CAPA 1 – EM OUTRO DOCUMENTO

CAPA 2 – EM OUTRO DOCUMENTO

CAPA 3 – EM OUTRO DOCUMENTO

RESUMO

A sociedade depende cada vez mais de algum tipo de software, sendo que o mesmo está se tornando imprescindível e cada vez mais utilizado nas atividades do dia a dia dos mais diversos setores. Essa dependência tem feito com que os usuários sejam cada vez mais críticos e exigentes em relação à qualidade dos produtos. Por outro lado, os profissionais que trabalham com desenvolvimento, tem o desafio de construir softwares que tragam maior satisfação aos seus usuários, e ao mesmo tempo tenham 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 que sejam utilizados diretamente na construção do produto final, a modelagem traz ganhos consistentes de produtividade. Tendo em vista esse 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 FIGURAS

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 como: industria, comércio e serviços. Além disso, softwares podem se utilizados em sistemas de diversas áreas como: 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 desenvolvesse novas tecnologias 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. Essas 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 a 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 etapa fundamental da execução de um projeto, pois nela são discutidos quais são os principais conceitos envolvidos no projeto bem como seus relacionamentos. A execução inadequada da modelagem, ou mesmo sua não execução implica diretamente na qualidade e manutenibilidade do produto final.

Diferentes modelos foram criadas para representar estruturas de arquitetura de software, sendo que podemos destacar o modelo UML (*Unified Modeling Language*), que tem seu foco na modelagem de uma aplicação orientada a objetos e o modelo ER (*Entity Relationship*), que tem seu foco na modelagem de estruturas de bancos de dados relacionais. Para criar esses modelos, ao longo do tempo foram criadas ferramentas, que ajudam 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 trazer ganhos consistentes de produtividade por meio da geração de artefatos concretos que podem ser utilizados diretamente na construção do produto final. Como exemplo, pode ser citado que algumas ferramentas de modelagem oferecem funcionalidades tais como 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 num modelo especifico e possuem limitações quanto ao que pode ser gerado a partir deste modelo. Além disso deve-se lembrar que a maioria das ferramentas mais reconhecidas no mercado como Astah, Rose, Star UML, Enterprise Architecture, Umbrello, não permitem ou possuem limitações em relação a 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) Estudo comparativo de ferramentas para modelagem de software;

c) Implementação de uma plataforma básica de modelagem online;

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

e) 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 proposta do presente trabalho onde serão abordadas as funcionalidades que serão desenvolvidas na plataforma de modelagem de software.

Por fim o capítulo 5 apresenta as considerações parciais 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, e outros estão relacionados. A presente proposta também possui um forte viés de arquitetura e reuso de software, tendo em vista o foco em modelagem de software, portanto os assuntos Arquitetura de software e reuso também serão abordado. 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. Dessa forma, o assunto padrões de projeto também será explorado.

2.1 Engenharia de Software

Softwares vem se tornando 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)

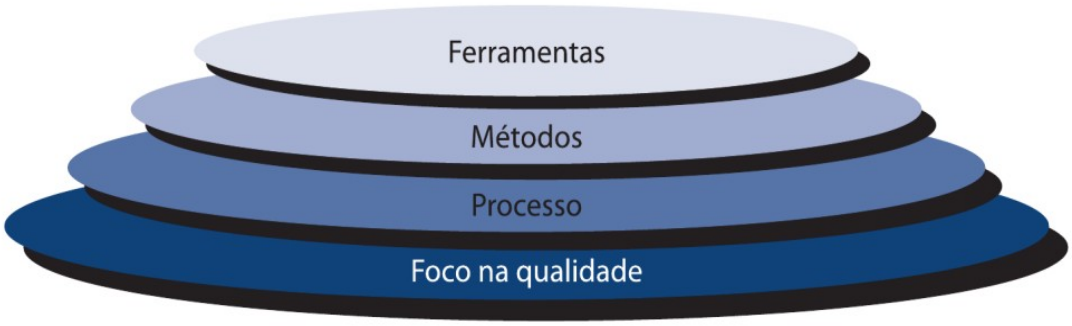
O objetivo da Engenharia de Software é viabilizar métodos, padrões, e estruturas que irão auxiliar no desenvolvimento de um software, tendo como foco a qualidade. Para alcançar

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 a 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 se apoiar num 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 construir eles. A ultima camada é a de ferramentas e tem como objetivo fornecer apoio automatizado e semi-automatizado 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 seus temas. Com a iniciativa da IEEE, foi criado o SWEBOK (*Software*

Engineering Body of Knowledge). Segundo o SWEBOK (2013), o livro é um guia de uso e aplicação 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 do software;

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 estabelecer 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 de 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 Bosh (2000) 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 sua definição de arquitetura bem estruturada, pois, consequentemente, ele terá um maior número de funcionalidades e processos, precisando 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 influencia 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 é muito próxima, 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: a arquitetura deve ser utilizar 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 e um só componente ou em poucos componentes. Reduzindo assim o custo e problemas de validação;

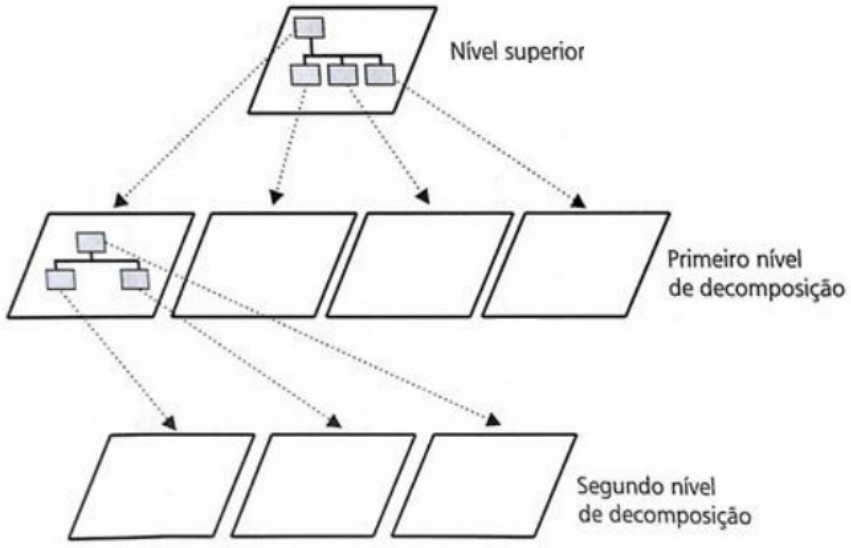
d) Disponibilidade: ao projetar a arquitetura, ela deve incluir componentes redundantes, para que seja possível substituir e atualizar componentes sem a necessidade de parar o sistema; (SOMMERVILLE, 2011)

e) Manutenção: a arquitetura do sistema deve ser projetada a partir de componentes autocontidos e 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, e definindo também a maneira que será feita 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 se encontram um nível acima de um módulo. Considerando o nível superior como o mais abstrato por ocultar as informações dos detalhes funcionais e de processamento. Essas 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 divido 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, tais como:

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 explicitas 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 interoperam. 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 irá abordar o conceito de arquitetura em camadas e orientada a serviços, pois nos próximos capítulos elas serão utilizadas como referência.

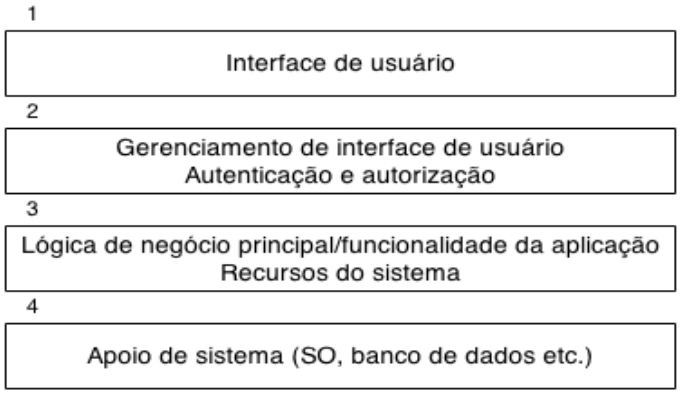
2.2.1 Arquitetura em camadas

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, os serviços dela podem ser disponibilizados para os usuários. Enquanto não ocorrer 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



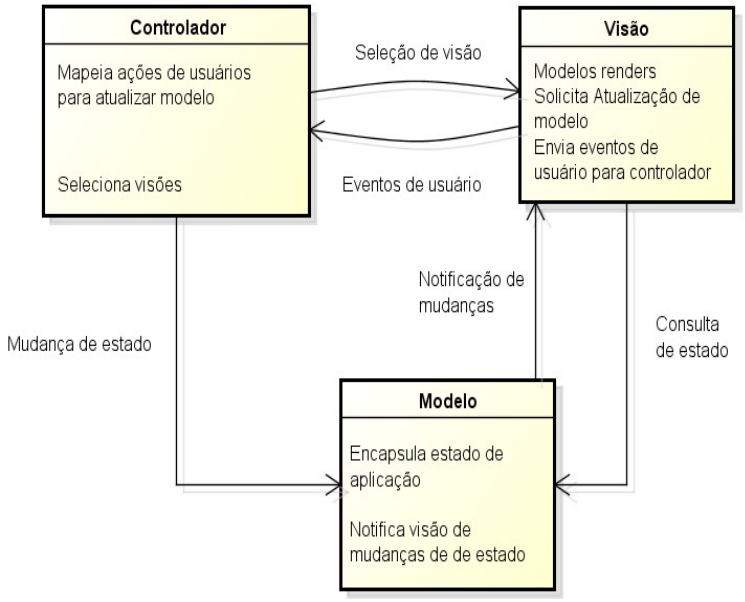
Fonte: Sommerville(2011, p. 111).

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, o que o torna 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 gerência 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.

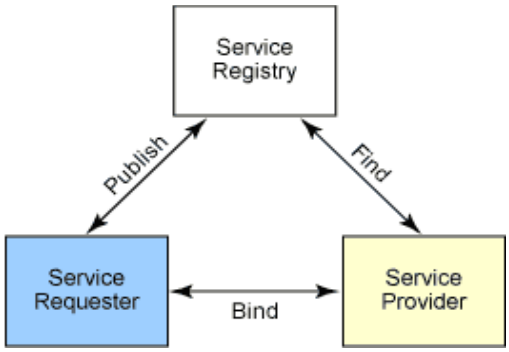
Tem como foco a interoperabilidade, podendo ser adaptável em quase todas as situações. Ela já está em constante utilização no desenvolvimento de aplicações (O'BRIEN; MERSON;BASS, 2007).

Em uma SOA, todos os componentes de software são modeladas como serviços. Tem como premissa, que todas as tarefas ou processos sejam construídos para que sejam designados como serviços para serem acessados através da rede. O 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 software geralmente já possui vários softwares os quais possuem semelhança entre os sistemas que já foram desenvolvidos por ela, os quais tem 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 caixa-branca. A reutilização caixa-preta consiste em reutilizar componentes inteiros, sem modificá-los, como um conjunto de funções de calculo 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: Nesse 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: Nesse nível são reutilizados objetos de uma biblioteca sem precisar reescrever um novo código;

c) Nível de componentes: Nesse nível se faz reúso de componentes que são coleção de objetos e classes de objetos 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: Nesse nível é feito o reúso completo de um sistema, onde são feitos alguns ajustes, adicionando novas funcionalidades através de adição e modificação de 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 que são: criar 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, logico e físico. Tradicionalmente o modelo conceitual é representado pelos diagramas ER e o modelo logico transforma o modelo ER em um modelo relacional, o qual define as tabelas e 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 a sua simplicidade, o que facilita o trabalho dos programadores, se comparado com modelos de dados anteriores como o de rede ou modelo hierárquico. (SILBERSCHATZ; KORTH; SUDARSHAN, 2012)

Um banco de dados relacional é composto por um conjunto de tabelas (entidades), onde cada uma tabela recebe um nome único, o qual a representa. Cada tabela possui campos (atributos) as quais também possuem um nome único e um tipo especifico 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 a 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 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. Num 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 as 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 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). Está 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: para realizar esta operação, 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 essa linguagem se tornou 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 ultima instância, reutilizáveis. Isso quer dizer que os próximos projetos 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 de 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 da 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 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) 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. (FOWLER, 2005, p. 25)

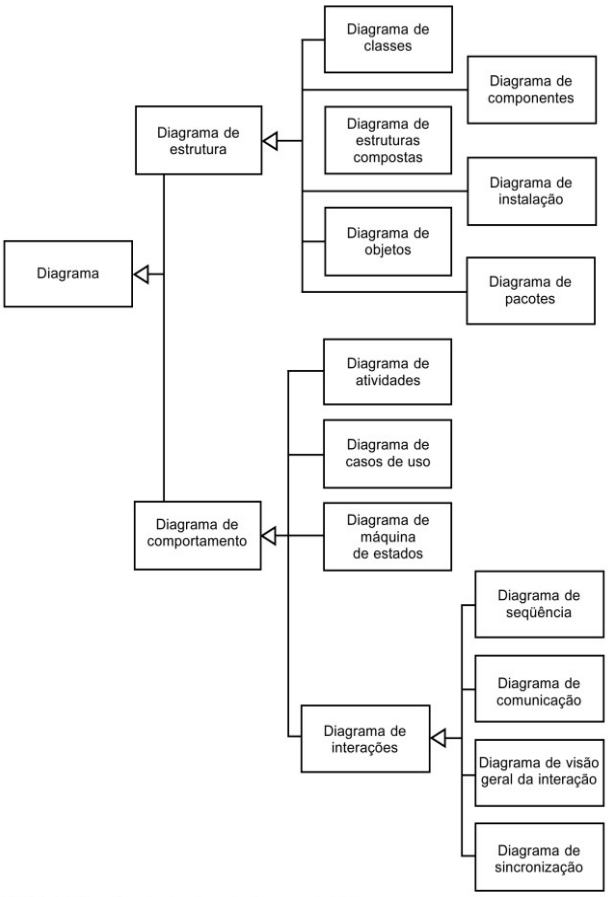
Deve-se lembrar que a UML não é uma linguagem de programação, e sim de modelagem de software. Tem como objetivo auxiliar engenheiros de software a definir as características do sistema, como os requisitos, comportamento, estrutura lógica, processos e até mesmo a estrutura física necessária para a implantação 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, 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 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 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: 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, Gilleanes, 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 de 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)

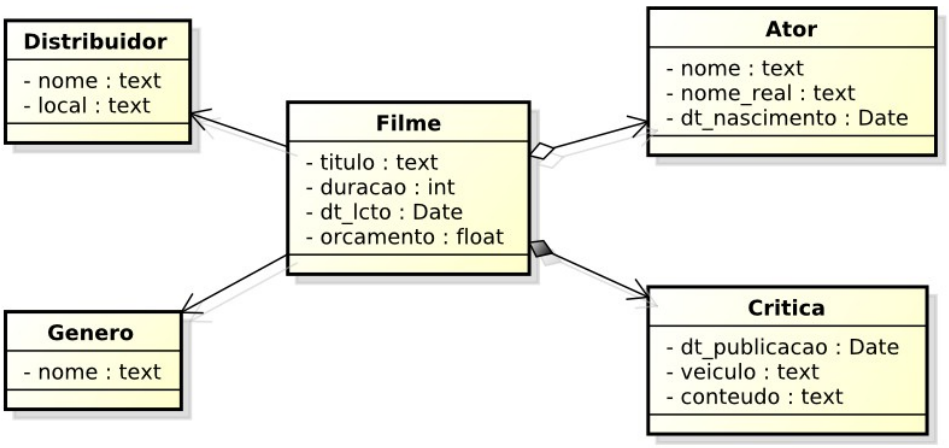
Como visto, a UML possui 13 diagramas. O presente trabalho tem como foco a geração de aplicações a partir do modelo de arquitetura de software. Assim, o foco será dado no modelo de classes, que é justamente o modelo que possui essa finalidade dentro dos diagramas da UML.

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 se relacionam, se complementam 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, serão apresentados nos capítulos 2.5.2 e 2.5.3.

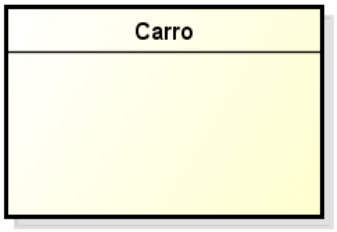
2.5.2 Conceitos sobre orientação a objetos

2.5.2.1 Classes

Na modelagem de um sistema orientada 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 até 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 de classes abstratas. (GUEDES, Gilleanes, 2009)

Figura 11 - Exemplo de uma classe



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

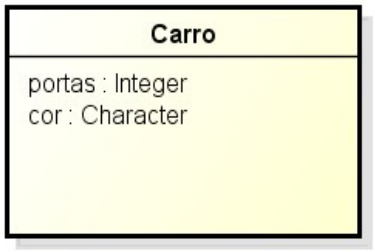
A Figura 11 apresenta o exemplo de uma classe que não possui métodos nem atributos. Nesta imagem tem apenas uma divisão que contém o nome da classe, pois não é obrigatório representar uma classe totalmente expandida, embora seja mais comum. (GUEDES, Gilleanes, 2009)

2.5.2.2 Atributos

Numa 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 as 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.

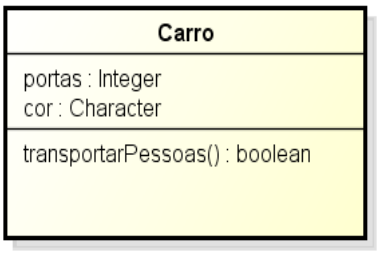
A Figura 12 apresenta o exemplo de uma classe com duas divisões, onde na segunda divisão se encontram os atributos da classe, bem como o seus nomes e respectivos tipos.

2.5.2.3 Métodos

Os métodos representam as funções que uma classe pode executar, representando assim 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.

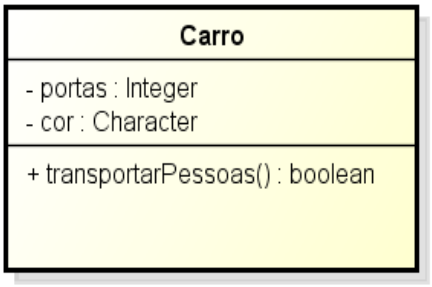
2.5.2.4 Visibilidade

A visibilidade é utilizada para indicar qual é o nível de acessibilidade de um atributo ou método, sendo apresentada a 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

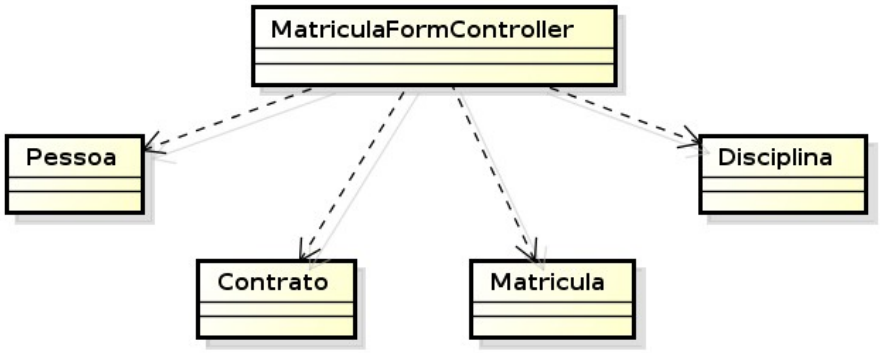
Na modelagem orientada a objetos, existem vários tipos de relacionamento entre classes como: dependência, associação, agregação, composição e herança. Nesta seção, serão apresentados os principais conceitos relativos à relacionamentos entre classes.

2.5.3.1 Dependência

A dependência é um tipo de relacionamento utilizado para identificar quando 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 que pode ser citado, ocorre quando a partir de uma classe de origem é realizada 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 são:

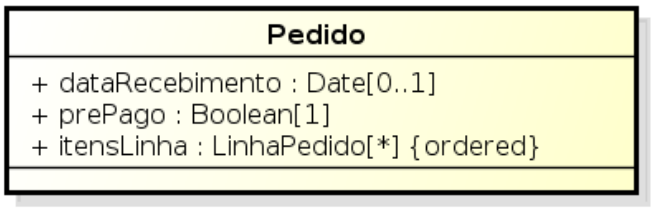
a) (1) - Exatamente uma instância: Um pedido deve ter um cliente ou seja exatamente um objeto do tipo cliente; (FOWLER, 2005)

b) (0..1) - Zero ou uma instância: Um pedido pode ter ou não uma data de recebimento; (FOWLER, 2005)

c) (*) - Zero ou muitas instâncias: Um cliente não precisa fazer um pedido, e não há nenhum limite quanto ao número de pedidos que ele ter. (FOWLER, 2005)

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



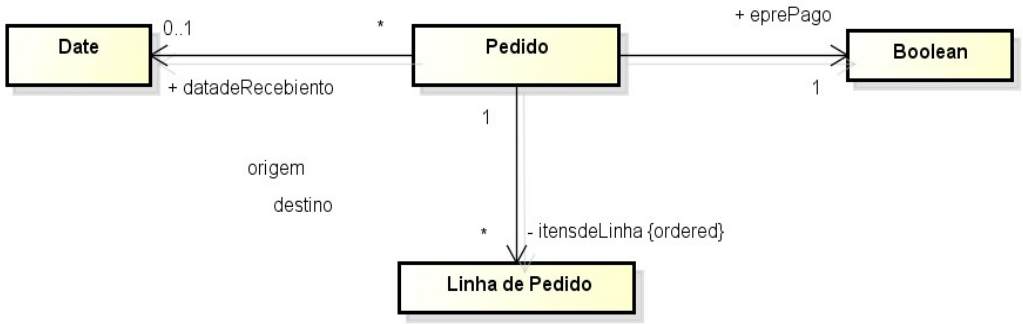
Fonte: Fowler(2005, p. 54) adaptado pelo autor.

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

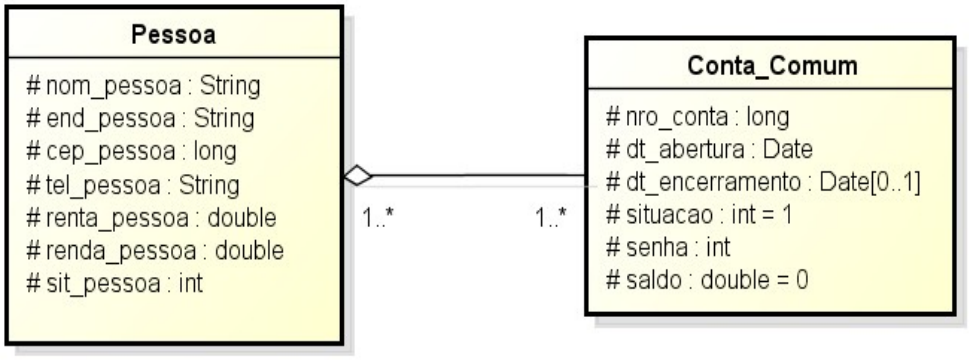


Fonte: Fowler(2005, p. 55)

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, os quais precisam ter suas informações completadas pelos objetos da classe Conta_Comum que são os objetos-parte.

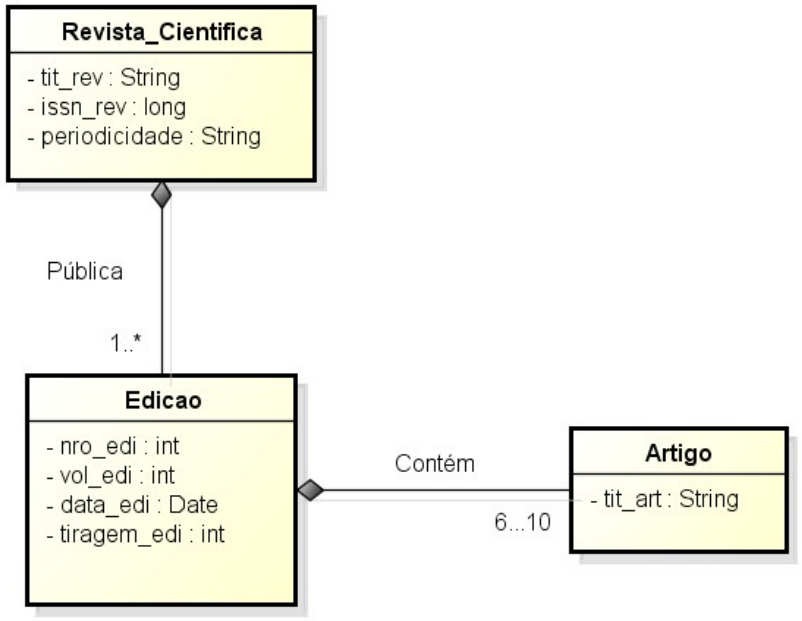
Uma agregação tem como característica possibilitar que os objetos-parte possam ser compartilhados por mais de um objeto-todo ou vice versa. Um exemplo disso seria um objeto da classe Pessoa possuir vários objetos da classe Conta_Comum, ou um objeto da classe Conta_Cumum estar relacionado com vários objetos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

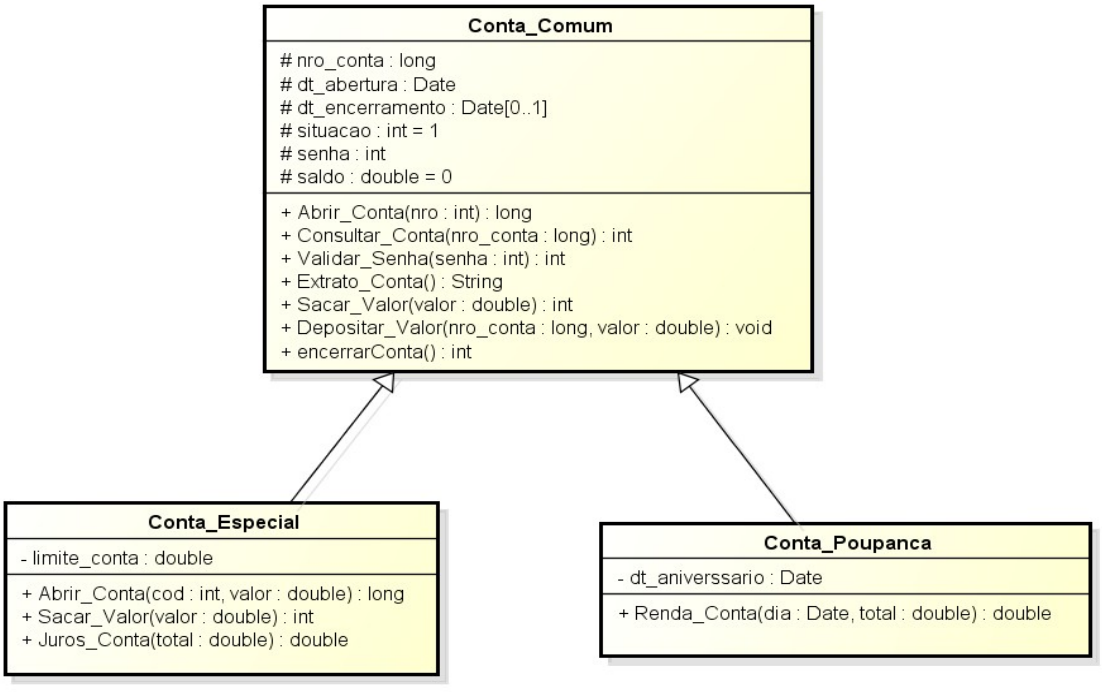
A generalização/especialização ocorre quando há duas ou mais classes com características muito semelhantes, evitando assim a redeclaração de atributos e/ou métodos idênticos. A utilização desta técnica permite o reaproveitamento de código, pois cria-se uma classe geral em que são declarados os atributos e métodos comuns de todas as classes envolvidas, e então são declaradas as subclasses ligadas a 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 continua contendo um triangulo não preenchido que aponta para a superclasse.

Figura 20 - Exemplo de Generalizacao/Especializacao 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 Gamta 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 sua referência, afim 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 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 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), que organizou estes em famílias de acordo com as características de cada padrão. A figura a seguir apresenta os 23 padrões de projetos "clássicos" documentados e catalogados por Gamma et al. (2008).

Figura 21 - Padrões de projeto clássicos

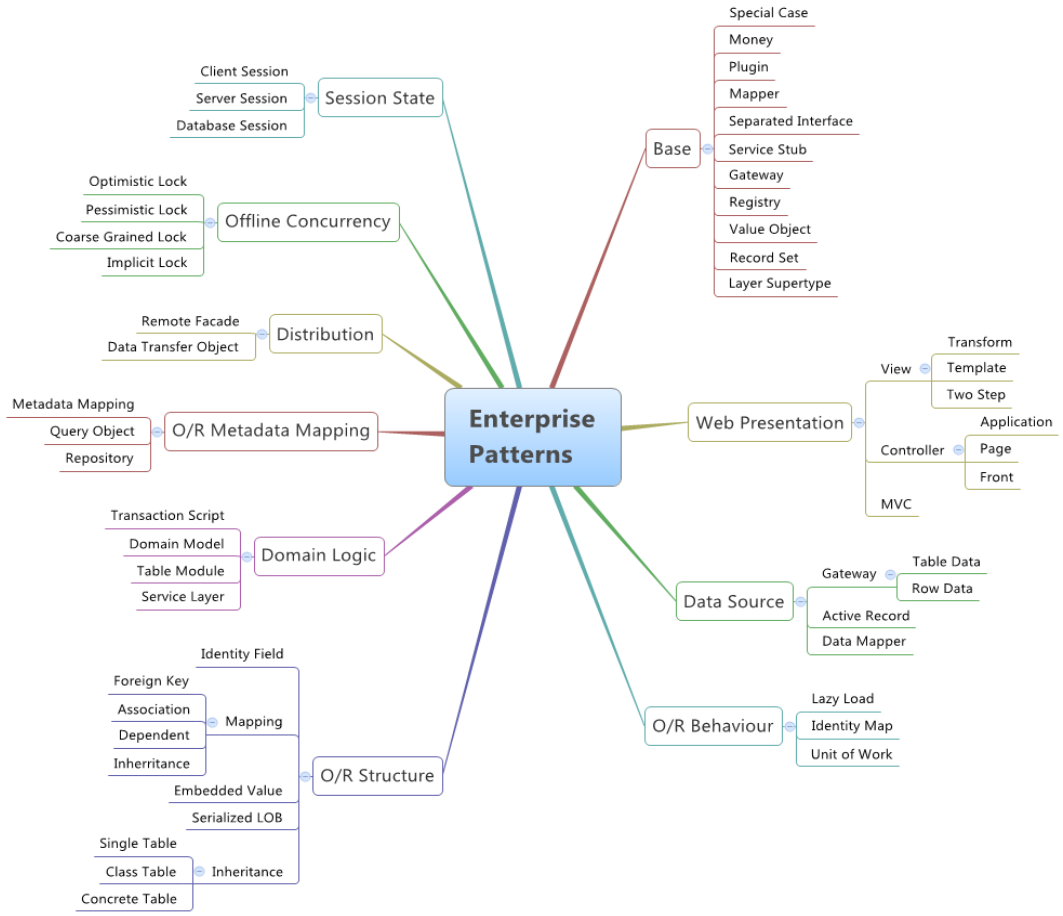
		Propósito		
		De criação	**Estrutural**	**Comportamental**
Escopo	**Classe**	Factory Method	Adapter (class)	Interpreter Template Method
	Objeto	Abstract Method Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsability Command Iterator Mediator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



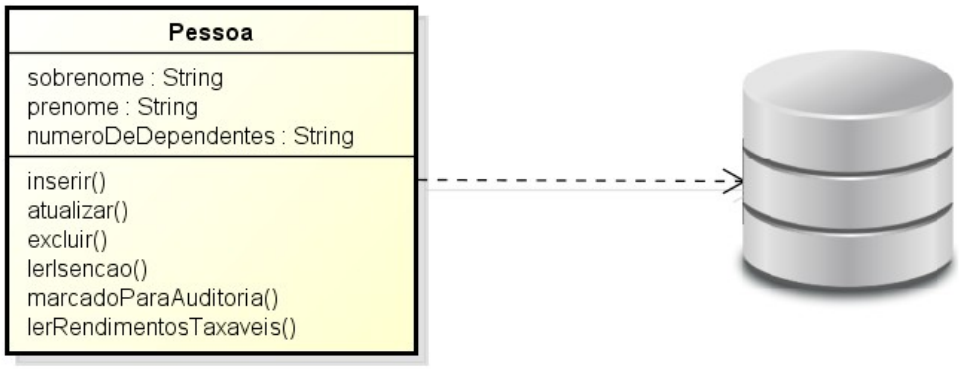
Fonte: Site da internet(http://logogin.blogspot.com.br/2013/03/fowlers-patterns-of-enterprise-mind-map.html)

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 adiciona 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 irá carregar 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. Dessa forma os objetos da classe Pessoa sabem como ler e gravar seus dados no banco de dados.

A essência do *Active Record* está na semelhança que as classes tem 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 a 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.

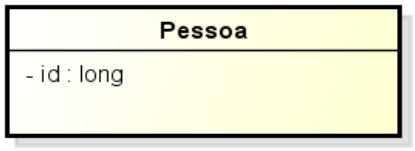
Além dos métodos utilizadas para realizar a leitura a gravação de atributos, os *Active Records* podem realizar outras operações de mais alto nível, como converter os tipos de dados orientados a SQL para um tipo mais apropriado para o usuário. Ao solicitar ao método de leitura uma tabela relacionada,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

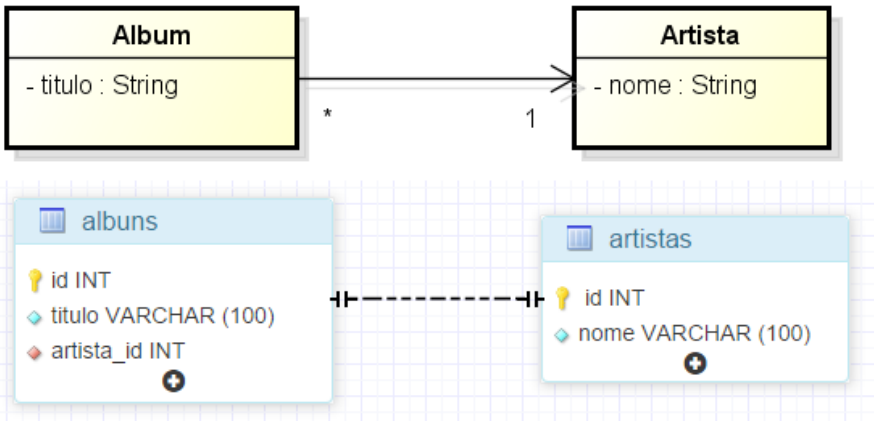
Para diferenciar os registros de uma linha de outra nos bancos de dados relacionais, é utilizado uma chave, geralmente a chave primária. Entretanto, nas linguagens de programação que possuem suporte a objetos, estes não precisam de tal chave, pois elas asseguram a identidade correta do objeto. Não há 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)

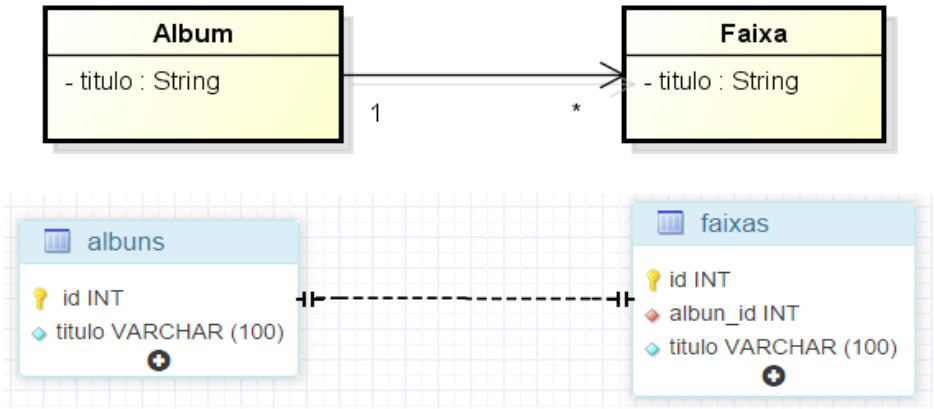
Figura 25 - Exemplo da utilização do padrão *Foreign Key Mapping*



Fonte: Fowler(2003, p. 236)

Em sistemas orientados a objetos, seja ele simples ou complexo, 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 instancias 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)

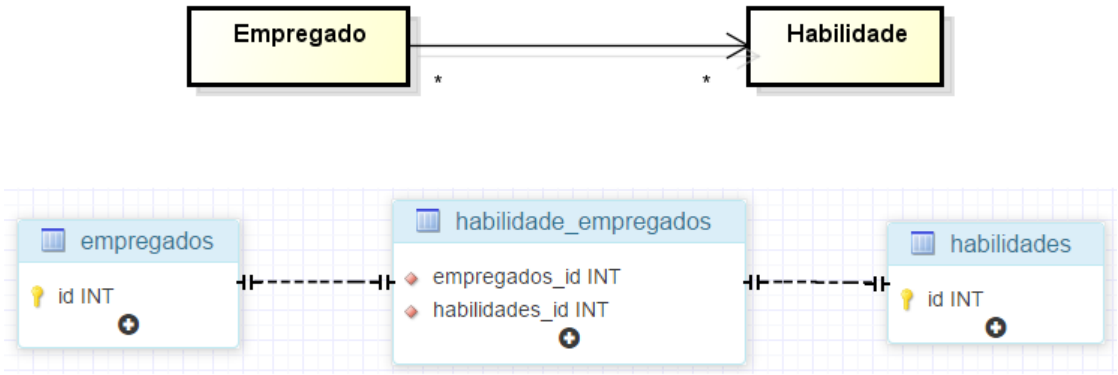
Segundo Fowler esses problemas podem ser resolvidos utilizando o padrão *Identity Field*, onde cada objeto irá armazenar a chave primária da respectiva tabela do banco de dados. Ao haver dois objetos conectados por meio de uma associação, esta deve ser substituída por uma chave estrangeira no banco de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

Figura 27 - Exemplo da associação



Fonte: Fowler(2003, p. 238)

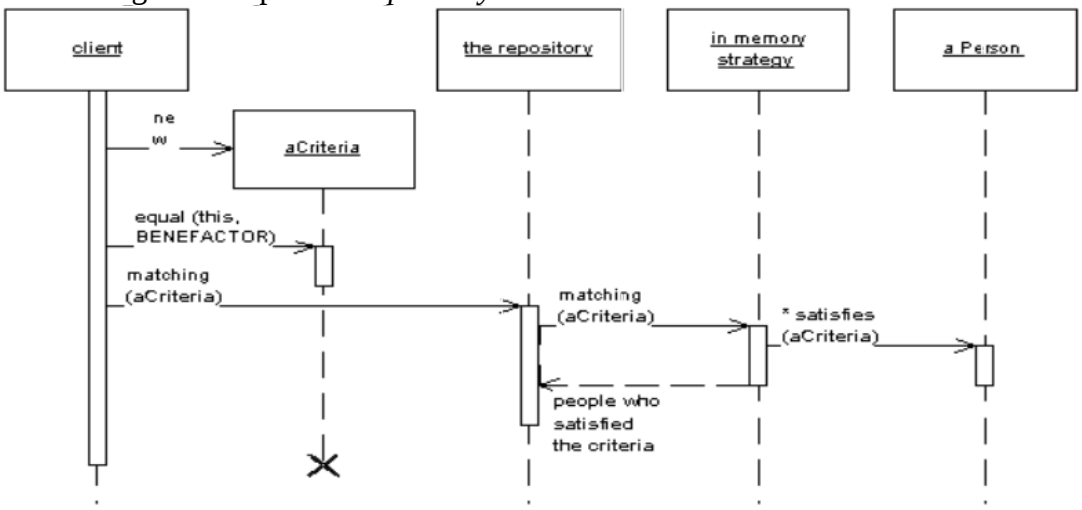
Em linhas gerais, bancos de dados relacionais não conseguem armazenar campos multivalorados para guardar a referência de outras tabelas. Já os objetos conseguem manipular com grande facilidade campos multivalorados, utilizando coleções como valores de campos. Quando se tem uma associação de um-para-muitos, 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. Nessa 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 logica de acesso a dados de qualquer outra lógica do software, criando assim, uma camada responsável pela comunicação entre a camada de mapeamento de dados com 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, onde 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 (1999, 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) para que, a partir da exploração destes temas, possa se desenvolver uma ferramenta em conformidade com os conceitos pré-existentes. Desta forma, este trabalho se caracteriza conforme Santos (1999), segundo 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 quantidade de usuários, de forma a gerar dados estatísticos, mas sim efetuar um estudo aprofundado sobre os temas que dão embasamento ao trabalho, bem como estabelecer sua relação com a prática, propondo uma solução, e avaliando a mesma de maneira qualitativa. Portanto, este estudo se trata 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 em conformidade com os conceitos já estabelecidos em relação às teorias relacional 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 num ambiente controlado, tendo como base projetos fictícios de software que serão utilizados como cenários de teste. Sendo assim, sua utilização se dará em ambiente de laboratório. Para Santos (1999) 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 o término do desenvolvimento da plataforma, estabelecida em laboratório, ela será submetida a especialistas da área de desenvolvimento de software para ser avaliada. A coleta dos dados dessa avaliação será feita através de um questionário qualitativo onde os especialistas irão avaliar questões como funcionalidades, facilidades de uso e os objetivos propostos pela plataforma. Segundo Wainer (2007):

> Questionários são uma forma rápida e simples para avaliar as opiniões, objetivos, anseios, preferências, crenças, etc. de pessoas. Mas por ser uma forma simples, se mal concebida, pode levar a um viés considerável. (WAINER, 2007, p. 15)

Através das avaliações feitas pelos especialistas da área de Engenharia de Software, busca-se validar a plataforma que será desenvolvida por este trabalho. Os resultados obtidos serão analisados e as devidas conclusões serão apresentadas.

3.2 Cronograma

Tabela 1 - Cronograma para desenvolver e concluir a proposta. Ano base, 2015

Tarefa / Mês	Jan		Fev		Mar		Abr		Mai		Jun	
Implementar a diagramação UML/ER												
Implementar a diagramação colaborativa												
Implementar o mecanismo de *plugins*												
Implementar *plugins* de exemplo												
Testes de todas as funcionalidades												
Simulação com casos reais												
Realizar avaliação da ferramenta												
Ajustes no documento final												

Fonte: Elaborado pelo autor.

Segue abaixo uma breve explicação de cada tarefa:

a) Implementar a diagramação UML/ER: Nesta etapa, serão desenvolvidas as funcionalidades que possibilitarão aos usuários da plataforma a criação dos diagramas UML/ER;

b) Implementar a edição colaborativa: Nesta etapa será desenvolvida a funcionalidade de edição colaborativa entre os usuários da plataforma que compartilharam diagramas entre si;

c) Implementar o mecanismo de *plugins*: Nesta etapa será desenvolvido o mecanismo que permitirá acoplar plugins de importação e exportação;

d) Implementar *plugins* de exemplo: Após a implementação do mecanismo de *plugins* serão criados exemplos de *plugin* de exportação (aplicação para o *framework* Adianti) e importação (banco de dados);

e) Testes de todas as funcionalidades: Após o desenvolvimento da plataforma, serão realizados vários testes, que poderão encontrar *bugs*, os quais serão resolvidos;

f) Simulações com casos reais: Nesta etapa a plataforma será utilizada para modelar um novo software, tendo em vista realizar um teste da mesma com cenários reais;

g) Realizar avaliação da ferramenta: Após a conclusão do desenvolvimento da ferramenta, ela será submetida a uma avaliação por profissionais da área de Engenharia de Software;

h) Ajustes no documento final: O documento final pode sofrer ajustes, a partir de constatações indicadas nas etapas do desenvolvimento da proposta.

4 PROPOSTA

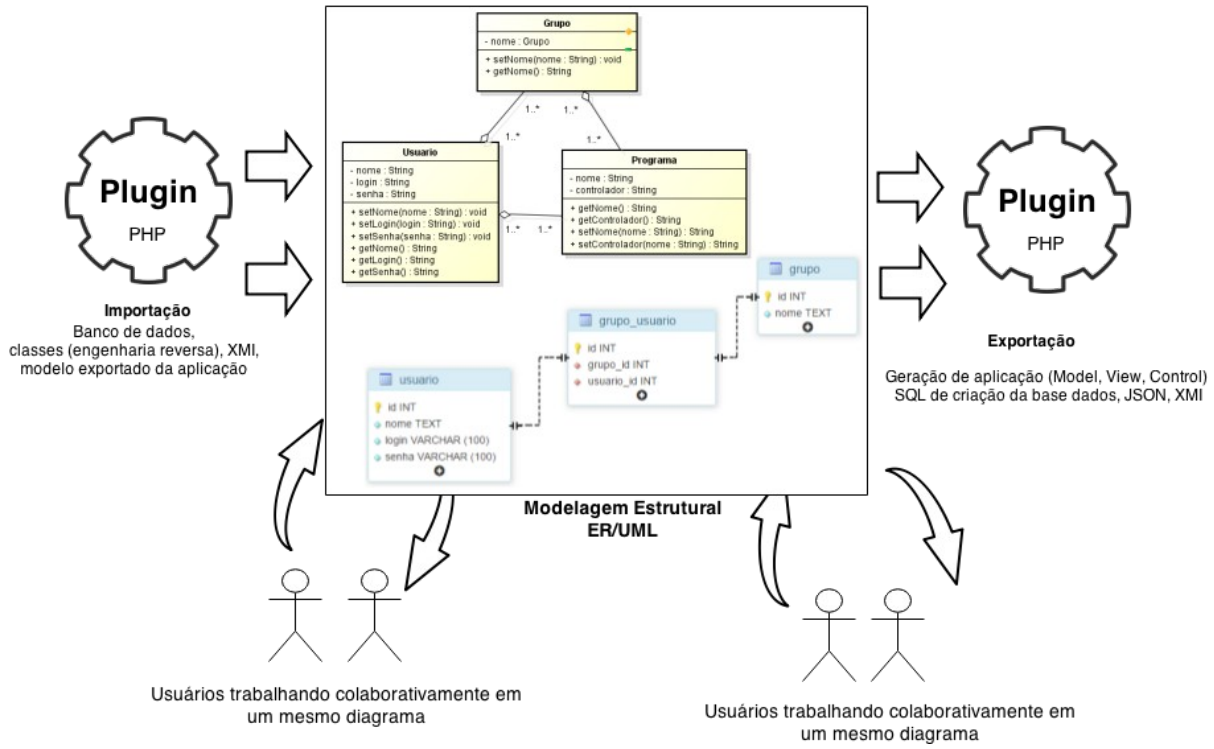
A proposta tem como objetivo a criação de uma plataforma web, online, colaborativa e extensível para a criação e modelagem estrutural de software.

4.1 Visão geral

A plataforma irá permitir a modelagem de aplicações orientadas a objetos (UML) e modelagem de estruturas de bancos de dados relacionais (ER), de forma online e colaborativa. Deverá permitir a interoperabilidade com diferentes modelos de estrutura de software, para isso, será disponibilizada uma especificação que ajudará os usuários da plataforma a criarem *plugins*, que permitirão a importação e exportação de diversos tipos de modelos. Os *plugins* de importação poderão importar dados a partir de classes, conexão com banco de dados, fazendo uma espécie de engenharia reversa, para um modelo UML ou ER. Os *plugins* responsáveis pela exportação do diagrama, poderão 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 poderão ser criados pela ferramenta. O núcleo de modelagem da ferramenta poderá 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), e outros. Também existirão *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 a serem desenvolvidos para a plataforma proposta:

a) Diagramação UML: A plataforma permitirá 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 permitirá 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 permitirá a seus usuários o compartilhamento de diagramas entre si, bem como a edição simultânea;

d) Importação do arquivo JSON: A plataforma permitirá a importação de um arquivo JSON de formato da própria ferramenta;

e) Exportação do modelo para JSON: A plataforma terá um mecanismo de exportação do modelo (ER ou UML) para um arquivo próprio no formato

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

f) *Plugins* de importação: Os *plugins* de importação poderão ser desenvolvidos pelos usuários da plataforma, utilizando a linguagem de programação PHP. Estes *plugins* terão como objetivo gerar um objeto no formato JSON padrão da plataforma, o qual terá em sua estrutura um modelo UML ou ER. Através destes *plugins* os usuários serão 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 dele;

g) *Plugins* de exportação: Os *plugins* de exportação poderão ser desenvolvidos pelos usuários da plataforma, utilizando a linguagem de programação PHP. Estes receberão um JSON como parâmetro, o qual terá em sua estrutura todo o diagrama. Através deles, os *plugins* serão 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: SQL de criação de base de dados no formato MySql e geração de aplicação (*model, view, controller*) para o *framework* PHP Adianti[1]. Os *plugins* serão submetidos a análise para posterior implementação na plataforma.

Para o desenvolvimento da seguinte proposta, será necessário utilizar diversas tecnologias existentes, que serão detalhadas no tópico 4.2. A seguir, serão listados os principais Requisitos não funcionais a serem atendidos para a plataforma proposta:

a) Compatibilidade com browsers Mozilla Firefox e Google Chrome;

b) Utilização do servidor de páginas Apache HTTP server;

c) Utilização de 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.

4.2 Tecnologias que serão utilizadas

Para apoiar a construção da plataforma, serão utilizadas várias ferramentas, as quais tem proposito de auxiliar no desenvolvimento do *front-end* e *back-end* da plataforma.

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

a) PHP: É uma linguagem de código aberto interpretada com propósito para o desenvolvimento web, é rápido, flexível e pragmática. O PHP[2] (PHP *Hypertext Preprocessor*) será utilizado para processar as requisições javascript feitas pelos usuários e 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] será utilizado para processar as requisições de arquivos PHP feitas pelos usuários, fazendo com que eles consigam ter acesso a plataforma online. (Apache, 2014)

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

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

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

a) HTML: É a linguagem de marcação da internet, utilizada para desenvolver as páginas na web, a qual é interpretada pelos navegadores. O HTML[4] (*HyperText Markup Language*) será 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[5] (*Cascading Style Sheets*) será 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, hoje já pode ser encontrada do lado do servidor. O JavaScript[6] será utilizado para tornar a interface HTML interativa e funcional, onde 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 aplicação responsivas para a web. O Bootstrap[7], será utilizado para desenvolver com mais facilidade a interface da plataforma; (BOOTSTRAP, 2014)

e) JQuery: É uma biblioteca de JavaScript que é rápida, pequena e muito rica em recursos, os quias facilitam o desenvolvimento e manipulação do HTML. O JQuery[8] será 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[9] será utilizado para facilitar o desenvolvimento das conexões entre as entidades do diagrama, as quais são elementos feitos com HTML. (JSPLUMB, 2014)

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

4.3 Protótipo

Até o presente momento no desenvolvimento desta proposta além ter definido a metodologia e realizado a pesquisa do referencial teórico, foi possível desenvolver um protótipo da plataforma com as seguintes funcionalidades:

a) Diagramação ER;

b) Criação de tabelas;

c) Criação de atributos;

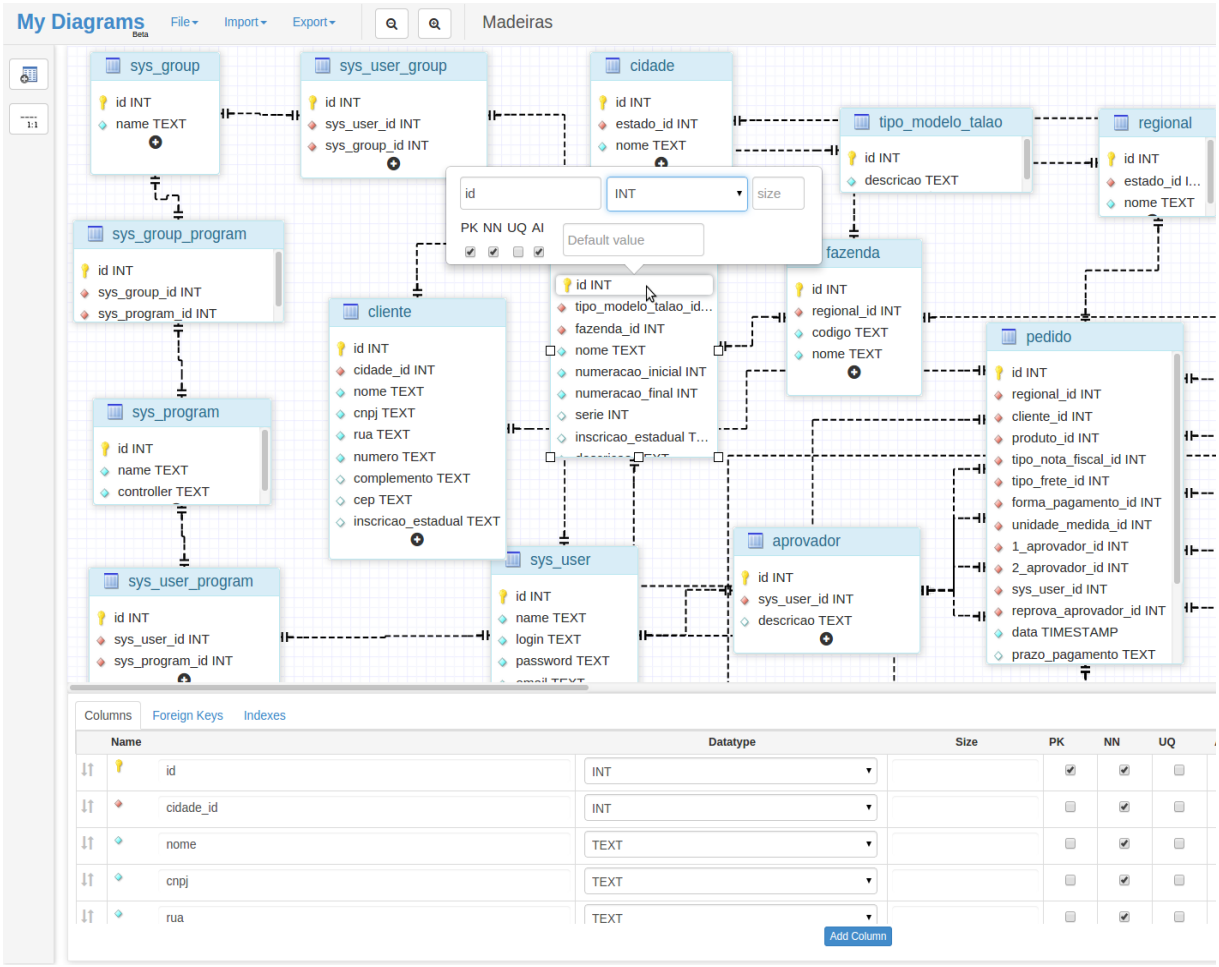
d) Criação de chaves estrangeiras;

e) Criação de índices;

f) Relacionamento entre as tabelas.

A Figura 30 apresenta as funcionalidades descritas á cima, onde é demonstrado um diagrama ER de um banco de dados modelado na plataforma.

Figura 30 - Plataforma proposta



Fonte: Feito pelo autor.

5 CONSIDERAÇÕES PARCIAIS

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

Até o presente momento todos os objetivos propostos na metodologia deste trabalho foram contemplados, bem como os estudos bibliográficos, a teorização dos conceitos abordados, bem como a criação de um protótipo para prova de conceito, que permitirá a continuidade do desenvolvimento da plataforma, visando atingir o restante dos requisitos funcionais e não funcionais.

As próximas etapas de desenvolvimento do presente trabalho envolverão a implementação da plataforma de modelagem de software estrutural visando atingir os requisitos restantes como modelagem UML, edição colaborativa, interoperabilidade por meio de *plugins*, testes de suas funcionalidades, avaliação qualitativa da plataforma e ajustes na redação do documento final.

REFERÊNCIAS

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

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.

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.

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

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.