop to Submission */
20      @OneToMany(fetch = FetchType.LAZY, cascade = CascadeType.NONE)
21      @OrderBy("natural")
22      public List<Submission> submission;
23
24      /** Workshop to Student */
25      @OneToMany(fetch = FetchType.LAZY, cascade = CascadeType.NONE)
26      @OrderBy("natural")
27      public List<Student> student;
28
29      /** TODO: generated by FrameWeb. Should be documented. */
30
31      private Date submissionDeadline;
32      /** TODO: generated by FrameWeb. Should be documented. */
33
```

```
34          private Date reviewDeadline;
35
36          // region Boilerplate Code
37
38          /** Getter for statistics. */
39          public Statistics getStatistics() {
40              return statistics;
41          }
42
43          /** Setter for statistics. */
44          public void setStatistics(final Statistics statistics) {
45              this.statistics = statistics;
46          }
47
48          /** Getter for submission. */
49          public List<Submission> getSubmission() {
50              return submission;
51          }
52
53          /** Setter for submission. */
54          public void setSubmission(final List<Submission> submission) {
55              this.submission = submission;
56          }
57
58          /** Getter for student. */
59          public List<Student> getStudent() {
60              return student;
61          }
62
63          /** Setter for student. */
64          public void setStudent(final List<Student> student) {
65              this.student = student;
66          }
67
68          /** Getter for submissionDeadline. */
69          public Date getSubmissionDeadline() {
70              return submissionDeadline;
71          }
72
73          /** Setter for submissionDeadline. */
74          public void setSubmissionDeadline(final Date submissionDeadline) {
75              this.submissionDeadline = submissionDeadline;
76          }
77
78          /** Getter for reviewDeadline. */
79          public Date getReviewDeadline() {
80              return reviewDeadline;
81          }
82
83          /** Setter for reviewDeadline. */
84          public void setReviewDeadline(final Date reviewDeadline) {
85              this.reviewDeadline = reviewDeadline;
86          }
87          // endregion
88
89      }
```

A.2 Modelo de Persistência

O Modelo de Persistência não tem propriedades a mais, então utilizamos apenas os valores base.

A.2.1 Exemplo de *Template* para o Modelo de Persistência

O modelo na Listagem A.3 foi criado utilizando como base o padrão repositório (classe similar a um *Data Access Object*) utilizado com o *framework Spring Data JPA* e representa a *interface* do repositório.

Listagem A.3 – Exemplo de *template* para uma *interface* do Modelo de Persistência.

```
1  <#ftl strip_whitespace=true>
2  <#import "MethodInterfaceTemplate.ftl" as method>
3  <#import "InterfaceGeneralizationTemplate.ftl" as generalization>
4  <#assign defaultGeneralization = "JpaRepository<${clazz.name?replace('
       Repository', '')}>">
5  package ${pack.name};
6
7  import org.springframework.data.jpa.repository.JpaRepository;
8
9  /** TODO: generated by FrameWeb. Should be documented. */
10 public interface ${clazz.name} <@generalization.generate_generalization
       generalization=clazz.generalization defaultGeneralization=
       defaultGeneralization/> {
11
12
13 <#-- METHODS -->
14 <@method.generate_methods methods=methods/>
15
16 }
```

Já o modelo apresentado na Listagem A.4 apresenta a implementação de um repositório utilizando o *framework Spring Data JPA*.

Listagem A.4 – Exemplo de *template* para uma classe do Modelo de Persistência.

```
1  <#ftl strip_whitespace=true>
2  <#import "MethodOverrideTemplate.ftl" as method>
3  <#import "ClassGeneralizationTemplate.ftl" as generalization>
4  <#import "ClassRealizationTemplate.ftl" as realizations>
5  <#assign defaultGeneralization = "">
6  <#assign defaultRealization = "${clazz.name?replace('Impl', '')}">
7  package ${pack.name};
8
9
10 import org.springframework.stereotype.Repository;
11 import javax.persistence.PersistenceContext;
12
13 @Repository
14 public class ${clazz.name}
15 <@generalization.generate_generalization generalization=clazz.generalization
```

```
        defaultGeneralization=defaultGeneralization/> <@realizations.
        generate_realization realizations=clazz.realizations defaultRealization=
        defaultRealization/> {
16
17      @PersistenceContext
18      EntityManager entityManager;
19
20
21      <#-- METHODS -->
22      <@method.generate_methods methods=methods/>
23
24      }
```

A.2.2 Código Gerado para uma Classe do Modelo de Persistência

O código da Listagem A.5 foi gerado utilizando o *template* apresentado na Listagem A.3 sobre a classe *WorkshopRepository* representada na Figura 28.

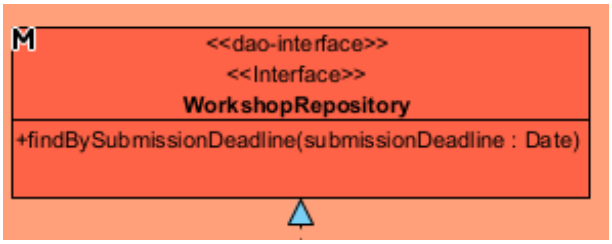

Figura 28 – Classe *WorkshopRepository* modelada no editor.

Listagem A.5 – Código gerado para a classe *WorkshopRepository* com base em um *template*.

```
1   package br.ufes.informatica.oldenburg.core.persistence;
2
3   import org.springframework.data.jpa.repository.JpaRepository;
4
5   /** TODO: generated by FrameWeb. Should be documented. */
6   public interface WorkshopRepository extends JpaRepository<Workshop> {
7
8       public void findBySubmissionDeadline(
9               final Date submissionDeadline);
10
11  }
```

Já o código da Listagem A.6 foi gerado utilizando o *template* apresentado na Listagem A.3 sobre a classe *WorkshopRepositoryImpl* representada na Figura 29. Em um projeto real, seria necessário criar uma *interface* extra que seria estendida pela *interface* de um *JpaRepository* e essa classe seria implementada, mas para fins de simplificação, modelamos uma referência direta entre a *interface* e a sua implementação.

Listagem A.6 – Código gerado para a classe *WorkshopRepositoryImpl* com base em um *template*.

```
1   package br.ufes.informatica.oldenburg.core.persistence;
```

Figura 29 – Classe *WorkshopRepositoryImpl* modelada no editor.

```
2
3     import org.springframework.stereotype.Repository;
4     import javax.persistence.PersistenceContext;
5
6     @Repository
7     public class WorkshopRepositoryImpl
8             implements
9             WorkshopRepository {
10
11        @PersistenceContext
12        EntityManager entityManager;
13
14        @Override
15        public void findBySubmissionDeadline(
16                final Date submissionDeadline) {
17            return;
18        }
19
20    }
```

A.3 Modelo de Aplicação

O Modelo de Aplicação não tem propriedades a mais, então utilizamos apenas os valores base.

A.3.1 Exemplo de *Template* para o Modelo de Aplicação

O modelo nas listagens A.7 e A.8 foram baseados em uma *interface* e sua implementação de serviço para o *framework Spring*.

Listagem A.7 – Exemplo de *template* para uma *interface* do Modelo de Aplicação.

```
1     <#ftl strip_whitespace=true>
2     <#import "MethodInterfaceTemplate.ftl" as method>
3     <#import "InterfaceGeneralizationTemplate.ftl" as generalization>
4     <#assign defaultGeneralization = "">
5     <#assign domainClassName = clazz.name?replace('Service', '') >
6     <#assign domainClassNameCamelCase = domainClassName?uncap_first>
7     package ${pack.name};
8
9
10    /** TODO: generated by FrameWeb. Should be documented. */
11    public interface ${clazz.name} <@generalization.generate_generalization
          generalization=clazz.generalization defaultGeneralization=
```

```
        defaultGeneralization/> {
12
13
14      <#-- METHODS -->
15      <@method.generate_methods methods=methods/>
16
17
18
19      public Optional<${domainClassName}> find${domainClassName}ById(Long id);
20      public List<${domainClassName}> findAll${domainClassName}s();
21      public ${domainClassName} save${domainClassName}(${domainClassName} ${
            domainClassNameCamelCase});
22      public void delete${domainClassName}(${domainClassName} ${
            domainClassNameCamelCase});
23
24      }
```

Listagem A.8 – Exemplo de *template* para uma classe do Modelo de Aplicação.

```
1       <#ftl strip_whitespace=true>
2       <#import "ServiceAttributeTemplate.ftl" as attr>
3       <#import "AttributeGetterSetterTemplate.ftl" as attrgs>
4       <#import "MethodOverrideTemplate.ftl" as method>
5       <#import "ClassGeneralizationTemplate.ftl" as generalization>
6       <#import "ClassRealizationTemplate.ftl" as realizations>
7       <#import "ServiceAssociationTemplate.ftl" as assoc>
8       <#import "ServiceAssociationMethodsTemplate.ftl" as assocmethods>
9       <#assign defaultGeneralization = "">
10      <#assign defaultRealization = "${clazz.name?replace('Impl', '')}">
11      package ${pack.name};
12
13
14      import org.springframework.stereotype.Service;
15
16      @Service
17      public class ${clazz.name}
18      <@generalization.generate_generalization generalization=clazz.generalization
            defaultGeneralization=defaultGeneralization/> <@realizations.
            generate_realization realizations=clazz.realizations defaultRealization=
            defaultRealization/> {
19
20      <#-- ASSOCIATIONS -->
21      <@assoc.generate_associations associations=associations/>
22
23      <#-- ATTRIBUTES -->
24      <@attr.generate_attributes attributes=attributes/>
25
26      <#-- METHODS -->
27      <@method.generate_methods methods=methods/>
28
29
30      <#-- GETTERS AND SETTERS ATTRIBUTES -->
31      <@attrgs.generate_getter_setter_attributes attributes=attributes/>
32
33      <#-- ASSOCIATIONS METHODS -->
34      <@assocmethods.generate_associations_methods associations=associations/>
```

```
35
36
37      }
```

A.3.2 Código Gerado para uma Classe do Modelo de Aplicação

O código da Listagem A.9 foi gerado utilizando o *template* apresentado na Listagem A.7 para a *interface WorkshopService* do SIW *Oldenburg*. A Figura 30 mostra a classe modelada no editor.

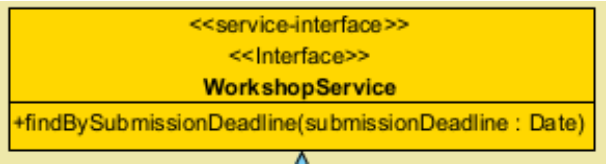

Figura 30 – Classe *WorkshopService* modelada no editor.

Listagem A.9 – Serviço a classe Workshop

```
38      package br.ufes.informatica.oldenburg.core.application;
39
40      /** TODO: generated by FrameWeb. Should be documented. */
41      public interface WorkshopService {
42
43          public void findBySubmissionDeadline(
44                  final Date submissionDeadline);
45
46          public Optional<Workshop> findWorkshopById(Long id);
47
48          public List<Workshop> findAllWorkshops();
49
50          public Workshop saveWorkshop(Workshop workshop);
51
52          public void deleteWorkshop(Workshop workshop);
53
54      }
```

Já código da Listagem A.10 foi gerado utilizando o *template* apresentado na Listagem A.8 para a classe *WorkshopServiceImpl* do SIW *Oldenburg*. A Figura 31 mostra a classe modelada no editor.

Listagem A.10 – Implementação do Serviço a classe Workshop

```
56      package br.ufes.informatica.oldenburg.core.application;
```

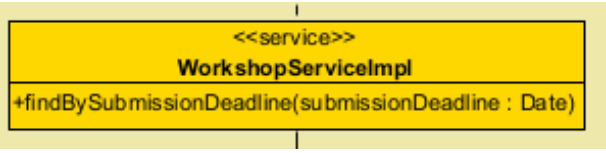

Figura 31 – Classe *WorkshopServiceImpl* modelada no editor.

```
57
58      import org.springframework.stereotype.Service;
59
60      @Service
61      public class WorkshopServiceImpl
62              implements
63              WorkshopService {
64
65          @Autowired
66          private WorkshopRepository workshopRepository;
67
68          @Override
69          public void findBySubmissionDeadline(
70                  final Date submissionDeadline) {
71              return;
72          }
73
74          @Override
75          public Optional<Workshop> findWorkshopById(Long id) {
76              return workshopRepository.findById(id);
77          }
78
79          @Override
80          public List<Workshop> findAllWorkshops() {
81              return workshopRepository.findAll();
82          }
83
84          @Override
85          public Workshop saveWorkshop(Workshop workshop) {
86              return workshopRepository.save(workshop);
87          }
88
89          @Override
90          public void deleteWorkshop(Workshop workshop) {
91              workshopRepository.delete(workshop);
92          }
93
94      }
```