CENTRO UNIVERSITÁRIO UNIVATES
CURSO DE ENGENHARIA DA COMPUTAÇÃO

MY DIAGRAMS: UMA PLATAFORMA COLABORATIVA EXTENSÍVEL PARA MODELAGEM DE SOFTWARE

Matheus Agnes Dias

Lajeado, junho de 2015.

Matheus Agnes Dias

MY DIAGRAMS: UMA PLATAFORMA COLABORATIVA EXTENSÍVEL PARA MODELAGEM DE SOFTWARE

Monografia apresentada ao Centro de Ciências Exatas e Tecnológicas do Centro Universitário UNIVATES, como parte dos requisitos para a obtenção do título de Bacharel em Engenharia da Computação.

Orientador: Prof. Ms. Pablo Dall'Oglio

Lajeado, junho de 2015.

Matheus Agnes Dias

MY DIAGRAMS: UMA PLATAFORMA COLABORATIVA EXTENSÍVEL PARA MODELAGEM DE SOFTWARE

A Banca examinadora abaixo, aprova a Monografia apresentada na disciplina Trabalho de Conclusão de Curso II, do Centro Universitário Univates, como exigência para obtenção do grau de Bacharel em Engenharia da Computação:

Prof. Ms. Pablo Dall'Oglio - orientador
Centro Universitário Univates

Prof. Ms. Alexandre Stürmer Wolf
Centro Universitário Univates

Prof. Ms. Vilson Cristiano Gärtener
Centro Universitário Univates

Lajeado, junho de 2015.

RESUMO

A sociedade depende cada vez mais de algum tipo de software, sendo que este vem se tornando imprescindível e cada vez mais utilizado nas atividades do dia a dia e, dos mais diversos setores. Essa dependência tem tornado os usuários cada vez mais críticos e exigentes em relação à qualidade dos produtos. Por outro lado, os profissionais que trabalham com desenvolvimento têm o desafio de construir softwares que tragam maior satisfação aos seus usuários, e, ao mesmo tempo tenham uma boa eficiência, usabilidade e manutenibilidade. Entretanto, com a constante mudança e crescimento da demanda de softwares pelo mercado, nem sempre é possível alcançar estas características. A forte demanda do mercado por soluções, muitas vezes, faz com que os profissionais envolvidos tenham de desenvolver soluções em um tempo cada vez menor, deixando de lado atividades essenciais como a modelagem do projeto, o que implica diretamente na qualidade do produto final. Quando trazida para dentro do ciclo de desenvolvimento de um produto, gerando artefatos concretos, e que sejam utilizados diretamente na construção do produto final, a modelagem traz ganhos consistentes de produtividade. Tendo em vista o cenário descrito, o objetivo do presente trabalho, é construir uma ferramenta extensível e colaborativa para modelagem estrutural de software, que permita gerar artefatos de desenvolvimento de alto nível, a fim de aumentar a produtividade em projetos de criação de software.

Palavras-chave: Engenharia de Software. Projeto de Software. Arquitetura em Camadas. Padrões de Projeto. *Frameworks*. Modelagem, ER, UML.

ABSTRACT

Society increasingly depends on some kind of software, and it is becoming essential and increasingly used in day to day activities in many sectors. This dependence has caused users to be increasingly critic and exigent about quality of products. Moreover, professionals who works with development has the challenge of building software that brings greater satisfaction to its users, while having good efficiency, usability and maintainability. However, with changing and growing demands of the software market, it is not always possible to achieve these characteristics. The strong market demand for solutions often causes professionals involved to have to develop solutions in an ever shorter time, leaving aside core activities such as design modeling, which directly involves end-product quality. When brought into the product development cycle, producing concrete articles that are directly used in the construction of the final product, the consistent modeling grants gains in productivity. Given this described scenario, the objective of this work is to build a collaborative and extensible tool for a structural modeling software that allows to generate high-level developing artifacts in order to increase productivity in software creation projects.

Keywords: Software Engineering, Software Design, Layered Architecture, Design Patterns, Frameworks, Modeling, ER, UML.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

CSS:	Cascading Style Sheets
ER:	Entity Relationship
HTML:	HyperText Markup Language
HTTP:	Hypertext Transfer Protocol
IEEE:	Institute of Electrical and Electronics Engineers
JSON:	JavaScript Object Notation
MVC:	Model, View, Controller
OMG:	Object Management Group
OMT:	Object Modeling Tecnique
PHP:	PHP Hypertext Preprocessor
SGBD:	Sistema de Gerenciamento de Banco de Dados
SOA:	Service Oriented Architeture
SOAP:	Simple Object Access Protocol

SQL:	Structured Query Language
SWEBOK:	Software Engineering Body of Knowledge
UML:	Unified Modeling Language
WSDL:	Web Service Description Language
XMI:	XML Metadata Interchange
XML:	Extensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

Softwares têm se tornado cada vez mais imprescindíveis nas atividades do dia a dia na sociedade, estando incorporados em diversos setores voltados à indústria, comércio e serviços. Além disso, softwares podem ser utilizados em sistemas de diversas áreas: entretenimento; transportes; medicina; indústria; militar e telecomunicações; onde se tornou indispensável pois agiliza e automatiza diversas operações e processos (PRESSMAN, 2010).

Segundo Pressman (2010), o aumento da importância do software e de sua utilização, fez com que a comunidade da área de Engenharia de Software criasse tecnologias inovadoras para tornar mais fácil, rápido e barato o desenvolvimento e a manutenção de sistemas de alta qualidade.

Um bom software deve ter características de manutenibilidade, confiança, eficiência e aceitabilidade. A manutenibilidade é obtida quando o software é escrito de forma que possa evoluir para atender futuras necessidades dos clientes, pois em um cenário de constante mudança essa situação é inevitável. Um software confiável não deve causar prejuízos em caso de falhas, tampouco deixar que usuários maliciosos possam acessar o sistema. Para ser considerado eficiente, o software deve fazer uso racional dos recursos computacionais. Por fim, para possuir aceitabilidade, ele deve ser compreensível, usável e compatível com vários sistemas já utilizados pelo usuário (SOMMERVILLE, 2011).

Para desenvolver um software com qualidade é muito importante planejar o seu desenvolvimento, bem como projetar boas estruturas. Estas estruturas facilitarão a posterior manutenção do software, o que acaba aumentado a sua vida útil. Neste sentido, é muito importante a preocupação com o *design* (arquitetura) do software, que por sua vez engloba os componentes, estruturas e camadas que farão parte dele.

Softwares construídos sem preocupação com a arquitetura, podem ter maior complexidade no desenvolvimento ou manutenção. Uma vez que, a agregação de novas funcionalidades poderá não se ajustar de forma transparente em uma arquitetura que não foi planejada com o objetivo de ser abstrata e extensível.

A forte demanda do mercado por soluções faz com que os profissionais envolvidos, muitas vezes, tenham de desenvolver soluções em um tempo cada vez menor, deixando de lado atividades essenciais como a modelagem do projeto. A modelagem é a etapa fundamental da execução de um projeto, pois nela são discutidos os principais conceitos envolvidos no projeto, bem como os seus relacionamentos. A execução inadequada da modelagem, ou mesmo a sua não execução, implica diretamente na qualidade e manutenibilidade do produto final.

Diferentes modelos foram criados para representar estruturas de arquitetura de software, sendo que podemos destacar o modelo UML (*Unified Modeling Language*), que tem o seu foco na modelagem de uma aplicação orientada a objetos, e, o modelo ER (*Entity Relationship*), que tem o seu foco na modelagem de estruturas de bancos de dados relacionais. Para criar esses modelos, ao longo do tempo foram criadas ferramentas que auxiliam a modelar os softwares. Dentre estas ferramentas podemos destacar: Astah; Rose; Star UML; Enterprise Architecture; e, Umbrello.

Quando trazida para dentro do ciclo de desenvolvimento de um produto, a modelagem pode acarretar em ganhos consistentes de produtividade, por meio da geração de artefatos concretos que podem ser utilizados diretamente na construção do produto final. Pode-se citar que algumas ferramentas de modelagem oferecem funcionalidades, como exemplo, tem-se a importação e exportação de formatos abertos (XML), e também a geração de *schemas* de bases de dados, bem como protótipos de classes em linguagens de programação como Java e C++, dentre outras tecnologias.

1.1 Motivação

A partir de um mesmo modelo de software, seja ele ER (*Entity Relationship*) ou UML (*Unified Modeling Language*), diferentes artefatos de desenvolvimento podem ser gerados. Por meio do modelo ER, é possível construir rotinas que gerem formulários básicos, relatórios

em formatos de tabela e outros elementos de baixa complexidade. Já o modelo UML, que carrega informações mais ricas sobre o relacionamento entre os objetos, permite gerar estruturas mais complexas, como classes, relacionamentos entre elas, e, mecanismos de persistência. Hoje em dia, a maioria dos desenvolvimentos de software usa como base *frameworks*, e as ferramentas de modelagem não estão preparadas para gerar artefatos de alto nível para *frameworks* específicos.

A maioria das ferramentas de modelagem são focadas em um modelo específico e, possuem limitações quanto ao que pode ser gerado a partir deste modelo. Além disso, deve-se lembrar que a maioria das ferramentas, dentre as mais reconhecidas no mercado, Astah, Rose, Star UML, Enterprise Architecture, Umbrello, não permitem ou possuem limitações em relação à modelagem colaborativa.

Tendo como base a necessidade de gerar artefatos de alto nível para *frameworks* específicos, e, tendo em vista a necessidade de colaboração na criação de modelos, o presente trabalho propõe a criação de uma plataforma colaborativa para modelagem estrutural de software que seja genérica, permitindo a interoperabilidade com diferentes estruturas de software. Esta plataforma poderá ser estendida com a adição de *plugins* que permitam a importação e exportação de diferentes formatos, permitindo a geração de artefatos de alto nível para *frameworks* específicos.

1.2 Objetivos

O objetivo geral do presente trabalho é o desenvolvimento de uma plataforma colaborativa e extensível para a criação e modelagem estrutural de software.

1.2.1 Objetivos específicos

Os objetivos específicos são:

a) Levantamento dos requisitos de uma ferramenta para modelagem de software;

b) Implementação de uma plataforma básica de modelagem *online*;

a) Implementação de *plugins* para geração de artefatos de alto nível;

b) Desenvolvimento da prova de conceitos com um ou mais *frameworks*.

1.3 Organização do trabalho

Para melhorar a compreensão do presente trabalho, seus capítulos foram divididos em uma ordem de apresentação.

O capítulo 2 apresenta o referencial teórico, utilizado como embasamento para o desenvolvimento do trabalho, que conta com os seguintes subcapítulos: Engenharia de Software; Arquitetura de Software; Reutilização de Software; Modelagem Relacional; Modelagem Orientada a Objetos; e, Padrões de Projeto.

No capítulo 3 é abordada a metodologia utilizada para a realização do trabalho, onde é apresentado o método científico em que este trabalho se enquadra. Além disso, são apresentadas as etapas para o desenvolvimento do trabalho.

O capítulo 4 apresenta a especificação do projeto proposto pelo presente trabalho, onde será demonstrada sua visão geral, casos de uso, cenários de utilização, bem como a especificação de suas classes.

No captítulo 5 é apresentada a validação da plataforma prosposta, bem como os resultados obtidos através dos testes realizados.

Por fim, o capítulo 5, apresenta as considerações finais do presente trabalho.

2 REFERENCIAL TEÓRICO

No presente capítulo serão apresentados os principais conceitos relacionados ao desenvolvimento da proposta. A Engenharia de software é a disciplina sob a qual todos os conhecimentos e habilidades, tais como: análise, *design*, desenvolvimento, dentre outros, estão relacionados. A presente proposta também possui um forte viés de arquitetura e reúso de software, tendo em vista o foco em modelagem de software, portanto, os assuntos Arquitetura de software e reúso, também serão abordados. Como a plataforma proposta fará uso de serviços em sua implementação, também será abordado o tema Arquitetura Orientadas a serviços. A Modelagem relacional e a modelagem orientada a objetos estão no cerne da solução, sendo expostas de maneira mais detalhada. Como a solução visa a geração de artefatos de alto nível para o desenvolvimento de sistemas, é essencial a adoção de padrões de projeto para tal, principalmente aqueles relacionados com a representação e conversão de modelos. Desta forma, o assunto padrões de projeto também será explorado.

2.1 Engenharia de Software

Softwares vêm tornando-se cada vez mais importantes no dia a dia da sociedade, pois podem ser utilizados em sistemas de diversas áreas como entretenimento, transportes, medicina, indústria, militar e telecomunicações. Software tornou-se indispensável pois agiliza e automatiza diversas operações e processos. O desenvolvimento, especificação, gerenciamento, e a evolução desses sistemas, constituem a disciplina da Engenharia de Software (SOMMERVILLE, 2011).

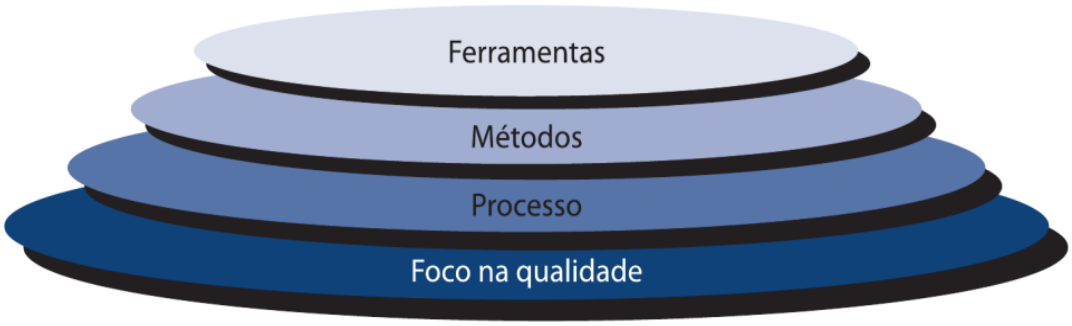
O objetivo da Engenharia de Software é viabilizar métodos, padrões, e estruturas que auxiliem no desenvolvimento de um software, tendo como foco a qualidade. Para alcançar

este objetivo, a Engenharia de Software abrange todas as fases de construção de um sistema, desde o levantamento de requisitos, especificação, desenvolvimento, gerenciamento, até a evolução e implantação do sistema (PRESSMAN, 2010).

A IEEE (*Institute of Electrical and Electronics Engineers*), define a Engenharia de Software como o estudo de abordagens, bem como, aplicação delas de uma forma sistemática, disciplinada e quantificável para o desenvolvimento, operação, e, manutenção do software, aplicando à engenharia ao software (IEEE, 1993).

A Engenheira de Software pode ser vista como um conjunto de camadas, que tem como base de apoio a qualidade. É dividida em quatro camadas: ferramentas, métodos, processo e foco na qualidade, conforme apresenta a Figura 1. Abordagens de engenharia devem apoiar-se em um compromisso organizacional com a qualidade, buscando constante aperfeiçoamento, o qual leva a um desenvolvimento constante de abordagens mais efetivas para a engenharia de software.

Figura 1 - Exemplo de estrutura organizacional (Vertical e Horizontal)



Fonte: Pressman(2010).

A primeira camada, que constitui a Engenharia de Software, foca na qualidade. A segunda camada é a de processo, a qual é considerada o alicerce da Engenharia de Software, pois formam a base para o controle gerencial dos projetos de software. Os métodos representam a terceira camada e, são responsáveis pelo desenvolvimento do software, fornecendo a técnica de "como fazer" para construí-los. A última camada é a de ferramentas, e tem como objetivo, fornecer apoio automatizado e semiautomatizado para os processos e para os métodos.

A disciplina de Engenharia de Software é muito ampla, englobando diversas áreas e assuntos. Em função de sua abrangência, surgiu a necessidade de criar uma referência e consenso sobre os seus temas. Com a iniciativa da IEEE, foi criado o SWEBOK (*Software*

Engineering Body of Knowledge). Segundo o SWEBOK (2014), o livro é um guia de uso e aplicações das melhores práticas de Engenharia de Software, tendo como principal objetivo, estabelecer um conjunto apropriado de critérios e normas para a prática profissional da Engenharia de Software.

De acordo com Sbrocco (2012), a Engenharia de Software está dividida em dez áreas no SWEBOK:

a) Gerência de engenharia: são as boas práticas que devem ser utilizadas na gerência do desenvolvimento do software;

b) Gerência de configuração: é a área responsável em identificar, controlar, documentar, auditar as mudanças que ocorrem durante o desenvolvimento do sistema;

c) Processos de engenharia: é responsável por estabelecer uma prática para auxiliar no desenvolvimento do software, definindo, implementando, avaliando e mensurando mudanças e melhorias ao longo do ciclo de vida deste;

d) Ferramentas e métodos: busca novas ferramentas e aplica novos métodos, tendo como objetivo aumentar a produção, qualidade, auxiliando na manutenção e no desenvolvimento de software;

e) Qualidade: utiliza padrões de qualidade e estabelece métodos para o desenvolvimento de produtos, atendendo as especificações de qualidade e o prazo estipulado;

f) Requisitos: faz uso de boas práticas para o levantamento de requisitos, bem como na análise, negociação, especificação e documentação dos mesmos;

g) *Design* de software: tradução dos requisitos para uma arquitetura, com objetivo de detalhar a estrutura da solução;

h) Construção de software: área responsável pela codificação do software;

i) Teste de software: tem como principal objetivo validar e verificar se os requisitos especificados foram atendidos e implementados da forma correta;

j) Manutenção de software: ajustes necessários para poder manter o software ao longo do tempo, como melhorias, novas funcionalidades e correções de problemas.

2.2 Arquitetura de software

Ao iniciar o desenvolvimento de um novo software, é de fundamental importância definir a arquitetura que será utilizada para a sua construção, pois softwares construídos sem planejamento da arquitetura podem ter maior complexidade em seu desenvolvimento ou manutenção. A falta de planejamento da arquitetura acaba implicando diretamente na qualidade do software, em futuras agregações de novas funcionalidades, as quais podem não se ajustar de forma transparente em uma arquitetura que não foi planejada, com o objetivo de ser abstrata e extensível.

Segundo Booch (2012), a arquitetura de software é importante, pois afeta o desempenho e a robustez, bem como a capacidade de distribuição e de manutenibilidade de um sistema. Softwares complexos e grandes devem ter a sua definição de arquitetura bem estruturada, pois, consequentemente, eles terão um maior número de funcionalidades e processos, necessitando assim, fornecer uma boa usabilidade, estabilidade e desempenho.

Um sistema tradicionalmente é formado por componentes, sendo que a forma como esses componentes estão organizados e se comunicam, implementam os requisitos de um sistema. Os requisitos não funcionais são influenciados pelos componentes escolhidos, mas a influência que predomina é a da arquitetura escolhida para o projeto de tal sistema (SOMMERVILLE 2011).

Como a relação entre os requisitos não funcionais e a arquitetura de software são muito próximas, a escolha da arquitetura irá depender dos requisitos não funcionais, escolhidos para o sistema, como:

a) Desempenho: arquitetura deve ser projetada para agrupar as operações críticas dentro de um pequeno número de componentes e implantados em um mesmo computador, ao invés de estarem distribuídas pela rede. É importante utilizar componentes grandes, para diminuir o número comunicações entre eles, bem

como considerar, a organização do sistema de modo que ele seja executado em diversos processadores (SOMMERVILLE, 2011);

b) Proteção: na arquitetura deve ser utilizada uma estrutura em camadas, que permita proteger ativos mais críticos nas camadas mais internas, aplicando um alto nível de validação e proteção nestas camadas (SOMMERVILLE, 2011);

c) Segurança: a arquitetura deve ser projetada para que as operações que precisam de mais segurança estejam localizadas em um só componente, ou em poucos componentes. Reduzindo assim, o custo e problemas de validação;

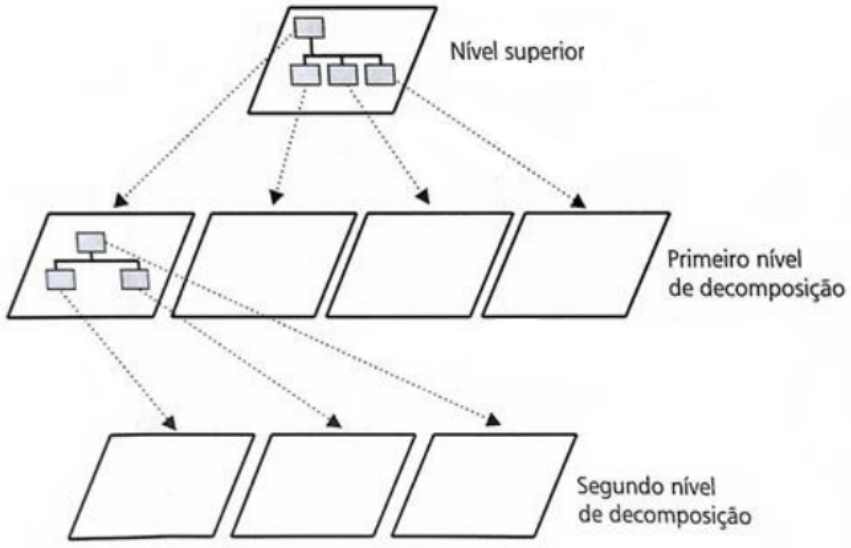
d) Disponibilidade: ao projetar a arquitetura, ela deve incluir componentes redundantes, para que seja possível substituir e atualizar componentes sem a necessidade de parar o sistema (SOMMERVILLE, 2011);

e) Manutenção: a arquitetura do sistema deve ser projetada a partir de componentes autocontidos, de baixa granularidade, e que possam ser facilmente alterados. Deve-se evitar a utilização de estruturas de dados compartilhadas (SOMMERVILLE, 2011).

Após a definição da arquitetura do sistema, a organização dele deve ser estabelecida, definindo um tipo de modelo para o seus componentes e módulos, definindo também a maneira como será realizada a separação e interação das funcionalidades. A arquitetura irá fornecer uma visão geral do sistema, auxiliando no planejamento, tomada de decisões, gerenciamento do projeto e construção do sistema.

O projeto de arquitetura de um sistema pode ser descrito, primeiramente em um nível superior, onde se encontram os seus principais elementos, os quais são os módulos. Após esta etapa, vem a refinação e detalhamento de funcionalidades e recursos, onde se define a integração entre eles. Esta etapa gera a decomposição do sistema em vários subsistemas. Conforme demonstrado na Figura 2.

Figura 2 - Níveis de decomposição



Fonte: Pfleeger(2007, p. 162).

O processo de decomposição visa aperfeiçoar e detalhar os componentes que encontram-se um nível acima de um módulo, considerando o nível superior como o mais abstrato por ocultar as informações dos detalhes funcionais e de processamento. Estas características permitem que um componente seja alterado, sem que as mudanças afetem outros componentes.

Para Pfleeger (2007), a utilização de modularidade é uma característica de um bom projeto, pois, fazendo o uso desta técnica, o sistema estará dividido em módulos ou componentes, onde cada um terá as suas respectivas entradas e saídas. Estes artefatos são organizados em uma hierarquia decorrente da decomposição, o que possibilita corrigir e encontrar eventuais problemas de uma forma mais fácil e rápida.

Para Sommerville (2011), existem algumas vantagens quando se projeta e documenta a arquitetura de software. São elas:

a) Comunicação de *stackholders*: gera uma apresentação do sistema em alto nível, o que facilita a compreensão do sistema e a comunicação com os membros da equipe;

b) Análise de sistema: ajuda a identificar aspectos e características explícitas do sistema, com objetivo de identificar se o sistema irá conseguir atender aos requisitos críticos (não funcionais), como desempenho, confiabilidade e manutenibilidade;

c) Reúso em larga escala: o modelo de arquitetura é a uma descrição compacta de como o sistema é organizado, e, como os componentes interromperam. Em sistemas de arquitetura semelhante, é possível reutilizar componentes em larga escala.

No desenvolvimento de software, existem vários modelos de padrões que podem ser adotados para a arquitetura, sendo que os mais utilizados, e que possuem bons princípios de projeto e de arquitetura, são: arquitetura em camadas; arquitetura de repositório; arquitetura cliente-servidor; arquitetura orientada a serviços; arquitetura de duto e filtro. O presente trabalho abordará o conceito de arquitetura em camadas e orientada a serviços, pois nos próximos capítulos estas serão utilizadas como referência.

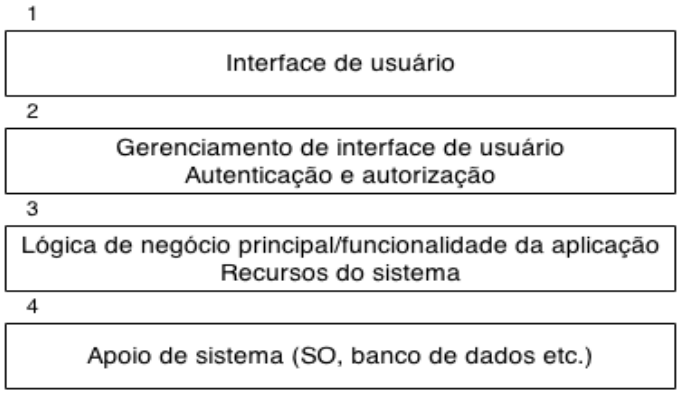
2.2.1 Arquitetura em camadas

O padrão da arquitetura em camadas separa os elementos de um sistema, permitindo alterá-los de forma independente, sem comprometer o funcionamento do sistema. Esta arquitetura organiza as funcionalidades do sistema em camadas separadas, sendo assim, cada camada é dependente dos recursos e serviços oferecidos pela camada abaixo dela. Os níveis de camadas mais baixos representam os principais serviços utilizados em todo o sistema (SOMMERVILLE, 2011).

Segundo Sommerville (2011), a arquitetura em camadas é mutável e portável, apoia o desenvolvimento incremental dos sistemas, pois quando uma camada é desenvolvida, seus serviços podem ser disponibilizados para os usuários. Enquanto não ocorrerem modificações na interface, uma camada pode ser substituída por outra equivalente. Quando ocorrer alguma mudança na camada de interface, apenas a camada adjacente é afetada.

A Figura 3 demonstra um exemplo de uma arquitetura em camadas, com quatro camadas. A quarta camada representa os softwares de apoio ao sistema, como banco de dados. A terceira camada é de aplicação, que inclui os componentes relacionados com as funcionalidades da aplicação e, componentes utilitários, utilizados por outros componentes da aplicação. A segunda camada tem como objetivo o gerenciamento da interface dos usuários, validando e autorizando-os com a camada superior, fornecendo recursos de interface com os usuários.

Figura 3 - Arquitetura em camadas genéricas



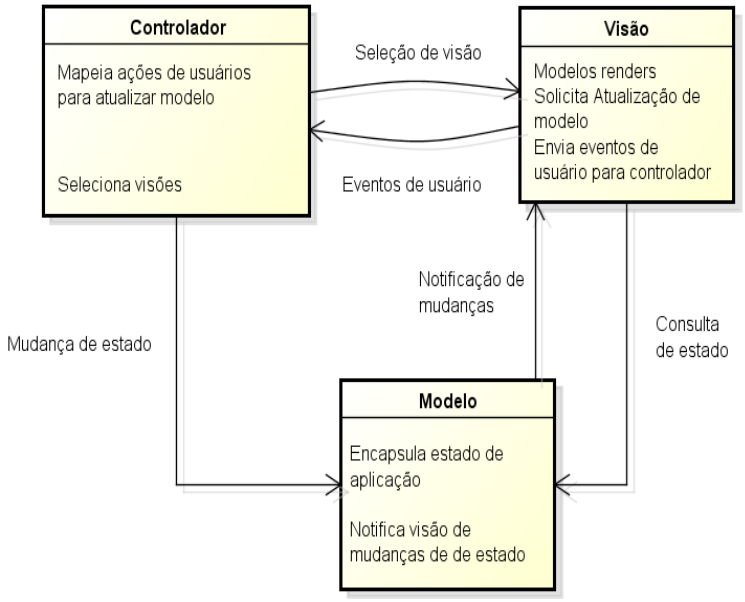
Fonte: Sommerville(2011, p. 111).

2.2.1.1 Arquitetura MVC

O padrão de arquitetura MVC (*Model, View, Controller*) é um dos padrões mais utilizados, seja em aplicações *desktop*, ou em sistemas web, pois permite mudar determinados elementos do sistema de uma forma independente, sem afetar o resto dele, tornando-o extensível.

Para conseguir realizar estas mudanças sem afetar o sistema por completo, o padrão MVC estrutura o sistema em três camadas que interagem entre si, contendo componentes de Modelo, Visão ou Controle. O componente Modelo é responsável pelo gerenciamento de dados e operações associadas a esses dados, como regras de negócio. O componente Visão, gerencia e define como os dados serão apresentados para o usuário. O componente Controlador, gerencia a interação do usuário, como por exemplo, cliques em botões, teclas, entre outros, os quais interagem com a Visão. O seu papel é validar e repassar as informações contidas na Visão, para que o Modelo possa gerenciá-las. A Figura 4 demonstra um modelo genérico de como é organizado o modelo MVC.

Figura 4 - Organização do modelo MVC



Fonte: Sommerville (2011, p. 111).

2.2.2 Arquitetura orientada a serviços

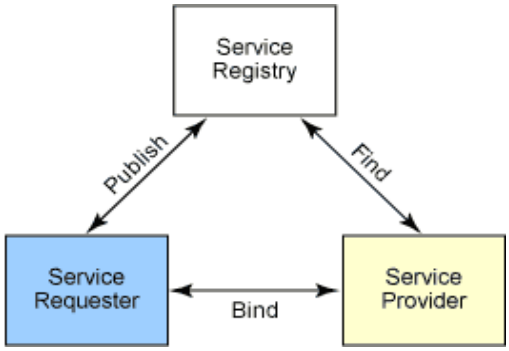
Arquitetura Orientada a Serviços, ou *Service Oriented Architecture* (SOA), é uma abordagem de desenvolvimento criada para interligar diferentes softwares de uma forma simplificada e unificada, por meio de serviços. Tem como foco a interoperabilidade, podendo ser adaptável em quase todas as situações. Ela já está em constante utilização no desenvolvimento de aplicações (O'BRIEN; MERSON;BASS, 2007).

Em uma SOA, todos os componentes de software são modelados como serviços. Tem como premissa que, todas as tarefas ou processos sejam construídos, para que sejam designados como serviços, sendo acessados através da rede. SOA trabalha utilizando *Web Services*, o que nada mais é do que uma forma de implementação do conceito de SOA.

Para garantir a interoperabilidade dos serviços *web* entre diversos sistemas, estes serviços utilizam basicamente dois padrões: *Web Service Description Language* (WSDL) e *Simple Object Access Protocol* (SOAP), que são padrões abertos e de ampla utilização.

A Arquitetura Orientada a Serviços pode ser representada a partir do processo de "*find-bind-execute paradigm*", ou “paradigma procura-consolida-executa". Na Figura 5, é exemplificada uma estrutura SOA.

Figura 5 - Paradigma procura-consolida-executa



Fonte: O'Brien, Merson, Bass (2007).

Uma SOA possui três responsabilidades:

a) *Service Provider* (Provedor de Serviço): é responsável por criar uma descrição de serviço;

b) *Service Requestor* (Requisitor de Serviço): é responsável por encontrar uma descrição de serviço;

c) *Service Registry* (Registro de Serviço): é responsável por referenciar um local, em que os provedores de serviço podem transmitir informações.

2.3 Reutilização de Software

Uma empresa que desenvolve softwares, geralmente já possui vários outros softwares desenvolvidos, e estes possuem semelhança entre si, ou seja, têm funcionalidades e objetivos iguais. Assim, ao construir um novo sistema, é possível o aproveitamento ou adaptação de componentes e estruturas utilizadas anteriormente.

Segundo Sommerville (2011), a utilização de técnicas de reúso de código resulta na redução dos custos de desenvolvimento do software, pois menos componentes precisam ser desenvolvidos, implementados e avaliados. Entretanto, essa redução não é a única vantagem, pois há também um aumento na velocidade de desenvolvimento, qualidade, maior confiança no software desenvolvido, redução na margem de erro e da estimativa de custos do projeto.

Para fazer uso da reutilização, Pfleeger (2007) afirma que existem basicamente duas formas: reutilização caixa-preta e reutilização caixa-branca. A reutilização caixa-preta consiste em reutilizar componentes inteiros, sem modificá-los, como um conjunto de funções

de cálculo de impostos. Já a reutilização caixa-branca, consiste na adaptação de componentes de acordo com as necessidades, como na utilização de um *framework*, que pode ser modificado e estendido de acordo com as necessidades do software.

Para Sommverville (2009) o reúso de software é possível em vários níveis diferentes:

a) Nível de abstração: neste nível o que é reutilizado é o conhecimento das abstrações de sucesso no projeto do software, como os padrões de projeto e de arquitetura utilizadas;

b) Nível de objeto: neste nível são reutilizados objetos de uma biblioteca, sem precisar reescrever um novo código;

c) Nível de componentes: neste nível se faz reúso de componentes, que são coleção de objetos e classes de objetos, e que interagem entre si para fornecer funções e serviços. Um exemplo deste tipo de reúso é a utilização de um *framework* para a construção de uma tela de autenticação de usuário;

d) Nível de sistema: neste nível é feito o reúso completo de um sistema, onde são feitos alguns ajustes, adicionando novas funcionalidades, através da adição e modificação do código, ou através da interface do sistema.

O reúso de software, além de garantir o aumento da produtividade, acaba também aumentando a qualidade, pois os componentes e processos que foram reutilizados, geralmente já foram testados e validados em outros sistemas, garantindo assim, o seu funcionamento.

Para que seja possível a reutilização de software, ao desenvolvê-lo deve-se aplicar algumas práticas de arquitetura, como a criação de componentes com baixo acoplamento, alta coesão e o princípio da responsabilidade única, tornando-os mais robustos.

2.4 Modelagem relacional

A construção de um projeto de banco de dados passa pela elaboração dos modelos conceitual, lógico e físico. Tradicionalmente, o modelo conceitual é representado pelos diagramas ER, e o modelo lógico transforma o modelo ER em um modelo relacional, o qual define as tabelas e os seus respectivos campos (HEUSER, 2009).

O modelo relacional é considerado, atualmente, como o principal modelo de dados utilizado em aplicações comerciais. Se tornou o principal modelo devido à sua simplicidade, o que facilita o trabalho dos programadores, se comparado com modelos de dados anteriores, como o modelo de rede ou modelo hierárquico (SILBERSCHATZ; KORTH; SUDARSHAN, 2012).

Um banco de dados relacional é composto por um conjunto de tabelas (entidades), onde cada tabela recebe um nome único, o qual a representa. Cada tabela possui campos (atributos), as quais também possuem um nome único e um tipo específico de dados (SILBERSCHATZ; KORTH; SUDARSHAN, 2012).

2.4.1 Tabela

A tabela é formada por um conjunto não ordenado de linhas, e cada linha é composta por uma série de campos, onde cada campo tem a informação referente à linha. A Figura 6 representa um exemplo de uma tabela “Empregado”, onde cada linha da tabela corresponde a um empregado, e, cada campo é uma informação referente a este empregado. (HEUSER, 2009)

Figura 6 - Exemplo de tabela

Empregado

CodigoEmp	Nome	CodigoDepto	CategFuncional
E5	Souza	D1	C5
E3	Santos	D2	C5
E2	Silva	D1	C2
E1	Soares	D1	-

Fonte: Heuser (2009, p. 121).

Os campos de uma tabela são identificados através de um nome único. Na figura 6, os nomes dos campos estão representados no cabeçalho da tabela. As tabelas de um banco de dados possuem as seguintes características:

a) As linhas de uma tabela não têm ordenação. O SGBD (Sistema de Gerenciamento de Banco de Dados) recupera as linhas de forma arbitrária. Entretanto, ao fazer uma consulta é possível definir a ordenação;

b) Os valores dos campos de uma tabela são atômicos, pois não podem ser compostos por outros campos;

c) Ao utilizar linguagens de consulta de base de dados relacionais, é possível ter acesso a diversos campos de uma ou mais linhas, através de critérios pré-definidos na consulta.

2.4.2 Chave

A chave em um banco de dados relacional tem como objetivo básico identificar linhas e estabelecer relações entre linhas de tabelas do banco de dados. Em um banco de dados relacional há dois tipos de chaves a considerar, chave primária e chave estrangeira.

2.4.2.1 Chave primária

A chave primária de uma tabela é uma coluna ou uma combinação de colunas, onde os valores irão distinguir, de forma única, uma linha das demais, dentro da tabela.

A Figura 7 apresenta o exemplo da tabela Dependente, que possui uma chave primária composta por duas colunas, CodEmp e NoDepen. Neste exemplo, tanto a coluna CodEmp, como a NoDepen, possuem valores que se repetem dentro delas. Sendo assim, é necessário considerar ambos os valores para identificar uma linha da tabela de forma única.

Figura 7 - Tabela com chave primária composta

Dependente

CodEmp	NoDepen	Nome	Tipo	DataNasc
E1	01	João	Filho	12/01/2001
E1	02	Maria	Filha	20/10/2003
E2	01	Ana	Esposa	12/12/1970
E5	01	Paula	Esposa	14/08/1981
E5	02	José	Filho	03/05/1985

Fonte: Heuser (2009, p. 122).

2.4.2.2 Chave estrangeira

A chave estrangeira de uma tabela é uma coluna ou uma combinação de colunas, onde os valores são uma referência para uma chave primária. O campo de uma chave estrangeira de uma tabela, possui o mesmo valor do campo de uma chave primária de outra tabela. A chave estrangeira permite a implementação dos relacionamentos entre tabelas em um banco de dados relacional.

A Figura 8 a seguir representa as tabelas de um banco de dados, onde a coluna CodigoDepto da tabela Emp (empregado) possui uma chave estrangeira para a chave primária da tabela Dept (departamento). Esta relação significa que, todo o empregado deve estar relacionado a um departamento.

Figura 8 - Chave estrangeira

Dept

CodigoDepto	NomeDepto
D1	Compras
D2	Engenharia
D3	Vendas

Emp

CodEmp	Nome	CodigoDepto	CategFuncional	CPF
E1	Souza	D1	-	132.121.331-20
E2	Santos	D2	C5	891.221.111-11
E3	Silva	D2	C5	341.511.775-45
E5	Soares	D1	C2	631.692.754-88

Fonte: Heuser (2009, p. 124).

A utilização de uma chave estrangeira em uma tabela impõe algumas restrições que devem ser atendidas ao executar algumas operações no banco de dados:

a) Quando ocorrer a inclusão de um novo registro em uma tabela que possui uma chave estrangeira, o valor da chave estrangeira deve existir na coluna da chave primária referenciada;

b) Quando ocorrer a alteração do valor da chave estrangeira, deve ser garantido que o novo valor da chave estrangeira exista na coluna da chave primária referenciada;

c) Quando ocorrer a exclusão de um registro da tabela que contém a chave primária, referenciada pela chave estrangeira, a coluna da chave estrangeira não pode possuir o valor da chave primária que está sendo excluída;

d) Quando ocorrer a alteração do valor da chave primária referenciada pela chave estrangeira, deve ser garantido que, o valor da chave estrangeira não apareça com o valor antigo da chave primária que está sendo alterada.

2.5 Modelagem orientada a objetos

A modelagem orientada a objetos pode ser realizada através da utilização da UML (*Unified Modeling Language*), que é uma linguagem visual, utilizada para modelar softwares baseados no paradigma da orientação a objetos. Nos últimos anos, esta linguagem tornou-se a linguagem padrão para a modelagem de software, sendo adotada internacionalmente pela indústria de Engenharia de Software (GUEDES, Gilleanes T. A, 2009).

Softwares projetados de maneira orientada a objetos, e que utilizam padrões de projeto, acabam, por sua vez, resolvendo problemas específicos de projetos, tornando-os mais flexíveis e, em última instância, reutilizáveis. Isto refere-se aos projetos futuros, uma vez que estes serão bem-sucedidos, pois podem ser baseados nas experiências dos projetos anteriores (GAMMA, ET ALL, 2008).

Existem várias maneiras de definir um modelo de software, entretanto, as mais comuns são provenientes da perspectiva de um algoritmo, ou da perspectiva orientada a objetos.

A visão tradicional de desenvolvimento de software adota a perspectiva de um algoritmo, onde o foco é a construção de uma função principal para o desenvolvimento dos sistemas. Nessa perspectiva os desenvolvedores focam em questões referentes ao controle e na decomposição de algoritmos maiores. Este tipo de modelo tem a tendência a tornar os sistemas instáveis e de difícil manutenibilidade.

A visão contemporânea adota a perspectiva orientada a objetos, na qual o foco de desenvolvimento dos sistemas se dá através da utilização de objetos ou classes. Estes irão fazer a interação com a camada de banco de dados, interface do usuário, interface intermediária e, por fim, visões de alto nível, relacionadas com as entidades do banco de dados.

2.5.1 UML

A UML é uma linguagem visual utilizada para auxiliar no processo de modelagem de softwares, baseados no paradigma de orientação a objetos. Ela permite especificar a construção do software e documentar os seus artefatos. Atualmente a UML está sendo adotada como padrão pela indústria de Engenharia de Software, em um âmbito internacional (GUEDES, 2009).

Segundo Fowler (2005, p.25), a UML pode ser definida como “uma família de notações gráficas, apoiada por um metamodelo único, que ajuda na descrição e no projeto de sistemas de software, particularmente daqueles construídos utilizando o estilo orientado a objetos”

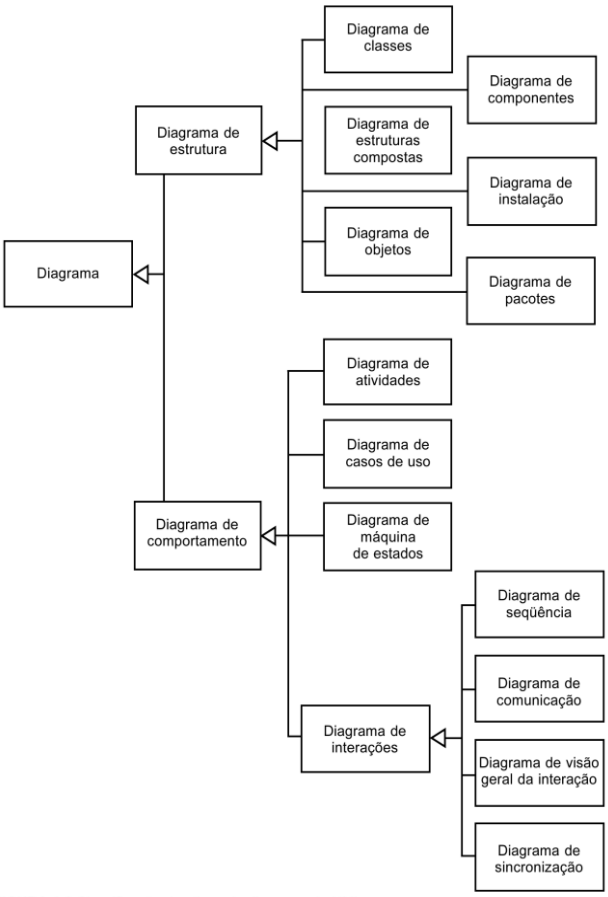
Deve-se lembrar que a UML não é uma linguagem de programação, e sim, de modelagem de software. Tem como objetivo, auxiliar engenheiros de software a definir as características do sistema, como os requisitos, comportamento, estrutura lógica, processos, e até mesmo, a estrutura física necessária para a implantação do sistema. Estas características podem ser definidas através do uso da UML, antes mesmo de começar o desenvolvimento do software. Deve-se destacar que, a UML não é um processo pré-definido de desenvolvimento de software, não estando ligada a uma forma exclusiva, sendo totalmente independente e podendo ser utilizada por diversos tipos de processos de software (GUEDES E GILLEANES, 2009).

Os diagramas da UML são genéricos, pois permitem documentar e especificar softwares de uma maneira simples, não dependendo da linguagem de programação escolhida para o desenvolvimento do mesmo. Um diagrama de classes feito em UML, pode ser implementado em qualquer linguagem, seja ela PHP, Java ou Python, dentre outras.

Conforme Guedes (2009), a UML teve a sua origem na união de outros três métodos de modelagem: Booch, *Object Modeling Tecnique* (OMT) e o *Object-Oriented Software Engineering* (OOSE). Até meados da década de 1990, estes métodos de modelagem orientada a objetos eram os mais utilizados pelos profissionais de desenvolvimento de software. A primeira versão da UML foi lançada em 1996, tendo o apoio de empresas da área de desenvolvimento de software para aprimorar e ampliar a linguagem. Em 1997, a UML foi adotada como linguagem padrão de modelagem pela OMG (*Object Management Group*). A versão 2.0 da UML foi lançada em julho de 2005, e atualmente encontra-se na versão 2.5.

A UML é composta por 13 diagramas, que estão divididos em diagramas de estrutura, e diagramas de comportamento, conforme demonstrado na Figura 9.

Figura 9 - Diagramas da UML



Fonte: Fowler (2005 p. 34).

São diagramas de estrutura dentro da UML:

a) Diagrama de Classes: que permite a visualização das classes de um sistema com seus respectivos atributos e métodos, demonstrando como as classes se relacionam, se complementam e transmitem informações. Apresenta uma visão estática das classes, definindo a estrutura lógica do sistema. Este tipo de diagrama é a base para muitos outros diagramas da UML (GUEDES; GILLEANES, 2009);

b) Diagrama de estruturas compostas: este diagrama pode ser utilizado para modelar colaborações e também para descrever a estrutura interna de um classificador. Possui uma semelhança com o diagrama de classes, porém apresenta uma visão estática da estrutura de classes (GUEDES; GILLEANES, 2009);

c) Diagrama de Objetos: tem como objetivo fornecer a visualização dos valores armazenados pelos objetos de uma classe, em um determinado momento do sistema, representando assim uma instância da classe (GUEDES; GILLENAES, 2009);

d) Diagrama de Componentes: tem como objetivo identificar os componentes que fazem parte de um sistema, subsistema, ou até mesmo de uma classe, podendo ser um componente lógico ou físico;

e) Diagramas de instalação: são utilizados para demonstrar o *layout* físico de um sistema, onde são apresentados em que parte do hardware, o software será executado (FOWLER, 2005);

f) Diagrama de Pacotes: descreve a forma como os elementos do modelo estão organizados em pacotes, demonstrando as dependências entre eles. Pode ser utilizado na modelagem de subsistemas, e demonstrar a arquitetura de uma linguagem (GUEDES; GILLEANES, 2009).

São diagramas de comportamento dentro da UML:

a) Diagrama de Atividades: é utilizado para modelar atividades, descrevendo a lógica dos processos, regras de negócio, fluxo de trabalho e algoritmos. Possui

uma grande semelhança com os fluxogramas, mas se difere por suportar comportamento paralelo (FOWLER, 2005);

b) Diagrama de Casos de Uso: possibilita a compreensão do comportamento externo do sistema. É utilizado no início da modelagem de sistemas, principalmente na etapa de levantamento e análise de requisitos. Tem como objetivo, apresentar uma visão externa geral das funcionalidades que o sistema deve ter (GUEDES; GILLEANES, 2009);

c) Diagrama de Máquina de Estados: utilizado para descrever o comportamento de um objeto. Na utilização da orientação a objetos, este diagrama demonstra o comportamento do ciclo de vida de um único objeto (FOWLER, 2005);

d) Diagrama de Sequência: tem como objetivo capturar o comportamento de um cenário, determinando a ordem em que os eventos ocorrem, mensagens enviadas, métodos que são chamados e como os objetos interagem dentro de um processo (GUEDES; GILLEANES, 2009);

e) Diagrama de Comunicação: este diagrama está amplamente associado ao diagrama de Sequência, entretanto, não está preocupado com a temporalidade do processo, mas concentra-se na maneira em que os elementos do diagrama estão vinculados e quais mensagens trocam entre si durante o processo (GUEDES; GILLEANES, 2009);

f) Diagrama de visão geral de interação: este diagrama é uma mistura entre os diagramas de atividade e de sequência. Este diagrama pode ser considerado um diagrama de atividade, onde as atividades são substituídas por pequenos diagramas de sequência (GUEDES; GILLEANES, 2009);

g) Diagrama de sincronização: possui algumas semelhanças com o diagrama de máquina de estados, entretanto, ele foca nas mudanças de estado de um objeto ao longo do tempo (GUEDES; GILLEANES, 2009).

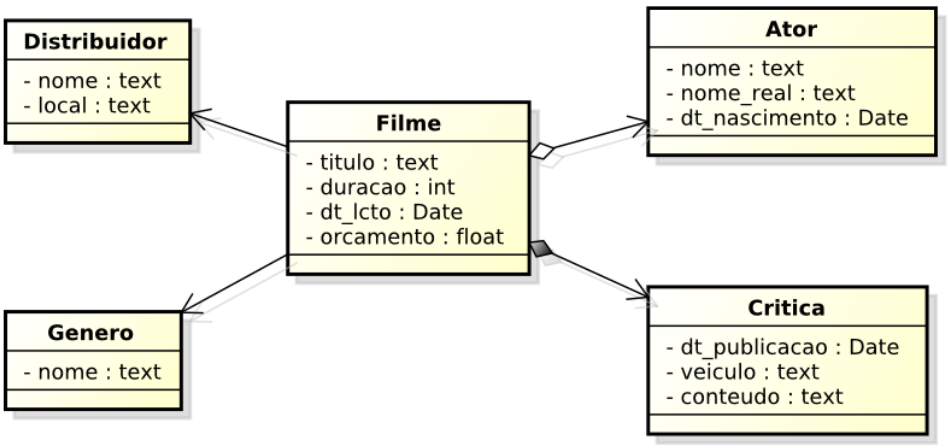
Como já citado, a UML possui 13 diagramas. O presente trabalho tem como foco a geração de aplicações, a partir do modelo de arquitetura de software. Assim, o foco será dado no modelo de classes, que é justamente o modelo que possui essa finalidade, dentro dos diagramas da UML.

2.5.1.1 Diagrama de classes

Segundo Guedes (2009), o diagrama de classes é o mais utilizado na UML sendo considerado como um dos mais importantes entre os 13 diagramas da UML. Tem como objetivo principal permitir a visualização das classes de um determinado sistema, com seus respectivos atributos e métodos, demonstrando como as classes relacionam-se, complementam-se e transmitem informações entre si.

A Figura 10 representa um diagrama de classes, onde é possível observar as classes e seus respectivos atributos.

Figura 10 - Exemplo de um diagrama de classes



Fonte: Feito pelo autor.

Os principais conceitos do diagrama de classes são: classes, atributos, métodos, visibilidade e tipos de relacionamento entre classes. Estes serão apresentados nas seções 2.5.2 e 2.5.3.

2.5.2 Conceitos sobre orientação a objetos

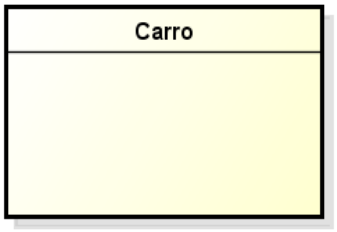
Nesta seção serão apresentados os principais conceitos sobre orientação a objetos, aplicados a diagrama de classes como: classes, atributos, métodos e visibilidade.

2.5.2.1 Classes

Na modelagem de um sistema orientado a objetos, as classes compõem a parte mais importante. Uma classe é um conjunto de objetos que compartilham os mesmos atributos, operações, relacionamentos e semântica. As classes podem ser utilizadas para representar itens de software, hardware e, inclusive itens que sejam puramente conceituais (BOOCH, ET ALL, 2012).

Na UML, uma classe é representada graficamente por um retângulo, que pode ter até três divisões. Na primeira divisão é armazenado o nome da classe que a identifica. A segunda divisão é composta pelos atributados da classe, e a terceira lista, os possíveis métodos dela. É possível encontrar classes que não tenham atributos ou métodos, ou até mesmo nenhuma dessas características, como no caso das classes abstratas (GUEDES; GILLEANES, 2009).

Figura 11 - Exemplo de uma classe



Fonte: Guedes(2009, p. 47) adaptado pelo autor.

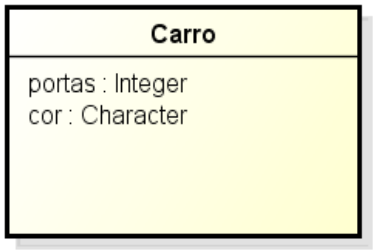
A Figura 11 apresenta o exemplo de uma classe que não possui métodos, nem atributos. Nesta imagem tem-se apenas uma divisão que contém o nome da classe, pois não é obrigatório representar uma classe totalmente expandida, embora seja mais comum (GUEDES; GILLEANES, 2009).

2.5.2.2 Atributos

Em uma classe, os atributos representam as características, peculiaridades que costumam variar de um objeto para outro, como a cor em um objeto da classe carro, e que, permitem diferenciar um objeto de outro da mesma classe, devido às diferentes características (GUEDES; GILLEANES, 2009).

Em uma classe, os atributos são apresentados na segunda divisão, contendo normalmente duas informações, o nome que o identifica, e o tipo de dado que o atributo irá armazenar, como por exemplo: *Integer, Float, String*, dentre outros.

Figura 12 - Exemplo de uma classe com atributos



Fonte: Guedes(2009, p. 48) adaptado pelo autor.

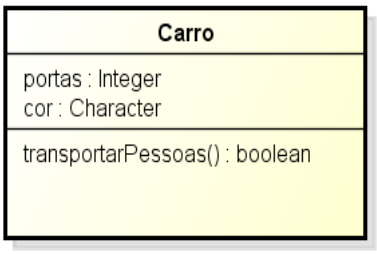
A Figura 12 apresenta o exemplo de uma classe com duas divisões, onde na segunda divisão se encontram os atributos da classe, bem como, seus nomes e respectivos tipos.

2.5.2.3 Métodos

Os métodos representam as funções que uma classe pode executar, ou seja, um conjunto de instruções que são executadas quando o método é chamado. Um exemplo seria um objeto da classe “Carro” que pode executar a atividade de transportar pessoas. A maior parte da codificação dos sistemas de informação orientadas a objetos está contida nos métodos definidos em suas classes.

Em uma classe, os métodos são apresentados na terceira divisão, conforme apresenta a Figura 13, que contém o método transportarPessoas().

Figura 13 - Exemplo de uma classe com método



Fonte: Guedes(2009, p. 49) adaptado pelo autor.

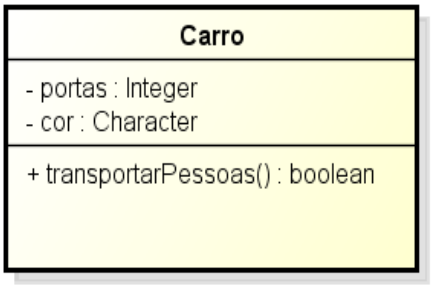
2.5.2.4 Visibilidade

A visibilidade é utilizada para indicar qual é o nível de acessibilidade de um atributo ou método, sendo apresentada à esquerda destes. Os principais modos de visibilidade são:

a) Visibilidade privada: é representada pelo símbolo de menos (-), e significa que somente objetos da própria classe do atributo ou método poderão visualizá-lo ou utilizá-lo;
b) Visibilidade protegida: é representada pelo símbolo de sustenido (#), que determina que os atributos e métodos da classe serão acessíveis pelos seus objetos e por suas subclasses;
c) Visibilidade pública: é representada pelo símbolo de mais (+), que determina se o atributo ou método pode ser utilizado por qualquer objeto, seja ele da própria classe ou não (GUEDES; GILLEANES, 2009).

A Figura 14 demonstra um exemplo de visibilidade, onde os atributos cor e portas são privados, e o método transportarPessoas(), é público.

Figura 14 - Exemplo de visibilidade



Fonte: Guedes(2009, p. 50) adaptado pelo autor.

2.5.3 Relacionamentos

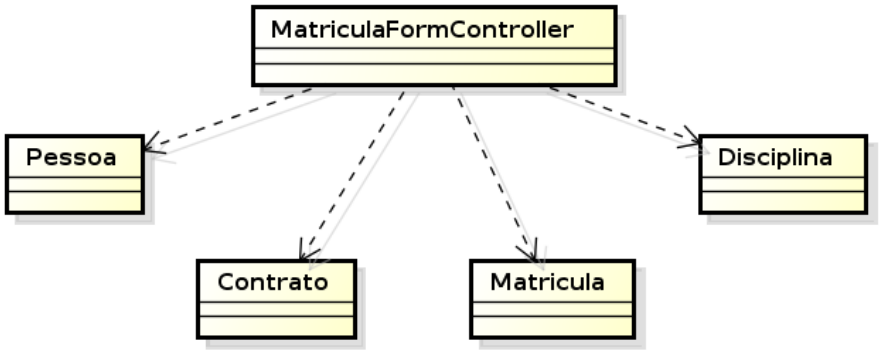
Na modelagem orientada a objetos, existem vários tipos de relacionamento entre classes como: dependência, associação, agregação, composição e herança. Nesta seção, serão apresentados os principais conceitos relativos a relacionamentos entre classes.

2.5.3.1 Dependência

A dependência é um tipo de relacionamento utilizado para identificar quando uma classe depende de outra. Este tipo de relacionamento é identificado através de uma linha tracejada entre as duas classes e uma seta que se encontra na outra extremidade do relacionamento, representando que a classe de origem é dependente da classe de destino (GUEDES; GILLEANES, 2009).

Um exemplo a ser citado, é quando, a partir de uma classe de origem, realiza-se uma chamada de um método da classe de destino. A Figura 15 demonstra um exemplo de dependência.

Figura 15 - Exemplo de dependência



Fonte: Feito pelo autor.

2.5.3.2 Multiplicidade

A multiplicidade tem como objetivo, identificar em um atributo, quantas instâncias de objetos podem preenchê-lo. As multiplicidades mais encontradas segundo Fowler (2005) são:

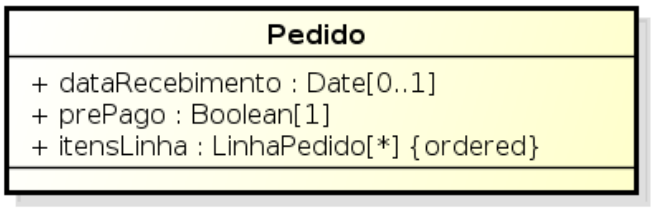
a) (1) - Exatamente uma instância: um pedido deve ter um cliente, ou seja, exatamente um objeto do tipo cliente;

b) (0..1) - Zero ou uma instância: um pedido pode ter, ou não, uma data de recebimento;

c) (*) - Zero ou muitas instâncias: um cliente não precisa fazer um pedido, e não há nenhum limite quanto ao número de pedidos, que ele possa ter.

Conforme demostra a Figura 16, a multiplicidade é encontrada na segunda divisão da classe, após o tipo do atributo.

Figura 16 - Mostrando as propriedades de um pedido como atributos



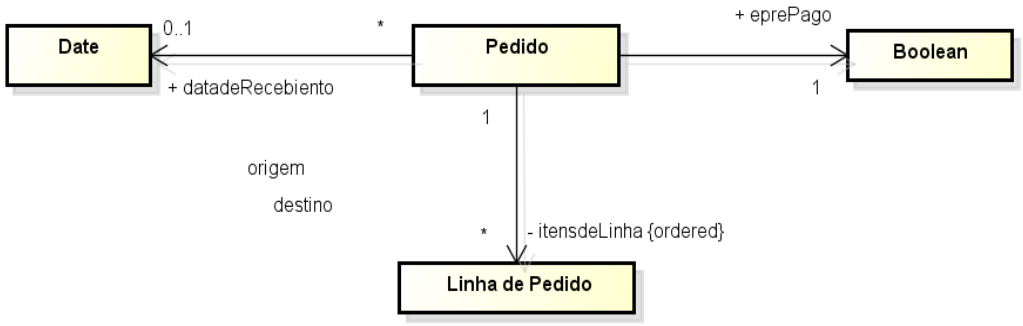
Fonte: Fowler(2005, p. 54) adaptado pelo autor.

2.5.3.3 Associação

Um atributo também pode ser representado através de uma associação, onde o objeto de uma classe está relacionado a uma instância de outra classe. Uma associação é representada por uma linha contínua, entre duas classes, e é direcionada da classe de origem, para a classe de destino. O nome do atributo é representado no destino final da associação junto com a sua multiplicidade. O destino final da associação tem como objetivo, vincular o atributo da classe de origem, com a classe que representa o seu tipo. (FOWLER, 2005)

A Figura 17 apresenta um exemplo de atributos de um pedido, sendo representados por associações.

Figura 17 - Mostrando as propriedades de um pedido como associação

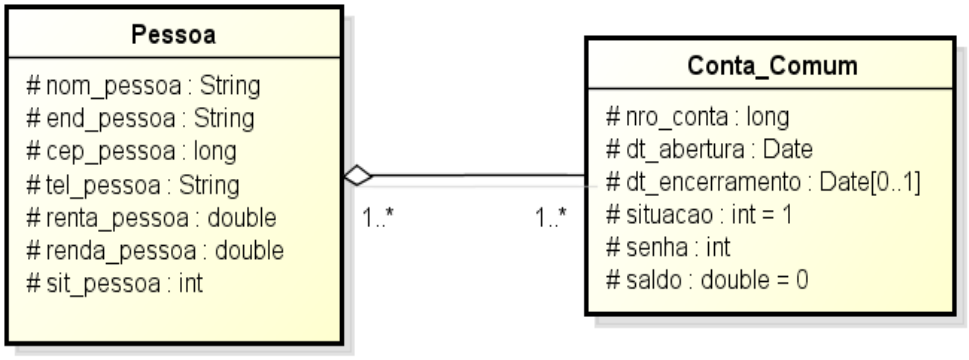


Fonte: Fowler(2005, p. 55)

2.5.3.4 Agregação

A agregação é um tipo de associação utilizada para demonstrar que as informações de um objeto (objetos-todo), precisam ser completadas pelas informações de outro objeto (objetos-parte), isso quer dizer que um objeto, contém outro objeto. A agregação tem como principal função, identificar a obrigatoriedade de um objeto-todo por seus objetos-parte. O símbolo que representa a agregação se difere da associação, pois possui um losango não preenchido na extremidade da classe, que contém os objetos-todo (GUEDES; GILLEANES, 2009).

Figura 18 - Exemplo de agregação



Fonte: Guedes(2009, p. 119)

A Figura 18, demonstra um exemplo de agregação entre uma classe Pessoa, e a classe Conta_Comum, onde é determinado que os objetos da classe Pessoa são os objetos-todo. Estes precisam ter as suas informações completadas pelos objetos da classe Conta_Comum, que são os objetos-parte.

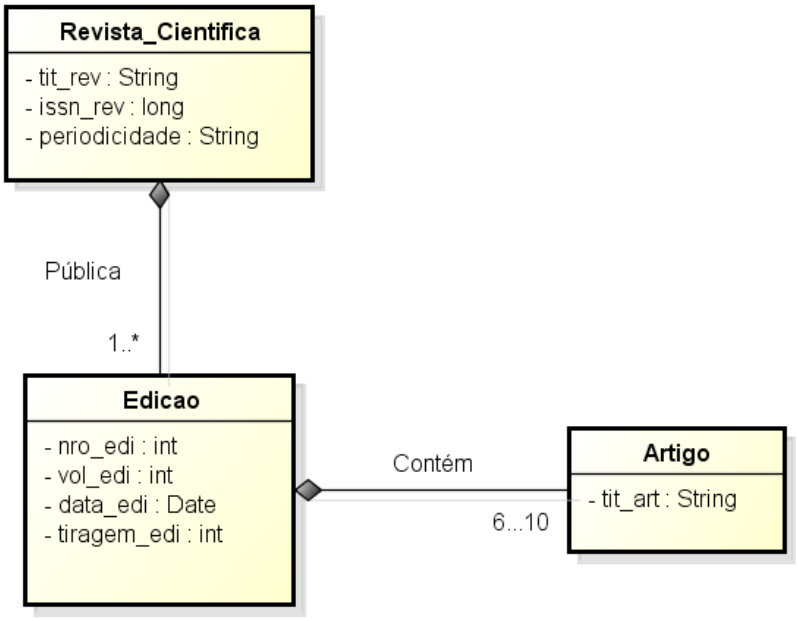
Uma agregação tem como característica, possibilitar que os objetos-parte possam ser compartilhados por mais de um objeto-todo ou vice versa. Um exemplo disso, seria um objeto da classe Pessoa, possuir vários objetos da classe Conta_Comum, ou, um objeto da classe Conta_Cumum, estar relacionado com vários objetos da classe Pessoa.

2.5.3.5 Composição

Uma associação do tipo composição é uma variação da agregação, pois neste tipo de associação os objetos-parte devem estar associados com um único objeto-todo. O símbolo, que representa a composição, se diferencia da agregação por utilizar um losango preenchido.

O losango deve ficar do lado do objeto-todo, da mesma forma que é utilizado na agregação (GUEDES; GILLEANES, 2009).

Figura 19 - Exemplo de composição



Fonte: Guedes(2009, p. 118)

A Figura 19 representa um exemplo de composição, onde é possível perceber que um objeto da classe Revista_Cientifica, refere-se a, no mínimo, um objeto da classe Edicao, e cada instância da classe Edicao, deve ser relacionada, única e exclusivamente, com uma instância específica da classe Revista_Cientifica, a qual não pode se relacionar com nenhuma outra.

2.5.3.6 Herança

Herança é um tipo especial de relacionamento, que tem como objetivo, representar o compartilhamento de atributos ou métodos entre as classes, identificando as superclasses e subclasses (classes especializadas), demonstrando a hierarquia entre as classes e possivelmente, métodos polimórficos nas classes especializadas (GUEDES; GILLEANES, 2009).

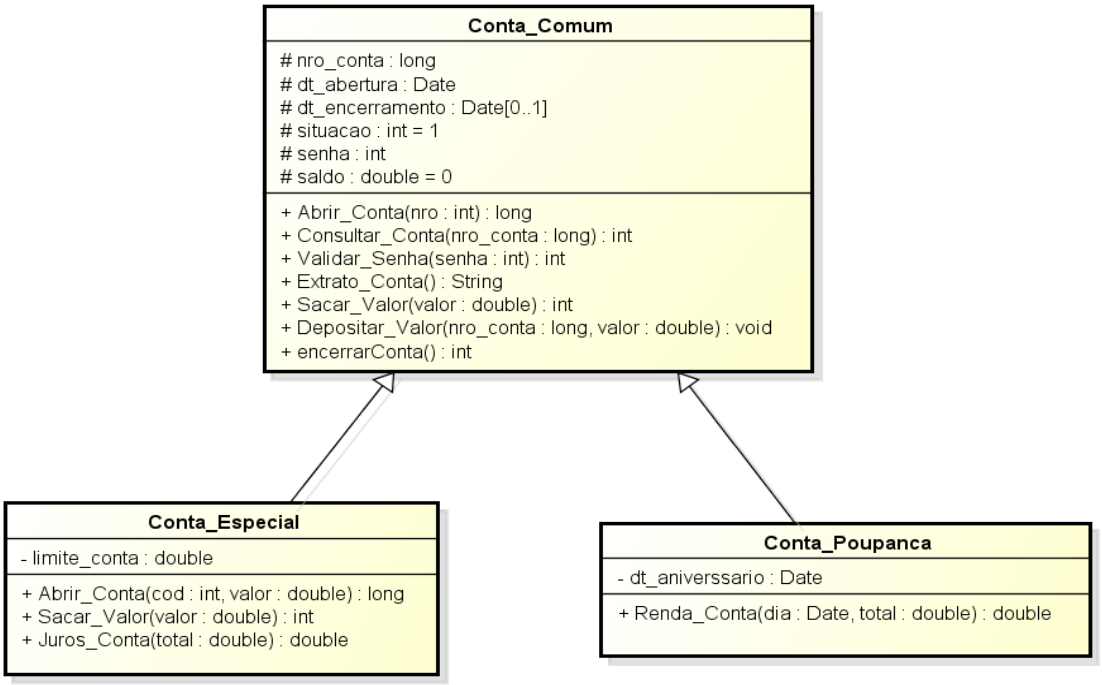
A generalização/especialização ocorre quando há duas ou mais classes com características muito semelhantes, evitando assim, a redeclaração de atributos e/ou métodos idênticos. A utilização desta técnica permite o reaproveitamento de código, pois cria-se uma

classe geral em que são declarados os atributos e métodos comuns de todas as classes envolvidas, e então, são declaradas as subclasses ligadas à superclasse, as quais irão herdar todas as suas características, podendo ter atributos e métodos próprios (GUEDES; GILLEANES, 2009).

Uma subclasse pode redeclarar métodos da superclasse, caso eles tenham o mesmo nome, entretanto, o seu comportamento será diferente. Isso permite que não haja a necessidade de alterar o código fonte do sistema em relação às chamadas das subclasses, pois o nome do método permanece o mesmo da superclasse, o qual irá se comportar de maneira diferente, quando for chamado por um objeto da subclasse.

A Figura 20, apresenta um exemplo de generalização/especialização, representado por três classes, onde a superclasse é a Conta_Comum; e as subclasses dela são a classe Conta_Especial e Conta_Poupanca. O relacionamento entre a subclasse e a superclasse é feito através de uma linha contínua, contendo um triângulo não preenchido, que aponta para a superclasse.

Figura 20 - Exemplo de Generalização/Especialização no diagrama de classes



Fonte: Guedes(2009, p. 119)

2.6 Padrões de projeto

Padrões de projetos, inicialmente foram moldados a partir de ideais apresentadas por Christopher Alexander (ALEXANDER et al., 1977), que sugere, adotar padrões comuns de projeto que eram inerentemente agradáveis e eficazes para construções, os quais podem ser reutilizados diversas vezes (SOMMERVILLE, 2011).

Os primeiros conceitos de padrões de projetos foram criados por Christopher Alexander, que afirma: “cada padrão descreve um problema no nosso ambiente e o cerne de sua solução, de tal forma que se possa usar essa solução mais de um milhão de vezes, sem nunca fazê-lo da mesma maneira” (ALEXANDER, 1977 apud GAMMA et al., 2008).

Segundo Gamma et al. (2008), a definição de padrão de projeto é:

> Um padrão de projeto sistematicamente nomeia, motiva e explica um design geral que endereça um problema recorrente de design em sistemas orientados a objeto. Ele descreve o problema, a solução, quando aplicar a solução e suas consequências. Ele também dá dicas de implementação e exemplos. A solução é um arranjo geral de objetos e classes que resolvem o problema. A solução é customizada e implementada para resolver o problema em um contexto em particular. (GAMMA et al, 2008, p. 333)

A utilização dos padrões de projetos possibilita desenvolver softwares e componentes, de uma maneira que a reutilização dos mesmos ocorra de forma eficiente. O padrão é uma descrição do problema com a essência da sua solução, o que irá possibilitar a reutilização dessa solução por diferentes sistemas. Ele descreve a experiência e o conhecimento acumulado numa solução comprovada para um problema em comum (SOMMERVILLE 2011).

2.6.1 Padrões de projeto clássicos

Gamma et al. (2008) definiu quatro elementos essências para especificação de padrões de projeto, os quais tem como objetivo descrever e documentar um padrão, sendo que estes elementos são:

a) O nome: um padrão deve ter um nome que seja a sua referência, a fim de identificá-lo, permitindo uma maior comunicação sobre ele, dentro da equipe;

b) O problema: visa descrever em que situação deve-se aplicar tal padrão, explicando o problema e o seu contexto;

c) A solução: descreve os elementos que compõem o padrão de projeto, seus relacionamentos, responsabilidades e colaborações. Fornecendo uma descrição abstrata do problema e de como um conjunto de classes e objetos o solucionam;

d) As consequências: são os resultados e análises das vantagens e desvantagens da aplicação do padrão.

Os padrões de projetos foram classificados por Gamma et al. (2008), em famílias, de acordo com as características de cada padrão. A Figura 20 apresenta os 23 padrões de projetos "clássicos", documentados e catalogados por Gamma et al. (2008).

Figura 21 - Padrões de projeto clássicos

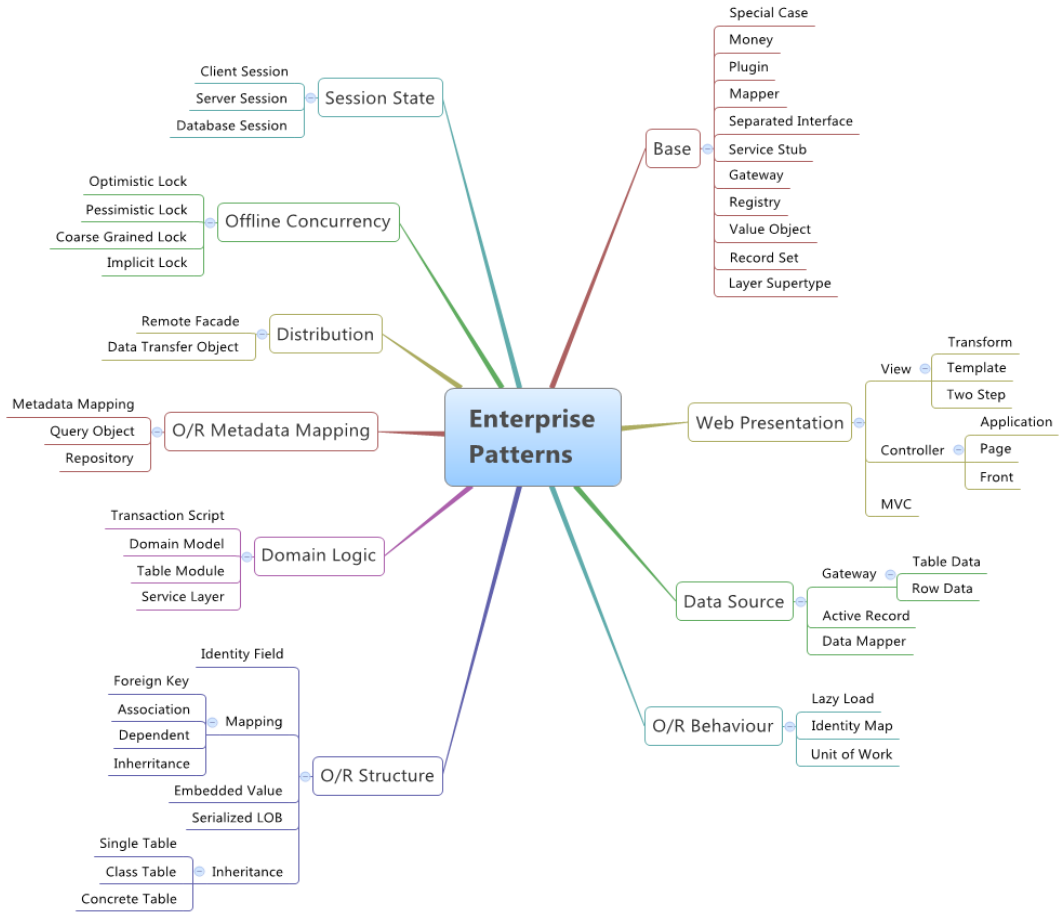
		Propósito		
		De criação	**Estrutural**	**Comportamental**
Escopo	**Classe**	Factory Method	Adapter (class)	Interpreter Template Method
	Objeto	Abstract Method Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsability Command Iterator Mediator Memento Observer State Strategy Visitor

Fonte: Guedes(2009, p. 119)

2.6.2 Padrões de projeto de negócio

Na obra Patterns of Enterprise Application Architecture (Fowler, 2003), Martin Fowler cataloga 51 tipos de padrões de projeto de negócio. A Figura 22 apresenta os 51 padrões em um mapa conceitual.

Figura 22 - Padrões de projeto de negócios



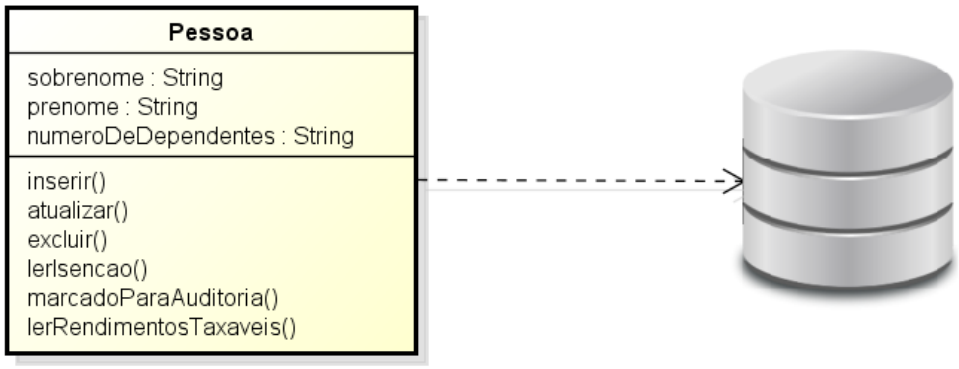
Fonte: Site da internet(http://logogin.blogspot.com.br/2013/03/fowlers-patterns-of-enterprise-mind-map.html)

Nas seções seguintes, serão abordados os principais padrões de projeto, catalogados por Fowler, que possuem forte relação com o desenvolvimento da atual proposta. O critério adotado para a seleção dos padrões foi a relação com a camada de modelo da aplicação.

2.6.2.1 Active Record

Active record é um padrão de projeto que integra a camada de Modelo da aplicação, para encapsular, em um objeto, os dados de uma linha de uma tabela ou visão de um banco de dados, o acesso ao banco de dados e, adicionar lógica de domínio a esses dados. A Figura 23 representa uma classe que utiliza o padrão *Active Record* (FOWLER, 2003).

Figura 23 - Padrão *Active Record*



Fonte: Fowler(2003, p. 160)

O objeto carregará os dados e o seu comportamento. A maioria desses dados são persistentes e precisam ser gravados em um banco de dados. O *Active Record* utiliza o método mais óbvio, que é colocar a lógica de acesso junto com o objeto de domínio. Desta forma, os objetos da classe Pessoa sabem como ler e gravar os seus dados, no banco de dados.

A essência do *Active Record* está na semelhança que as classes têm com a estrutura das tabelas correspondentes a elas no banco de dados. Cada *Active Record* é responsável por salvar e buscar seus dados no banco de dados e, por qualquer outra lógica de domínio necessária. A estrutura de uma classe de *Active Record* deve corresponder à tabela do banco de dados, onde deve-se ter um atributo na classe para cada coluna na tabela. Lembrando que, o tipo dos atributos, deve ser o mesmo das colunas.

A classe Active Record, geralmente possui métodos que fazem as seguintes funções:

a) Constroem a instância de um *Active Record* a partir dos dados resultantes de uma consulta SQL(*Structured Query Language*);

b) Constroem uma nova instância, para posterior inserção em uma tabela;

c) Métodos de busca estáticos que encapsulam consultas SQL e retornam objetos do tipo *Active Record*;

d) Atualizam a base de dados e inserem os dados do *Active Record*;

e) Gravam e leem atributos (*get and sets*);

f) Implementam algumas lógicas de negócio.

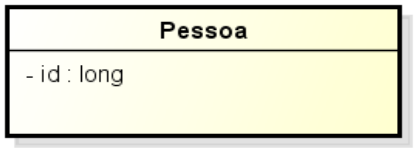
Além dos métodos utilizados para realizar a leitura, a gravação de atributos, os *Active Records* podem realizar outras operações de mais alto nível, como converter os tipos de dados orientados a SQL para um tipo mais apropriado para o usuário. Ao solicitar ao método de leitura uma tabela relacionada, tal método pode retornar o Active Record apropriado.

Fowler (2003) sugere a utilização do *Active Record* quando a lógica de domínio não for muito complexa, e, para evitar problemas de duplicação de código.

2.6.2.2 Identity Field

O *Identity Field* é um padrão utilizado para manter os dados de um objeto, em memória, relacionados com uma linha do banco de dados. O *Identity Field* irá armazenar o campo "id", que é a chave primária da tabela do banco de dados. A Figura 24 representa a aplicação desse padrão em uma classe.

Figura 24 - Classe utilizando o *Identity Field*



Fonte: Fowler(2003, p. 216)

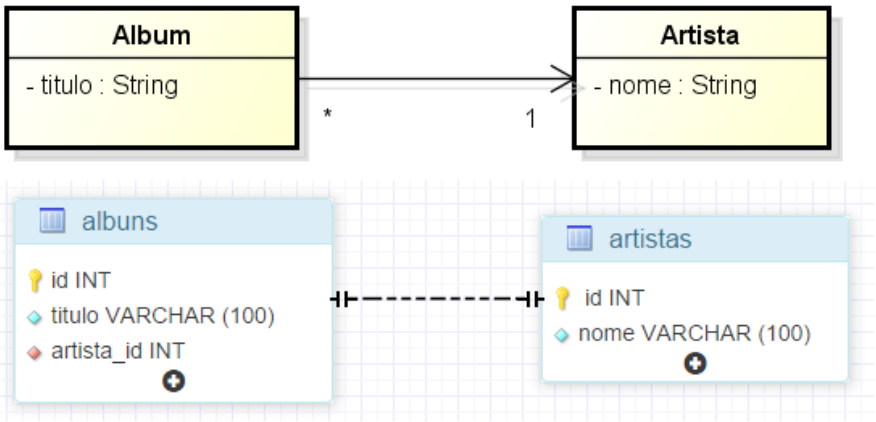
Para diferenciar os registros de uma linha de outra nos bancos de dados relacionais, é utilizada uma chave, geralmente a chave primária. Entretanto, nas linguagens de programação que possuem suporte a objetos, estes não precisam de tal chave, pois elas asseguram a identidade correta do objeto. Não há problemas em relação à leitura dos dados do banco de dados, mas sim, na hora de gravá-los, pois é necessário vincular as linhas da tabela banco de dados com objetos que estão em memória (FOWLER, 2003).

O *Identity Field* é muito utilizado na implementação de um ORM (*Object Relational Mapping*), como o *Active Record*. O padrão é usado para vincular a representação relacional com a representação orientada a objetos dos dados.

2.6.2.3 Foreigen Key Mapping

No modelo orientado a objetos, os relacionados entre os objetos são realizados através de uma referência. O padrão *Foreign Key Mapping*, permite realizar o mapeamento dessa referência, para então, obter as respectivas chaves estrangeiras, entre as tabelas do banco de dados, que estes objetos representam. A Figura 25 demonstra este padrão, onde podem ser visualizadas as classes acima, e as tabelas abaixo (FOWLER, 2003).

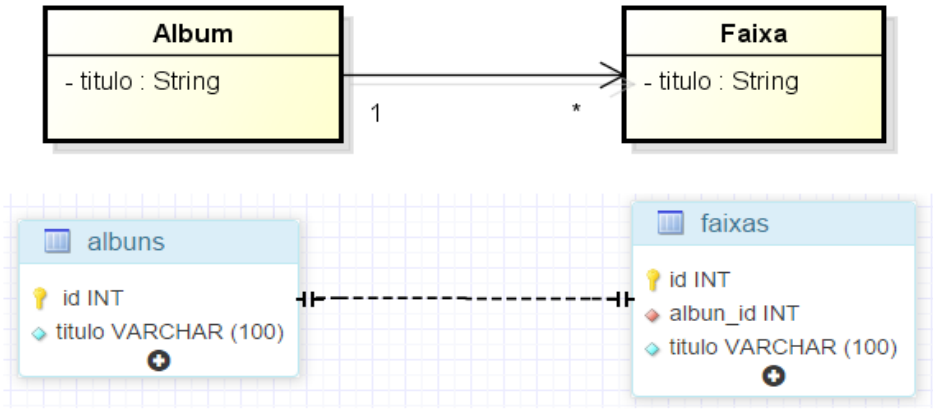
Figura 25 - Exemplo da utilização do padrão *Foreign Key Mapping*



Fonte: Fowler(2003, p. 236)

Em sistemas orientados a objetos, sejam eles simples ou complexos, há grupos de objetos que estão diretamente conectados entre si, através de referência. Para ser possível a gravação desses objetos em um banco de dados, deve-se armazenar essas referências. Entretanto, como os dados dessas referências são instâncias de outros objetos, não é possível gravar o valor destes dados em estado bruto. Outro problema ocorre quando, estes objetos possuem uma coleção de referências para outros objetos, conforme representado na Figura 26. Essa estrutura de objetos, acaba violando a forma na qual os bancos de dados relacionais funcionam.

Figura 26 - Classes e tabelas para uma relação multivalorada



Fonte: Fowler(2003, p. 238)

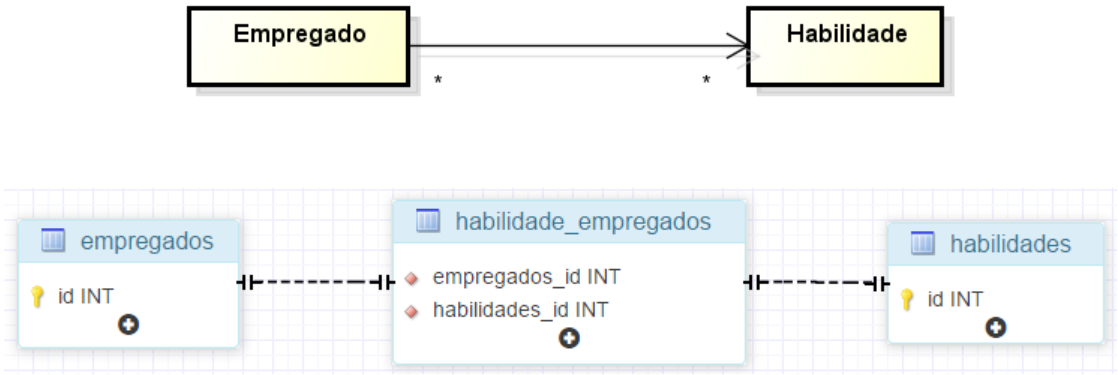
Segundo Fowler, esses problemas podem ser resolvidos utilizando o padrão *Identity Field*, onde cada objeto irá armazenar a chave primária da respectiva tabela do banco de dados. Ao haver dois objetos conectados, por meio de uma associação, esta deve ser substituída por uma chave estrangeira no banco de dados, e também, o valor dessa chave estrangeira deve ser armazenada em um *Identity Field*, para posterior utilização.

Fowler (2003), sugere a utilização do padrão *Foreign Key Mapping* em quase todas as associações entre classes. Entretanto há um caso onde não é possível utilizá-lo, que é o caso da associação muitos-para-muitos. Para este tipo de associação deve-se utilizar o padrão *Association Table Map* que será visto na próxima seção.

2.6.2.4 Association Table Map

Este padrão é utilizado em associações muitos-para-muitos, onde a associação é gravada como uma tabela, a qual utiliza chaves estrangeiras, para as tabelas que estão vinculas pela associação. A Figura 27 demonstra esta associação.

Figura 27 - Exemplo da associação



Fonte: Fowler(2003, p. 238)

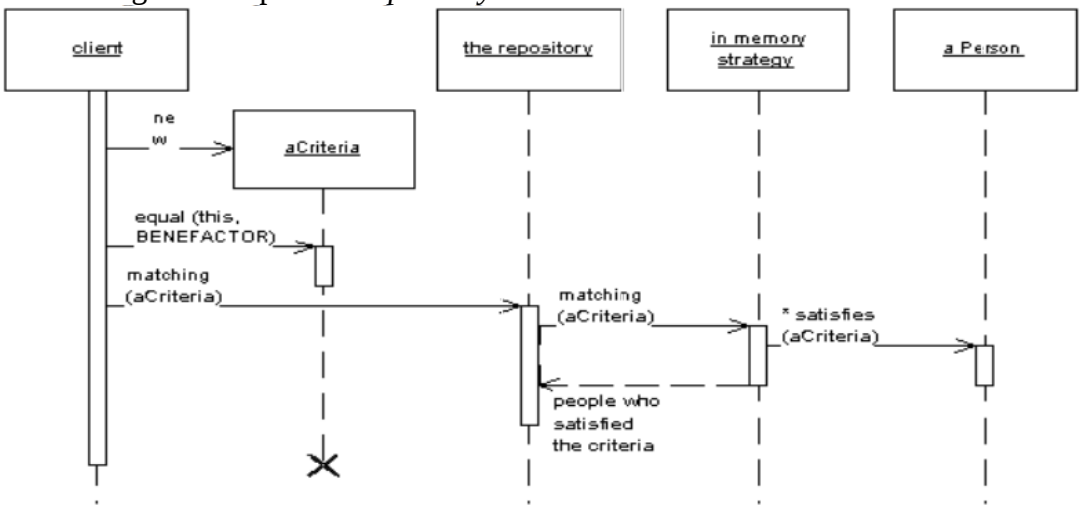
Em linhas gerais, bancos de dados relacionais não conseguem armazenar campos multivalorados para guardar a referência de outras tabelas. Já os objetos, conseguem manipular com grande facilidade campos multivalorados, utilizando coleções como valores de campos. Quando se tem uma associação de um-para-muitos, o mapeamento é feito através de *Foreign Key Mapping*, como visto anteriormente, que armazena a chave estrangeira no lado univalorado da associação. Já em uma associação de muitos-para-muitos, não é possível fazer isso, pois não há um lado univalorado para armazenar a chave estrangeira (FOWLER, 2003).

Segundo Fowler (2003), este problema pode ser resolvido utilizando o padrão *Association Table Map*, o qual utiliza uma tabela auxiliar para armazenar essa associação. Nesta tabela são armazenados os IDs das chaves estrangeiras, para as duas tabelas vinculadas a ela, possuindo assim, uma linha para cada par de objetos.

2.6.2.5 Repository

O objetivo principal deste padrão é isolar a lógica de acesso a dados de qualquer outra lógica do software, criando assim, uma camada responsável pela comunicação entre a camada de mapeamento de dados e a camada de domínio da aplicação. A camada criada pelo *Repository* trata os objetos persistidos no formato de coleção e concentra o código das consultas ao banco de dados (FOWLER, 2003).

Figura 28 - Diagrama do padrão *Repository*



Fonte: Fowler(2003, p. 238)

O *Repository* irá fornecer uma interface para realizar a manipulação e persistência dos dados de uma maneira mais orientada a objetos, onde cada tabela ou conjunto de tabelas é tratado como uma coleção de objetos em memória, permitindo realizar as operações de inserção, edição e exclusão de dados.

Este padrão irá auxiliar na separação da camada de mapeamento de dados, da camada de domínio, diminuindo a dependência entre elas. Essa separação das camadas, permite abstrair a forma de como um sistema irá se comunicar com o banco de dados, podendo então, desenvolver separadamente a camada de mapeamento de dados, informando apenas uma interface para o *Repository*, permitindo assim, trocar a forma como o *Repository* acessa os dados.

A utilização desse padrão permite especificar as consultas ao banco de dados de um sistema, de uma forma totalmente orientada a objetos. São criados objetos com critérios, para filtrar os registros do banco de dados, sem ter a necessidade de escrever as consultas SQL.

Fowler (2003), sugere a utilização do padrão *Repository* em sistemas grandes e complexos, os quais geralmente possuem muitos objetos de domínio e demandam muitas consultas. Através dele é possível reduzir a quantidade de códigos que realizam tais consultas, concentrando-os em um mesmo local, minimizando assim a duplicidade de código, levando em consideração que isso será feito de uma forma encapsulada e orientada a objetos.

3 METODOLOGIA

Para a realização do presente trabalho, é essencial definir e seguir uma metodologia. Conforme Marconi e Lakatos (2003), “Todas as ciências caracterizam-se pela utilização de métodos científicos; em contrapartida, nem todos os ramos de estudo que empregam estes métodos são ciências”. Ainda segundo Marconi e Lakatos (2003), “podemos concluir que a utilização de métodos científicos não é da alçada exclusiva da ciência, mas não há ciência sem o emprego de métodos científicos”. O presente trabalho irá utilizar um método científico que será apresentado neste capítulo.

3.1 Delineamento

Segundo Wainer (2007, p.1), “pesquisa em Ciência da Computação envolve, na maioria dos casos, a construção de um programa, de um modelo, de um algoritmo ou de um sistema novo”. Embora o fato de criar um sistema não seja suficiente para conceder o mérito à pesquisa, a partir disso é necessário avaliar a ferramenta de forma metodológica (Wainer, 1999).

Conforme apresentado anteriormente, o presente trabalho buscará construir uma plataforma *online*, colaborativa e extensível para a criação e modelagem estrutural de software, que possibilitará a interoperabilidade, com diferentes modelos de estruturas de software. Para tal, serão realizados estudos sobre os temas relacionados (modelagem, padrões de projeto). A partir da exploração destes temas, que possa-se desenvolver uma ferramenta, em conformidade com os conceitos pré-existentes. Desta forma, este trabalho caracteriza-se, conforme Santos (1999), e segundo o seu objetivo, como pesquisa exploratória.

> Explorar é tipicamente a primeira aproximação de um tema e visa criar maior familiaridade a um fato ou fenômeno. Quase sempre busca-se essa familiaridade pela prospecção de materiais que possam informar ao pesquisador a real importância do problema, o estágio em que se encontram as informações já disponíveis a respeito do assunto, e até mesmo, revelar ao pesquisador novas fontes de informação (SANTOS, 1999, p.26).

Este trabalho não tem como objetivo realizar uma pesquisa para quantificar dados com uma grande volume de usuários, de forma a gerar dados estatísticos, mas sim, efetuar um estudo aprofundado sobre os temas que dão embasamento ao trabalho; estabelecer a sua relação com a prática, propondo uma solução, e, avaliando a mesma, de maneira qualitativa. Portanto, este estudo trata-se de uma pesquisa qualitativa que, segundo Wainer (2007), é definida como:

> Métodos qualitativos diferem de métodos quantitativos, porque se ocupam de variáveis que não podem ser medidas, apenas observadas. Essa é uma dicotomia muito simplista. Métodos qualitativos vêm das ciências sociais, em oposição aos métodos quantitativos que derivam das ciências naturais. Essa diferença na origem já é suficiente para que visões diferentes sobre o que é ciência, e como se faz ciência, tornem definições sucintas sobre o que é um ou outro método muito difícil. De um modo geral, métodos qualitativos em Ciência da Computação são métodos que se caracterizam por ser um estudo aprofundado de um sistema no ambiente onde ele está sendo usado, ou, em alguns casos, onde se espera que o sistema seja usado. Métodos qualitativos sempre envolvem pessoas, e na maioria das vezes sistemas (WAINER, 2007, p.27).

Para que a plataforma proposta estivesse em conformidade com os conceitos já estabelecidos e, em relação às teorias relacionais e orientada a objetos, foi necessário utilizar o método de pesquisas bibliográficas, que segundo Fonseca (2002):

> A pesquisa bibliográfica é feita a partir do levantamento de referências teóricas já analisadas, e publicadas por meios escritos e eletrônicos, como livros, artigos científicos, páginas de web sites. Qualquer trabalho científico inicia-se com uma pesquisa bibliográfica, que permite ao pesquisador conhecer o que já se estudou sobre o assunto. Existem porém pesquisas científicas que se baseiam unicamente na pesquisa bibliográfica, procurando referências teóricas publicadas com o objetivo de recolher informações ou conhecimentos prévios sobre o problema a respeito do qual se procura a resposta. (Fonseca, 2002, p. 32)

A plataforma proposta será utilizada e testada em um ambiente controlado, tendo como base, projetos fictícios de software, que serão utilizados como cenários de teste. Sendo assim, a sua utilização se dará em um ambiente de laboratório. Para Santos (1999, p.31), o ambiente de laboratório é caracterizado como:

> Ambiente de laboratório é um espaço e momento de uma pesquisa caracterizada por duas situações: a interferência artificial na produção do fato/fenômeno ou a artificialização de sua leitura, geralmente melhorando as capacidades humanas naturais de percepção. Com efeito, os fatos/fenômenos que acontecem na realidade, no campo, muitas vezes escapam ao padrão desejável de observação. Por isso são reproduzidos de forma artificial e controlada, e permitem assim captação adequada para descrição e análise. Outras vezes, os mecanismos naturais de observação se mostram insuficientes, seja em alcance, seja em acuidade. Daí a necessidade de artificializar o ambiente ou os mecanismos de percepção, para que o fato/fenômeno seja produzido/percebido adequadamente.

Após análise dos resultados obtidos, a partir dos testes em laboratório, busca-se validar a plataforma desenvolvida por este trabalho. Os resultados obtidos serão analisados e as devidas conclusões serão apresentadas.

4 ESPECIFICAÇÃO DO PROJETO

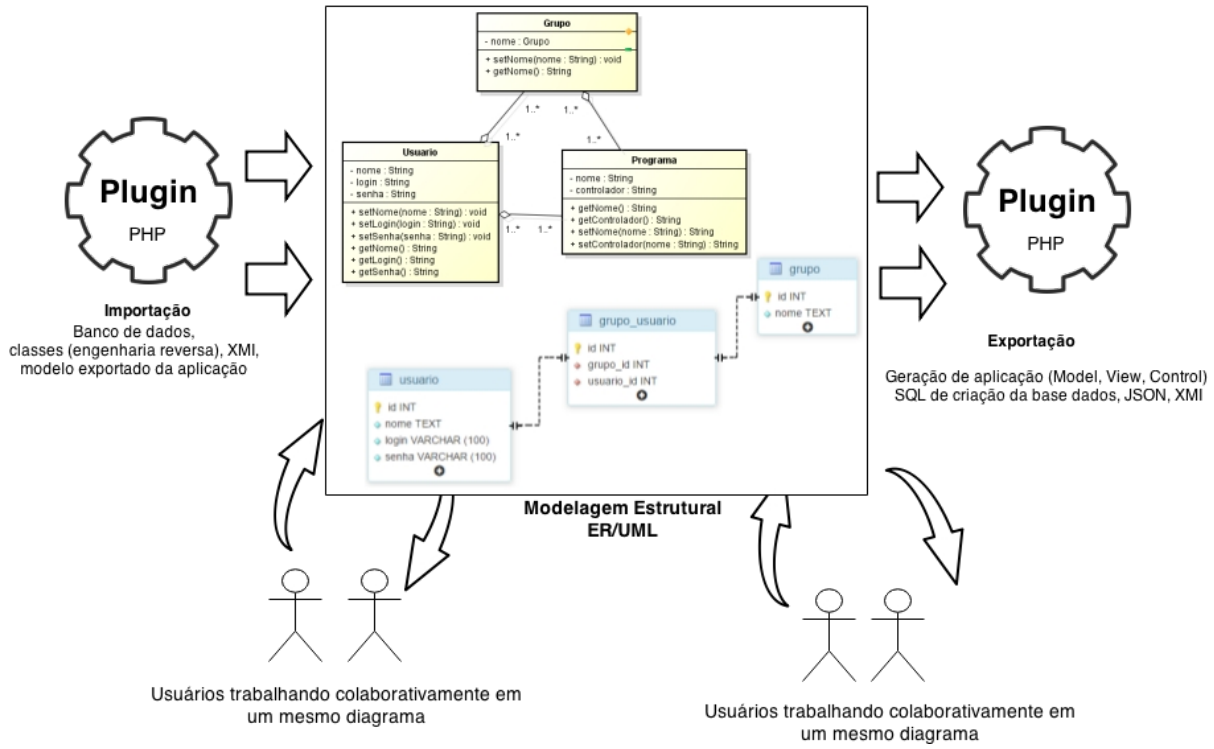
O objetivo do presente capítulo é apresentar o projeto proposto, desde a sua visão geral, casos de uso, cenários de utilização, bem como, a especificação de suas classes.

4.1 Visão geral do projeto

A plataforma proposta deve permitir a modelagem de aplicações orientadas a objetos (UML), e de estruturas de bancos de dados relacionais (ER), de forma *online* e colaborativa, permitindo a interoperabilidade com diferentes modelos de estrutura de software. Para tal, a plataforma proposta permite a inserção de *Plugins*, que permitem a importação e exportação de diversos tipos de modelos. Os *plugins* de importação podem ser construídos para importar dados a partir de classes, conexões com bancos de dados, fazendo uma espécie de engenharia reversa, para um modelo de classes UML ou ER. *Plugins* de exportação podem ser construídos para gerar uma aplicação (*model*, *view*, *control*), SQLs de criação para diferentes bancos de dados, XMI (XML *Metadata Interchange*), JSON (*JavaScript Object Notation*), dentre outros.

Conforme pode ser visto na Figura 29, na parte central, estão representados os diferentes tipos de modelos que podem ser criados pela ferramenta. O núcleo de modelagem da ferramenta pode ser alimentado a partir de um *plugin*, que pode importar diferentes tipos de artefatos de software, como código-fonte, tabelas de um banco de dados (para engenharia reversa), outros. Também existem *plugins* de exportação, que irão ler o modelo construído e gerar artefatos finais para construção do software, como código-fonte, SQL, dentre outros.

Figura 29 - Visão geral da plataforma proposta



Fonte: Feito pelo autor

A seguir, serão listados os principais grupos de requisitos funcionais que foram desenvolvidos para a plataforma:

a) Diagramação UML: a plataforma deve permitir a modelagem de diagrama de classes, englobando a criação de classes, definição de atributos e métodos, bem como relacionamentos entre classes (associação, agregação, composição);

b) Diagramação ER: a plataforma deve permitir a modelagem de estrutura de banco de dados relacionais, englobando a criação de tabelas, atributos e definição de chaves primárias e estrangeiras;

c) Edição colaborativa: a plataforma deve permitir a seus usuários o compartilhamento de diagramas entre si, bem como a edição simultânea;

d) Importação do arquivo JSON: a plataforma deve permitir a importação de um arquivo JSON, de formato da própria ferramenta;

e) Exportação do modelo para JSON: a plataforma deve ter um mecanismo de exportação do modelo (ER ou UML), para um arquivo próprio no formato

JSON (JavaScript *Object Notation*). O arquivo JSON resultante, pode ser utilizado pelo projetista para gerar outros modelos;

f) *Plugins* de importação: a plataforma deve permitir criar *plugins* de importação, que podem ser desenvolvidos pelos usuários da plataforma, utilizando a linguagem de programação PHP. Estes *plugins* tem como objetivo, gerar um objeto no formato JSON padrão da plataforma, o qual terá em sua estrutura, uma representação de um modelo de classes UML ou ER. Através destes *plugins*, os usuários são capazes de importar arquivos de classes de modelo, SQLs de criação de base de dados, XML, XMI, modelos de outras ferramentas. Para ajudar na criação do *plugin*, a plataforma virá com um exemplo de importação de base de dados que gerará o modelo ER e UML a partir dele;

g) *Plugins* de exportação: a plataforma deve permitir criar *plugins* de exportação, que podem ser desenvolvidos pelos usuários da plataforma, utilizando a linguagem de programação PHP. Estes recebem um JSON como parâmetro, o qual terá em sua estrutura todo o diagrama. Com base no JSON, os *plugins* são capazes de gerar SQLs de criação de base de dados, arquivos para aplicações (*model, view, controller*), XML, XMI, bem como qualquer outro modelo. Para ajudar na criação do *plugin*, a plataforma virá com dois exemplos de *Plugins*: SQL de criação de base de dados no formato MySql e geração da camada de modelo da aplicação (*model*) para o *framework* PHP Adianti[1]. Os *plugins* são submetidos a análise para posterior implementação na plataforma.

Para o desenvolvimento da plataforma, foi necessário utilizar diversas tecnologias existentes, que serão detalhadas no tópico 4.2. A seguir, serão listados os principais requisitos não funcionais atendidos pela plataforma:

a) Compatibilidade com browsers Mozilla Firefox e Google Chrome;

b) Utilização do servidor de páginas Apache HTTP server;

c) Utilização da linguagem de programação PHP no back-end;

d) Utilização do banco de dados Mysql para armazenar os diagramas criados pelos usuários;

[1]http://www.adianti.com.br/framework

e) Desenvolvimento do Front-end, seguindo os padrões HTML5, e CSS3;

f) Utilização da linguagem de programação Javascript no front-end;

g) Utilização das bibliotecas JQuery e Bootstrap no front-end;

h) Utilização da biblioteca JSPlumb para conectar os elementos do diagrama no front-end;

i) Utilização do WebSocket com JavaScript no front-end para permitir a edição colaborativa;

j) Utilização da biblioteca PHP Ratchet no back-end para realizar a comunicação com o WebSocket, e permitir a edição colaborativa.

4.2 Tecnologias utilizadas

Para possibilitar a construção da plataforma, foram utilizadas várias ferramentas, as quais tinham como propósito, auxiliar no desenvolvimento do *front-end* e *back-end* da plataforma.

Para o desenvolvimento do *back-end* foram utilizadas as seguintes tecnologias:

a) PHP: é uma linguagem de código aberto, interpretada com o propósito para o desenvolvimento web, é rápida, flexível e pragmática. O PHP[2] (PHP *Hypertext Preprocessor*) é utilizado para processar as requisições javascript feitas pelos usuários, gravar e recuperar os diagramas feitos a partir da plataforma, no banco de dados (PHP, 2014);

b) Apache HTTP Server: é um software de código aberto, que tem como objetivo servir páginas HTML à web, através de requisições feitas pelos usuários. O Apache[3] é utilizado para processar as requisições de arquivos PHP feitas pelos usuários, fazendo com que eles consigam ter acesso à plataforma *online* (Apache, 2014);

[2]http://www.php.net
[3] http://httpd.apache.org

c) MySql: é um sistema gerenciador de banco de dados robusto, de código-fonte aberto, que oferece multiprocessamento, multiusuário e velocidade. Está sendo utilizado para armazenar os diagramas feitos pelos usuários (MySql, 2014);

d) Ratchet: é uma biblioteca PHP, de baixo acoplamento, que fornece ferramentas para criar aplicações em tempo real, bidirecionais entre clientes e servidores, através de *WebSockets*. O Ratcchet[4] está sendo utilizado na diagramação colaborativa entre os usuários da plataforma (Ratchet, 2014).

No desenvolvimento do *front-end*, serão utilizadas as seguintes tecnologias e bibliotecas:

a) HTML: é a linguagem de marcação da internet, utilizada para desenvolver as páginas na web, a qual é interpretada pelos navegadores. O HTML[5] (*HyperText Markup Language*) foi utilizado para criar a interface da plataforma (HTML, 2014);

b) CSS: é uma linguagem de folhas de estilo, utilizada para definir a apresentação visual das páginas da web. O CSS[6] (*Cascading Style Sheets*) foi utilizado para melhorar a apresentação, estilo da interface para os usuários da plataforma (CSS, 2014);

c) JavaScript: é uma linguagem de programação, interpretada e dinâmica, utilizada originalmente nos navegadores, e desenvolvida inicialmente para executar scripts do lado do cliente. Atualmente, já pode ser encontrada do lado do servidor. O JavaScript[7] foi utilizado para tornar a interface HTML interativa e funcional, onde o usuário poderá diagramar e se comunicar com o *back-end* da plataforma (JAVASCRIPT, 20014);

d) Bootstrap: é um *framework* de HTML, CSS e JavaScript, utilizado para o desenvolvimento de aplicações responsivas para a web. O Bootstrap[8] foi utilizado para desenvolver com mais facilidade, a interface da plataforma (BOOTSTRAP, 2014);

[4]http://socketo.me/
[5]http://www.w3.org/html/
[6]http://www.w3.org/Style/CSS/
[7]https://developer.mozilla.org/en-US/docs/Web/JavaScript
[8]http://getbootstrap.com/

e) JQuery: é uma biblioteca de JavaScript, rápida, pequena e muito rica em recursos, os quais facilitam o desenvolvimento e manipulação do HTML. O JQuery[9] foi utilizado para facilitar e agilizar o desenvolvimento das funcionalidades, bem como as interações do usuário com a interface da plataforma (JQUERY, 2014);

f) JSPlumb: é uma biblioteca que fornece vários recursos visuais, que permitem conectar elementos HTML em uma página da web. O JSPlumb[10] foi utilizado para facilitar o desenvolvimento das conexões entre as entidades do diagrama, as quais são elementos feitos com HTML (JSPLUMB, 2014);

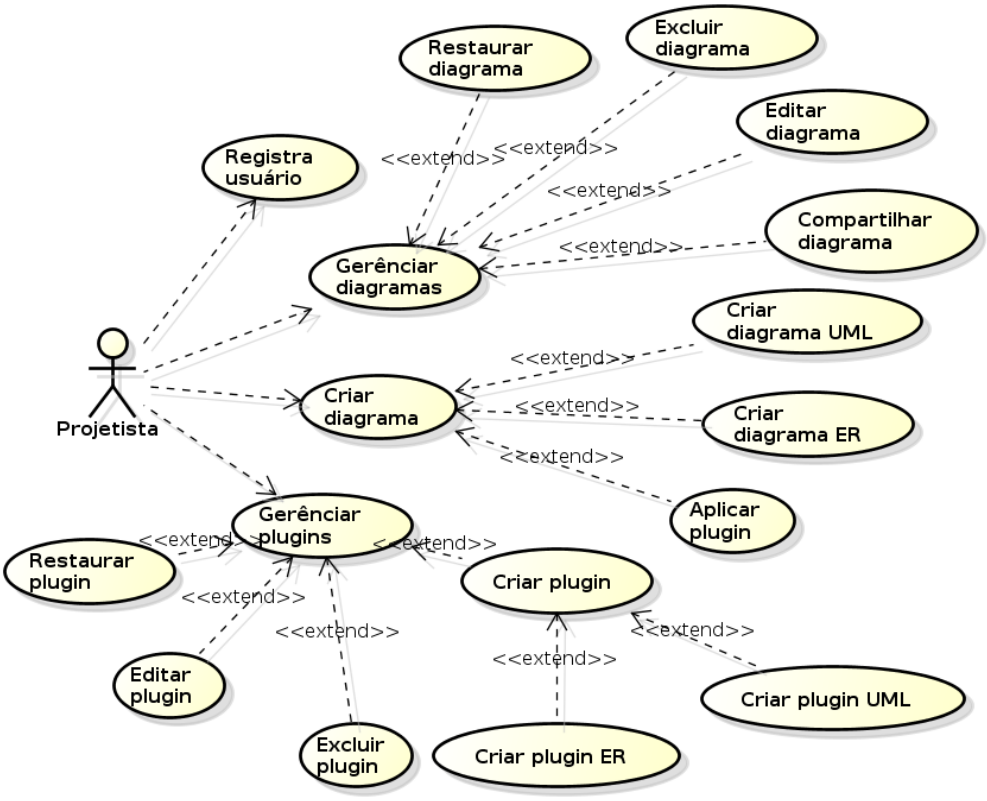
g) WebSocket: é uma interface JavaScript, que permite estabelecer uma conexão bidirecional, através de canais *full-duplex*, utilizando um único soquete, onde mensagens são enviadas entre cliente e servidor. O WebSocket[11] permite a diagramação colaborativa entre os usuários da plataforma.

4.3 Visão de casos de uso

O diagrama de casos de uso será utilizado para demonstrar os principais papéis dos usuários envolvidos na plataforma. Conforme pode ser visto na Figura 30, a plataforma tem apenas um tipo de usuário, que é o projetista. Este por sua vez, pode realizar ações de gerenciar diagramas (criar, editar, excluir, compartilhar, restaurar); gerenciar *plugins* (criar, editar, excluir, compartilhar); dentre outros, detalhados Tabela 1.

[9]http://jquery.com/
[10]https://jsplumbtoolkit.com
[11]https://www.websocket.org/

Figura 30 - Casos de uso da plataforma



Fonte: Feito pelo autor

A Tabela 1 detalha os casos de uso ilustrados na Figura 30.

Tabela 1 - Casos de uso da plataforma

Caso de uso	Descrição
Registra usuário	Permite que os usuários da plataforma possam realizar o seu auto cadastro, preenchendo algumas informações (nome, e-mail, senha).
Gerenciar diagramas	Permite que os usuários possam visualizar todos os diagramas criados por eles, e também, os que foram compartilhados. Através dela, eles têm acesso às ações de manipulação dos diagramas.
Restaurar diagrama	Permite que os usuários possam restaurar um diagrama excluído.
Excluir diagrama	Permite que os usuários possam excluir um diagrama criado por ele.
Editar diagrama	Permite que os usuários possam editar um diagrama, abrindo a interface de diagramação.
Compartilhar diagrama	Permite que os usuários possam compartilhar um diagrama criado por ele, com outros usuários.
Criar diagrama	Permite que os usuários possam criar um novo diagrama.
Criar diagrama UML	Permite que os usuários possam criar um novo diagrama UML.
Criar diagrama ER	Permite que os usuários possam criar um novo diagrama ER.
Gerenciar *plugins*	Permite que os usuários possam visualizar todos os *plugins* criados

	por eles, e também, os que foram criados de forma pública.
Restaurar *plugin*	Permite que os usuários possam restaurar um *plugin* excluído.
Editar *plugin*	Permite que os usuários possam editar um *plugin*, visualizando o código fonte do mesmo.
Excluir *plugin*	Permite que os usuários possam excluir um *plugin* criado por ele.
Criar *plugin*	Permite que os usuários possam criar um novo plugin, escrevendo o seu código fonte.
Criar *plugin* UML	Permite que os usuários possam criar um novo *plugin* para diagramas UML, onde ele escreve o código fonte do mesmo.
Criar *plugin* ER	Permite que os usuários possam criar um novo *plugin* para diagramas ER, onde ele escreve o código fonte do mesmo.

Fonte: Feito pelo autor

4.4 Modelo de domínio

O modelo de domínio representa um problema ou definição de uma estrutura. É um modelo conceitual, que auxilia no desenvolvimento e definição da estrutura a ser utilizada entres os envolvidos do processo. Este modelo faz uma relação entre entidades, bem como atributos e métodos de uma classe.

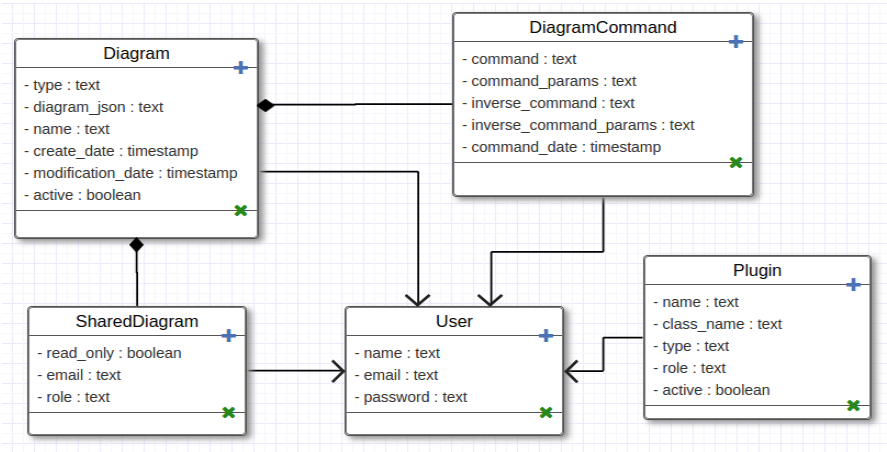
4.4.1 Modelo de classes

O modelo de classes, demonstrado na Figura 31, representa os conceitos envolvidos da plataforma. Conforme descrito na seção 2.5.1.1 deste documento, o diagrama de classes tem como objetivo, a visualização das classes de um determinado sistema, bem como os seus atributos e métodos.

Conforme pode ser visto no modelo da Figura 31, a principal classe do modelo é Diagram, que representa um diagrama criado pelo usuário da plataforma. Um diagrama é composto de uma sequência de comandos (DiagramCommand), que foram utilizados para a sua construção (Ex: Criar, excluir, arrastar). Tanto Diagram, quanto DiagramCommand, são realizados por um usuário (User). Como os diagramas podem ser compartilhados entre usuários, a classe ShareDiagram representa o compartilhamento de um diagrama entre diferentes usuários da plataforma. Por fim, a classe Plugin representa um *plugin* criado por

um usuário, que pode ser aplicado pelo mesmo sobre os seus diagramas, a fim de realizar transformações (Ex: exportação).

Figura 31 - Diagrama de classes da plataforma



Fonte: Feito pelo autor

As Tabelas enumeradas de 2 a 6, dispõem de uma breve descrição das classes, atributos e métodos existentes.

Tabela 2 - Descrição da Classe Diagram

Classe	Diagram.
Representa um diagrama criado pelo usuário.	
Atributos	
type	Tipo do diagrama, UML ou ER.
diagram_json	Estrutura do diagrama, armazenada no formato JSON.
name	Nome do diagrama.
create_date	Data de criação do diagrama.
modification_date	Última data de modificação do diagrama.
active	Atributo que indica se o diagrama está ativo ou não.

Fonte: Feito pelo autor

Tabela 3 - Descrição da Classe User

Classe	User.
Representa um usuário da plataforma.	
Atributos	
name	Nome do usuário.
email	E-mail do usuário.
password	Senha do usuário.

Fonte: Feito pelo autor

Tabela 4 - Descrição da Classe SharedDiagram

Classe	SharedDiagram.
Representa o compartilhamento de diagramas entre os usuários. Precisa estar vinculada com a classe Diagram e User.	
Atributos	
read_only	Indica se o diagrama compartilhado com algum usuário é somente leitura, ou se ele pode realizar a edição do mesmo.
email	Atributo utilizado para armazenar o e-mail com o qual o diagrama foi compartilhado, caso o usuário não seja encontrado na base de dados.

Fonte: Feito pelo autor

Tabela 5 - Descrição da Classe DiagramCommand

Classe	DiagramCommand.
Representa um comando realizado por um usuário em um diagrama, para que o mesmo possa ser distribuído para os demais usuários, com os quais o diagrama foi compartilhado.	
Atributos	
command	Método que foi executado pelo usuário ao diagramar.
command_params	Parâmetros utilizados pelo método ao diagramar.
inverse_command	Método inverso ao que foi utilizado pelo usuário ao diagramar.
inverse_command_params	Parâmetros do método inverso que foi utilizado ao diagramar.
command_date	Data e hora em que o comando foi executado.

Fonte: Feito pelo autor

Tabela 6 - Descrição da Classe Plugin

Classe	Plugin.
Representa os *plugins* criados pelos usuários, deve estar relacionado com um objeto da classe User.	
Atributos	
name	Nome do *plugin*.
class_name	Nome da classe do *plugin* criada pelo usuário.
type	Tipo do *plugin*, se é um *plugin* para diagramas do tipo ER ou UML.
role	Atributo utilizado para indicar se o *plugin* é público ou privado.
active	Indica se o *plugin* ainda está ativo.

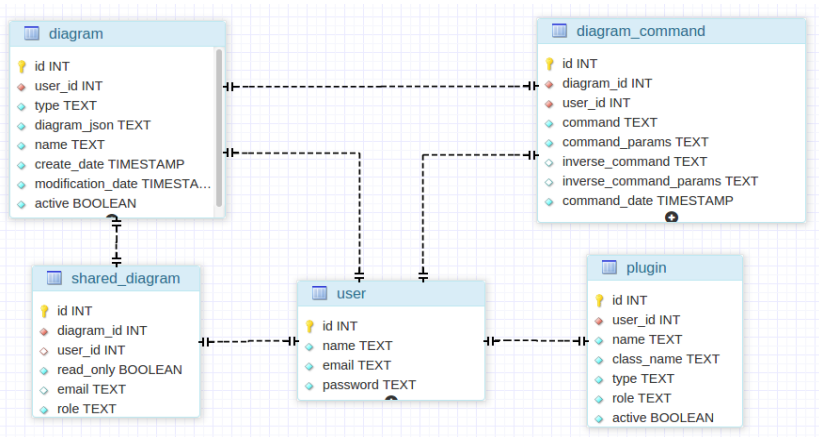
Fonte: Feito pelo autor

4.4.2 Modelo relacional

A modelagem relacional foi baseada no modelo de classes e representa como será modelada a base de dados da plataforma MyDiagrams. O diagrama ER (Entidade Relacionamento), demonstra todas as tabelas da base de dados, e suas respectivas dependências. A Figura 32 demonstra o modelo relacional.

Conforme padrões de mapeamento Objeto-Relacional, tanto as chaves estrangeiras, quanto as composições, podem ser mapeadas por meio do padrão de projetos *Foreign Key Mapping*. Este, trata de converter os relacionamentos em chaves estrangeiras, como pode ser visto na seção 2.6.2.3.

Figura 32 - Diagrama ER da plataforma



Fonte: Feito pelo autor

As Tabelas 7 a 11 dispõem de uma breve descrição das tabelas e dos campos do Diagrama ER, da plataforma MyDiagrams.

Tabela 7 - Descrição da tabela diagram

Tabela	diagram			
Descrição				
Tabela destinada a armazenar os diagramas feitos pelos usuários da plataforma.				
Campos/Atributos				
Chave	**Nome**	**Tipo**	**Nulo**	**Descrição/Restrições**
PK	id	inteiro	Não	Código único, chave primária da tabela.
FK	user_id	inteiro	Não	Usuário criador do diagrama, chave estrangeira para a tabela de usuários.
	type	texto	Não	Representa o tipo do diagrama, UML ou ER.
	diagram	texto	Não	JSON que representa o diagrama.
	name	texto	Não	Representa o nome do diagrama.
	create_date	data/hora	Não	Data e hora de criação do diagrama.
	modification_date	data/hora	Não	Data e hora da última modificação do diagrama.
	active	boleano	Não	Representa se o diagrama está ativo ou inativo.

Fonte: Feito pelo autor

Tabela 8 - Descrição da tabela shared_diagram

Tabela	shared_diagram			
Descrição				
Tabela destinada a armazenar os diagramas compartilhados entre os usuários da plataforma.				
Campos/Atributos				
Chave	**Nome**	**Tipo**	**Nulo**	**Descrição/Restrições**
PK	id	inteiro	Não	Código único, chave primária da tabela.
FK	diagram_id	inteiro	Não	Diagrama que foi compartilhado, chave estrangeira para a tabela de diagramas.
FK	user_id	inteiro	Sim	Diagrama, o qual foi compartilhado com um usuário, chave estrangeira com a tabela de usuários.
	read_only	boleano	Não	Se o diagrama compartilhado é somente leitura ou não.
	email	texto	Não	E-mail do usuário, ao qual o diagrama foi compartilhado.

Tabela 9 - Descrição da tabela *plugin*

Tabela	*plugin*			
Descrição				
Tabela destinada a armazenar os *plugins* criados pelos usuários da plataforma.				
Campos/Atributos				
Chave	**Nome**	**Tipo**	**Nulo**	**Descrição/Restrições**
PK	id	inteiro	Não	Código único, chave primária da tabela.
FK	user_id	inteiro	Não	Usuário criador do *plugin*, chave estrangeira para a tabela de usuários.
	name	texto	Não	Nome do *plugin*.
	class_name	texto	Não	Nome da classe criada pelo usuário.
	type	texto	Não	Tipo do *plugin*, para diagramas UML ou ER.
	role	texto	Não	Regra para saber se um *plugin* é privado ou público.
	active	boleano	Não	Utilizada para saber se um *plugin* ainda está ativo.

Fonte: Feito pelo autor

Tabela 10 - Descrição da tabela user

Tabela	user			
Descrição				
Tabela destinada a armazenar os usuários da plataforma.				
Campos/Atributos				
Chave	**Nome**	**Tipo**	**Nulo**	**Descrição/Restrições**
PK	id	inteiro	Não	Código único, chave primária da tabela.
	name	texto	Não	Nome do usuário.
	email	texto	Não	E-mail do usuário, usado como identificador de *login*, devendo ser único.
	password	texto	Não	Senha do usuário.
Índices				
Tipo			**Campos/Atributos**	
Único			email	

Fonte: Feito pelo autor

Tabela 11 - Descrição da tabela diagram_command

Tabela	diagram_command			
Descrição				
Tabela destinada a armazenar os comandos realizados pelos usuários ao diagramarem.				
Campos/Atributos				
Chave	**Nome**	**Tipo**	**Nulo**	**Descrição/Restrições**
PK	id	inteiro	Não	Código único, chave primária da tabela.
FK	diagram_id	inteiro	Não	Diagrama que está sendo criado, chave estrangeira para a tabela de diagramas.
FK	user_id	inteiro	Não	Usuário que está executando os comandos, chave estrangeira para a tabela de usuários.
	command	texto	Não	Função que foi utilizada para diagramar.
	command_params	texto	Não	Parâmetros utilizados pela função ao diagramar.
	inverse_command	texto	Sim	Função inversa a que foi utilizada ao diagramar.
	inverse_command_params	texto	Sim	Parâmetros da função inversa que foi utilizada ao diagramar.
	command_date	data/hora	Não	Data e hora que o comando foi executado.

Fonte: Feito pelo autor

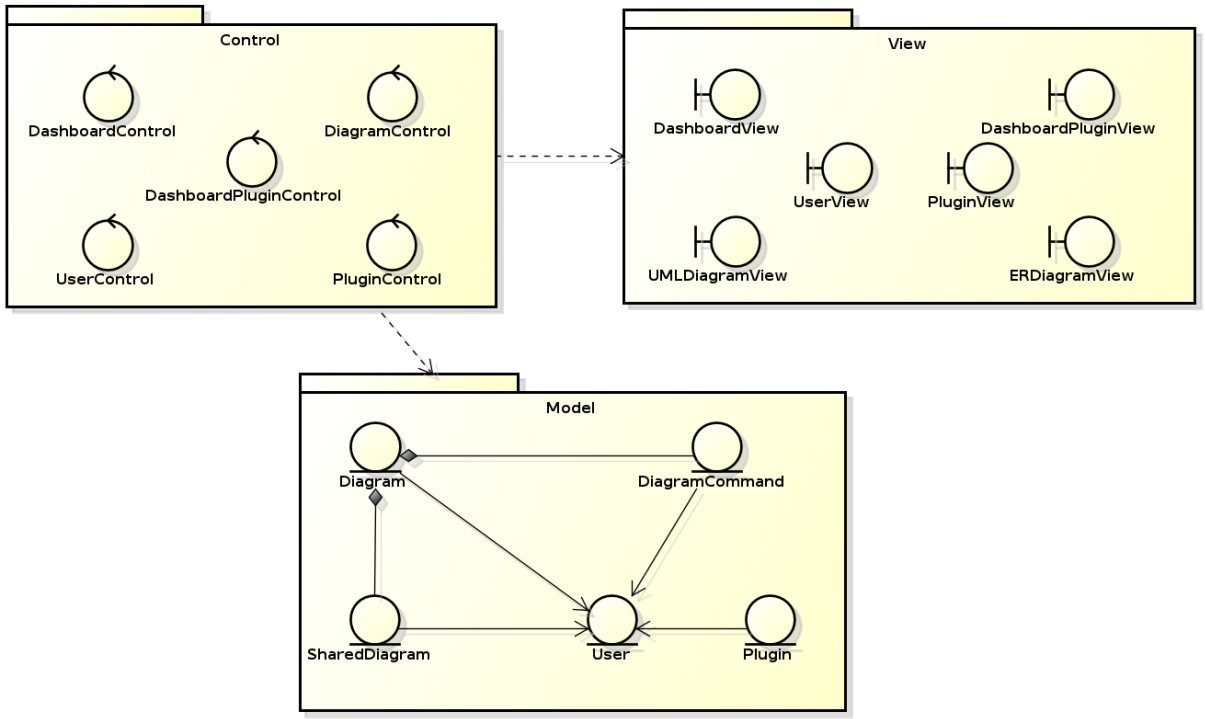
4.5 Especificação da arquitetura

Nesta seção, será detalhada a arquitetura da plataforma, estruturada a partir do modelo MVC. Neste sentido, será apresentada a arquitetura que roda ao lado do servidor (PHP), bem como a arquitetura que roda no lado do cliente (Javascript).

4.5.1 O modelo MVC

A arquitetura MVC separa as camadas de uma aplicação em *Model*, *View* e *Control*, como visto na seção 2.2.1.1. A Figura 33 demonstra o diagrama do modelo MVC da plataforma. Dentro do pacote *Control*, estão os controladores da plataforma, que gerenciam os principais aspectos: Painéis, Dashboards, Diagramas, Usuários e *Plugins*. No pacote *Model*, estão os modelos que fazem conexão direta às tabelas do banco de dados, e seus relacionamentos, como já apresentado na seção (2.4). Por último, no pacote *View*, estão as classes utilizadas para construir a interface com os usuários da plataforma. Cada classe de controle, aciona uma classe correspondente de *View* para apresentar, ou coletar dados dos usuários.

Figura 33 - O modelo MVC da plataforma

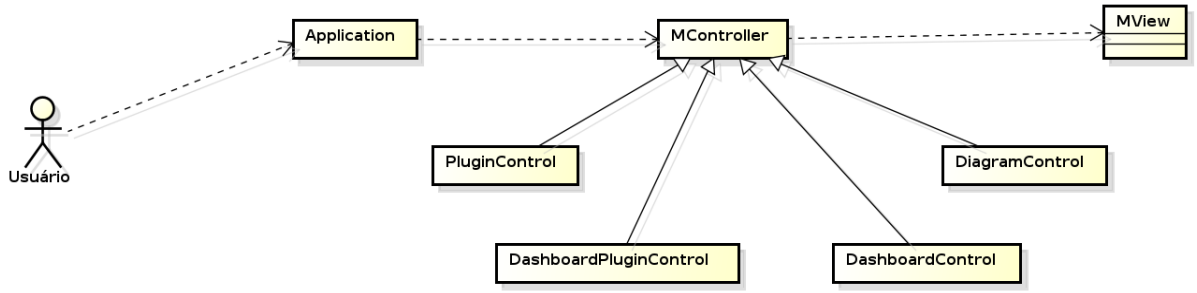


Fonte: Feito pelo autor

4.5.2 Arquitetura server-side

O diagrama apresentado na Figura 34 apresenta o *Design Pattern*, utilizado para gerenciar o fluxo de execução das ações na plataforma, chamado *Front Controller.* A classe MController é uma implementação do padrão *Front Controller.* Conforme Fowler (2003), um *front controller* é representado por uma classe, que recebe todas as requisições feitas para um sistema, e direciona estas, para classes específicas. Como pode ser visto na Figura 34, as requisições são direcionadas para as classes filhas do Mcontroller, para então serem tratadas. Por meio da Figura 34, pode-se visualizar as principais classes de controle da plataforma: PluginControl (ações de gerenciamento dos *plugins*); DashboardPluginControl (utilizada para gerenciar os *plugins*); DashboardControl (gerência os diagramas); DiagramControl (responsável pelas ações dos diagramas).

Figura 34 - *Front Controller* da plataforma



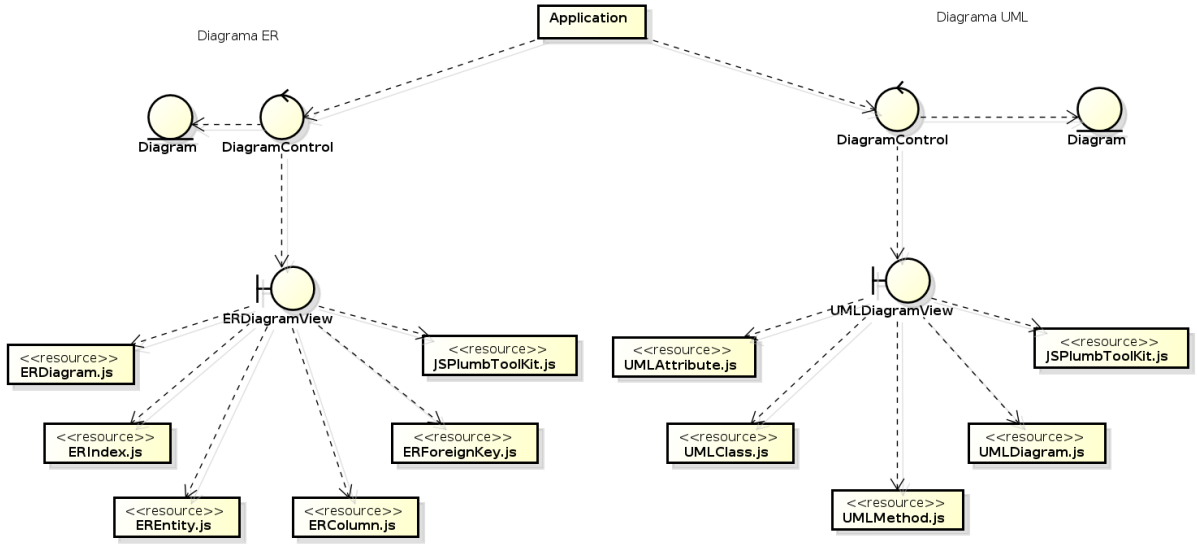
Fonte: Feito pelo autor

A plataforma está dividida em dois cenários principais de utilização, a diagramação UML e a diagramação ER. A Figura 35 demonstra esses cenários, bem como, as dependências de cada um deles. As classes de controle, como a DiagramControl, dependem das classes de visão ERDiagramView e UMLDiagramView, que por sua vez, fazem a interface com o usuário, permitindo a diagramação dos dois tipos de diagramas.

Para que o usuário possa diagramar, as classes de visão dependem de uma série de bibliotecas *JavaScript*, que encontram-se ao lado do cliente. Ao criar um diagrama ER, a classe DiagramControl instancia um objeto da classe ERDiagramView, que por sua vez, aciona ao lado do cliente várias classes Javascript. Estas representam as entidades de um diagrama, tais como: ERDiagram, ERIndex, EREntity, ERColumn, ERForeignKey e a classe JSPlumbToolKit, responsável por conectar visualmente as entidades. Sempre que um diagrama UML for criado, a classe DiagramControl instancia um objeto da classe UMLDiagramView, que por sua vez, aciona ao lado do cliente várias classes Javascript que

representam as classes de um diagrama, tais como: UMLDiagram, UMLClass, UMLAttribute, UMLMethod e JSPLumbToolKit, responsável por conectar visualmente as classes.

Figura 35 - As dependências da plataforma

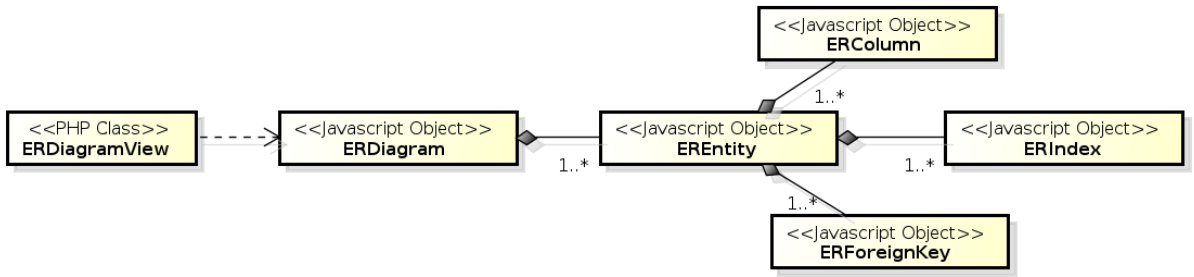


Fonte: Feito pelo autor

4.5.3 Arquitetura client-side

O Digrama da Figura 36, demonstra as relações de dependência entre as classes Javascript, responsáveis por criar os diagramas ER. Por meio da Figura 36, pode-se observar que a classe ERDiagramView, aciona a classe JavaScript ERDiagram, que por sua vez, pode criar diversas instâncias de EREntity para formar um diagrama. Cada objeto EREntity pode ser composto por uma ou várias instâncias de ERColumn, ERIndex ou ERForeignKey.

Figura 36 - Dependências para diagramação ER

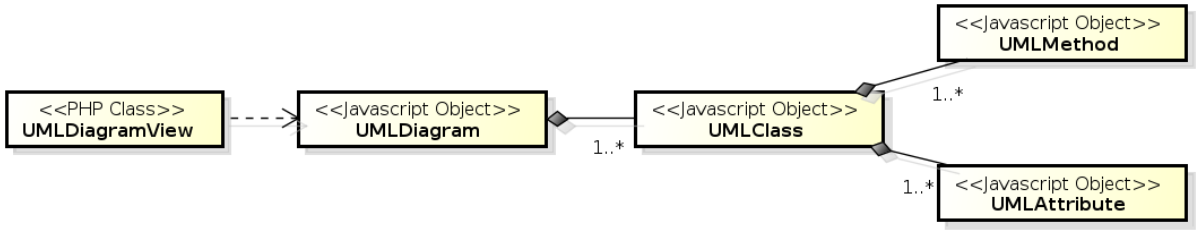


Fonte: Feito pelo autor

O diagrama da Figura 37, demonstra as dependências entre as classes Javascript, responsáveis por criar os diagramas UML. Por meio da Figura 37, pode ser visto que, a classe UMLDiagramView aciona a classe JavaScript UMLDiagram, que por sua vez, pode criar

várias instâncias de UMLClass para formar um diagrama. Cada objeto UMLClass pode ser formado ainda por instâncias de UMLMethod e UMLAttribute.

Figura 37 - Dependências para diagramação UML



Fonte: Feito pelo autor

4.5.4 Especificação dos cenários

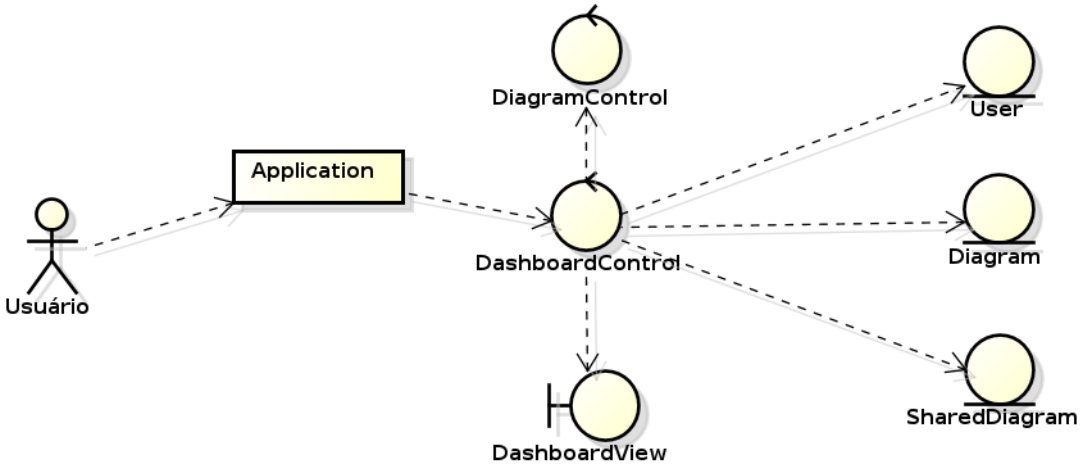
Nesta seção, serão apresentados os principais cenários do uso da plataforma. Cada cenário será apresentado com o seu diagrama correspondente, bem como uma figura para ilustrar a sua utilização pelo usuário.

4.5.4.1 Gestão de diagramas

O cenário de gestão de diagramas é apresentado ao usuário logo após o seu acesso ao sistema por meio do login. A Figura 38 apresenta as classes envolvidas neste cenário. Pode-se assim, verificar que a aplicação aciona a classe DashboardControl, que controla as ações para o gerenciamento de diagramas. Para gerenciar os diagramas são necessárias as classes DashboardView, que são utilizadas para apresentar uma lista de diagramas ao usuário.

Por meio dessa interface, o mesmo poderá realizar ações, como criar, editar, excluir, restaurar e compartilhar diagramas. Para realizar estas ações, são necessárias ainda, algumas classes de modelo para prover informações, tais como: User (para obter informações do usuário que criou os diagramas); Diagram (para fornecer a lista de diagramas, bem como as suas informações); e SharedDiagram (para obter uma lista de diagramas compartilhados com o usuário).

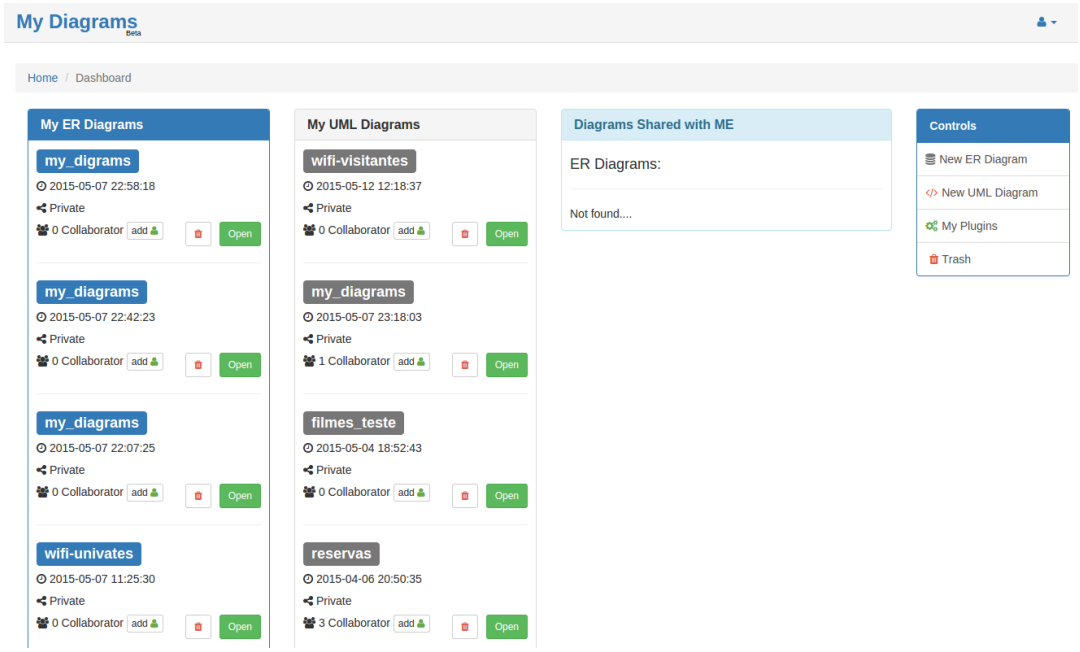
Figura 38 - DashboardControl



Fonte: Feito pelo autor

A Figura 39, ilustra a tela de administração de diagramas. Conforme a imagem, pode-se visualizar mais à esquerda, a lista de diagramas ER, criados pelo usuário. Um pouco mais à direita, pode-se visualizar a lista de diagramas UML. Em seguida, os diagramas criados por outros usuários que foram compartilhados. Logo após os diagramas compartilhados, encontram-se alguns links de controle, os quais permitem criar um diagrama UML ou ER, visualizar o Dashboard de *plugins*, e, visualizar os diagramas excluídos pelo usuário. Através da listagem dos diagramas é possível editá-los, compartilhá-los e excluí-los, entretanto, estas duas últimas opções só são disponibilizadas para os diagramas criados pelo usuário.

Figura 39 - Visualização do Dashboard dentro da plataforma



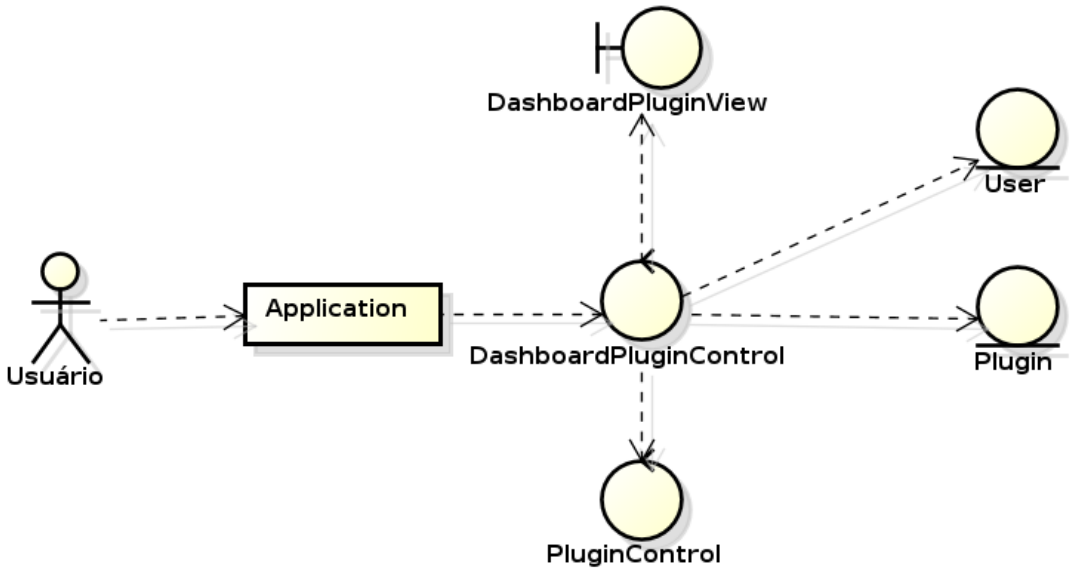
Fonte: Feito pelo autor

4.5.4.2 Gestão de *plugins*

O cenário de gestão de *plugins* é apresentado ao usuário quando ele acessa o DashboardControl e clica no link *Plugins* que se encontra dentro da lista de controles, à direita da tela. A Figura 40 apresenta as classes envolvidas neste cenário. Pode-se visualizar que, a aplicação aciona a classe DashboardPluginControl, que controla as ações para gerenciamento de *plugins*. Para gerenciar os *plugins*, são necessárias as classes DashboardPluginView, que são utilizadas para apresentar uma lista de *plugins* ao usuário.

Por meio dessa interface, o usuário poderá realizar ações, como criar, editar, excluir e restaurar *plugins*. Para realizar estas ações, são necessárias mais duas classes de modelo para prover informações: User (para obter informações do usuário que criou os *plugins*) e Plugin(para fornecer a lista de *plugins*, bem como as suas informações).

Figura 40 - Visualização do dashboard dos *plugins* dentro da plataforma

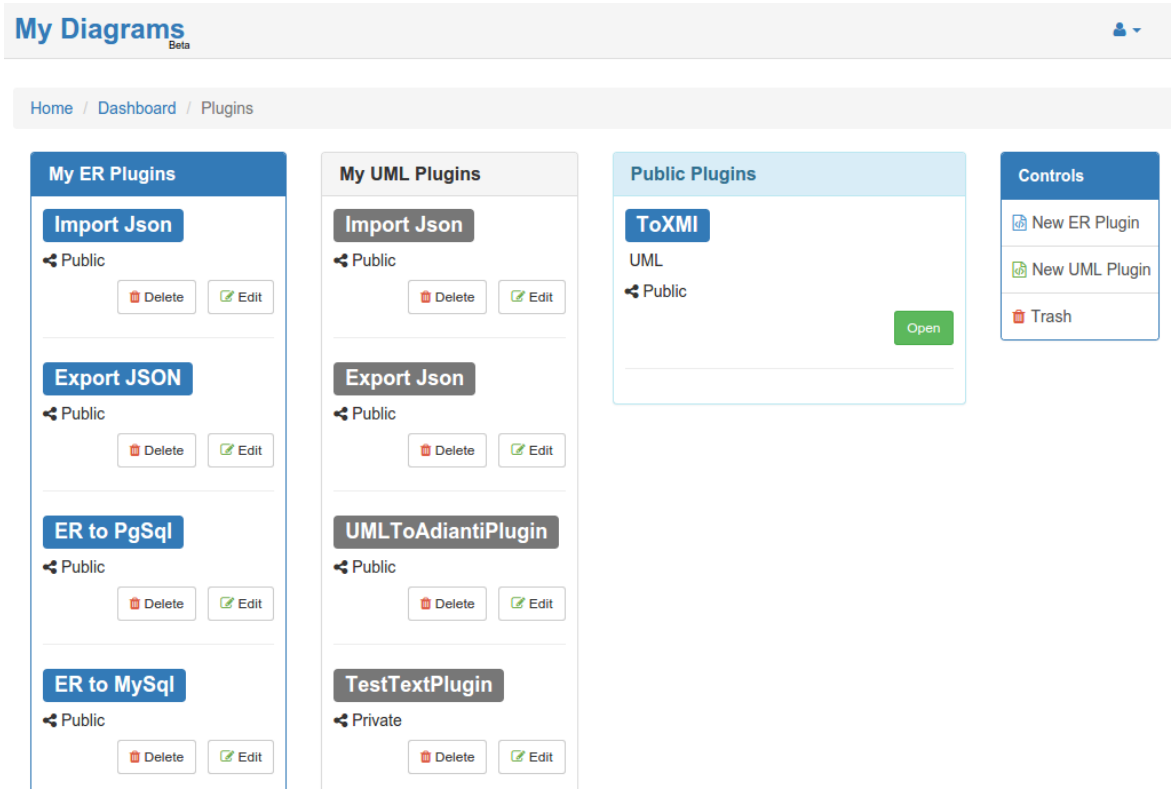


Fonte: Feito pelo autor

A Figura 41 ilustra a tela de administração de *plugins*. Conforme demonstrado na imagem, mais à esquerda, encontra-se a lista de *plugins* para diagramas ER, criados pelo usuário. Um pouco mais à direita, a lista de *plugins* para diagramas UML. Em seguida, os *plugins* públicos, criados por outros usuários. Logo após os *plugins* públicos, encontram-se alguns links de controle, os quais permitem criar um *plugin* para diagrama UML ou ER, e visualizar os *plugins* excluídos pelo usuário. Através da listagem dos *plugins* é possível editá-

los e excluí-los, sendo que na listagem de *plugins* públicos somente pode-se visualizar o código-fonte deles.

Figura 41 - DashboardPluginControl



Fonte: Feito pelo autor

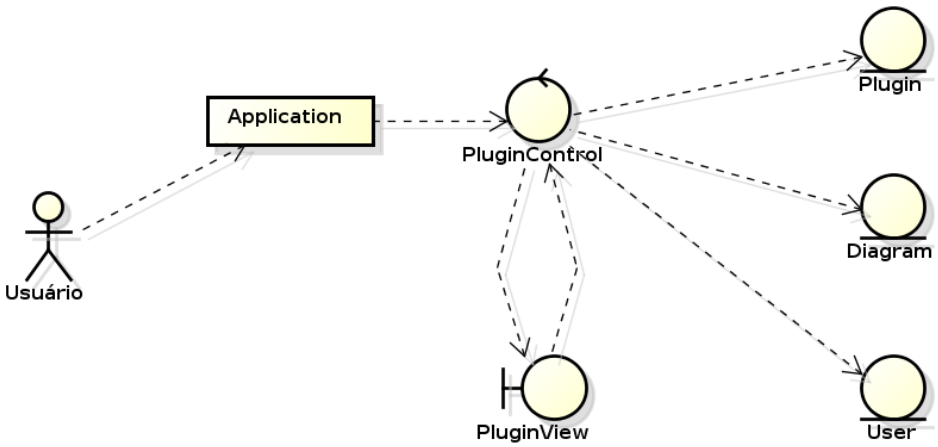
4.5.4.3 Arquitetura de *plugins*

O cenário dos *plugins* é apresentado ao usuário quando ele acessa o DashboardPluginControl e realiza ações, como criar, editar e restaurar *plugins*. O usuário também irá interagir com os *plugins* na hora de realizar uma importação e exportação de diagramas. A Figura 42 apresenta as classes envolvidas neste cenário. Pode-se constatar que a aplicação aciona a classe PluginControl, que controla as ações de gerenciamento (criar, editar, excluir, restaurar *plugins*) dos *plugins*. Para realizar tais ações, são necessárias as classes PluginView, que é utilizada para apresentar o código fonte do *plugin* ao usuário, bem como mostrar os *plugins* já excluídos.

A classe DiagramControl também é utilizada para criar uma listagem dos *plugins* na tela de diagramação, na qual o usuário pode importar ou exportar os diagramas. Para realizar estas ações, são necessárias ainda duas classes de modelo para prover informações: User (para

obter informações do usuário que criou os *plugins*), Plugin(para fornecer as informações do *plugin*) e Diagram (para obter as informações do diagrama).

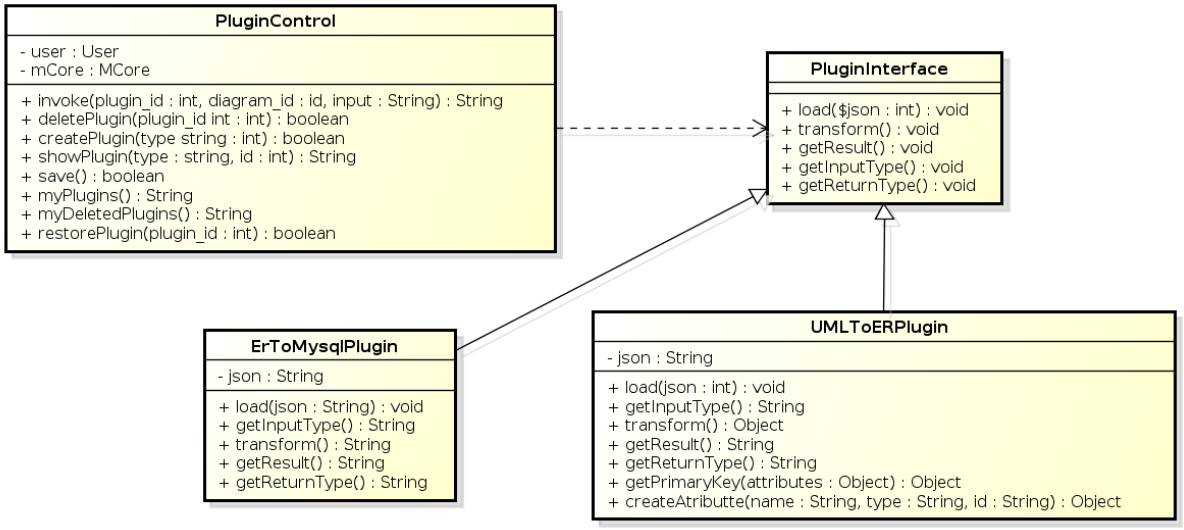
Figura 42 - *Plugins*



Fonte: Feito pelo autor

A Figura 43, ilustra a estrutura definida para criação de *Plugins*. PluginControl é uma classe da plataforma, utilizada para gerenciar os *plugins* do usuário. Por um lado, a classe PluginControl depende, que sejam fornecidas para ela, classes que implementem a interface (PluginInterface), ou seja, que forneça implementação destes métodos. Por outro lado, o projetista (usuário da plataforma), pode desenvolver *plugins* que atendam aos métodos definidos nesta interface. Ao atender esses requisitos, criando uma classe que implemente a interface PluginInterface, o desenvolvedor poderá construir um *plugin* que realize uma grande quantidade de transformações sobre o modelo.

Figura 43 - Funcionamento dos *plugins*



Fonte: Feito pelo autor

Para criar um *plugin*, o desenvolvedor precisará implementar os seguintes métodos definidos pela interface PluginInterface, conforme demonstrado na tabela.

Tabela 12 - Descrição dos métodos da interface PluginInterface

Interface	PluginInterface.
Métodos	
load	Carrega o json do diagrama, ou outros tipos de dados informados pelo usuário.
transform	Transforma os dados carregados pelo método load.
getResult	Retorna os dados que foram modificados pelo método transform.
getInputType	Retorna qual será o tipo de entrada de dados ao usuário.
getReturnType	Retorna qual será o tipo de saída de dados ao usuário.

Fonte: Feito pelo autor

A Figura 44 apresenta a estrutura de código fonte mínima para ser implementada na criação de novos *plugins*. Como pode ser visto, a figura contém os protótipos dos métodos apresentados na Tabela 12.

Figura 44 - Estrutura de código fonte de um novo *plugin*

```php
<?php

    class MyUMLPlugin implements PluginInterface{

        private $json;
        const DOWNLOAD = "download";
        const SHOW = "show";
        const TOUML = "toUml";
        const TOER = "toEr";
        const TEXTAREA = "textArea";
        const FILEUPLOAD = "fileUpload";
        const FROMDIAGRAM = "fromDiagram";

        public function load($json){
            $this->json = $json;
        }

        public function transform(){
            return $newThing;
        }

        public function getResult(){
            return $result;
        }

        public function getInputType(){
            return self::FROMDIAGRAM; // TEXTAREA, FILEUPLOAD
        }

        public function getReturnType(){
            return self::TOER; // DOWNLOAD, SHOW, TOUML, TOER
        }
    }
?>
```

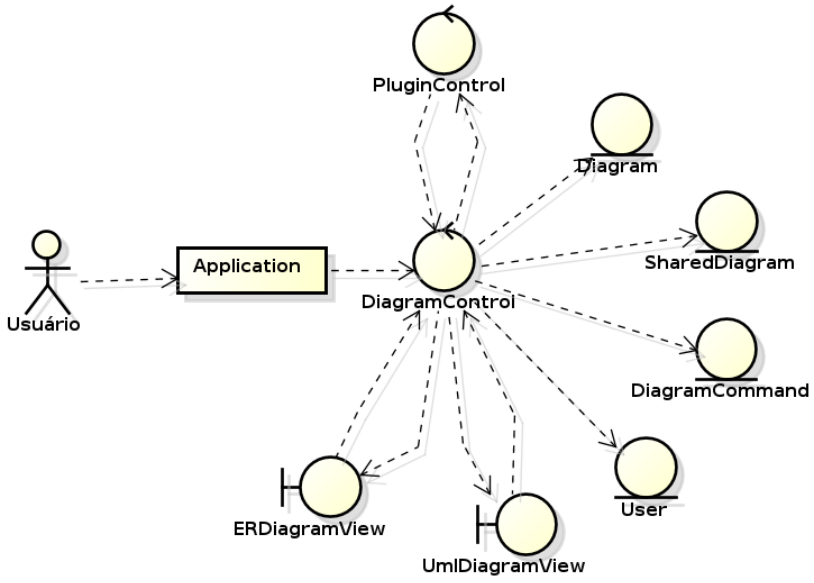
Fonte: Feito pelo autor

4.5.4.4 Criação dos diagramas

O ambiente de criação de diagramas é apresentado ao usuário quando ele acessa o DashboardControl, e realiza ações de criar ou editar um diagrama. A Figura 42 apresenta as classes envolvidas neste cenário. Pode ser visto que a aplicação aciona a classe DiagramControl, que controla as ações de diagramação, edição colaborativa, gravação, e execução de *plugins*. Para realizar tais ações, são necessárias as classes ERDiagramView, UMLDiagramView e PluginControl.

A classe ERDiagramView é responsável por criar a interface de diagramação ER da plataforma, já a classe UMLDiagramView cria a interface de diagramação UML, e por fim, a classe PluginControl é utilizada para montar uma lista de *plugins* disponíveis, permitindo que o usuário faça uso dos mesmos. Para realizar essas ações, são necessárias ainda quatro classes de modelo para prover informações, salvar os diagramas, bem como permitir a edição colaborativa: User (para obter informações do usuário que criou os diagramas), Diagram (para fornecer as informações do diagrama), SharedDiagram (para saber se o diagrama está compartilhado com mais usuários) e DiagramCommand (para guardar os comandos feitos pelos usuários da plataforma, que por sua vez são enviados através de um websocket, permitindo a diagramação coolaborativa).

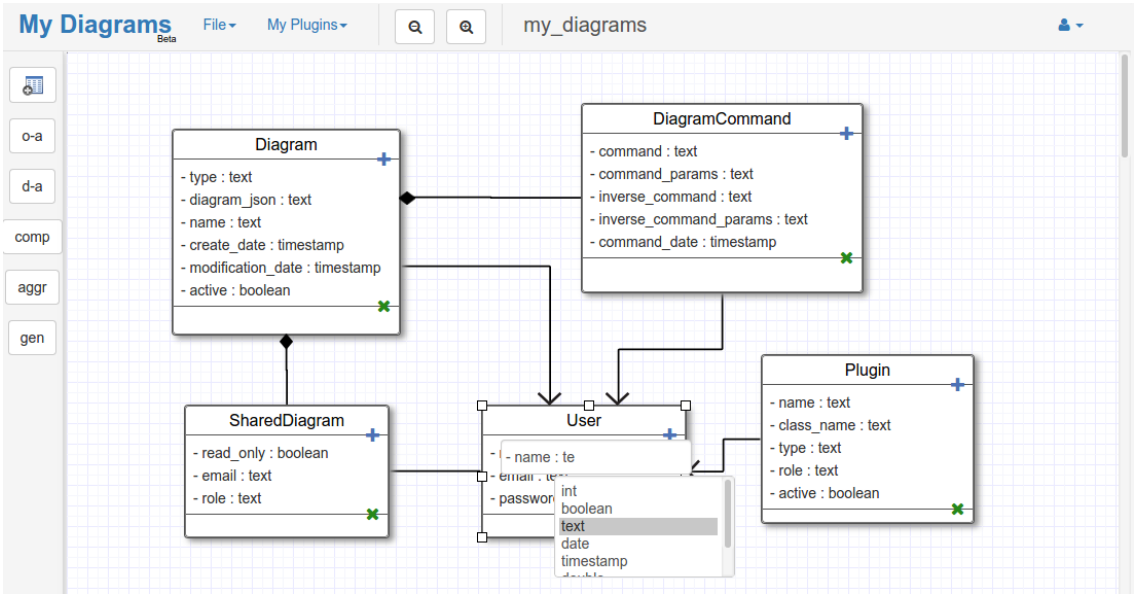
Figura 45 - DiagramControl



Fonte: Feito pelo autor

A Figura 46 demonstra um diagrama UML criado pela plataforma. Como pode-se visualizar, neste contexto temos uma barra de botões à esquerda, com as seguintes funcionalidades: adicionar uma nova classe, relação de dependência, agregação e associação entre classes. No centro da figura podem ser observados os elementos que representam as classes do diagrama, os quais possuem algumas funcionalidades, por exemplo, acrescentar um novo atributo e método. Na parte superior da imagem encontra-se uma barra de ferramentas, que tem como principal ação a aplicação dos *plugins* criados pelos usuários. Estes permitem a importação ou exportação do diagrama em diferentes formatos (JSON, classes de modelo para o *framework* Adianti, diagrama ER na plataforma).

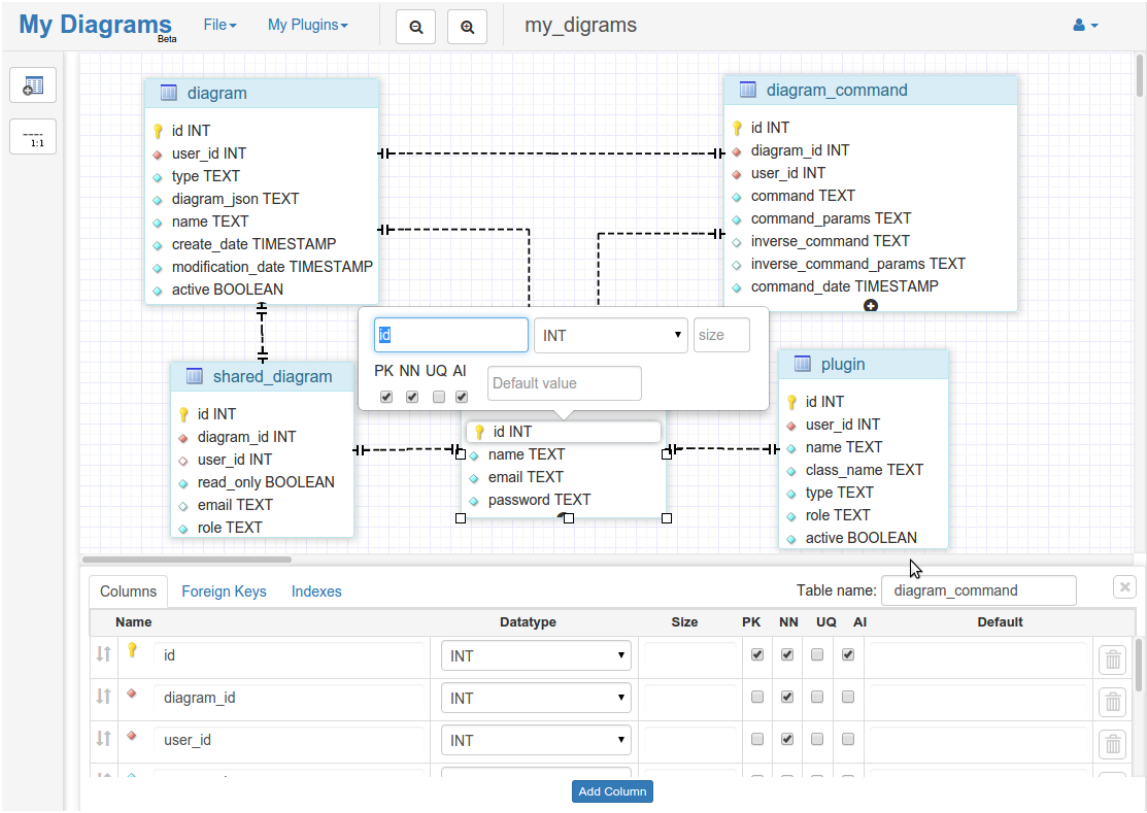
Figura 46 - Diagrama UML elaborado na plataforma



Fonte: Feito pelo autor

A Figura 47 demonstra um diagrama ER criado pela plataforma. Como pode ser visto neste contexto, temos uma barra de botões à esquerda, com as funcionalidades de adicionar uma nova entidade, e adicionar uma relação entre entidades. No centro da figura podem ser observados os elementos que representam as entidades do diagrama, as quais possuem algumas funcionalidades, como acrescentar uma nova coluna e alterar o nome da entidade. Na parte inferior da imagem, encontra-se o formulário de edição completo de uma entidade, no qual pode-se editar o nome dela, acrescentar uma nova coluna, adicionar um índice, bem como acrescentar uma relação de chave estrangeira. Na parte superior da imagem, encontra-se uma barra de ferramentas, que tem como principal ação a aplicação dos *plugins* criados pelos usuários, onde estes permitem a importação ou exportação do diagrama em diferentes formatos (JSON, SQL de criação de base MySQL e PostgreSQL).

Figura 47 - Diagrama ER elaborado na plataforma



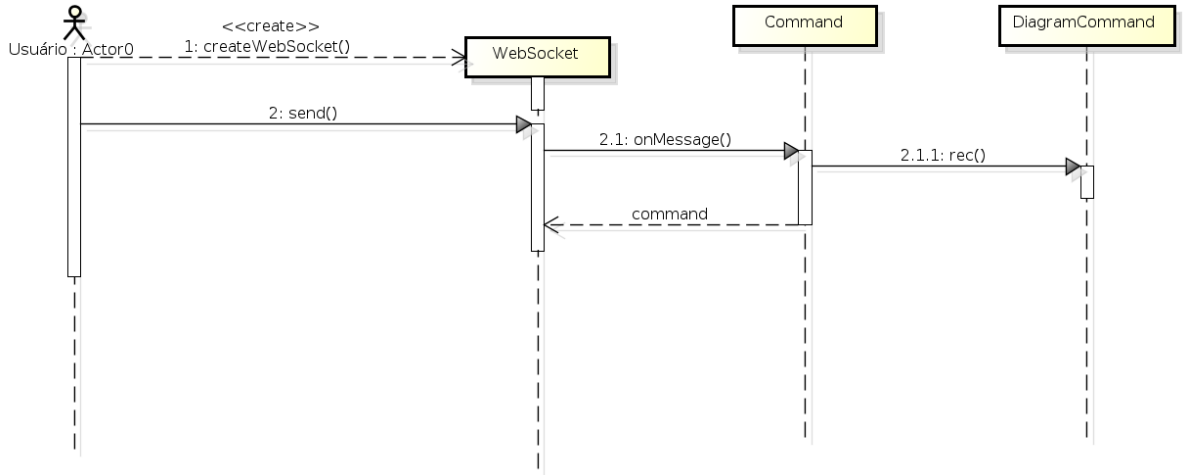
Fonte: Feito pelo autor

4.5.4.5 Websocket

Como a plataforma fornece o recurso de edição compartilhada de diagramas, foi necessário utilizar uma ferramenta assíncrona, que permitisse a execução de comandos enviados pelo servidor ao lado do cliente. Para gerenciar essa comunicação, foi utilizada a biblioteca Rachet, que implementa o conceito de WebSocket.

O diagrama da Figura 48, representa a comunicação de WebSocket utilizada para a sincronização de comandos utilizados para a edição de diagramas. Como pode ser visto, o processo se inicia quando, a partir da interface cliente, é criado um objeto WebSocket. Então, sempre que o usuário realiza algum comando, este é enviado por meio do método “send”, do WebSocket em forma de mensagem. Esta, por sua vez, é recebida no servidor e tratada pela classe Command. A classe Command instancia a classe DiagramComand para gravar a mensagem tratada na base de dados e, em seguida, envia a mesma mensagem para os demais usuários que estão colaborando na construção do mesmo diagrama.

Figura 48 - WebSocket



Fonte: Feito pelo autor

4.6 Trabalhos futuros

Conforme poderá ser visto no capítulo 5, o presente trabalho cumpre com os objetivos propostos. Porém, existem melhorias em algumas funcionalidades existentes, bem como novas funcionalidades que puderam ser percebidas durante o desenvolvimento do mesmo. Dentre estas, podem ser citadas:

a) Desenvolver a funcionalidade de desfazer e refazer ações (Ex: movimentação de entidades, exclusão de colunas);

b) Desenvolver a funcionaliade de copiar e colar entidades ou classes;

c) Desenvolver a funcionalidade de selecionar várias entidades ou classes, para realizar ações em grupo como: mover, excluir, copiar e colar;

d) Desenvolver a funcionalidade de exportar o diagrama em formato de imagem;

e) Desenvolver um bate-papo para os usuários que estão trabalhando em um mesmo diagrama, para que estes possam se comunicar através da plataforma;

f) Utilizar a linguagem de programação Node.js no lugar do PHP no back-end da plataforma, a fim de reaproveitar as classes JavaScript utilizadas no front-end, para facilitar modificações no JSON dos diagramas gerados;

g) Desenvolver um assistente para ajudar a criar um novo *plugin;*

h) Poder criar assistentes de geração de aplicações para os *plugins;*

i) Permitir a criação de formuláros de entrada de dados (assistentes), tanto nas rotinas de importação como nas de exportação de *plugins*, a fim de utilizar estas informações fornecidas pelo usuário para a criação de aplicações mais completas *(model, view, controller).*

5 VALIDAÇÃO

5.1 Introdução

Neste capítulo será apresentada a validação da plataforma prosposta, bem como os resultados obtidos. A validação da plataforma será realizada por meio de uma comparação de um sistema existente, e em produção, em relação à sua variante modelada pelo MyDiagrams.

O sistema escolhido para conduzir a validação é o sistema de inscrições em processos, desenvolvido internamente pela Univates. O sistema em questão está em produção desde um de maio de 2014, e, é totalmente orientado a objetos, desenvolvido com arquitetura MVC, em PHP, e com o Adianti *Framework*.

Por um lado tem-se um grupo de classes de modelo já existentes e em operação, em um sistema que já está validado e em produção, por outro lado, o mesmo conjunto de classes será replicado na plataforma MyDiagrams, que será utilizada para gerar novamente o código-fonte da camada de modelo da aplicação. O código-fonte gerado não será 100% igual ao atualmente em uso, que possui por exemplo, regras de negócio específicas. Porém, diversos métodos que tratam do relacionamento entre os objetos, bem como persistência dos mesmos, poderão ser comparados.

Dessa forma, a validação da plataforma proposta envolverá a comparação de funcionalidades existentes, relativas à camada de modelo do atual sistema de inscrições em processos, com o mesmo, porém modelado, no MyDiagrams. A comparação será realizada por meio de scripts que simulam testes unitários, realizando chamadas de métodos da camada Model, para persistir, bem como retornar informações pertinentes ao negócio. Os *scripts* serão

rodados sobre as classes geradas, e os resultados serão comparados para analisar se a plataforma gerou de maneira correta, as classes da camada de modelo.

5.2 Descrição do cenário

O cenário escolhido para realizar a validação da plataforma é o sistema de inscrições em processos (provas de nivelamento, processo seletivo de docente, inscrições de mestrado, etc.), da Univates. Este sistema possui uma página de acesso público, na qual os participantes podem se cadastrar e se inscrever em um processo de inscrição (eventos). Nesta página, o participante pode informar o seu e-mail e senha, caso já possua um cadastrado no sistema. Caso não esteja cadastrado, deve cadastrar-se, informando os campos básicos do cadastro (nome, cpf, endereço, e-mail, senha), para que o sistema gere um novo cadastro.

O participante pode a qualquer momento acessar o sistema. Neste caso, ele tem acesso à opção de alteração de dados pessoais (nome, telefone, etc). Para eventos em aberto, o usuário pode reimprimir o boleto ou visualizar os dados da inscrição e as suas atividades selecionadas, refazendo a sua inscrição em forma de edição.

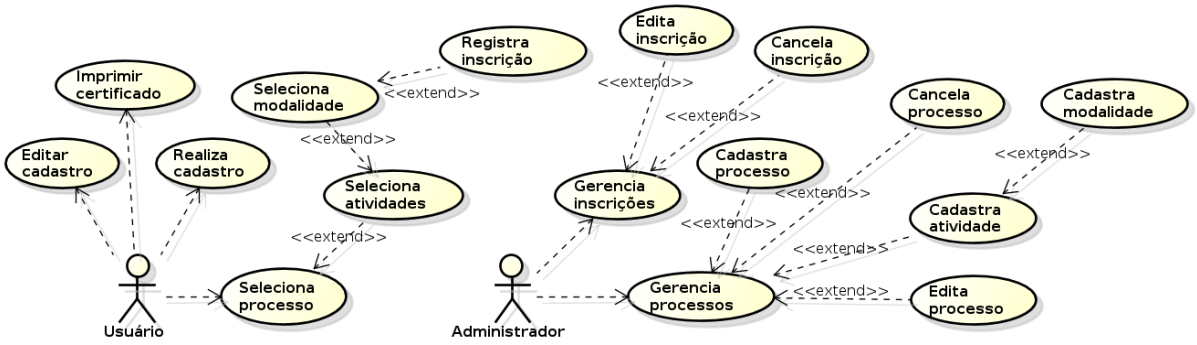
Para funcionar, o sistema precisa de usuários realizando operações de retaguarda (verificar pagamentos, planejar o evento, etc.). Para tal, o sistema permite que o administrador do sistema cadastre novos usuários, informando o nome, login e senha. Estes usuários administrativos são responsáveis em gerenciar e cadastrar os processos de inscrições, bem como as suas atividades em um evento (Ex: oficinas, minicursos), bem como modalidades de pagamento (Ex: taxa de inscrição gratuita ou com valor fixo). Além disso, os administradores são responsáveis por verificar se as taxas de inscrições foram pagas, gerar listas de inscritos, entre outros.

5.3 Casos de uso

O diagrama de casos de uso será utilizado para demonstrar os principais papéis de usuários envolvidos no sistema a ser validado. Conforme pode ser visto na Figura 49, o sistema possui dois tipos de usuários, um deles é o próprio usuário e o outro é o administrador do sistema. O usuário por sua vez pode realizar ações, como selecionar um processo, selecionar atividades, realizar a sua inscrição, realizar o seu cadastro, dentre outros. Já o

administrador do sistema, pode realizar ações de gerenciar processos (criar, editar, cancelar), gerenciar inscrições (editar, cancelar), dentre outros.

Figura 49 - Casos de uso do sistema de inscrições



Fonte: Feito pelo autor

A Tabela 13 detalha os casos de uso ilustrados na Figura 49.

Tabela 13 - Casos de uso da plataforma

Caso de uso	Descrição
Realiza cadastro	Permite que os usuários possam realizar o seu autorregistro, preenchendo algumas informações (nome, e-email, senha).
Editar cadastro	Permite que os usuários possam alterar as suas informações pessoais, preenchidas no cadastro.
Imprimir certificado	Permite aos usuários, a impressão de certificados de processos de inscrição que ele participou.
Seleciona processos	Permite aos usuários escolher um processo de inscrição.
Seleciona atividades	Permite aos usuários escolher atividades de um processo de inscrição.
Seleciona modalidades	Permite aos usuários escolher uma modalidade de pagamento, para cada uma das atividades escolhidas de um processo de inscrição.
Gerencia inscrições	Permite ao administrador do sistema, visualizar todas as inscrições feitas pelos usuários do mesmo.
Cancela inscrição	Permite que o administrador do sistema cancele uma ou mais inscrições, feitas em algum processo de inscrição.
Edita inscrição	Permite que o administrador do sistema possa editar a inscrição feita por algum usuário.
Gerencia processos	Permite ao administrador do sistema, visualizar todos os processos de inscrição cadastrados no sistema.
Cadastra processos	Permite que o administrador do sistema, cadastre as informações básicas de um novo processo de inscrição.
Cancela processos	Permite que o administrador do sistema possa cancelar algum processo de inscrição.

Cadastra atividade	Permite que o adminstrador do sistema, cadastre as modalidades de pagamento de um processo de inscrição criado por ele.
Cadastra modalidade	Permite ao administrador, cadastrar as modalidades de pagamento para cada atividade cadastrada no sistema.
Edita processo de inscrição	Permite ao administrado do sistema, editar todas as informações de um processo de inscrição, bem como as suas atividades e modalidades.
Registra inscrição	Permite que o usuário realize ou não a sua inscrição em um processo.

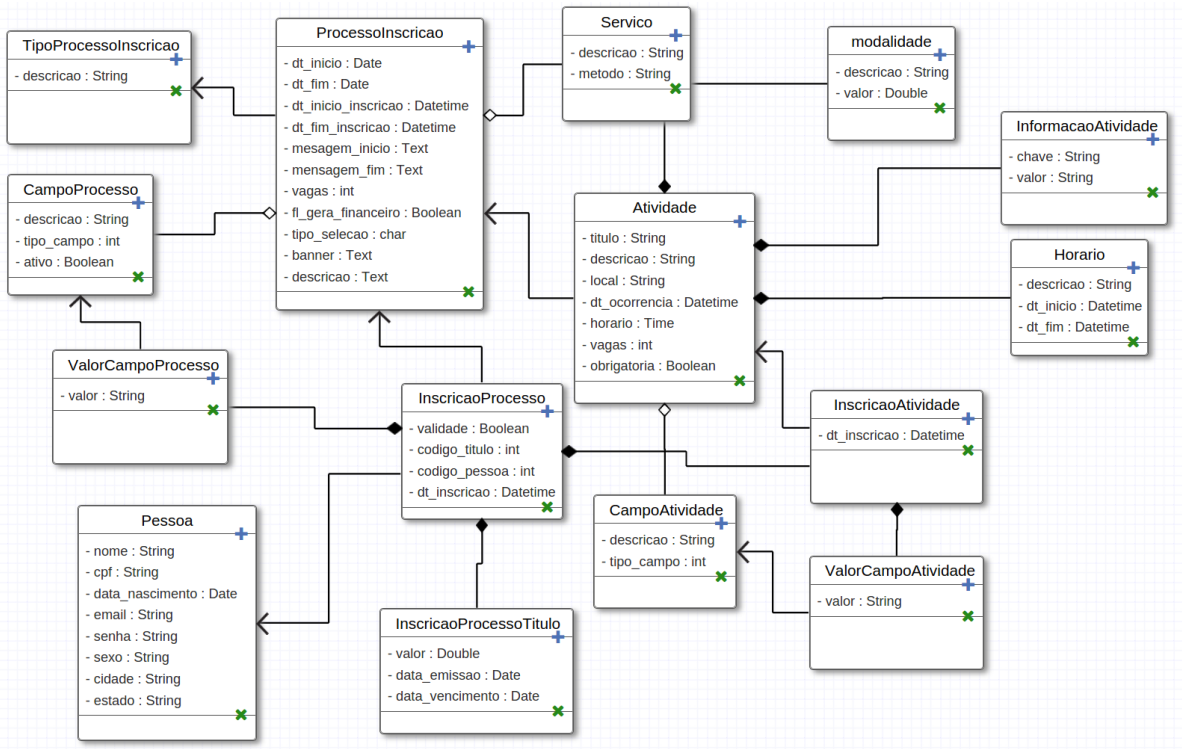
Fonte: Feito pelo autor

5.4 Modelo UML

O modelo de classes, demonstrado na Figura 50, representa os conceitos envolvidos do sistema de inscrições. Conforme pode ser visto no modelo da Figura 50, a principal classe do modelo é ProcessoInscricao, que representa um processo de inscrição criado no sistema. Um processo de inscrição é composto de atividades (Atividade) e de modalidades (Modalidade).

Conforme pode ser visto na Figura 50, a classe mais relevante do sistema é ProcessoInscricao, que por sua vez, pode ter campos específicos por meio de uma agregação com a classe CampoProcesso. Um processo de inscrição (Ex: inscrições para o “Projeto i”, prova de proficiência, pós-graduação, etc.), poderá ter vinculado a ele várias atividades, que são representadas pela classe Atividade (Ex: mini curso de inglês, prova de inglês, prova de espanhol), que por sua vez podem ter inscrições, estas vinculadas com a classe InscricaoAtividade, a qual possui uma composição com a classe InscricaoProcesso, que por sua vez está vinculada com a classe ProcessoInscricao.

Figura 50 - Diagrama de classes do sistema de inscrições



Fonte: Feito pelo autor

Os itens a seguir descrevem as classes do modelo de classes que estão no sistema de inscrições:

a) ProcessoInscricao: classe que representa um processo de inscrição;

b) Pessoa: classe que representa um usuário do sistema, contendo as suas informações pessoais e credenciais de acesso ao sistema;

c) InscricaoProcesso: classe que representa a inscrição de um usuário em um processo;

d) TipoProcessoInscricao: classe que representa os tipos de processos de inscrição existentes no sistema;

e) Atividade: classe que representa as atividades que um processo de inscrição pode ter;

f) InscricaoAtividade: classe que representa a inscrição de um usuário em uma atividade de um processo;

g) Modalidade: classe que representa a modalidade de pagamento de uma atividade;

h) InformacaoAtividade: classe que representa valores de campos dinâmicos, que uma atividade pode ter;

i) Horario: classe que representa os horários em que uma atividade irá ocorrer;

j) Servico: classe que representa um *Web Service* que pode ser executado para realizar algum tipo de bloqueio ou validação de regra de negócio, durante o processo de inscrição;

k) CampoAtividade: classe que representa um campo dinâmico, cadastrado para uma atividade específica, que será solicitado no momento da inscrição;

l) ValorCampoAtividade: classe que representa o conteúdo preenchido pelo usuário para cada objeto do tipo CampoAtividade, solicitado na inscrição;

m) CampoProcesso: classe que representa um campo dinâmico, cadastrado para um processo específico, que será solicitado no momento da inscrição;

n) ValorCampoProcesso: classe que representa o conteúdo preenchido pelo usuário para cada objeto do tipo CampoProcesso, solicitado na inscrição;

o) InscricaoProcessoTitulo: classe que representa um título de cobrança gerado por uma inscrição realizada por algum usuário.

5.5 Geração de código

A plataforma possibilita a geração de código-fonte para *frameworks* específicos, desde que seja criado um *plugin* para esta finalidade. O código é gerado através de um *plugin* criado na plataforma, que permite obter as informações necessárias, e gerar as classes através da estrutura do diagrama de classes. Por meio da utilização deste *plugin*, será possível a geração de todas as classes de modelo, que foram previamente modeladas pelo usuário da plataforma. O código gerado será no padrão de model para o *framework* Adianti.

Na Figura 51 é apresentada uma classe de modelo, gerada a partir do *plugin* de geração de código-fonte para o *framework* Adianti. Nela são apresentados os métodos de relacionamento, que representam as associações e agregações existentes entre algumas classes da aplicação. Os métodos que representam agregação na classe ProcessoInscricao são addServico() e addCampoProcesso(), os quais utilizam um vetor de objetos em suas classes, para aramzenar essa estrutura. Já os métodos que representam as associações são set_tipo_processo_inscricao() e get_tipo_processo_inscricao(), os quais fazem associação com a classe TipoProcessoInscricao. O método getInscricoesProcessos(), retorna os objetos vinculados da associação com a classe InscricaoProcesso, e por último, o método getAtividades(), retorna os objetos vinculados da associação com a classe Atividade.

Figura 51 - Métodos de relacionamento da classe ProcessoInscricao

```php
class ProcessoInscricao{

    const TABLENAME = 'ProcessoInscricao';
    const PRIMARYKEY= 'id';
    const IDPOLICY =  'serial';
    private $servicos;
    private $campo_processos;
    private $tipo_processo_inscricaos;

    public function addServico(Servico $object){
        $this->servicos[] = $object;
    }

    public function getServicos(){
        return $this->servicos;
    }

    public function addCampoProcesso(CampoProcesso $object){
        $this->campo_processos[] = $object;
    }

    public function getCampoProcessos(){
        return $this->campo_processos;
    }

    public function get_tipo_processo_inscricao(){
        if(empty($this->tipo_processo_inscricao))
            $this->tipo_processo_inscricao = new TipoProcessoInscricao($this->tipo_processo_inscricao_id);

        return $this->tipo_processo_inscricao;
    }

    public function set_tipo_processo_inscricao(TipoProcessoInscricao $object){
        $this->tipo_processo_inscricao_id = $object->id;
        $this->tipo_processo_inscricao = $object;
    }

    public function getInscricoesProcesso(){
        $criteria = new TCriteria;
        $criteria->add(new TFilter('processo_inscricao_id', '=', $this->id));
        return InscricaoProcesso::getObjects( $criteria );
    }

    public function getAtividades(){
        $criteria = new TCriteria;
        $criteria->add(new TFilter('processo_inscricao_id', '=', $this->id));
        return Atividade::getObjects( $criteria );
    }
```

Fonte: Feito pelo autor

A continuação da classe ProcessoInscricao pode ser obervada na Figura 52, onde são apresentandos os métodos de persistência, responsáveis por carregar, deletar e salvar dados no banco de dados. O método load() é responsável por carregar em memória, os dados do objeto, e os dados agregados das classes ProcessoInscricaoServico e CampoProcesso. O método

store() é reponsável por salvar os dados das classes ProcessoInscricao, bem como os objetos agregados CampoProcesso e ProcesoInscricaoServico. Para realizar a exclusão dos dados é utilizado o método delete(), que exclui primeiramente os dados das classes CampoProcesso e ProcessoInscricaoServico, e por fim, da própria classe ProcessoInscricao.

Figura 52 - Métodos de persistência da classe ProcessoInscricao

```
    public function load($id){
        $this->servicos = parent::loadAggregate("Servico", "ProcessoInscricaoServico",
                                                "processo_inscricao_id", "servico_id", $id);
        $this->campo_processos = parent::loadAggregate("CampoProcesso", "ProcessoInscricaoCampoProcesso",
                                                       "processo_inscricao_id", "campo_processo_id", $id);
        return parent::load($id);
    }

    public function store(){
        parent::store();
        parent::saveAggregate("ProcessoInscricaoServico", "processo_inscricao_id",
                              "servico_id", $this->id, $this->servicos);
        parent::saveAggregate("ProcessoInscricaoCampoProcesso", "processo_inscricao_id",
                              "campo_processo_id", $this->id, $this->campo_processos);
    }

    public function delete($id = NULL){
        $id = isset($id) ? $id : $this->id;
        parent::deleteComposite("ProcessoInscricaoServico", "processo_inscricao_id", $id);
        parent::deleteComposite("ProcessoInscricaoCampoProcesso", "processo_inscricao_id", $id);

        parent::delete($id);
    }
}
```

Fonte: Feito pelo autor

Com a utilização do *plugin*, foi possível gerar todas as classes de modelo para a aplicação. Entretanto, apenas a mais importante foi descrita neste tópico, sendo que as outras classes do sistema podem ser encontradas no apêndice A, do presente trabalho.

5.6 Testes unitários

Para realizar a validação da plataforma, foram desenvolvidas algumas funções, que têm como objetivo comparar a integridade e saída das informações do banco de dados do sistema. As funções criadas são utilizadas para verificar se, a plataforma conseguiu alcançar o seu objetivo de gerar um código correto e coeso, para um *framework* específico. Foram desenvolvidos quatro métodos, a fim de realizar esta verificação, sendo eles, novaInscricao(), listarInscricoes(), listarAtividades() e inscricoesAtividade(). A seguir, os métodos serão apresentados, juntamente com os resultados de sua execução e uma amostra dos dados persistidos por estes, na base de dados.

Na Figura 53 é apresentado o código-fonte de um método criado com a finalidade de simular uma nova inscrição. O método se chama novaInscricao(), e recebe como parâmetro quatro variáveis (um objeto pessoa, o código de algum processo de inscrição, um vetor de

códigos de atividades e um vetor de objetos da classe Modalidade). Internamente este método carrega os objetos de atividades, modalidades, processo de inscrição em memória, para posterior gravação na base de dados, através da classe InscricaoProcesso, a qual executa o método store(), que por sua vez, realiza esta operação.

Figura 53 - Método que cria uma nova inscrição

```
function novaInscricao($pessoa, $processoInscricaoId, $atividades, $modalidades)
{
    try {
        TTransaction::open('inscricoes');
        $inscricaoAtividade = new InscricaoAtividade();
        $inscricaoProcesso  = new InscricaoProcesso();
        $processoInscricao  = new ProcessoInscricao($processoInscricaoId);

        $inscricaoProcesso->processo_inscricao = $processoInscricao;
        $inscricaoProcesso->pessoa = $pessoa;

        if($atividades)
        {
            foreach ($atividades as $atividade_id)
            {
                $atividade = new Atividade($atividade_id);
                if($modalidades)
                {
                    foreach ($modalidades as $modalidade)
                    {
                        if($modalidade->atividade_id = $atividade->id)
                        {
                            $atividade->setModalidadeSelecionada($modalidade);
                        }
                    }
                }

                $inscricaoAtividade->atividade = $atividade;
                $inscricaoProcesso->addInscricaoAtividade($inscricaoAtividade);
            }
        }

        $inscricaoProcesso->store();
        TTransaction::close();

        echo "Inscrição criada com sucesso";

    } catch (Exception $e) {
        TTransaction::rollback();
        echo "Ocorreu um erro ao salvar a inscrição";
    }
}
```

Fonte: Feito pelo autor

Após ligar os *logs* de comandos SQL executados na aplicação, foi executado o método novaInscricao(), que retornou uma mensagem de sucesso, “Inscrição criada com sucesso”. A fim de validar a execução do método, foram utilizados os *logs* previamente habilitados. Através dos *logs* foi possível verificar todos os comandos SQL, executados internamente no método, como pode ser visto na Figura 54. Pode ser constatado, que as instruções de SQL gerados, condizem perfeitamente com os métodos de persistência executados.

Figura 54 - Comandos SQL executados pelo método novaInscricao

```
SELECT * FROM processo_inscricao WHERE id = 5;
SELECT * FROM processo_inscricao_campo_processo WHERE processo_inscricao_id = 5;
SELECT * FROM processo_inscricao_servico WHERE processo_inscricao_id = 5;

SELECT * FROM atividade WHERE id = 10;
SELECT * FROM atividade WHERE id = 11;

INSERT INTO inscricao_processo (id, pessoa_id, processo_inscricao_id, dt_inscricao)
VALUES (NULL, 2, 5, '2015-10-01');

INSERT INTO inscricao_atividade (id, atividade_id, inscricao_processo_id, modalidade_id)
VALUES (NULL, 10, 15, 3);

INSERT INTO inscricao_atividade (id, atividade_id, inscricao_processo_id, modalidade_id)
VALUES (NULL, 11, 15, 4);

INSERT INTO inscricao_processo_titulo (id, inscricao_processo_id, valor, data_emissao, data_vencimento)
VALUES (NULL, 15, 50, 25.30, '2015-10-01', '2015-10-10');
```

Fonte: Feito pelo autor

Para validar a listagem de inscrições em um determinado processo, a partir das classes geradas pela plataforma, foi desenvolvido o método listarInscricoes(), o qual é demonstrado na Figura 55. Este método recebe, por parâmetro, o código de um processo de inscrição, que é utilizado para instanciar a classe ProcessoInscricao, sobre a qual é executado o método getInscricoesProcesso(). Este, retorna um vetor de objetos da classe InscricaoProcesso. Os objetos retornados pelo método getInscricaoProcesso(), são utilizados para gerar uma listagem de inscritos no processo de inscrição. Os resultados exibidos pela Figura 56 puderam comprovar que, os métodos de relacionamento utilizados para carregar os dados funcionaram perfeitamente, conforme o esperado.

Figura 55 - Método que lista inscrições em um processo

```
function listarInscricoes($processo_inscricao_id)
{
    try {
        TTransaction::open('inscricoes');

        $processoInscricao = new ProcessoInscricao($processo_inscricao_id);
        $inscricoes = $processoInscricao->getInscricoesProcesso();

        if($inscricoes)
        {
            foreach($inscricoes as $inscricao)
            {
                echo "{$inscricao->processo_inscricao->descricao} - {$inscricao->pessoa->nome} -
                      {$inscricao->dt_inscricao} <br>";
            }
        }
        else
        {
            echo "Não foram encontradas inscrições!";
        }

        TTransaction::close();

    } catch (Exception $e) {
        TTransaction::rollback();
        echo "Ocorreu um erro ao listar as inscrições";
    }
}
```

Fonte: Feito pelo autor

A fim de validar a integridade dos dados de uma nova inscrição, o método listarInscricoes(), foi executado, passando como parâmetro o código do proceso de inscrição

“cinco” (Prova de nivelamento do Projeto I), que por sua vez, retornou as informações demonstradas na Figura 56.

Figura 56 - Retorno do método “listarInscricoes”

```
Prova de nivelamento do Projeto I - João Carlos    - 01/10/2015
Prova de nivelamento do Projeto I - Rubens Fonceca - 29/09/2015
Prova de nivelamento do Projeto I - Matheus Dias   - 25/09/2015
Prova de nivelamento do Projeto I - Augusto Wolf   - 22/09/2015
```

Fonte: Feito pelo autor

Após a execução do método listarInscricoes(), foi possível verificar todos os comandos de SQL executados internamente, por meio dos logs da aplicação. Os comandos SQL obtidos, estão listados na Figura 57, onde pode ser constatado que as instruções de SQL geradas, condizem perfeitamente com os métodos de persistência executados.

Figura 57 - Logs de SQL executados internamente pelo método “listarInscricoes”

```
SELECT * FROM processo_inscricao WHERE id = 5;
SELECT * FROM processo_inscricao_campo_processo WHERE processo_inscricao_id = 5;
SELECT * FROM processo_inscricao_servico WHERE processo_inscricao_id = 5;

SELECT * FROM inscricao_processo WHERE processo_inscricao_id = 5;

SELECT * FROM inscricao_atividade WHERE inscricao_processo_id = 15;
SELECT * FROM valor_campo_processo WHERE inscricao_processo_id = 15;
SELECT * FROM inscricao_processo_titulo WHERE inscricao_processo_id = 15;

SELECT * FROM pessoa WHERE id = 2;
SELECT * FROM pessoa WHERE id = 5;
SELECT * FROM pessoa WHERE id = 6;
SELECT * FROM pessoa WHERE id = 10;
```

Fonte: Feito pelo autor

Para realizar a validação do relacionamento da classe ProcessoInscricao, com a classe Atividade, foi desenvolvido o método listarAtividades(), que tem como objetivo listar todas as atividades de um determinado processo de inscrição. Este método recebe como parâmetro o código de um processo de inscrição, e internamente, instancia a classe ProcessoInscricao, executando o método getAtividades(). Este método retorna um vetor de objetos da classe Atividade, para posterior exibição dos dados destas. Na Figura 58 é apresentado o código-fonte do método listarAtividades().

Figura 58 - Método que lista as atividades de um processo

```php
function listarAtividades($processo_inscricao_id)
{
    try {
        TTransaction::open('inscricoes');

        $processoInscricao = new ProcessoInscricao($processo_inscricao_id);
        $atividades = $processoInscricao->getAtividades();

        if($atividades)
        {
            foreach($atividades as $atividade)
            {
                echo "{$atividade->titulo} - {$atividade->vagas} - {$atividade->local} <br>";
            }
        }
        else
        {
            echo "Não foram encontradas atividades!";
        }

        TTransaction::close();

    } catch (Exception $e) {
        TTransaction::rollback();
        echo "Ocorreu um erro ao listar as atividades de um processo de inscrição";
    }
}
```

Fonte: Feito pelo autor

O método “listarAtividades” foi executado passando como parâmetro o código do proceso de inscrição “5” (Prova de nivelamento do Projeto I), o qual retornou as atividades cadastradas nele, conforme a listagem demostrada na Figura 59.

Figura 59 - Atividades do processo de inscrição “Prova de nivelamento Projeto I”

```
Prova de nivelamento em Inglês nível I   - 30 - Sala 205-03
Prova de nivelamento em Inglês nível II  - 30 - Sala 206-03
Prova de nivelamento em Espanhol nível I - 25 - Sala 405-11
```

Fonte: Feito pelo autor

Após a execução do método “listarAtividades”, foi possível verificar todos os comandos de SQL executados internamente, dentro do método, por meio dos logs da aplicação. Estes comandos SQL estão listados na Figura 60, onde pode ser constatado que, as instruções de SQL geradas, condizem perfeitamente com os métodos de persistência executados.

Figura 60 - Logs de SQL executados internamente pelo método “listarAtividades”

```sql
SELECT * FROM processo_inscricao WHERE id = 5;
SELECT * FROM processo_inscricao_campo_processo WHERE processo_inscricao_id = 5;
SELECT * FROM processo_inscricao_servico WHERE processo_inscricao_id = 5;

SELECT * FROM ativiadde WHERE processo_inscricao_id = 5;
```

Fonte: Feito pelo autor

Na Figura 61 é apresentado o código-fonte do método inscricoesAtividade(), criado com a finalidade de exibir as inscrições realizadas em uma determinada atividade. Este método recebe como parâmetro o código de uma atividade. Internamente, ele carrega o objeto da classe Atividade, sobre o qual é executado o método getInscricoesAtividades(), que retorna um vetor de objetos da classe InscricaoAtividade. Desta forma, é possível obter uma listagem das pessoas que se inscreveram em uma determinada atividade.

Figura 61 - Método que lista as inscrições em uma determinada atividade

```
function inscricoesAtividade($atividade_id)
{
    try {
        TTransaction::open('inscricoes');

        $atividade = new Atividade($atividade_id);
        $inscricoesAtividades = $atividade->getInscricoesAtividade();

        if($inscricoesAtividades)
        {
            foreach($inscricoesAtividades as $inscricaoAtividade)
            {
                $inscricaoProcesso = new InscricaoProcesso($inscricaoAtividade->inscricao_processo_id);
                echo "{$inscricaoProcesso->id} - {$atividade->titulo} -
                      {$inscricaoProcesso->pessoa->nome} - {$inscricaoProcesso->dt_inscricao}";
            }
        }
        else
        {
            echo "Não foram encontradas atividades!";
        }

        TTransaction::close();

    } catch (Exception $e) {
        TTransaction::rollback();
        echo "Ocorreu um erro ao listar as atividades de um processo de inscrição";
    }
}
```

Fonte: Feito pelo autor

Ao executar o método inscricoesAtividade(), passando como parâmetro a atividade de código “10” (Prova de nivelamento em Inglês nível I), foram exibidos os dados de inscrição de uma pessoa inscrita nesta atividade, conforme pode ser observado na Figura 62.

Figura 62 - Inscritos na atividade “Prova de nivelamento em Inglês nível I”

```
15 - Prova de nivelamento em Inglês nível I - João Carlos - 01/10/2015
```

Fonte: Feito pelo autor

Após a execução do método inscricoesAtividade(), foi possível verificar todos os comandos de SQL executados internamente, dentro do método, por meio dos logs da aplicação que foram previamente ativados. Estes comandos SQL estão listados na Figura 63, onde pode ser constatado que, as instruções de SQL geradas, condizem perfeitamente com os métodos de persistência executados.

Figura 63 - Logs de SQL executados internamente pelo método “inscriçõesAtividade”

```
SELECT * FROM horario WHERE atividade_id = 10;
SELECT * FROM modalidade WHERE atividade_id = 10;
SELECT * FROM informacao_atividade WHERE atividade_id = 10;
SELECT * FROM campo_atividade WHERE atividade_id = 10;
SELECT * FROM atividade WHERE id = 10

SELECT * FROM inscricao_processo WHERE id = 15;
SELECT * FROM inscricao_atividade WHERE inscricao_processo_id = 15;
SELECT * FROM valor_campo_processo WHERE inscricao_processo_id = 15;
SELECT * FROM inscricao_processo_titulo WHERE inscricao_processo_id = 15;
SELECT * FROM pessoa WHERE id = 2;
```

Fonte: Feito pelo autor

5.7 Resultados

Os testes unitários criados tiveram o objetivo de executar métodos das classes de modelo criadas pela plataforma. O código-fonte das classes foi criado por meio do *plugin* de exportação, construído no presente trabalho. Este *plugin* de exportação recebeu o modelo de classes no formato JSON, e gerou as classes de modelo compatíveis com o Adianti *Framework*, também usado pela Univates, no desenvolvimento do Sistema de Inscrições, que serviu de estudo de caso.

Por meio de análise e resultados dos testes, foi possível verificar que os mesmos são compatíveis com o comportamento esperado pelas classes. Tal conclusão foi possível a partir da verificação dos dados persistidos na base de dados, após a execução dos testes unitários.

Desta maneira, verificou-se que, a camada de persistência gerada pelo *plugin* de exportação é compatível com a aplicação real, e demonstrou, o comportamento adequado para as funções executadas.

6 CONSIDERAÇÕES FINAIS

Por meio dos estudos realizados, foi possível constatar a importância da modelagem de software no desenvolvimento de um novo produto de software. Quando esta não é executada de forma adequada, pode influenciar diretamente no tempo de desenvolvimento, bem como na qualidade e manutenibilidade do produto final.

Atualmente, empresas de desenvolvimento de software buscam aumentar a sua produtividade, a fim de se manterem competitivas. Uma das maneiras encontradas para o aumento da produtividade é a utilização da geração automática de artefatos, que por sua vez, pode ser realizada a partir de modelos de projeto de software, tais como o modelo ER e UML.

A plataforma criada no presente trabalho, demonstrou ser capaz de gerar artefatos de alto nível para *frameworks* específicos, de maneira colaborativa, por meio de *plugins* que podem ser personalizados para gerar modelos, a partir de fontes externas (Ex: base de dados), bem como transformar os modelos criados em artefatos concretos (Ex: classes de modelo). Com isso, consegue-se atingir grande produtividade em estágios iniciais de um projeto.

Após a conclusão da plataforma MyDiagrams, foi possível validá-la em um ambiente real. Nesta validação, as classes de modelo geradas através do *plugin* de geração de código para o Adianti *Framework*, foram submetidas a testes unitários. Os resultados obtidos através desta validação foram analisados, a fim de verificar se, a solução proposta, cumpriu os objetivos descritos no presente documento. A validação da plataforma, em um ambiente real, mostrou que a mesma pode ser usada em cenários reais. Dessa forma, pode-se concluir que o presente trabalho cumpriu com os objetivos propostos em sua metodologia, bem como os estudos bibliográficos e a teorização dos conceitos abordados.

REFERÊNCIAS

ABRAN, Alain; MOORE, James W.; BOURQUE, Pierre; DUPUIS, Robert; TRIPP, Leonard L. **Guide to the Software Engineering Body of Knowledge**. California: IEEE, 2004.

ALEXANDER, Christopher. **A Pattern Language: Towns, Buildings, Construction**. Estados Unidos: Oxford University Press, 1977.

BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. **UML: guia do usuário**. 2. ed. Rio de Janeiro: Elsevier, 2012.

FONSECA, J. J. S. **Metodologia da pesquisa científica**. Fortaleza: UEC, 2002.

FOWLER, Martin. **UML Essencial**: **um breve guia para a linguagem-padrão de modelagem de objetos**. 3. ed. Porto Alegre: Bookman, 2005.

FOWLER, Martin. **Patterns of Enterprise Application Architecture**. Boston: Addison-Wesley, 2003

GAMMA, Erich; HELM, Richard; JOHNSON, Ralph. **Padrões de projeto: soluções reutilizáveis de software orientado a objetos**. Porto Alegre: Bookman, 2008.

GUEDES, Gilleanes T. A. **UML 2: uma abordagem prática**. São Paulo: Novatec, 2009.

HEUSER, Carlos Alberto. **Projeto de banco de dados**. 6. ed. Porto Alegre: Bookman, 2009.

IEEE, **Institute of Electrical and Electronics Engineers: Software Engineering**, IEEE Standard 610.12 – 1990, IEEE, 1993.

MARCONI, Marina de Andrade; LAKATOS, Eva Maria. **Fundamentos de Metodologia Científica**. 5ª Edição. São Paulo: Editora Atlas, 2003.

O'BRIEN, L.; MERSON, P.; BASS, L. **Quality Attributes for Service-Oriented Architectures**. In: International Workshop on Systems Development in SOA Environments, IEEE Computer Society Washington, DC, USA, 7 p., 2007. Disponível em: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4273291. Acesso em: 15 set. 2014.

PFLEEGER, Shari Lawrence. **Engenharia de software: teoria e prática**. 2. ed. São Paulo: Prentice Hall, 2007.

PRESSMAN, Roger S.; PENTADO, Rosângela Ap. D. **Engenharia de software**. 6. ed. Porto Alegre: AMGH, 2010.

SILBERSCHATZ, Abraham; KORTH, Henry F.; SUDARSHAN, S. **Sistema de banco de dados**. Rio de Janeiro: Elsevier, 2012.

SANTOS, Antonio Raimundo dos. **Metodologia científica: a construção do conhecimento**. 2ª ed. Rio de Janeiro: DP&A editora, 1999.

SBROCCO, José Henrique Teixeira de Carvalho; MACEDO, Paulo Cesar. **Metodologias ágeis: engenharia de software sob medida**. 1ª ed. São Paulo: Érica, 2012.

SOMMERVILLE, Ian; MELNIKOFF, Selma Shin Shimizu; ARAKAKI, Reginaldo. **Engenharia de software**. 9. ed. São Paulo: Pearson Prentice Hall, 2011.

SWEBOK. **The Guide to the Software Engineering Body of Knowledge**. Disponível em: <http://www.swebok.org/>. Acesso em Setembro 2014.

WAINER, Jacques. **Métodos de pesquisa quantitativa e qualitativa para a Ciência da Computação**. In: KOWALTOWSKI, Tomasz; BREITMAN, Karin; organizadores. Atualizações em Informática 2007. Rio de Janeiro: Ed. PUC-Rio; Porto Alegre: Sociedade Brasileira de Computação, 2007.

APÊNDICES

APÊNDICE A – Classes de modelo geradas através do *plugin* de geração de código-fonte para o Adianti *Framework*.

Código-fonte da classe Atividade:

```
<?php

class Atividade

{
    const TABLENAME = 'Atividade';
    const PRIMARYKEY= 'id';
    const IDPOLICY =  'serial'; // {max, serial}

    private $horarios;
    private $modalidades;
    private $informacao_atividades;
    private $processo_inscricaos;
    private $campo_atividades;

    public function __construct($id = NULL, $callObjectLoad = TRUE)
    {
        parent::__construct($id, $callObjectLoad);
        parent::addAttribute('titulo');
        parent::addAttribute('descricao');
        parent::addAttribute('local');
        parent::addAttribute('dt_ocorrencia');
        parent::addAttribute('horario');
        parent::addAttribute('vagas');
        parent::addAttribute('obrigatoria');
    }

    public function addHorario(Horario $object)
    {
        $this->horarios[] = $object;
    }
    public function getHorarios()
    {
        return $this->horarios;
    }
    public function addmodalidade(modalidade $object)
    {
        $this->modalidades[] = $object;
    }
    public function getmodalidades()
    {
        return $this->modalidades;
```

```
    }
    public function addInformacaoAtividade(InformacaoAtividade $object)
    {
        $this->informacao_atividades[] = $object;
    }
    public function getInformacaoAtividades()
    {
        return $this->informacao_atividades;
    }
    public function get_processo_inscricao()
    {
        if(empty($this->processo_inscricao))
            $this->processo_inscricao = new ProcessoInscricao($this->processo_inscricao_id);

        return $this->processo_inscricao;
    }
    public function set_processo_inscricao(ProcessoInscricao $object)
    {
        $this->processo_inscricao_id = $object->id;
        $this->processo_inscricao = $object;
    }
    public function addCampoAtividade(CampoAtividade $object)
    {
        $this->campo_atividades[] = $object;
    }

   public function getCampoAtividades()
    {
        return $this->campo_atividades;
    }

       public function getInscricoesAtividade()
       {
              $criteria = new TCriteria;
        $criteria->add(new TFilter('atividade_id', '=', $this->id));
        return InscricaoAtividade::getObjects( $criteria );
       }

    public function load($id)
    {
        $this->atividade = parent::loadComposite("Horario", "atividade_id", $id);
        $this->atividade = parent::loadComposite("Modalidade", "atividade_id", $id);
        $this->atividade = parent::loadComposite("InformacaoAtividade", "atividade_id", $id);
        $this->campo_atividades = parent::loadAggregate("CampoAtividade",
"AtividadeCampoAtividade", "atividade_id", "campo_atividade_id", $id);
        return parent::load($id);
    }
    public function store()
    {
        parent::store();
        parent::saveComposite("Horario", "atividade_id", $this->id, $this->horarios);
        parent::saveComposite("Modalidade", "atividade_id", $this->id, $this->modalidades);
        parent::saveComposite("InformacaoAtividade", "atividade_id", $this->id, $this-
>informacao_atividades);
        parent::saveAggregate("AtividadeCampoAtividade", "atividade_id", campo_atividade_id,
$this->id, $this->campo_atividades);
    }
    public function delete($id = NULL)
    {
        $id = isset($id) ? $id : $this->id;
        parent::deleteComposite("Horario", "atividade_id", $id);
        parent::deleteComposite("Modalidade", "atividade_id", $id);
        parent::deleteComposite("InformacaoAtividade", "atividade_id", $id);
        parent::deleteComposite("AtividadeCampoAtividade", "atividade_id", $id);

        parent::delete($id);
    }
}
?>
```

Código-fonte da classe InscricaoProcesso:

```
<?php
class InscricaoProcesso
{
    const TABLENAME = 'InscricaoProcesso';
    const PRIMARYKEY= 'id';
    const IDPOLICY =  'serial'; // {max, serial}

    private $inscricao_atividades;
    private $processo_inscricaos;
    private $valor_campo_processos;
    private $inscricao_processo_titulos;
       private $pessoas;

    public function __construct($id = NULL, $callObjectLoad = TRUE)
    {
        parent::__construct($id, $callObjectLoad);
        parent::addAttribute('validade');
        parent::addAttribute('codigo_titulo');
        parent::addAttribute('codigo_pessoa');
        parent::addAttribute('dt_inscricao');
        parent::addAttribute('processo_inscricao_id');
    }

    public function addInscricaoAtividade(InscricaoAtividade $object)
    {
        $this->inscricao_atividades[] = $object;
    }
    public function getInscricaoAtividades()
    {
        return $this->inscricao_atividades;
    }
    public function get_processo_inscricao()
    {
        if(empty($this->processo_inscricao))
            $this->processo_inscricao = new ProcessoInscricao($this->processo_inscricao_id);

        return $this->processo_inscricao;
    }
    public function set_processo_inscricao(ProcessoInscricao $object)
    {
        $this->processo_inscricao_id = $object->id;
        $this->processo_inscricao = $object;
    }
    public function addValorCampoProcesso(ValorCampoProcesso $object)
    {
        $this->valor_campo_processos[] = $object;
    }
    public function getValorCampoProcessos()
    {
        return $this->valor_campo_processos;
    }
    public function addInscricaoProcessoTitulo(InscricaoProcessoTitulo $object)
    {
        $this->inscricao_processo_titulos[] = $object;
    }
    public function getInscricaoProcessoTitulos()
    {
        return $this->inscricao_processo_titulos;
    }
    public function get_pessoa()
    {
        if(empty($this->pessoa))
            $this->pessoa = new Pessoa($this->pessoa_id);

        return $this->pessoa;
    }
    public function set_pessoa(Pessoa $object)
    {
        $this->pessoa_id = $object->id;
        $this->pessoa = $object;
    }
```

```
    public function load($id)
    {
        $this->inscricao_processo = parent::loadComposite("InscricaoAtividade",
"inscricao_processo_id", $id);
        $this->inscricao_processo = parent::loadComposite("ValorCampoProcesso",
"inscricao_processo_id", $id);
        $this->inscricao_processo = parent::loadComposite("InscricaoProcessoTitulo",
"inscricao_processo_id", $id);
        return parent::load($id);
    }
    public function store()
    {
        parent::store();
        parent::saveComposite("InscricaoAtividade", "inscricao_processo_id", $this->id, $this-
>inscricao_atividades);
        parent::saveComposite("ValorCampoProcesso", "inscricao_processo_id", $this->id, $this-
>valor_campo_processos);
        parent::saveComposite("InscricaoProcessoTitulo", "inscricao_processo_id", $this->id,
$this->inscricao_processo_titulos);
    }
    public function delete($id = NULL)
    {
        $id = isset($id) ? $id : $this->id;
        parent::deleteComposite("InscricaoAtividade", "inscricao_processo_id", $id);
        parent::deleteComposite("ValorCampoProcesso", "inscricao_processo_id", $id);
        parent::deleteComposite("InscricaoProcessoTitulo", "inscricao_processo_id", $id);

        parent::delete($id);
    }
}
?>
```

Código-fonte da classe InscricaoAtividade:

```
<?php
class InscricaoAtividade
{
    const TABLENAME = 'InscricaoAtividade';
    const PRIMARYKEY= 'id';
    const IDPOLICY =  'serial'; // {max, serial}

    private $atividades;
    private $valor_campo_atividades;
    private $valor_campo_atividades;

    public function __construct($id = NULL, $callObjectLoad = TRUE)
    {
        parent::__construct($id, $callObjectLoad);
        parent::addAttribute('dt_inscricao');
        parent::addAttribute('atividade_id');
        parent::addAttribute('inscricao_processo_id');
    }

    public function get_atividade()
    {
        if(empty($this->atividade))
            $this->atividade = new Atividade($this->atividade_id);

        return $this->atividade;
    }
    public function set_atividade(Atividade $object)
    {
        $this->atividade_id = $object->id;
        $this->atividade = $object;
    }
    public function addValorCampoAtividade(ValorCampoAtividade $object)
    {
        $this->valor_campo_atividades[] = $object;
    }

    public function getValorCampoAtividades()
    {
        return $this->valor_campo_atividades;
```

```
    }
    public function load($id)
    {
        $this->inscricao_atividade = parent::loadComposite("InscricaoAtividade",
"inscricao_atividade_id", $id);
        return parent::load($id);
    }
    public function store()
    {
        parent::store();
        parent::saveComposite("ValorCampoAtividade", "inscricao_atividade_id", $this->id,
$this->valor_campo_atividades);
    }
    public function delete($id = NULL)
    {
        $id = isset($id) ? $id : $this->id;
        parent::deleteComposite("ValorCampoAtividade", "inscricao_atividade_id", $id);

        parent::delete($id);
    }
}
?>
```