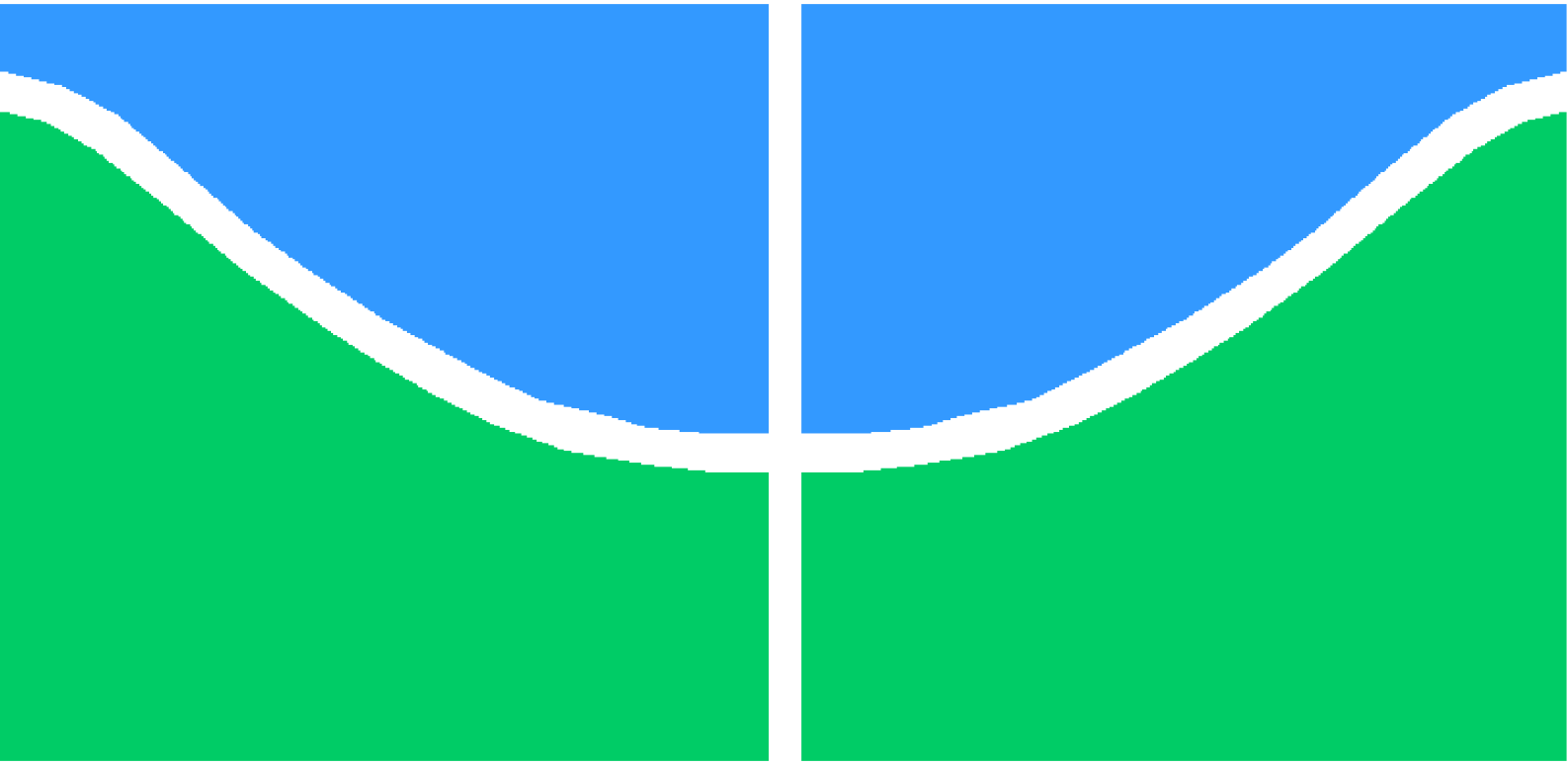


Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Engenharia de Software

Framework para Redes Sociais baseadas em Compartilhamento de Rotas e Agendas

Autores: Álex Silva Mesquita, Jefferson Nunes de Sousa Xavier
Orientador: Prof. Dr. Maurício Serrano
Coorientador: Profª. Drª. Milene Serrano

Brasília, DF
2016

Álex Silva Mesquita, Jefferson Nunes de Sousa Xavier

Framework para Redes Sociais baseadas em Compartilhamento de Rotas e Agendas

Monografia submetida ao curso de graduação em Engenharia de Softwareda Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dr. Maurício Serrano

Coorientador: Prof[a]. Dr[a]. Milene Serrano

Brasília, DF

2016

Álex Silva Mesquita, Jefferson Nunes de Sousa Xavier
Framework para Redes Sociais baseadas em Compartilhamento de Rotas e Agendas/ Álex Silva Mesquita, Jefferson Nunes de Sousa Xavier. – Brasília, DF, 2016-
141 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Maurício Serrano
Coorientador: Profª. Drª. Milene Serrano

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2016.

1. Framework. 2. Reutilização de software. I. Prof. Dr. Maurício Serrano. II. Profª. Drª. Milene Serrano. III. Universidade de Brasília. IV. Faculdade UnB Gama. V. Framework para Redes Sociais baseadas em Compartilhamento de Rotas e Agendas

CDU 02:141:005.6

Álex Silva Mesquita, Jefferson Nunes de Sousa Xavier

Framework para Redes Sociais baseadas em Compartilhamento de Rotas e Agendas

Monografia submetida ao curso de graduação em Engenharia de Softwareda Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, 06 de Julho de 2016:

Prof. Dr. Maurício Serrano
Orientador

Profª. Drª. Milene Serrano
Coorientador

Prof. Dr. Edson Alves da Costa Júnior
Convidado 1

Prof. Dr. Fabricio Ataides Braz
Convidado 2

Brasília, DF
2016

Agradecimentos

À Deus, que nos deu saúde e forças para prosseguirmos, vencendo todas as dificuldades.

À família, pelo amor, incentivo e apoio que sempre nos ajudaram a continuar com este trabalho.

Aos orientadores, Prof. Dr. Maurício Serrano e Profª. Drª. Milene Serrano pelo excelente suporte, e pelas revisões que nos ajudaram a chegar até aqui.

E a cada um que possa ter nos ajudado direta ou indirentamente, mesmo sem saber, e fizeram parte de nossa história nessa formação.

"Não reze por uma vida fácil,
mas sim para ter forças
para uma vida difícil.
(Bruce Lee, informação verbal, tradução nossa)

Resumo

Redes sociais vêm ganhando um grande destaque em meio à sociedade nos tempos atuais. A cada dia, novas redes são desenvolvidas, merecendo a atenção de diversos usuários. Nesse contexto, torna-se necessário o desenvolvimento de ferramentas que auxiliem desenvolvedores a criar novas redes de uma maneira mais rápida e prática, sem se preocupar com problemas recorrentes nesse tipo de sistema. No intuito de colaborar com esse contexto tecnológico, este trabalho propõe-se a desenvolver um *framework* que possa fornecer recursos de rotas e agendas, bem como recursos comuns no desenvolvimento de redes sociais como, por exemplo, o relacionamento de usuários. Dessa forma, o presente trabalho cria a lógica para atender o gerenciamento de rotas e agendas e de relacionamento entre usuários em uma rede social virtual. Essa lógica é extendida e utilizada em aplicações que instanciam o suporte desenvolvido, no caso, um *framework*. Dessa forma, um desenvolvedor que instancie o *framework* proposto, terá disponível esses recursos, permitindo aplicá-los conforme as suas necessidades em uma rede social, sem, necessariamente, se preocupar com a lógica implementada. Tal suporte, o qual transcende recursos mais básicos de redes sociais, procura atender perfis de usuários, interessados em redes sociais mais específicas. Esse cenário tem se tornado algo desejado nos tempos atuais.

Palavras-chaves: *Framework*. Reutilização de software. Redes sociais. Grafos. Rotas. Agendas.

Abstract

Social networks come gaining a big highlight between the society in present times. Each day, new networks are developed, deserving the attention of several users. On that context, it becomes necessary developing tools that assists developers create new networks in a way faster and practical, without not worry with recurring problems in this type of system. In order to collaborate with this technological context, this work proposes developing a framework to provide resources of routes and schedules, as well common resources in social networks development, for example, the users relationship. In this way, this present work creates the logic to control the management of routs and schedules and the relationship between users in a virtual social network. That logic is extended and used in applications that instanciation the developed support, in case, a framework. Beyond this resource, the framework implements some extra resources. This resources will be for domains schedulers and routes. In this way, a developer who uses the presented framework, will be available this resources, allowing apply them as needed in a social network, without, necessarily, not worry with the implemented logic. Such support, which transcends most basic social network resources, seeks to meet users profiles, interested in most specific soccial networks. This scenario has become something desired in present times.

Key-words: Framework. Software reuse. Social Network. Graphs. Routes. Schedules.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

BFS	*Breadth-First Search*
DFS	*Depth-First Search*
HTTP	*Hypertext Transfer Protocol*
JSON	*JavaScript Object Notation*
REST	*Representational State Transfer*
SOAP	*Simple Object Access Protocol*
TCC	Trabalho de Conclusão de Curso
WSDL	*Web Services Description Language*
XHTML	*eXtensible Hypertext Markup Language*
XML	*eXtensible Markup Language*
XP	*Extreme Programming*

Lista de símbolos

$\in$	Pertence
$\subseteq$	Subconjunto
$\cap$	Intersecção
$\cup$	União
$\oslash$	Conjunto nulo
ψ	Letra grega minúscula psi
Δ	Letra grega maiúscula delta
δ	Letra grega minúscula delta
Θ	Letra grega maiúscula teta
θ	Letra grega minúscula teta
$\sum$	Soma

Códigos

Sumário

1 Introdução

Este capítulo apresenta a contextualização deste trabalho, o tema estudado, o problema resolvido, os objetivos alcançados e como o trabalho está organizado.

Desde a sua concepção, *sites* de redes sociais têm atraído milhões de usuários, muitos dos quais têm integrado esses *sites* em suas rotinas diárias. Existem centenas de redes sociais com diversas capacidades tecnológicas suportando uma ampla gama de interesses e práticas. Os principais recursos tecnológicos envolvidos são bem consistentes e semelhantes, porém, as culturas que estão impregnadas podem variar bastante. Alguns *sites* tendem a atender diversos tipos de pessoas com interesses variados, outros se voltam a grupos mais específicos, que se comunicam em uma linguagem comum, possuindo outros interesses (BOYD; ELLISON, 2007).

O aumento do número de usuários de redes sociais dá-se, principalmente, pela facilidade de acesso das pessoas a aparelhos que permitem se conectar à internet a qualquer hora e em qualquer lugar, como é o caso dos *smartphones*. Como pode ser visualizado na Figura 1, o Brasil, até o ano de 2017, tem uma projeção de que cerca de 110 milhões de pessoas utilizem as redes sociais, o que representa mais de 89% dos usuários com acesso à internet (EMARKETER, 2013).

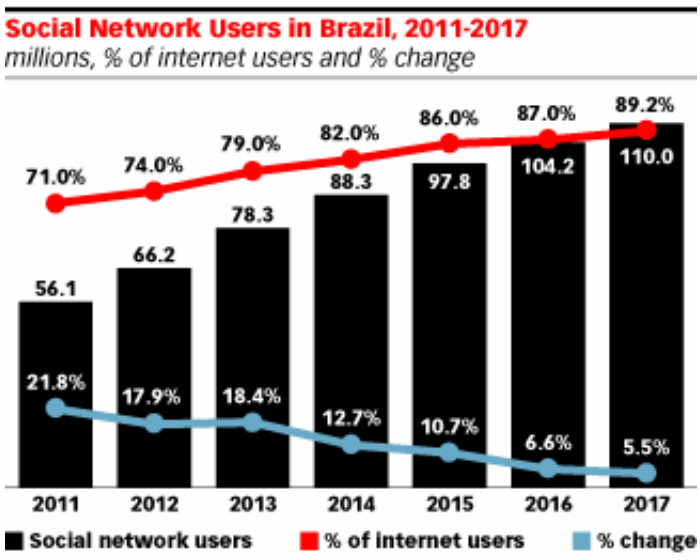


Figura 1 – Uso de redes sociais no Brasil (EMARKETER, 2013)

Visando colaborar com o contexto de redes sociais, busca-se apresentar uma ferramenta que auxilie desenvolvedores a criar sistemas desse tipo, oferecendo recursos que são comuns à maioria das aplicações sociais. Mais precisamente, esse trabalho consistiu em desenvolver um *framework* que oferece esses recursos para o desenvolvedor. Nas próximas seções, essa ideia será contextualizada e justificada.

1.1 Contextualização

Uma rede social virtual é uma comunidade *online* que representa um conjunto de participantes (pessoas, organizações ou outras entidades), unindo ideias e recursos em torno de relacionamentos, valores e interesses compartilhados (MARTELETO, 2014).

Diversos sistemas conhecidos parecem estruturados como uma rede. Para a Biologia, há o interesse em saber quem se alimenta de quem, quando se estuda a cadeia alimentar. O cérebro faz ligações entre neurônios; as sinapses, para que a pessoa lembre ou resolva algum problema ou questão. A Internet é uma rede na qual as pessoas se conectam e se comunicam. As doenças, por exemplo, podem se propagar de uma pessoa para outras, deflagrando uma epidemia (GOULAR, 2001).

Assim, a análise da relação entre os nós e da estrutura formada pela rede fornece informações a respeito de diversos fenômenos e situações: como o cérebro funciona, como a doença se propaga, como as pessoas se comunicam e trocam informações, isto é, as relações ou interações influenciam a própria rede (GOULAR, 2001).

A relação entre os nós da rede tem várias denominações apresentadas em trabalhos científicos: vínculo, ligação, arco, interação, conexão e/ou relação. Os nós da rede, também chamados de atores, estão ligados por essas relações. Por exemplo, os atores podem se classificar como amigos, quem troca informação com quem, quem confia em quem. A relação pode indicar que os atores fazem parte de um clube, de uma associação, trabalham no mesmo departamento ou que mantêm transações comerciais, trocam mensagens, trabalham em equipe ou cooperam entre si para algum tipo de trabalho. Os atores podem ser pessoas ou empresas que estão relacionadas por alguma atividade, além de grupos, localidades, cidades, regiões, entre outros (HANNEMAN; RIDDLE, 2005).

John Scott, em seu livro (SCOTT; CARRINGTON, 2011), diz que redes sociais são formadas por dois tipos principais de dados: dados de atributos e dados de relacionamentos. Diz-se que os dados de atributos são as opiniões e os comportamentos dos agentes demonstrados dentro da rede. Portanto, são qualidades e características que pertencem a eles como indivíduos. Esses dados podem ser quantificados e analisados. Os dados de relacionamentos dizem respeito aos contatos, laços e conexões. Esses dados não são dados de um agente, e sim, de um conjunto de agentes conectados que formam um sistema de relacionamentos. É possível também quantificar e analisar esses tipos de dados, podendo encontrar padrões de relacionamentos em grupos.

Para abstrair o contexto do mundo real para o mundo computacional, deve-se utilizar um software para prover mecanismos de análise para melhor planejar e manter a rede de uma organização. Tendo como foco o produto de software e baseando-se nas plataformas tecnológicas, deve-se levar em conta a Reutilização de Software. Essa necessidade é evidenciada, pois uma empresa geralmente deixa de construir um produto de software

isolado, e busca parcerias para construir suas soluções (LIMA, 2015).

A reutilização de software é o processo de criar sistemas de software a partir de um software já existente, ao invés de criar a partir do zero (KRUEGER, 1992). A meta da reutilização de software é reciclar o *design*, código e outros componentes de um software e, assim, reduzir o custo, o tempo e melhorar a qualidade do produto (KESWANI; JOSHI; JATAIN, 2014).

Mesmo com todos os benefícios mencionados anteriormente, ao lidar com reutilização de software, deve-se planejar e conhecer bem quais são os objetivos esperados a partir dessa prática, como afirmam os autores Keswani, Joshi e Jatain, em sua publicação (KESWANI; JOSHI; JATAIN, 2014), sobre reutilização de software: "*Para que um programa de reutilização de software confira o retorno apropriado, o mesmo deve ser sistematizado e planejado. Uma organização que implementa reutilização deve identificar os melhores métodos e estratégias para alcançar máxima produtividade*".

1.2 Questão de Pesquisa

Esse trabalho tem como intuito responder com a seguinte questão: É possível oferecer um *framework* que auxilie no desenvolvimento de redes sociais, disponibilizando recursos gerais de relacionamentos e específicos de definições de rotas e agenda, proporcionando ao desenvolvedor facilidade ao lidar com preocupações intrínsecas desse contexto?

Foi desenvolvido o SocialFramework para responder essa questão, além da rede social SocialBike para um uso comprovado dos recursos fornecidos pelo *framework*. A documentação completa desses, assim como, os resultados encontrados no desenvolvimento pode ser encontrada nos capítulos 6 e 7 deste trabalho.

1.3 Justificativa

As redes sociais ultrapassaram o âmbito acadêmico/científico, conquistando espaço em outras esferas. É possível observar esse movimento na Internet, onde encontramos adeptos com objetivos específicos (TOMAE; ALCARA; CHIARA, 2005).

Como pode ser observado na Figura 2, o crescimento de número de usuários em mídias sociais cresceu 12% de janeiro de 2014 para janeiro de 2015 (SOCIAL, 2015).

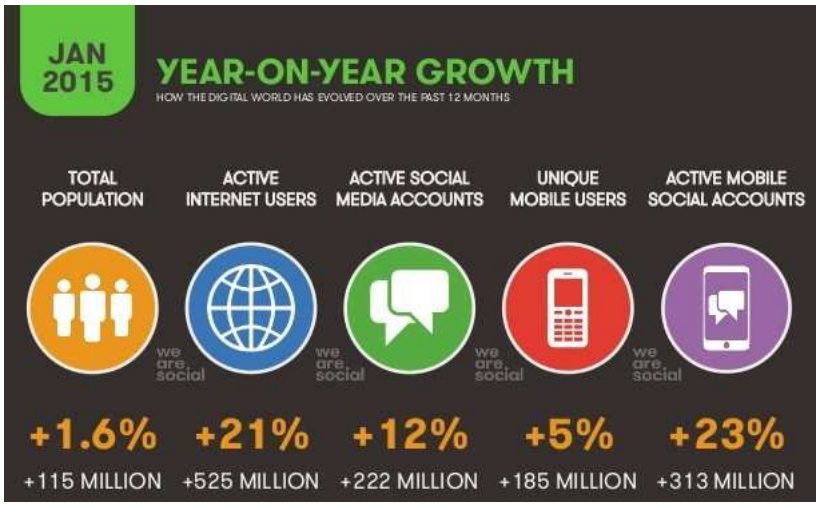


Figura 2 – Crescimento de usuários em mídias sociais (SOCIAL, 2015)

De acordo com o relatório da agência de *marketing* social We Are Social (SOCIAL, 2015), em janeiro de 2015, 47% dos brasileiros utilizam ao menos uma mídia social, como pode ser visualizado na Figura 3. No mundo essa valor fica em torno de 29%.

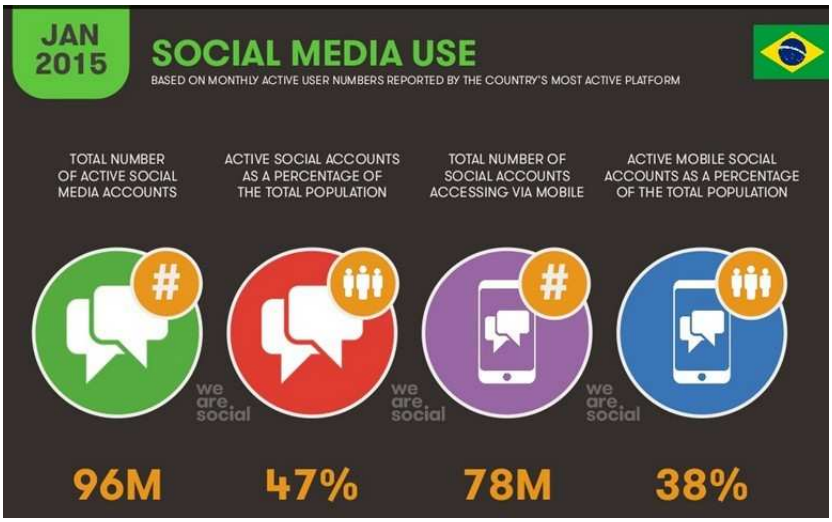


Figura 3 – Quantidade de usuários de mídias sociais no Brasil (SOCIAL, 2015)

Com tal crescimento e repercussão, torna-se pertinente a criação de um suporte que auxilie a produção de novas redes sociais digitais. Diante dessa questão, este trabalho focou suas atenções na criação de um *framework* em atendimento a esse tópico, trabalhando com a ideia de redes sociais mais específicas, as quais fazem uso de definições de rotas ou controle de relacionamento de agendas.

1.4 Objetivos

Esse trabalho teve como objetivo atingir objetivos geral e específicos, conforme colocado nos subtópicos a seguir apresentados.

1.4.1 Objetivo Geral

Oferecer um *framework* para ser utilizado no desenvolvimento de redes sociais, o qual disponibiliza recursos gerais de relacionamentos e específicos de definição de rotas e agenda, procurando auxiliar o desenvolvedor de software ao lidar com preocupações intrínsecas desse contexto.

1.4.2 Objetivos Específicos

A partir do objetivo geral, pôde-se definir os objetivos específicos, os quais são:

1. Definir uma arquitetura com base no uso de grafos, alinhada às boas práticas da Engenharia de Software, para representar os relacionamento entre as pessoas bem como as rotas percorridas por elas;

2. Evoluir a arquitetura proposta, visando projetar algoritmos para compatibilizar agendas entre os interessados;

3. Usar estruturas de dados e algoritmos específicos, visando desempenho e facilidades na manutenção evolutiva do software;

4. Instanciar um produto de software - i.e. uma rede social específica - a partir do *framework*, no intuito de coletar as primeiras impressões acerca do suporte desenvolvido como tema foco desse trabalho.

1.5 Organização do documento

Este documento foi dividido em seis capítulos, além da introdução. Esses estão organizados da seguinte forma:

- Referencial Teórico: Levantamento bibliográfico de tópicos relevantes para alcançar o embasamento teórico necessário para a realização deste trabalho;

- Suporte Tecnológico: Apresenta todas as principais ferramentas utilizadas para desenvolvimento deste trabalho;

- Metodologia: Apresentação de como foi conduzido o trabalho, abordando modelos de desenvolvimento e modelos de pesquisa utilizados;

- Proposta: Apresentação completa quanto à proposta deste trabalho;

- SocialFramework: Apresentação do *framework* desenvolvido durante este trabalho;

- Resultados: Apresentação dos resultados obtidos durante todo o desenvolvimento do trabalho e da rede social SocialBike;
- Conclusão: Apresentação do que se pôde concluir com o trabalho desenvolvido, assim como, trabalhos futuros esperados.

2 Referencial Teórico

Neste capítulo, serão abordados os tópicos teóricos associados às redes sociais, aos grafos e à reutilização de software. Esses tópicos conferiram o embasamento teórico para realização deste trabalho.

2.1 Redes Sociais

Quando se pensa em rede, surge a ideia de um conjunto de nós interligados entre si, como uma teia que ocupa um determinado espaço em um ambiente. Os nós ou pontos estão ligados em pares e podem representar várias situações em áreas de interesse em comum (NEWMAN, 2010).

Para Sodré (SODRE, 2002), rede é onde as conexões e as interseções tomam o lugar do que seria antes apenas linearidade. Essas conexões e interações ocorrem pelo contato direto, face a face, e pelo contato indireto, utilizando-se um veículo mediador, como o telefone.

As pessoas estão inseridas na sociedade por meio das relações que desenvolvem durante toda sua vida, primeiro no âmbito familiar, em seguida na escola, na comunidade em que vivem e no trabalho; enfim, as relações que as pessoas desenvolvem e mantêm fortalecem a esfera social. A própria natureza humana nos liga a outras pessoas, estruturando a sociedade em rede (TOMAE; ALCARA; CHIARA, 2005).

Os tipos de relações também podem ser de movimentação entre lugares, como migração, mobilidade física ou social; de conexão física, como uma estrada, um rio ou uma ponte que conecta dois lugares; de relações de autoridade, ou de relação biológica, como descendência, por exemplo (WASSERMAN; FAUST, 1994).

Com base em seu dinamismo, as redes, dentro do ambiente organizacional, funcionam como espaços para o compartilhamento de informação e conhecimento. Espaços que podem ser tanto presenciais quanto virtuais, em que pessoas com os mesmos objetivos trocam experiências, criando bases e gerando informações relevantes para o setor em que atuam (TOMAE; ALCARA; CHIARA, 2005).

> “[...] na era da informação – na qual vivemos – as funções e processos sociais organizam-se cada vez mais em torno de redes. Quer se trate das grandes empresas, do mercado financeiro, dos meios de comunicação ou das novas ONGs globais, constatamos que a organização em rede tornou-se um fenômeno social importante e uma fonte crítica de poder.” (CAPRA, 2002)

O contexto em que estamos inseridos desencadeia uma série de mudanças na rotina dos indivíduos, e uma delas evidencia as redes como ponto de convergência da informação e do conhecimento (TOMAE; ALCARA; CHIARA, 2005).

O conhecimento que a rede possui repercute sobre o meio que esta se encontra, pois Wellman (WELLMAN, 1996) verifica, na rede, sua identidade singular em determinada situação, isto é, a representação e a interpretação das relações em rede estão fortemente ligadas à realidade que a cerca; a rede é influenciada pelo seu contexto e esse por ela. Portanto, para (MARTELETO, 2014), os efeitos das redes podem ser percebidos fora de seu espaço, nas interações com o estado, na sociedade ou em outras instituições representativas.

A interação constante ocasiona mudanças estruturais e, em relação às interações em que a troca é a informação, a mudança estrutural que pode ser percebida é a do conhecimento. Quanto mais informação troca-se com o ambiente, com os atores da rede, maior será a bagagem de conhecimento, e maior será o estoque de informação, e é nesse conjunto de significados que se inserem as redes sociais (TOMAE; ALCARA; CHIARA, 2005).

Milgram, em sua tese (MILGRAM, 1967), defende que qualquer pessoa está distante de qualquer outra pessoa do mundo, a no máximo seis graus de separação. Essa tese ficou conhecida como “mundo pequeno” e “teoria dos seis degraus”. Sua pesquisa demonstra que a rede social constitui importante recurso profissional e pessoal. Estar em contato com pessoas que conheçam uma pessoa-alvo, em razão de um interesse específico, já é um passo além para a conquista de um objetivo.

2.2 Teoria dos Grafos

A análise de redes sociais tem como fundamento a teoria dos grafos. A seguir serão apresentados os principais conceitos sobre grafos.

2.2.1 História

O primeiro livro sobre a teoria dos grafos foi publicado por König (1936). Isso levou ao desenvolvimento de uma forte escola de teóricos em grafos na Hungria que incluíram P. Erdős e T. Gallai. Também na década de trinta, H. Whitney publicou uma série de artigos influentes (BONDY; MURTY, 2007).

De acordo com Ore (ORE, 1963), Leonhard Euler[1] teria sido um dos matemáticos mais importantes quando se refere a teoria dos grafos. Contam que o povo da cidade de

[1] Grande matemático e físico suíço, 1707-1783, fez importantes descobertas em campos variados como Cálculo e Grafos.

Königsberg [2], localizado na Prússia e cortada pelo Rio Pregel, com sete pontes ligando duas ilhas e as margens opostas do rio, como pode ser visto na Figura 4, propôs, ao então famoso matemático Leonard Euler, o seguinte problema:

"Será possível fazer um passeio pela cidade, começando e terminando no mesmo lugar, cruzando cada ponte exatamente uma vez?"(ORE, 1963)

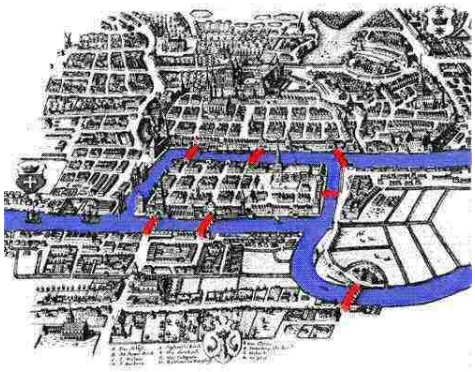


Figura 4 – Pontes de Königsberg[3]

Não se tem conhecimento sobre o Euler ter resolvido esse problema usando a representação de grafos estudada hoje. A modelagem do problema por grafo passa por uma representação, na qual cada porção de terra é representada por um ponto e as pontes estariam representadas por linhas (ORE, 1963), conforme apresentado na Figura 5.

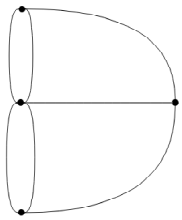


Figura 5 – Problema das sete pontes (ORE, 1963)

Analisando, então, este problema, Euler resolveu a questão provando que uma caminhada assim é possível se, e somente se, o grafo for conectado e todos os seus vértices tiverem grau par, termos estes que serão tratados mais à frente (MALTA, 2008).

Assim, Euler mostrou que, uma vez que o grafo de Königsberg tem vértices de grau ímpar, a resposta ao problema era que tal caminhada era impossível. Desde então, todo grafo conexo, cujos vértices possuem grau par, é chamado de grafo euleriano, e um caminho fechado em um grafo que passe por cada aresta deste exatamente uma vez, é chamado de circuito (ou ciclo) euleriano (MALTA, 2008).

[2] Atualmente Caliningrado.
[3] <http://www.mat.uc.pt/~alma/escolas/pontes/>

2.2.2 Definições

Definição: Um grafo é um par $G = (V, E)$ de conjuntos de tal modo que $E \subseteq [V]^2$. Assim, os elementos de E são subconjuntos de pares ordenados de elementos de V. Os elementos de V são os vértices (ou nós ou pontos) do grafo G, os elementos de E são arestas (ou linhas) (DIESTEL, 1997).

Grafos são assim chamados porque eles podem ser representados graficamente, e é esta representação gráfica que possibilita entender muitas de suas propriedades. Cada vértice é indicado por um ponto, e cada aresta por uma linha que une os pontos que representam as suas extremidades (BONDY; MURTY, 2007). Um exemplo pode ser observado na Figura 6.

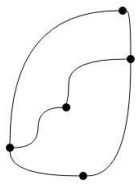


Figura 6 – Exemplo de grafo (BONDY; MURTY, 2007)

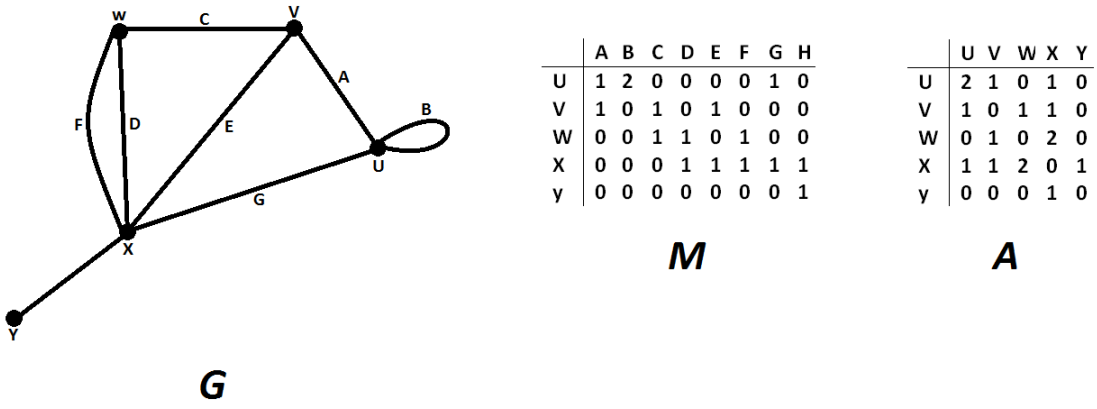
A maioria das definições e dos conceitos na teoria dos grafos são sugeridos por esta representação gráfica. As extremidades de uma aresta são referidas como sendo incidentes com a aresta, e vice-versa. Dois vértices que são incidentes com uma aresta em comum são chamados de adjacentes, e dois vértices adjacentes distintos são chamados de vizinhos (COSTA, 2011).

2.2.3 Representação de Grafos

Embora desenhos são um meio conveniente de especificação de grafos, eles claramente não são adequados para armazenar grafos em computadores, ou para a aplicação de métodos matemáticos para estudar suas propriedades. Para estes fins, são consideradas duas matrizes associadas com um grafo; uma matriz de incidência e uma matriz de adjacência (BONDY; MURTY, 2007).

Seja G um grafo, com conjunto de vértices V e conjunto de arestas E. A matriz de incidência de G é a matriz $M_G := (m_{ve})$, com dimensões $n \times m$, onde m_{ve} é o número de vezes que o vértice v e a aresta e estão conectados (BONDY; MURTY, 2007).

A matriz de adjacência de G é a matriz $A_G := (a_{uv})$, com dimensões $n \times n$, em que a_{uv} é o número de arestas que unem os vértices u e v. É possível verificar que a quantidade de memória necessária para essa representação é $\Theta(|V|^2)$ (BONDY; MURTY, 2007). A Figura 7 ilustra um grafo G representado na matriz de incidência M e na matriz de adjacência A.



	A	B	C	D	E	F	G	H
U	1	2	0	0	0	0	1	0
V	1	0	1	0	1	0	0	0
W	0	0	1	1	0	1	0	0
X	0	0	0	1	1	1	1	1
y	0	0	0	0	0	0	0	1

	U	V	W	X	Y
U	2	1	0	1	0
V	1	0	1	1	0
W	0	1	0	2	0
X	1	1	2	0	1
y	0	0	0	1	0

Figura 7 – Representação em matriz (BONDY; MURTY, 2007)

Como a maioria dos grafos possui um número bem maior de arestas do que os vértices, a matriz de adjacência de um grafo geralmente é menor do que a sua matriz de incidência e, assim, necessita de menos espaço de armazenamento. Ao lidar com grafos, é possível utilizar uma representação otimizada. Para cada vértice v, os vizinhos de v são armazenados em uma lista. A lista $(N(v)\colon\ v \in V)$ é chamada de lista de adjacência do grafo, onde $N(v)$ representa os vizinhos do vértice v. Um exemplo pode ser observado na Figura 8. A quantidade de memória necessária para uma lista de adjacências é $\Theta(|V|+|E|)$. Grafos são, normalmente, armazenados em computadores como listas de adjacência. Entretanto, em casos em que o grafo possua muitos vértices e poucas arestas, a representação em matriz de incidência é eficiente. O mesmo acontece para a representação em matriz de de adjacência, quando o grafo possui poucos vértices e muitas arestas (COSTA, 2011).

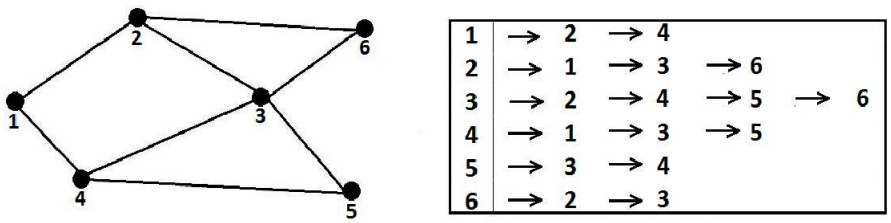


Figura 8 – Representação em lista de adjacência (COSTA, 2011)

Um conceito que torna-se muito importante ao se trabalhar com algoritmos complexos é a complexidades desses algoritmos, e como estes irão se comportar em uma aplicação. Certas complexidades podem tornar um algoritmo impraticável, a depender do caso. A seguir é apresentado um pouco mais sobre aspectos de complexidade de algoritmos.

2.3 Complexidade de algoritmos

Quando existem vários algoritmos que podem solucionar um problema, qual se deve escolher? Uma opção pode ser os algoritmos que possuem um fácil entendimento e depuração, porém dependendo do tipo de aplicação em que o algoritmo irá rodar, utilizar algoritmos que fazem um uso eficiente dos recursos do computador é essencial.

Para determinar qual algoritmo é mais eficiente que outro em determinado quesito, deve-se utilizar o cálculo de suas complexidades (custo). Este custo pode ser referente ao tempo de sua execução ou ao seu consumo de memória (ALBUQUERQUE, 2004).

Para o cálculo de complexidade, pode-se medir o número de passos de execução em um modelo matemático denominado maquina de Turing, ou medir o número de segundos gastos em um computador específico. A medida de complexidade é o crescimento assintótico dessa contagem de operações (JúNIOR, 2014).

A objetivo de se fazer a análise de complexidade de um algoritmo é obter estimativas de tempos de execução do algoritmo (JúNIOR, 2014).

2.3.1 Big O

Uma das preocupações com a eficiência é com problemas que envolvem um grande número de elementos. Se existir uma tabela com apenas dez elementos, mesmo o algoritmo considerado menos eficiente resolvem o problema, no entanto, à medida que o número de elementos aumenta, o esforço necessário começa a fazer diferença de algoritmo para algoritmo. A notação *Big O* faz uma estimativa de uma função de como o algoritmo se comporta com um número de elementos fixo ou tendendo ao infinito (JúNIOR, 2014). Pode-se observar na Figura 9 as principais funções que são utilizadas nesta notação.

Por exemplo, é mais importante saber que o número de operações executadas num algoritmo dobra se dobrarmos o valor de n, do que saber que para n igual a 100 são executadas 300 operações (JúNIOR, 2014).

Os resultados expressos em notação *Big O* devem ser interpretados com cuidado, pois indicam apenas que o tempo de execução do programa é proporcional a um determinado valor ou que nunca supera determinado valor. Na realidade o tempo de execução pode ser inferior ao valor indicado e pode ser que o pior caso nunca ocorra (JúNIOR, 2014).

Além dos conceitos de grafos e complexidade apresentados, este trabalho propõe-se também a trabalhar bastante com conceitos de Reutilização de Software, a seguir serão apresentados alguns, dos principais contextos desse tema.

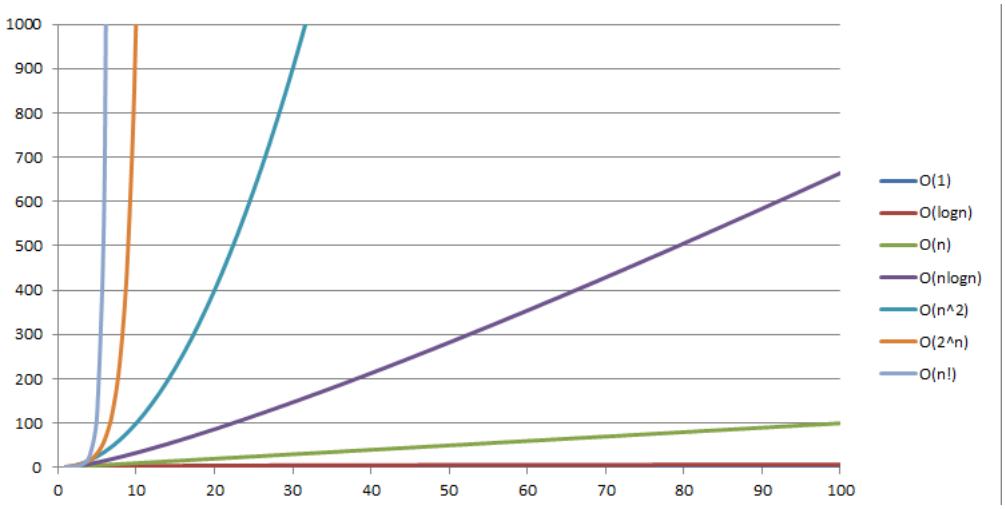


Figura 9 – Complexidade Big O

2.4 Reutilização de Software

A reutilização de software tem como objetivos aumentar a qualidade e a produtividade no desenvolvimento, pois busca evitar duplicidade de código e do esforço aplicado para desenvolver determinadas tarefas, reaproveitando o máximo possível experiências passadas (LUCRéDIO, 2009).

Todas as formas de reutilização de software usam algum tipo de abstração. Essa é uma característica importante presente nas técnicas de reutilização, pois facilita aos desenvolvedores o uso de objetos reutilizáveis. Sem abstrações, os desenvolvedores deveriam analisar todos os artefatos reutilizáveis buscando entender como cada um funciona e como e quando devem ser utilizados (KRUEGER, 1992).

Os próximos subtópicos irão apresentar temas deste trabalho que estão diretamente relacionados à reutilização de software.

2.4.1 Frameworks

Frameworks compartilham técnicas de reutilização em geral e são considerados uma importante parte da cultura de desenvolvimento no mundo da orientação a objetos (JOHNSON, 1997).

Fayad e Schimidt, em seu artigo (FAYAD; SCHMIDT, 1997) sobre *frameworks* de aplicações orientadas a objetos, mostram quais são os principais benefícios no uso de *frameworks*, dentre eles: modularização, reutilização, extensibilidade e inversão de controle.

- **modularização:** *Frameworks* encapsulam e interfaceiam alguns detalhes de implementação. Isso reduz o esforço necessário para entender e manter partes do software

existente, pois basta ao desenvolvedor usar o que lhe é oferecido sem necessariamente entender qual a implementação do *framework*;

- **reutilização:** As interfaces providas por *frameworks* ajudam também na reutilização através da definição de componentes genéricos, que podem ser aplicados em outras aplicações. Assim sendo, soluções comuns para sistemas diferentes podem ser usadas da mesma forma sem a necessidade de recriação das mesmas. Entende-se então, que estas soluções são pensadas uma única vez, e ao estarem presentes em um *framework*, basta que sejam usadas;

- **extensibilidade:** Este é um dos principais pontos positivos dos *frameworks*, pois estes fornecem métodos e interfaces estáveis que outras aplicações irão utilizar. Recomenda-se que essas aplicações façam uso desses métodos, visando resolver problemas parecidos em diferentes contextos. Uma boa estrutura de extensibilidade é essencial para garantir a customização de novos serviços e funcionalidades das aplicações, e;

- **inversão de controle:** A inversão de controle ocorre devido a forma como serão processados e entendidos muitos dos eventos de uma aplicação, que ficam invisíveis ao desenvolvedor quando este usa um *framework*, pois é o próprio *framework* quem decide o conjunto de métodos que será invocado para realizar uma determinada tarefa da aplicação.

2.4.1.1 Frameworks e Reutilização de Software

A tecnologia de reutilização ideal provê componentes que podem facilmente ser conectados para criar um novo sistema. Não é necessário ao desenvolvedor - cliente do framework - ter conhecimento de como o componente é implementado, e geralmente é fácil para ele aprender como o utilizar. O resultado é que o sistema tende a ser eficiente, fácil de manter e confiável (JOHNSON, 1997).

Frameworks são aplicações especializadas em prover classes e componentes abstratos que podem ser usados por outros sistemas. Estes proveem técnicas de reutilização robustas e de maior granularidade. Sendo aplicações independentes, é mais fácil usá-los em um maior número de sistemas (JOHNSON; FOOTE, 1988).

Para se alcançar a aplicação efetiva de um dado *framework*, é necessário ao desenvolvedor conhecer as interfaces que o *framework* proporciona antes de poder usá-las. Como podem existir diversas interfaces complexas, aprender a usar um novo *framework* pode ser difícil. Porém, os *frameworks* são poderosos e o tempo gasto em sua aprendizagem tende a ser recompensado, pois podem reduzir a quantidade de esforço aplicado para se desenvolver uma nova aplicação que os use (JOHNSON, 1997).

Ao longo do tempo, tornou-se muito caro desenvolver aplicações complexas a partir do zero. Isso ocorre, pois os componentes que são desenvolvidos devem passar por um criterioso processo de validação e manutenção, e isso procede sempre que um novo sistema é desenvolvido. Ao se usar *frameworks*, pode-se desenvolver componentes comuns e os processos citados são feitos em um único local (FAYAD; SCHMIDT, 1997).

As técnicas de reutilização são diferentes de acordo como o tipo do *framework* utilizados. Esses tipos podem ser "*white box*" ou "*black box*". O primeiro diz respeito a quando o código do *framework* é aberto e visível ao desenvolvedor, dessa forma, este pode estudar a implementação do *framework* e modificar o código de determinadas partes de acordo com suas necessidades. Os *frameworks* do tipo "*black box*" disponibilizam apenas interfaces ao desenvolvedor para que este possa usá-las. A forma como tudo é implementado e processado é desconhecida. No primeiro tipo, têm-se uma maior flexibilidade, porém, o uso é mais complexo ao desenvolvedor. No segundo, o uso é bem simples, porém, não existe flexibilidade para mudança da implementação (KROTH, 2000).

Existe também uma metodologia no desenvolvimento de *frameworks* que une as técnicas "*white box*" e "*black box*", assim, uma parte do *framework* é fechada e o desenvolvedor não pode alterar o código, apenas usar as funções disponíveis; outra parte é aberta ao desenvolvedor para realizar alterações no código. Essa abordagem é conhecida como *frameworks* do tipo "*gray box*" ou "caixa cinza" (KRISTENSEN; MADSEN; JøRGENSEN, 2004).

Além dos tipos de *frameworks*, estes também podem ser divididos quanto a sua aplicabilidade. Podem ser desenvolvidos para serem aplicados em qualquer domínio, de forma genérica sem se preocupar com algo específico, *frameworks horizontais*; ou podem ser desenvolvidos visando atender um tipo específico de domínio de problemas, *frameworks verticais*. Essas características dependem das necessidades apresentadas ao se trabalhar com *frameworks*, e isso impacta como será aplicada a reutilização (KROTH, 2000).

Na engenharia de software, busca-se cada vez mais o aumento da produtividade e da qualidade dos sistemas desenvolvidos. A reutilização de software, ao contrário de todas as demais partes de um sistema, é um fator que pode acarretar o aumento desses fatores, considerando que, ao se utilizar componentes já desenvolvidos e depurados, pode-se reduzir o tempo de desenvolvimento, de testes e as chances de ocorrência de erros que poderiam advir se fosse necessária a criação destes novos artefatos (SILVA, 2000).

Na qualidade, a reutilização advinda dos *frameworks* pode trazer ganhos de desempenho, confiabilidade e interoperabilidade de software (FAYAD; SCHMIDT, 1997).

2.4.1.2 Frameworks e Padrões

Padrões representam soluções recorrentes para problemas no desenvolvimento de software em um contexto específico. Tanto os padrões como os *frameworks* são técnicas de reutilização. A grande diferença é que os *frameworks* concentram-se na reutilização de estruturas, algoritmos e implementações em uma dada linguagem de programação. Já os padrões focam em apresentar desenhos abstratos de como resolver problemas; como microarquiteturas de software (FAYAD; SCHMIDT, 1997).

Um padrão descreve um problema a ser resolvido, apresenta uma solução e o contexto em que essa solução funciona, nomeia uma técnica, e descreve seus custos e benefícios (JOHNSON, 1997).

Quando um *framework* é implementado diversas vezes, este também pode ser considerado um padrão (JOHNSON, 1997). O MVC (*Model / View / Controller*) é um *framework* conceitual de interface com usuário que é considerado um padrão arquitetural(ALMEIDA, 2006).

Segundo (FAYAD; SCHMIDT, 1997), quando usados em conjunto com padrões, os *frameworks* podem aumentar significativamente a qualidade do software e reduzir o esforço de desenvolvimento.

2.4.2 Padrões de Projeto

Os padrões de Projeto são parte da vanguarda da tecnologia orientada a objetos, e este tema tem estado em constante crescimento no decorrer dos tempos(SHALLOWAY; TROTT, 2004). A proposição por trás dos padrões é que a qualidade do software pode ser medida objetivamente. Tal fato considera que, ao se analisar, o *design* de um padrão, este pode ser considerado bom ou ruim, e assim resultando em uma qualidade adequada ou inadequada(SHALLOWAY; TROTT, 2004).

Alexander, em seu livro (ALEXANDER, 1979) sobre padrões de construção, diz:

> "Cada padrão descreve um problema no nosso ambiente e o cerne da sua solução, de tal forma que você possa usar essa solução mais de um milhão de vezes, sem nunca fazê-lo da mesma maneira."

No livro de padrões de projeto (GAMMA et al., 1995), os autores concordam com as afirmações de Alexander. A diferença principal é que, no âmbito de software, os padrões são expressos em termos de objetos e interfaces ao invés de paredes e portas. Porém, ambos os tipos de padrões dizem respeito à solução para um problema em um contexto geral. Os autores consideram ainda que um padrão é dividido em quatro partes principais, que são:

- **nome:** Usado para descrever um problema de projeto, suas soluções e consequências em uma ou duas palavras. Quando um padrão possui um nome fica mais fácil de definir um vocabulário comum para tratar deste com outras pessoas;
- **problema:** O problema está ligado diretamente à situação em que deve ser aplicado o padrão, ou seja, qual o contexto. Algumas vezes o problema pode conter uma lista de condições que devem ser satisfeitas para que se possa alcançar sentido na aplicação do padrão;
- **solução:** Esta descreve todos os elementos que compõem o padrão, seus relacionamentos, responsabilidades e colaborações. É fornecida uma descrição abstrata de um problema de projeto e o arranjo geral de classes e objetos que resolvam o padrão. Não há uma solução ou implementação concreta, pois padrões são desenhados para serem usados em diferentes situações, e;
- **consequências:** As consequências apresentam uma série de resultados concernentes da aplicação do padrão, normalmente apresentando vantagens e desvantagens. São elementos críticos que entram na decisão da aplicação ou não do padrão que está em questão.

2.4.2.1 Por que usar Padrões de Projeto?

Os três pontos que justificam de forma adequada o uso de padrões de projeto são brevemente descritos a seguir, de acordo com (SHALLOWAY; TROTT, 2004):

- **soluções reutilizáveis:** O tempo gasto para aprender a utilizar determinado padrão é compensado pela reutilização que é oferecida. Têm-se o benefício de aplicar o que foi aprendido para diversos projetos. Depois de ter o conhecimento fixado, não é necessário reinventar soluções para problemas recorrentes, basta reutilizar o que os padrões oferecem;
- **terminologia comum:** Quando se está em um grupo de trabalho, são necessários uma base de vocabulário e pontos de visão do problema comuns. Os padrões de projeto providenciam um ponto comum de referência durante as fases de análise e *design* de um projeto, e;
- **perspectiva de alto nível do problema:** Os padrões dão aos desenvolvedores essa perspectiva, e ajudam na análise e entendimento dos problemas, facilitando a elaboração de uma solução mais adequada.

Os padrões ainda podem ajudar na refatoração de projetos. Uma das grandes dificuldades no desenvolvimento de software é que este tem de ser frequentemente reorganizado ou refatorado. Os padrões de projeto, ao oferecerem soluções comuns e já

consolidadas, podem reduzir a quantidade de refatoração que deverá ser feita mais tarde (GAMMA et al., 1995).

2.4.2.2 Classificação de Padrões de Projeto

Em (GAMMA et al., 1995), foram definidos e classificados 23 padrões. Essa classificação está feita de acordo com dois critérios: **finalidade** e **escopo**. O primeiro critério diz respeito ao que o padrão faz. A finalidade pode ser de criação, estrutural ou comportamental. Os padrões com finalidade de criação se preocupam com o processo de criação de objetos. Os estruturais focam em composições de classes ou objetos. Os comportamentais caracterizam as maneiras que as classes ou objetos interagem e distribuem responsabilidades.

O segundo critério especifica se o padrão é de classe ou objeto. Os padrões de classe lidam com relacionamentos entre classes e suas subclasses, para isso são utilizados relacionamentos de herança. E os padrões de objetos são mais dinâmicos, pois lidam com relacionamentos entre objetos que podem ser mudados em tempo de execução.

Nos apêndices em Reutilização de Software são encontrados alguns outros aspectos de reutilização, como serviços e o detalhamento de três padrões de projeto que foram implementados no desenvolvimento deste trabalho.

2.5 Resumo do Capítulo

Este capítulo apresentou o levantamento bibliográfico dos temas relacionados ao desenvolvimento deste trabalho. A seguir são apresentados os temas descritos nesse capítulo.

- Redes Sociais: este é o tema em que este trabalho é desenvolvido. Como o presente trabalho focou no desenvolvimento de um *framework* para redes sociais, é essencial o conhecimento e o entendimento deste tema para a realização deste tema.
- Teoria dos Grafos: o desenvolvimento de redes sociais têm como base o uso de grafos e, sendo assim, este é também um tema de suma importância para este trabalho.
- Reutilização de Software: este tópico representa um importante objetivo investigado ao longo desse trabalho. A reutilização de software é apresentada no uso de *frameworks*, padrões de projeto e serviços, todos estes utilizados na implementação do *framework*.

Outros detalhes de alguns tópicos apresentados neste capítulo podem ser encontrados na parte de apêndices deste documento.

O próximo capítulo apresenta a metodologia usada no decorrer do desenvolvimento deste trabalho.

3 Suporte Tecnológico

A fim de manter e disponibilizar o código e os artefatos gerados a partir deste trabalho, bem como o ambiente utilizado para a construção do *framework*, serão apresentadas as principais ferramentas que foram utilizadas durante o seu desenvolvimento.

3.1 Ferramentas de Desenvolvimento

3.1.1 Git

Este trabalho utilizou o Git[1] (versão 1.9.1) como o sistema de controle de versão. O Git foi projetado basicamente para facilitar a vida de quem quer executar projetos em equipe de forma segura. Foi criado por Linus Torvalds, que precisava de um controle de versão rápido e que pudesse lidar com as atividades envolvidas no desenvolvimento do Kernel do Linux. Linus desejava um ferramenta livre; não encontrando, decidiu criar o Git. Esse suporte foi nomeado em referência ao próprio Linus, pois no inglês britânico, *git* é uma gíria para "cabeça-dura".

Uma vantagem do Git é a possibilidade de controlar o projeto de forma descentralizada, ou seja, sem a exigência de um servidor mestre. A cada arquivo rastreado pelo Git, tem seu conteúdo verificado através do algoritmo de criptografia SHA-1[2].

O que faz o Git funcionar é sua habilidade de detectar mudanças em arquivos, não só que uma mudança ocorreu, mas também onde essa mudança ocorreu. Dessa forma, as alterações com problemas podem ser desfeitas, voltando para a versão estável.

3.1.2 GitHub

O GitHub[3], lançado em 2008 e escrito em Ruby on Rails, provê um armazenamento em nuvem (*Cloud*), onde se pode hospedar projetos que utilizam o Git como controle de versão. O GitHub possui funcionalidades de uma rede social como *feeds*, *followers*, wiki e gráficos para apresentar como os desenvolvedores trabalham em um repositório. Este lado social é interessante para descobrir novos projetos e receber ajuda em projetos particulares. É importante ressaltar que o repositório fornecido pelo GitHub é gratuito, e este repositório fica como de acesso público. Entretanto, existem planos comerciais nos quais o repositório pode se tornar privado.

---

1 <https://git-scm.com/>
2 <http://www.training.com.br/lpmaia/pub_seg_cripto.htm>
3 <https://github.com>

A seguir é apresentada, na Figura 10, a tela de um repositório hospedado no GitHub.

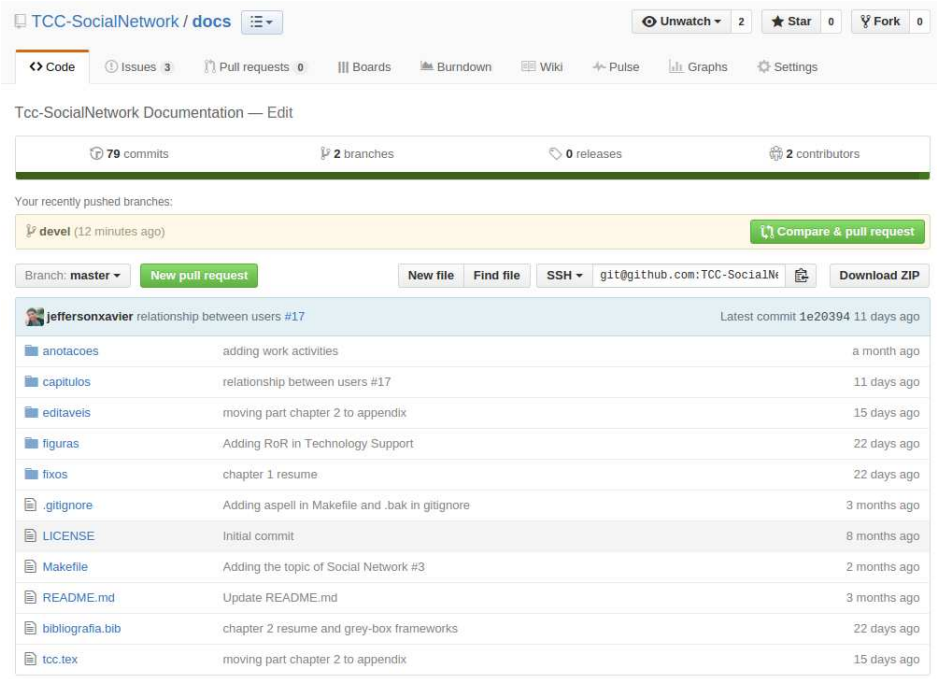


Figura 10 – Repositório no GitHub

3.1.3 Waffle

O Waffle[4] é um gerenciador de projetos gratuito conectado aos *Pull Requests* e as *Issues* do GitHub, que visa facilitar o trabalho das equipes de engenharia no que se refere ao acompanhamento das tarefas de uma forma visual, mostrando-as em um quadro com divisões para cada fase.

O Waffle escuta as ações em seu fluxo de trabalho para saber quando o trabalho é iniciado, quando está pronto para revisão, ou quando está finalizado e atualiza seu status automaticamente.

A Figura 11 apresenta um exemplo de um *board* do Waffle.

4 <https://waffle.io>

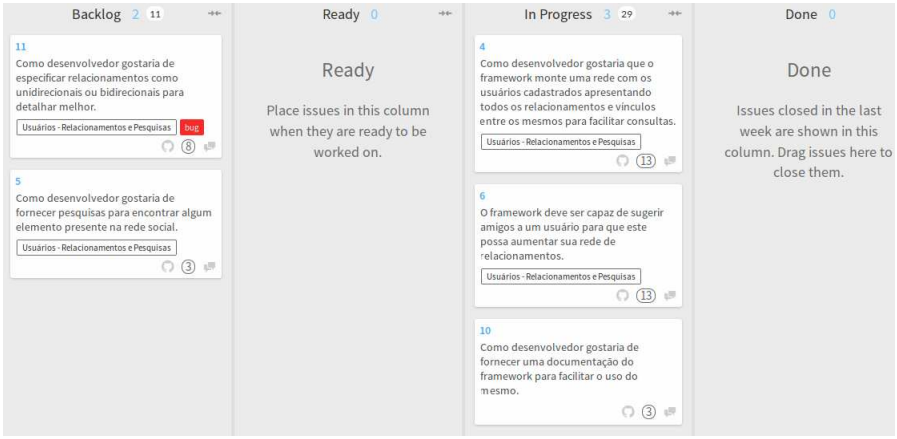


Figura 11 – Board gerado pelo Waffle

3.1.4 LaTeX 3

O LaTeX[5] é um pacote de macros ou marcações para o processador de textos TeX, utilizado amplamente pela comunidade científica devido à grande qualidade tipográfica. Adicionalmente, torna mais fácil e rápida a produção de documentos em TeX. O LaTeX foi desenvolvido por Leslie Lamport a partir do TeX, criado por Donald Knuth, ambos de código aberto.

O objetivo desse suporte é que o autor se distancie da apresentação visual do trabalho, concentrando-se no seu conteúdo. Para isto, ele possui formas de se lidar facilmente com estruturas como, por exemplo, bibliografias, citações, formatos de páginas e referências. O LaTeX não é algo imutável, e como tal, suporta várias maneiras de estilizar e formatar os documentos.

3.1.5 Sublime Text 3

Este trabalho utilizou o Sublime Text (*build 3103*) como editor de texto. Inicialmente pensado para ser uma extensão do Vim, o Sublime Text[6] é um editor de texto multiplataforma, escrito em linguagem C++. Com o Sublime é possível automatizar várias tarefas a partir de recursos como, por exemplo, macros, autocompletar, repetição de ações e construção automática.

Outros recursos como dividir a tela em várias janelas, auto *save*, navegação entre páginas por meio de abas e suporte a várias linguagens, por exemplo, C, C++, C#, CSS, HTML, Java, LaTeX, PHP, Ruby, SQL, XML, JavaScript e Groovy, fazem do Sublime uma ferramenta popular entre os programadores.

A seguir, na Figura 12, é apresentada uma imagem da ferramenta.

[5] <https://www.latex-project.org>
[6] <http://www.sublimetext.com/>

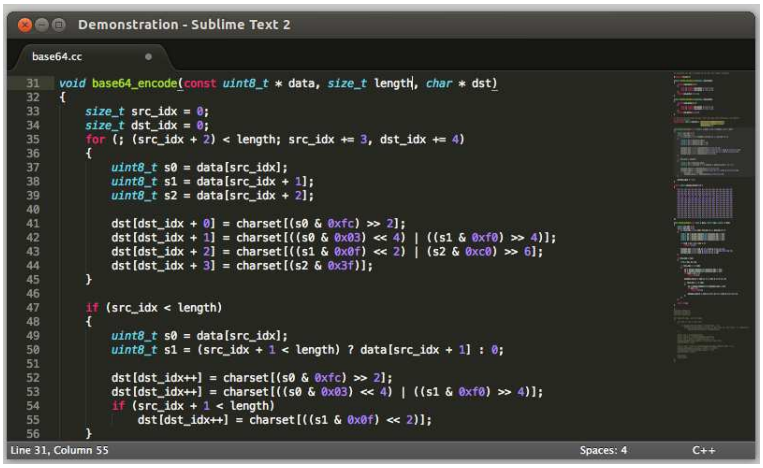


Figura 12 – Sublime Text 3

3.1.6 Bizagi Modeler

Bizagi[7] (utilizado na versão 3.0) é uma ferramenta para modelar processos. Esta abrange tanto o mapeamento de processos de trabalho quanto a automação de processos a partir do mapeamento, permitindo que os usuários possam desenhar, documentar e compartilhar seus processos de trabalho usando a notação BPMN (*Business Process Management Notation*). Um diferencial do Bizagi é a possibilidade de realizar tarefas em conjunto, utilizando um ambiente virtual.

A seguir é apresentado um exemplo de uso do Bizagi, na Figura 13.

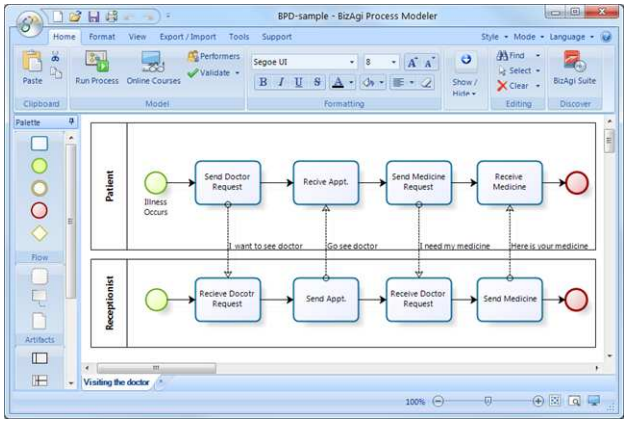


Figura 13 – Bizagi Modeler

3.1.7 Ubuntu

Ubuntu[8] é uma palavra de origem africana que significa: “Humanidade para os outros” ou ainda “Sou o que sou pelo que nós somos”. A distribuição Ubuntu traz o espírito desta palavra para o mundo do software livre. Baseado em Linux e criado a partir

7 <http://www.bizagi.com>
8 <http://www.ubuntu.com/>

do Debian, o Ubuntu é patrocinado pela Canonical Ltda., e é licenciado pela licença GPL (*General License Public*). Para o desenvolvimento deste trabalho foi utilizado o Ubuntu na versão 14.04 LTS.

3.1.8 Ruby on Rails

Ruby on Rails [9] (utilizado na versão 4.2.5) é um *framework* de desenvolvimento de aplicações web escrito na linguagem Ruby, projetado para tornar a programação de aplicações web mais fácil. A filosofia Rails inclui dois princípios orientadores:

- **DRY** (*Don't Repeat Yourself*), ou não repita a si mesmo, é um princípio de desenvolvimento de software que afirma que "Cada pedaço de conhecimento deve ter uma única representação inequívoca e autoritária dentro de um sistema"(WILSON et al., 2012).
- **Convenção sobre configuração** é um modelo onde o desenvolvedor precisa definir apenas aspectos não convencionais da aplicação e os aspectos convencionais são pré-estabelecidos. Tal estratégia evita o uso maciço de arquivos de configuração.

3.2 Resumo do Capítulo

Este capítulo descreveu as principais ferramentas que foram utilizadas no decorrer deste trabalho. Tais ferramentas visam auxiliar o desenvolvimento do trabalho, tanto na sua documentação, quanto no planejamento e na implementação.

No próximo capítulo, são apresentadas as metodologias de pesquisa e desenvolvimento que serão utilizadas no presente trabalho.

[9] <http://guides.rubyonrails.org/>

4 Metodologia

A partir das pesquisas realizadas e documentadas no Apêndice Classificações de metodologias, pôde-se definir a metodologia aplicada ao trabalho. Este trabalho possui metodologias específicas para a realização do levantamento bibliográfico bem como para estabelecer como proceder nas atividades de desenvolvimento.

A seguir serão apresentadas as duas metodologias e como estas foram aplicadas ao longo deste trabalho.

4.1 Metodologia de Pesquisa

Analisando-se o tema proposto para este trabalho, é possível perceber que se trata de uma pesquisa aplicada, uma vez que todo levantamento bibliográfico visa a compreensão dos temas, sendo esses conhecimentos adquiridos aplicados no desenvolvimento. A abordagem é tanto quantitativa quanto qualitativa.

Com relação aos objetivos da pesquisa, considera-se que é uma pesquisa exploratória, devido à necessidade encontrada em entender adequadamente o problema apresentado e se obter um maior entendimento de todos os temas relacionados ao que se propõe.

No contexto dos procedimentos técnicos, foi aplicada a pesquisa bibliográfica, visando o levantamento de fontes para se alcançar o conhecimento necessário para a realização do trabalho.

Por fim, no intuito de analisar os resultados obtidos no projeto, utilizou-se a pesquisa-ação. Nesse caso, foram conduzidos ciclos evolutivos visando:

1. A identificação de não conformidades;
2. A execução de ações estratégicas no intuito de melhorar essas não conformidades;
3. A análise dos resultados obtidos com a aplicação dessas ações.

4.2 Metodologia de Desenvolvimento

A metodologia de desenvolvimento seguiu algumas práticas ágeis já consolidadas. Basicamente, utilizou-se as práticas ágeis propostas na metodologia *Scrum*[1], como o uso de histórias de usuários, reuniões diárias (no contexto do trabalho foram de dois em dois dias), *sprints* de quinze dias, *backlog do produto* e *backlog de sprint*. Quando necessário, voltou-se

[1] <http://www.desenvolvimentoagil.com.br/scrum/>

também à algumas práticas do *XP*[2], como o uso de integração contínua, pareamento e *planning poker* para estimativa de pontos das histórias.

4.3 Fluxo de Trabalho

A Figura 14 apresenta o fluxo de trabalho que orientou a condução desse projeto.

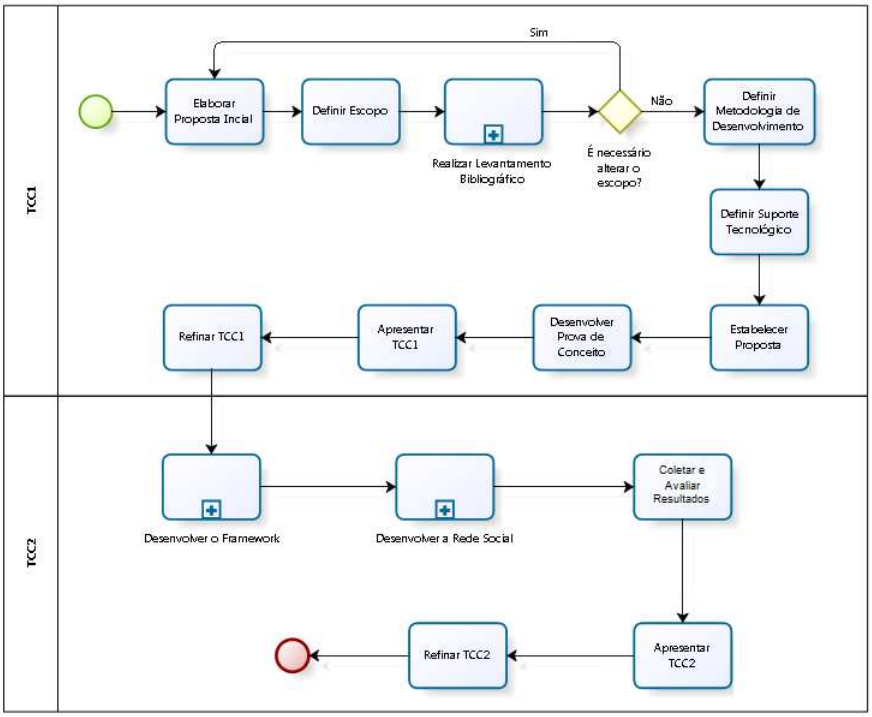


Figura 14 – Fluxo de trabalho

Primeiramente, a ideia inicial para o projeto foi definida bem como trabalhada, obtendo como resultado a proposta (em seu esboço preliminar). A partir dessa proposta, o escopo inicial foi definido, conferindo base para a realização do levantamento bibliográfico. Ao final, voltou-se o fluxo ao início, visando um melhor refinamento do que já havia sido realizado.

A partir da proposta inicial, foram definidos a metodologia de desenvolvimento, o suporte tecnológico e uma proposta mais concreta, a qual foi de fato desenvolvida ao longo do TCC. Para comprovar a viabilidade da proposta, uma prova de conceito foi elaborada.

Por fim, o trabalho foi apresentado e refinado de acordo com as observações da banca examinadora.

A segunda parte do fluxo apresenta as atividades relacionadas ao desenvolvimento do *framework* e da rede social que exemplificou o uso deste *framework*, comprovando

[2] <http://www.desenvolvimentoagil.com.br/xp/>

assim a sua instanciação e aplicabilidade. Na última etapa, os resultados obtidos foram coletados e o trabalho final foi apresentado e refinado.

Como pode ser visto no processo apresentado na Figura 14, existem duas etapas envolvendo desenvolvimento, no caso: "Desenvolver o *Framework*" e "Desenvolver a Rede Social". Cada uma dessas etapas, seguiu um Processo alternativo, como mostrado na Figura 15.

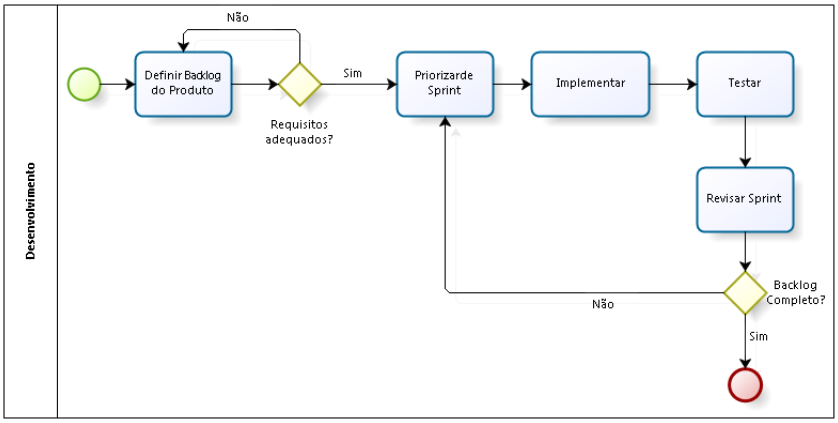


Figura 15 – Processo de desenvolvimento

Ao se observar a Figura 15, teve-se a etapa inicial de criação do *backlog* do produto. Quando completo, deu-se inicio a um fluxo de desenvolvimento que passou por etapas de desenvolvimento e testes em *sprints*.

4.4 Cronograma

A seguir, nas Figuras 16 e 17, são apresentados os cronogramas preliminares referentes às atividades que foram realizadas durante todo este trabalho. Estes cronogramas estão divididos entre as atividades das etapas um (que abrange o período entre agosto de 2015 e dezembro de 2015) e dois (que abrange o período entre março e julho de 2016).

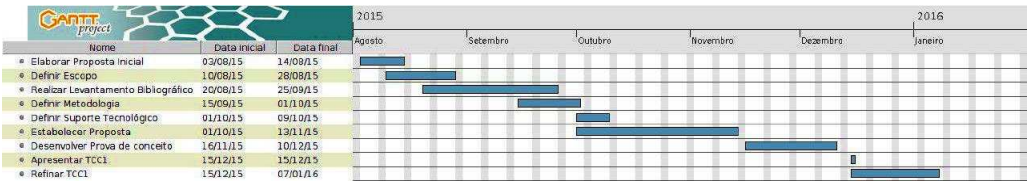


Figura 16 – Cronograma primeira parte

4.5 Resumo do Capítulo

Este capítulo apresentou as metodologias que foram utilizadas durante a realização das pesquisas e do desenvolvimento deste trabalho.

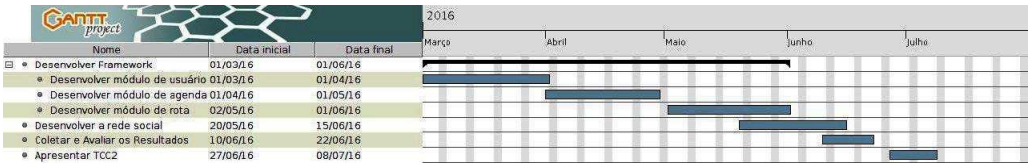


Figura 17 – Cronograma segunda parte

Inicialmente, foram documentadas, no apêndice, diversas metodologias. Essas metodologias estão divididas de acordo com o ponto de vista da natureza da pesquisa, da forma de abordagem, dos objetivos e dos procedimentos técnicos. Com a apresentação dessas metodologias, foram selecionadas as que melhor se adequavam a este trabalho. A metodologia de desenvolvimento seguiu práticas ágeis e está baseada no Scrum.

Foi apresentado também o fluxo de trabalho com as atividades definidas para as duas etapas de desenvolvimento deste trabalho bem como o cronograma proposto para cumprimento destas atividades.

O próximo capítulo apresentará a proposta consolidada, a qual orientou o processo de desenvolvimento desse trabalho.

5 Proposta

A seguir será apresentado cada ponto proposto em tópicos mais detalhados. Dessa forma, o capitulo está organizado para apresentar a visão geral do *framework* proposto, enfatizado como o mesmo será utilizado, e alguns trabalhos relacionados.

Ao final deste capítulo, pretende-se que o leitor forme uma ideia mais concreta sobre a proposta deste trabalho.

5.1 O Framework

A proposta deste trabalho foi desenvolver um *framework* que auxilie no desenvolvimento de redes sociais. Este *framework* deve oferecer recursos gerais com toda a lógica de usuários e relacionamentos, e recursos mais específicos para redes sociais baseadas em rotas e agendas. O recurso de rotas deve ser capaz de fazer um mapeamento de rotas de interesse de um determinado usuário e auxiliar este a fazer comparações com rotas de outros usuários. A agenda deve oferecer recursos de controle de ocupações no decorrer de dias e horários e auxiliar um usuário a encontrar dias e/ou horários comuns entre um grupo de usuários qualquer.

Parte do *framework* é extensível, possibilitando ao desenvolvedor reutilizá-lo a partir de seus pontos flexíveis, modelados e implementados usando o conceito de herança. O desenvolvedor poderá, portanto, acrescentar e/ou alterar suas implementações conforme as suas necessidades. O *framework*, por si só, oferecerá recursos que poderão ser reutilizados diretamente pelo desenvolvedor, não necessitando estendê-los. A extensibilidade só será necessária caso sejam desejados níveis de especificidade maiores.

A seguir, têm-se um maior detalhamento de cada um dos três principais componentes fornecidos pelo *framework*.

5.1.1 Relacionamento de Usuários

O relacionamento em redes sociais dá-se por meio de interações entre os usuários. Estas interações variam de acordo com a rede em que o usuário está inserido. Alguém pode apenas seguir outras pessoas e acompanhar suas postagens. Este é um relacionamento unidirecional, pois não é necessário que uma pessoa seguida acompanhe também postagens de seus seguidores. Existe também outro tipo de relacionamento, onde é necessário que as duas pessoas estejam diretamente ligadas entre si, o que o torna necessariamente bidirecional. Neste tipo, podem ser considerados relacionamentos entre conhecidos, amigos, namorados, familiares, entre outros.

Na montagem da estrutura de relacionamentos entre os usuários para redes sociais, foi necessário fazer uso de grafos, onde os usuários serão representados como os vértices e os relacionamentos como as arestas. No caso dos relacionamentos unidirecionais, apenas uma aresta é criada. Esta aresta faz uma ligação a partir do vértice do usuário seguidor para o vértice do usuário que este deseja seguir, como pode ser observado na Figura 18. No caso dos relacionamentos bidirecionais, a ligação deverá ser feita usando-se duas arestas. É necessário que para ambos os usuários exista uma possibilidade de se chegar ao outro, portanto, deve existir uma aresta que ligue um usuário "A" a um usuário "B", e uma aresta que ligue o usuário "B" ao usuário "A", vide a Figura 19.

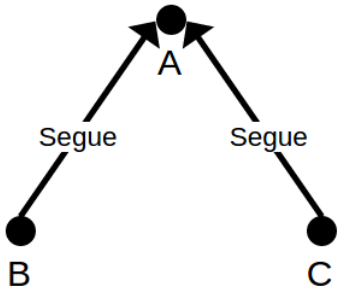


Figura 18 – Exemplo de aresta de seguir usuário

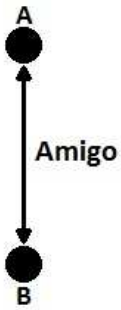


Figura 19 – Exemplo de aresta bidirecional de amizade

Cada aresta deverá possuir uma ou mais descrições que detalham qual o relacionamento entre os usuários. Isso fica mais claro ao se falar de relacionamentos bidirecionais, onde pode existir entre duas pessoas um relacionamento de amizade e parentesco, por exemplo. Esse relacionamento pode ser visualizado na Figura 20.

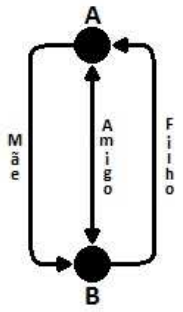


Figura 20 – Exemplo de aresta bidirecional de parentesco

5.1.2 Controle de Rotas

Rota[1] é um itinerário que se percorre para ir de um lugar a outro, indicando a direção ou rumo a ser percorrido. Um exemplo de rota pode ser visualizado na Figura 21.

Figura 21 – Exemplo de rota[2]

O *framework* oferece recursos que auxiliem o usuário a definir as rotas que percorrem e buscar rotas de outros usuários que coincidam, em parte ou integralmente, com as suas. Redes sociais podem utilizar estes dados para aplicações diversas como, por exemplo, caronas e encontros para ciclistas.

Conforme mostrado na Figura 21, pode-se ver uma rota que leva do ponto "0" ao ponto "5". Podem existir diversos usuários com rotas comuns nesse caminho, por exemplo, um usuário que siga todo o caminho traçado, ou um usuário que partisse do ponto "2" e chegasse ao ponto "4", e assim por diante. Uma das funcionalidades do *framework*, neste contexto, seria encontrar todos os usuários que compartilhassem esses caminhos, ou encontrar um caminho compartilhado entre um grupo de usuários, por exemplo.

[1] <http://www.dicio.com.br/rota/>
[2] <http://sede.wikidot.com/andy-s-brainstorm>

5.1.3 Controle de Agenda

Outra funcionalidade que o *framework* fornece é a conciliação de agendas entre os usuários, possibilitando, assim, encontrar dias da semana e horários específicos que são comuns a um determinado grupo. Esta funcionalidade pode auxiliar os usuários em diversos aspectos como, por exemplo, encontrar um horário em comum para realizar uma tarefa.

A Figura 22 mostra um exemplo de uma agenda semanal com todos os horários livres.

Horário	Segunda	Terça	Quarta	Quinta	Sexta	Sábado	Domingo
05:00							
06:00							
07:00							
08:00							
09:00							
10:00							
11:00							
12:00							
13:00							
14:00							
15:00							
16:00							
17:00							
18:00							
19:00							
20:00							
21:00							
22:00							
23:00							
00:00							
01:00							
02:00							
03:00							
04:00							

Figura 22 – Exemplo de agenda semanal[3]

Agendas dessa forma auxiliam usuários a terem um melhor controle de suas tarefas semanais, marcarem eventos, entre outros aspectos. O período compreendido por uma agenda pode variar de acordo com cada contexto. Pode ser uma agenda semanal, conforme a apresentada, mensal, anual, ou ainda um período não necessariamente pré-definido, o que poderia auxiliar usuários a marcarem eventos para datas futuras. De modo simplificado, basta dados como dia e horários de início e fim de uma tarefa para aplicar um controle de agenda.

O *framework* oferece o suporte para criação de agendas conforme apresentado. Assim, é possível, a partir de um período pré-determinado, encontrar um horário e dia comuns a um conjunto de usuários. Isso pode auxiliar a criação e marcação de eventos visando atender o maior número de usuários, conforme o contexto da aplicação.

5.1.4 Modelo Inicial

A Figura 23 apresenta os principais componentes que são oferecidos pelo *framework*. Fica evidenciado, no modelo, o relacionamento de *Users* com *Routes* e *Schedules*.

[3] <http://universovocesaude.com.br/wp-content/uploads/2014/08/horario-semanal_JPG.jpg>

Em cada um desses componentes, existem classes que implementam todo o modelo e as regras de negócio propostas.

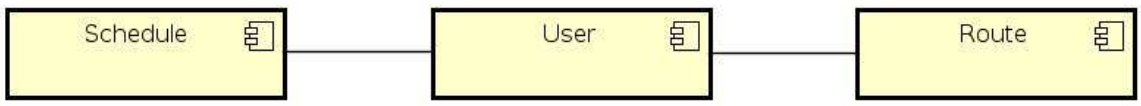


Figura 23 – Diagrama de componentes inicial

O diagrama mostra apenas um modelo preliminar que conferiu uma base para a implementação das funcionalidades propostas. Cada componente contém algoritmos relacionados e aplica padrões de projetos adequados, buscando proporcionar qualidade, soluções reutilizáveis, fácil manutenção e evolução.

5.2 Uso do Framework

O *framework* é *open sorce*, desenvolvido em *"Ruby on Rails"*, e tem sua *"Gem"*[4] no repositório oficial do Rails[5] para uso de outros desenvolvedores. O endereço completo para a *Gem* do SocialFramework é <https://rubygems.org/gems/social_framework>.

Para comprovação do funcionamento do *framework*, este trabalho se propôs a desenvolver uma rede social que faça uso do mesmo. Esta rede faz uso dos principais recursos providos pelo *framework*. Possibilitando assim, avaliar o seu uso na prática.

A Figura 24 apresenta um modelo que procura ilustrar e contextualizar um possível uso para o *framework*.

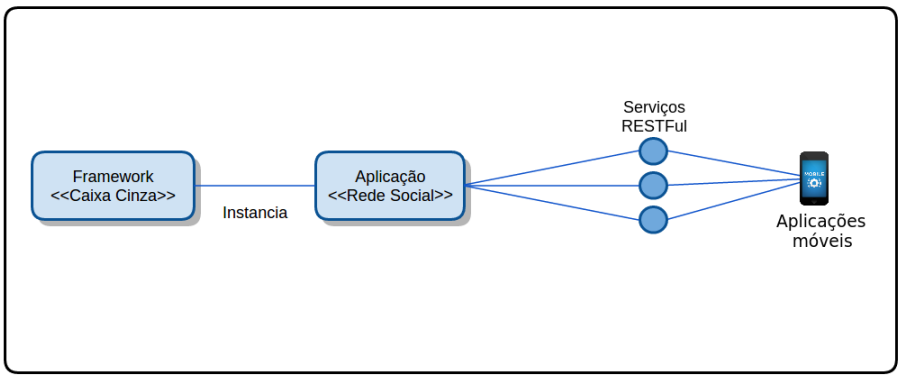


Figura 24 – Modelo ilustrando um possível uso do *Framework*

O modelo apresentado na Figura 24 mostra a instanciação do *framework* visando o desenvolvimento de uma rede social. Essa instanciação traz consigo toda a implementação desenvolvida no *framework* e a disponibiliza para ser usada na aplicação. Ao analisar a

[4] <httpttp://www.akitaonrails.com/2009/2/2/entendendo-rubygems>
[5] <https://rubygems.org/>

Figura 24, percebe-se um *framework "black box"*. Entretanto, a real proposta apresenta um *framework "grey box"*, pois, grande parte das classes disponibilizadas podem ser estendidas e ter os métodos alterados, ou atributos acrescentados, entre outros aspectos. Ressalta-se ainda, que os componentes mostrados no diagrama na seção anterior são disponibilizados como pacotes na forma de *hotspots* (FRANCA; STAA, 2001), garantindo, assim, uma maior flexibilidade ao desenvolvedor na escolha dos componentes que serão utilizados.

O modelo apresentado na Figura 24, é indicado o uso de serviços *RESTFul* para a aplicação desenvolvida. Esses serviços reforçam mais uma vez a ideia de reutilização de software ao fornecerem recursos simples e leves para utilização em dispositivos móveis, por exemplo, sem a necessidade de replicação da lógica de negócio desenvolvida por trás da aplicação. Contudo, esse é um uso recomendado, e cabe ao desenvolvedor disponibilizar esses serviços em sua rede social.

Com o objetivo de entender detalhes básicos de redes sociais, e compreender algumas das dificuldades para o pleno desenvolvimento do *framework* proposto, foi desenvolvida uma Prova de Conceito. Essa prova de conceito procurou lidar com uma funcionalidade chave em redes sociais, no caso, o relacionamento de usuários. Portanto, foi um passo inicial para o desenvolvimento da proposta. O desenvolvimento completo foi realizado na segunda parte deste trabalho, conforme evidenciado no fluxo de trabalho bem como no cronograma, ambos apresentados no capítulo de Metodologia.

5.3 Trabalhos Relacionados

Em algumas pesquisas relacionadas sobre trabalhos parecidos com o que se propôs, pode-se encontrar alguns sistemas semelhantes, os quais serão apresentados a seguir.

Existe uma comparação entre vinte e cinco plataformas, as quais permitem começar um serviço de rede social[6] próprio. Porém, algumas das plataformas apresentadas são pagas ou não são voltadas para desenvolvedores, ou seja, são plataformas que fornecem funcionalidades para usuários comuns poderem montar uma rede social a partir de um *template* visual já definido.

A Tabela 1 apresenta um comparativo entre algumas das plataformas que apresentam suporte à criação de redes sociais. Durante a pesquisa realizada para constatar as características de cada plataforma, percebeu-se que as plataformas feitas a partir da linguagem de programação PHP são predominantes. Porém, no intuito de focar nas plataformas que fornecem suporte para outras linguagens, deu-se um enfoque maior para essas outras plataformas apresentando-as na tabela.

[6] <http://www.tripwiremagazine.com/2010/07/25-best-social-networking-platforms-to-start-your-own-service.html>

Tabela 1 – Comparação entre os trabalhos relacionados

Rede Social	Licença	Preço	Linguagem	Customizável
SocialEngine	Customizada	A partir de $ 299,99	PHP	Sim
Elgg	GPL 2.0	Grátis	PHP	Extensível via plugins e com uma API flexível
XOOPS	GPL 2.0	Grátis	PHP	Extensível via módulos
mooSocial	Customizada	A partir de $ 149,99	PHP	
Anahita Social Engine	GPL 2.0	Grátis	PHP	Sim
Telligent Community	Customizada	Grátis com limitações	ASP .NET	Extensível via plugins, widgets, tarefas, eventos e REST
Newebe	AGPL	Grátis	Python/Coffeescript	Sim
Buddycloud	Apache 2.0	Grátis	NodeJS/Java	Sim
eXo Platform	LGPL	Grátis	Java	Extensível via plugins e com uma API flexível
Kune	AGPLv3	Grátis	Java	Extensível via widgets e módulos
Diaspora	AGPLv3	Grátis	Ruby	Sim
Insoshi	MIT	Grátis	Ruby	Sim
Libertree	AGPLv3	Grátis	Ruby	Sim

Outra característica importante que foi observada na comparação é que os componentes customizáveis que as plataformas afirmam possuir são, em sua maioria, via interface gráfica. Em poucas plataformas, é possível a customização via código, sendo, nesses casos, a maior parte feita via construção de *plugins*.

Existem também alguns *frameworks* semelhantes desenvolvidos em *Ruby*[7]. Estes, em sua maioria, fornecem recursos a desenvolvedores, porém, apenas a parte geral de relacionamento de usuários. A seguir é possível visualizar, na Figura 25, uma comparação da popularidade desses *frameworks*. O cálculo da popularidade é feito com base na quantidade de observadores e de *forks* que o repositório do *framework* possui.

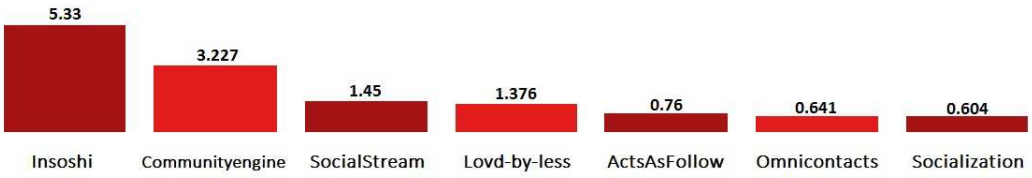


Figura 25 – Popularidade dos *Frameworks* em *Ruby*

5.4 Resumo do Capítulo

Este capítulo descreveu a proposta deste trabalho apresentando seus pontos principais e alguns detalhes técnicos.

A partir desse escopo inicial, e da prova de conceito realizada, desenvolveu-se um *framework* capaz de auxiliar interessados no desenvolvimento de redes sociais, das funcionalidades básicas às específicas dos domínios de rota e agenda, e esse será apresentado no próximo capítulo.

[7] <https://www.ruby-toolbox.com/categories/social_networking>

6 SocialFramework

O SocialFramework é uma *engine* do Rails[1] que ajuda a construir redes sociais com recursos comuns e específicos.

O SocialFramework é dividido em três módulos, que são: usuários, rotas e agendas. No módulo de usuários são providos os principais recursos para usuários, como autenticação, registro e pesquisas. No módulo de rotas, o *framework* provê recursos para checar a compatibilidade de rotas. No módulo de agenda, são oferecidos recursos para definir agendas de usuários e a compatibilidade de horários entre usuários.

Portanto, o SocialFramework pode ajudar a construir redes sociais genéricas e específicas, procurando evitar ao usuário, a preocupação com problemas recorrentes nestes tipos de sistemas.

Todo o código desenvolvido do SocialFramework pode ser encontrado no endereço: <https://github.com/TCC-SocialNetwork/social_framework>.

A seguir é apresentada uma documentação de todo o funcionamento do SocialFramework, passando pela instalação, uso e interface pública do mesmo.

6.1 Instalação

A instalação do SocialFramework possui dois passos simples.

Primeiro, deve-se adicionar a seguinte linha no arquivo Gemfile:

```
gem 'social_framework'
```

Código 6.1 – Gemfile

Após a adição da *gem* no Gemfile, deve-se proceder com a instalação, executando o seguinte comando:

```
$ bundle install
```

Código 6.2 – Instalação

Tal procedimento irá instalar o SocialFramework na aplicação.

Nas seções a seguir, será apresentado o funcionamento do SocialFramework como um *framework* do tipo caixa preta e do tipo caixa cinza. Mais especificamente, nas seções Primeiros Passos e Módulos do SocialFramework, será apresentado seu uso como um *framework* caixa preta. Já na seção Hotspots, será apresentado o seu uso como um *framework* caixa cinza.

[1] <http://rubyonrails.org/>

6.2 Primeiros Passos

O uso do SocialFramework é indicado para quem já possui algum conhecimento do *framework* Rails para um maior entendimento e aproveitamento de seus recursos fornecidos.

O SocialFramework utiliza o Devise[2], que é uma solução de autenticação flexível para Rails. Toda a documentação do Devise pode ser encontrada no repositório do projeto no GitHub, no endereço <https://github.com/plataformatec/devise>.

A classe de usuário foi implementada no SocialFramework, e possui algumas diferenças da classe de usuário padrão do Devise, como a adição do atributo nome e dos relacionamentos dos usuários. As *controllers* e *views* do Devise também foram alteradas para adicionar os novos recursos.

Inicialmente, alguns arquivos devem ser adicionados na aplicação para que as configurações do SocialFramework e do Devise sejam importadas. Esses arquivos são os *initializers*, o arquivo de internacionalização (i18n) do Devise, as rotas e as *views* de *registration* e *session* para criar e autenticar usuários. Para isso deve-se executar:

```
$ rails generate social_framework:install
```

Código 6.3 – Gerando as configurações do SocialFramework e do Devise

Este comando irá criar o arquivo "config/initializers/devise.rb", contendo as configurações do Devise; o arquivo "config/initializers/social_framework.rb", que contém as configurações do SocialFramework; o arquivo i18n do Devise, e irá adicionar a rota "devise_for", para mapear as *controllers* e *views* do Devise. Dessa forma, a aplicação estará preparada para usar o módulo de usuários com as configurações e comportamentos padrões.

Todas as tabelas do *framework* deverão criadas no banco de dados da aplicação. Portanto, para testar a aplicação não se deve esquecer de executar as *migrations*:

```
$ rake db:create
$ rake db:migrate
```

Código 6.4 – Migrations

Para acessar a página de autenticação, deve-se acessar a rota "/users/sign_in". Esta página está preparada para autenticar usuários com *email* ou nome de usuário. Para cadastrar um novo usuário, deve-se acessar a rota "/users/sign_up". Ao criar um novo usuário, este estará automaticamente conectado.

Quando se usa o Devise Mailer como o módulo de confirmação, é necessário adicionar em seu ambiente as configurações para a ação *mailer*. Por exemplo, se estiver no

[2] <https://github.com/plataformatec/devise>

ambiente de desenvolvimento, deve-se adicionar as seguintes configurações no arquivo 'development.rb'.

```
config.action_mailer.default_url_options = {host: 'localhost', port: 3000}

config.action_mailer.delivery_method = :smtp

config.action_mailer.smtp_settings = {
  address: "smtp.gmail.com",
  port: 587,
  domain: ENV["GMAIL_DOMAIN"],
  authentication: "plain",
  enable_starttls_auto: true,
  user_name: ENV["GMAIL_USERNAME"],
  password: ENV["GMAIL_PASSWORD"]
}
```

Código 6.5 – Configurações de email

Pode-se alterar os valores em 'domain', 'user_name', e 'password' ou criar as variáveis de ambiente locais, sendo indicado para esconder essas informações sigilosas e garantir uma maior segurança. As mesmas configurações são válidas para os ambientes de teste e produção, nos arquivos 'test.rb' e 'production.rb'.

6.2.1 Controller Filters and Helpers

O Devise fornece alguns elementos para serem usados nas *controllers* e *views*. Para configurar uma *controller* com uma autenticação de usuário, basta adicionar o seguinte filtro. Isto funciona porque o SocialFramework já contém a classe de usuário.

```
before_action :authenticate_user!
```

Código 6.6 – Requer autenticação de usuário

Outros elementos são:

```
user_signed_in?
```

Código 6.7 – Verifica se o usuário está logado

```
current_user
```

Código 6.8 – Recupera o usuário logado

```
user_session
```

Código 6.9 – Recupera a sessão para este escopo

Após o usuário logar, confirmar a conta ou atualizar a senha, o Devise irá procurar por um caminho *root* para redirecionar. Para alterar esse comportamento padrão, pode-se

sobrescrever os métodos “after_sing_in_path_for” e “after_sing_out_path_for”, para definir para onde o usuário deve ser redirecionado após realizar o *login* e o *logout* respectivamente.

6.2.2 Configuração das Modelos

A classe de usuário no SocialFramework implementa os módulos padrões do Devise. Na documentação completa do Devise, é possível encontrar todos os módulos e suas características.

Além dos módulos do Devise, a classe de usuário do SocialFramework possui métodos que implementam os comportamentos para os relacionamentos de usuários. É possível sobrescrever qualquer comportamento, estendendo a classe em outra modelo, como:

```
class OtherUserClass < SocialFramework::User
  # Your code goes here
end
```

Código 6.10 – Estensão da classe de usuário

É necessário mudar o nome da classe no arquivo “routes.rb” para a nova classe criada, assim o Devise poderá enxergar a nova classe. Para fazer isso:

```
devise_for :users, class_name: 'OtherUserClass',
    controllers: {sessions: 'users/sessions',
                  registrations: 'users/registrations',
                  passwords: 'users/passwords'}
```

Código 6.11 – Configurando a nova classe de usuário para o Devise

Além disso, deve-se colocar o nome da nova classe de usuário também no arquivo “social_framework.rb” para que o *factory* das classes de modelo possa funcionar adequadamente. Nesse caso, deve-se alterar a linha com o nome da classe de usuário deixando-a:

```
config.user_class = 'OtherUserClass'
```

Código 6.12 – Configurando a nova classe de usuário para o *framework*

Maiores detalhes sobre esse aspecto podem ser encontrados na seção Hotspots.

6.2.3 Configuração das Migrations

A classe de usuário fornece como padrão os atributos nome de usuário, *e-mail* e senha. Para adicionar ou remover atributos para esta ou qualquer outra classe, pode-se adicionar a *migrate* na aplicação. O SocialFramework dispõe de um gerador, sendo, neste caso, necessário executar:

```
$ rails generate social_framework:install_migrations –m users
```

Código 6.13 – Geração das *migrations*

A *migrate* de usuário será adicionada na aplicação e poderá ser alterada. O parâmetro '-m' é usado para gerar *migrations* específicas. O *script* espera os nomes das *migrations*. Caso não exista a *migration* na aplicação, esta será gerada.

Caso seja adicionado ou removido um atributo para o usuário, deve-se notificar o Devise, usando um filtro *before* na "ApplicationController":

```
class ApplicationController < ActionController::Base
  before_action :configure_permitted_parameters, if: :devise_controller?

  protected

  def configure_permitted_parameters
    devise_parameter_sanitizer.for(:sign_up) << :new_attribute
    devise_parameter_sanitizer.for(:account_update) << :new_attribute
  end
end
```

Código 6.14 – Configuração para adicionar um novo atributo de usuário

Isto faz com que o *sign_up* e o *account_update* receba o novo atributo. Pode-se usar o método de remoção para remover os atributos existentes. Para adicionar ou remover o parâmetro em outras ações, por exemplo o *sign_in*, deve-se fazer o mesmo procedimento.

6.2.4 Configuração das Controllers

Todas as *controllers* do Devise podem ser estendidas e ter o seus métodos sobrescritos. Por exemplo:

```
class OtherRegistrationControllerClass < Users::RegistrationsController
  # Your code goes here
end
```

Código 6.15 – Estensão de uma controller

Outras *controllers* são: "confirmations", "omniauth_callbacks", "passwords", "sessions" e "unlocks". Para usar os módulos de "omniauth_callbacks", "unlocks" e "confirmations", é necessário adicionar esses respectivos módulos do Devise na aplicação. Todas as *controllers* do Devise no SocialFramework possuem o prefixo "Users".

6.2.5 Configuração das Rotas

Quando se substitui algumas *controllers* do Devise deve-se definir essas novas *controllers* nas rotas. Isto pode ser feito alterando o caminho para a *controller* na configuração *devise_for*, como no exemplo a seguir:

```
devise_for :users, class_name: 'SocialFramework::User',
  controllers: {sessions: 'users/sessions',
                registrations: 'new_registration_controller_path',
                passwords: 'users/passwords'}
```

Código 6.16 – Configuração de uma *controller*

A *controller* de registro foi estendida, e a nova *controller* foi adicionada nas configurações de rotas do Devise.

6.2.6 Configuração das Views

Para adicionar as *views* do Devise na aplicação, o SocialFramework possui um gerador para executar esta tarefa. Para executá-lo, deve-se usar o seguinte comando:

```
$ rails generate social_framework:views
```

Código 6.17 – Geração das *views*

Para adicionar uma *view* específica, deve-se usar o parâmetro '-v', e passar o nome das *views*, como no exemplo a seguir:

```
$ rails generate social_framework:views -v registrations sessions passwords
```

Código 6.18 – Geração das *views*

Este comando irá adicionar as *views* "registrations", "sessions" e "passwords" na aplicação. Outras *views* são: "confirmations", "mailer" e "unlocks". Inicialmente, o SocialFramework adiciona as *views* "registrations" e "sessions" na aplicação, fornecendo autenticação e registro para os usuários.

6.3 Módulos do SocialFramework

Atualmente, o SocialFramework possui os módulos de usuário, de rotas e de agendas. Estes módulos serão detalhados a seguir.

6.3.1 Módulo de Usuários

Este módulo fornece a lógica principal às redes sociais, como criar, confirmar e remover relacionamentos entre os usuários, além de buscas, registros, atualizações e autenticações.

A estrutura de relacionamentos foi construída com um relacionamento de muitos para muitos entre usuários, através de arestas. As arestas possuem: o usuário de origem; o usuário de destino; o *status*, o qual pode estar ativo ou inativo; se é bidirecional ou não, o que especifica se um relacionamento é bidirecional ou unidirecional, e um rótulo

com o nome do relacionamento. Podem existir múltiplos relacionamentos entre os mesmos usuários. Cada relacionamento é representado com uma aresta.

Para criar um novo relacionamento entre dois usuários, deve-se utilizar o método 'create_relationship', definido na classe de usuário. Esse método possui a seguinte assinatura:

```
create_relationship(destiny, label, bidirectional=true, active=false)
```

Código 6.19 – Método para criar relacionamento

O usuário que pede o relacionamento deve chamar este método e passar o usuário de destino e rótulo para o relacionamento. Por padrão, é criado um relacionamento bidirecional e inativo entre esses usuários. É possível mudar esse comportamento passando outros parâmetros para o método.

Para ativar o relacionamento criado, é necessária uma confirmação. Assim, o usuário de destino deve invocar o método 'confirm_relationship'. Esse método possui a seguinte assinatura:

```
confirm_relationship(user, label)
```

Código 6.20 – Método para confirmar relacionamento

É necessário passar como parâmetro o usuário de origem e o rótulo com o tipo de relacionamento. O único usuário que pode confirmar o relacionamento é o usuário de destino.

Para remover algum relacionamento, os usuários devem chamar o método 'remove_relationship', passando outro usuário do relacionamento e o rótulo do relacionamento. Esse método possui a seguinte assinatura:

```
remove_relationship(destiny, label)
```

Código 6.21 – Método para remover relacionamento

A classe de usuário fornece também um método para obter os usuários relacionados a um outro usuário específico. Para isso, deve-se chamar o seguinte método:

```
relationships(label, status = true, created_by = "any")
```

Código 6.22 – Método para recuperar usuários com um tipo de relacionamento

O rótulo representa o tipo de relacionamento a partir do qual se deseja obter os usuários. O *status* é usado para os usuários com relacionamentos ativos ou inativos. O padrão é *true*. O parâmetro '*created_by*' é usado para especificar qual usuário pode ter criado o relacionamento; caso seja "any", qualquer usuário poderá ter criado o relacionamento; caso seja "self", os relacionamentos devem possuir como usuário de origem o próprio usuário ou, caso seja "other", os relacionamentos, devem possuir, como usuário de destino, o próprio usuário.

O módulo de usuários usa um grafo para fornecer algumas funcionalidades, como buscas na rede e sugestões de relacionamentos. Todas estas funcionalidades podem ser acessadas através da classe *GraphContext*, que é usada para implementação do Padrão Strategy. Informações mais detalhadas sobre essa implementação podem ser encontradas na seção Hotspots deste capítulo.

O grafo pode ser acessado a partir do método 'graph', que está presente na classe de usuário. No momento de *login*, o grafo é construído até a profundidade especificada no *initializer* 'social_framework.rb', na variável 'depth_to_build' com o usuário logado como raiz (*root*). O valor de profundidade padrão definido é 3, e pode ser alterado conforme as necessidades do usuário do *framework*.

O método 'graph' da classe de usuário possui a seguinte assinatura:

```
graph(graph_strategy = NetworkHelper::GraphStrategyDefault,
    elements_factory = ElementsFactoryDefault)
```

Código 6.23 – Método acessar o grafo de usuários

Os parâmetros *graph_strategy* e *elements_factory* são usados quando não se está usando a estratégia padrão do grafo ou a fábrica padrão de vértices e arestas. Mais uma vez, a seção Hotspots detalha esse aspecto.

O método *build* é o responsável pela construção do grafo, e possui a seguinte assinatura:

```
build(root, attributes = SocialFramework.attributes_to_build_graph,
    relationships = "all")
```

Código 6.24 – Método para construir o grafo

O parâmetro "root" é o usuário que irá ser a raiz do grafo. O parâmetro "attributes" corresponde aos atributos de usuário e evento que serão mapeados para os vértices. Por padrão, contém o nome de usuário, o email e o título do evento. O valor padrão desse atributo pode ser alterado no *initializer* 'social_framework.rb'. O parâmetro "relationships" representa os tipos de relacionamentos que serão considerados para a construção do grafo, devendo ser uma *string* para especificar um único relacionamento, ou um *array* para especificar mais de um relacionamento. Caso seja passada a *string* "all", o grafo é construído considerando todos os relacionamentos. Na ação de *login*, o grafo é construído com os atributos padrões definidos, além do 'id'.

Com o grafo construído, é possível sugerir relacionamentos. Para isso, utiliza-se o algoritmo DFS para analisar o terceiro nível do grafo e encontrar relacionamentos comuns, com o tipo especificado no *initialize* 'social_framework.rb', na variável 'relationship_type_to_suggest', que possui o seu valor padrão como "friend". A variável 'amount_relationship_to_suggest' especifica a quantidade de relacionamentos comuns

que devem existir entre dois usuários para que estes recebam a sugestão de amizade. Nesse caso, o valor padrão é cinco. Esse método possui a seguinte assinatura.

```
suggest_relationships(type_relationships = SocialFramework.
    relationship_type_to_suggest,
  amount_relationships = SocialFramework.amount_relationship_to_suggest)
```

Código 6.25 – Método para sugerir relacionamento

Considerando os valores padrão, se um usuário 'A' e um usuário 'C' tem cinco relacionamentos com o tipo "friend" com outros cinco usuários, o sistema irá sugerir ao usuário 'A' começar o mesmo relacionamento com o usuário 'C', e vice-versa.

A busca no grafo é executada usando o algoritmo BFS. O método da busca possui a seguinte assinatura.

```
search(map, search_in_progress = false, elements_number = SocialFramework.
    elements_number_to_search)
```

Código 6.26 – Método para fazer pesquisa

É passado um mapa para ser usado na pesquisa. Este mapa representa as chaves e os valores para comparar com os vértices, por exemplo, o mapa "{username: 'user', email: 'user', title: 'event'}" fará com que uma pesquisa retorne qualquer vértice que contém a *string* 'user' no nome de usuário ou no *e-mail*, ou que contenha a *string* "event" no título. O parâmetro 'search_in_progress' é usado para continuar uma busca para encontrar mais resultados. Nesse caso, deve-se passar *true*. O parâmetro 'elements_number' define a quantidade de resultados para retornar, sendo este valor especificado no *initializer* 'social_framework.rb', na variável 'elements_number_to_search'. O valor padrão é cinco.

Um exemplo para continuar uma pesquisa é mostrado no Código 6.27. Por padrão, quando se continua a procurar, o parâmetro 'elements_number' é usado para incrementar o número máximo de resultados que devem ser retornados. Neste caso, a primeira chamada retorna os primeiros cinco elementos encontrados no grafo; a segunda chamada retorna mais cinco elementos, totalizando no final dez elementos, e assim por diante. O retorno do método é uma *hash* com duas chaves; a primeira é "users" e seu valor é um *array* com os usuários encontrados, e a segunda é "events" e seu valor é um *array* com os eventos encontrados.

```
map = {username: 'user', email: 'user', title: 'event'}
graph.search(map)
graph.search(map, true, 10)
```

Código 6.27 – Exemplo de continuação de pesquisa

Pode-se alterar o comportamento padrão para continuar pesquisas, passando um bloco para o método. Este bloco irá indicar como o "elements_number" deve ser incrementado. Por exemplo:

```
map = {username: 'user', email: 'user', title: 'event'}
graph.search(map, false, 1)
graph.search(map, true) { |elements_number| elements_number *= 2 }
```

Código 6.28 – Bloco para aumentar o valor do "elements_number"

Nesse caso, o 'elements_number' vai dobrar a cada método de pesquisa chamado com esse bloco. Portanto, a primeira chamada retorna um elemento, devido ao valor passado; o segundo dois; a terceira quatro, e assim por diante.

Quando a pesquisa chega ao fim do grafo e ainda não encontrou todos os elementos necessários, uma pesquisa no banco de dados é realizada para completar a pesquisa.

6.3.2 Módulo de Agendas

Este módulo fornece a lógica para trabalhar com a agenda dos usuários de redes sociais, como criar, inserir, convidar, confirmar, sair e remover eventos.

A estrutura de relacionamentos foi construída do seguinte modo: uma agenda pertence a um usuário e tem um relacionamento de muitos para muitos com o evento através de um "Participant_Event"; o "Participant_Event" tem um atributo confirmado que indica se o usuário confirmou o evento ou se a confirmação está pendente. Além de um evento ser capaz de estar presente em várias agendas, um evento tem um título, uma descrição, uma data e hora de início, uma data e hora de término, e pode ou não ter uma rota associada. Caso o evento seja particular, outro usuário só poderá participar do evento se for convidado. Caso contrário, ou seja, não sendo particular, qualquer usuário pode entrar no evento.

A classe "Participant_Event" possui também o atributo 'role' que representa qual o papel que o usuário terá dentro de um evento. Os papeis definidos por padrão no SocialFramework são: 'creator', 'admin', 'inviter' e 'participant'. No *initialize* 'social_framework.rb', estão definidas todas as permissões para cada tipo desses papéis. Tais permissões dizem respeito a quais ações um usuário pode executar dentro de um evento, como remover ou convidar usuários, por exemplo.

Para criar um novo evento, utiliza-se o método 'create_event', definido na classe de agenda. Pode-se acessar a agenda a partir de um usuário pelo método 'schedule'. A assinatura no método 'create_event' é mostrada a seguir.

```
create_event(title, start, duration = nil, description = "", particular =
    false)
```

Código 6.29 – Método para criar evento

Caso a duração do evento não seja passada, o término do evento será marcado para o fim do dia. Para o evento ser criado, o usuário não poderá ter eventos em sua agenda que coincidam com o horário do evento que se deseja criar.

Para um usuário que queira entrar em um evento público sem um convite, deve-se usar o método 'enter_in_event', definido na classe de agenda. Este método possui a seguinte assinatura:

```
enter_in_event(event)
```

Código 6.30 – Método para entrar em um evento

É necessário passar como parâmetro o evento que o usuário deseja entrar.

A confirmação de um convite para um evento dá-se por meio do método 'confirm_event', definido na classe de agenda. Este método possui a seguinte assinatura:

```
confirm_event(event)
```

Código 6.31 – Método para confirmar um evento

É necessário passar como parâmetro o evento que o usuário deseja confirmar.

Para sair de um evento, deve-se invocar o método 'exit_event', definido na classe de agenda. Esse método possui a seguinte assinatura:

```
exit_event(event)
```

Código 6.32 – Método para sair de um evento

É necessário passar como parâmetro o evento que o usuário deseja remover da sua agenda. Mas para conseguir sair do evento, o usuário não pode ter o papel de 'creator' do evento. Caso esse usuário seja o criador do evento, este deve passar o papel para outro usuário, antes de sair do evento.

Para apagar um evento, é necessário invocar o método 'remove_event', definido na classe de agenda. Esse método possui a seguinte assinatura:

```
remove_event(event)
```

Código 6.33 – Método para remover um evento

É necessário passar como parâmetro o evento que se deseja apagar. Somente o usuário criador de um evento tem a permissão para remover o mesmo.

Para verificar se um usuário possui eventos em um intervalo de tempo definido, pode-se usar o método 'events_in_period', definido na classe de agenda. Esse método retorna todos os eventos que estão dentro desse intervalo de tempo e possui a seguinte assinatura:

```
events_in_period(start, finish = start.end_of_day)
```

Código 6.34 – Método para recuperar os eventos dentro de um período de tempo

O valor padrão para o parâmetro 'finish' é o fim do dia da data de 'start'.

Para convidar um usuário para um evento, deve-se invocar o método 'invite', definido na classe evento. Esse método possui a seguinte assinatura:

```
invite(inviting, guest, relationship = SocialFramework.
    relationship_type_to_invite)
```

Código 6.35 – Método para convidar um usuário para um evento

É necessário passar como parâmetro o usuário que está convidando e o usuário convidado. Caso o tipo de relacionamento que os usuários devem possuir não seja passado, será utilizado o tipo definido no arquivo de configuração. Para convidar um usuário para um evento, o usuário que convida deve estar confirmado no evento, ter a permissão de 'invite', e ter o convidado em seu círculo de relacionamentos com o tipo de relacionamento especificado. Deve-se utilizar o tipo 'all' para qualquer tipo de relacionamento.

Para alterar o papel de um participante em um evento, deve-se invocar o método 'change_participant_role', definido na classe evento. Esse método possui a seguinte assinatura:

```
change_participant_role(maker, participant, action)
```

Código 6.36 – Método para mudar o papel de um usuário

É necessário passar como parâmetro o usuário que irá mudar o papel, o usuário que terá o seu papel alterado e a ação que será executada. Esta ação deve ter como prefixo 'maker' ou 'remove', que indica se será atribuído ou removido um papel, seguido por um '_' e o papel da ação, por exemplo: ':make_admin'. Para que a mudança de papel seja realizada com sucesso, o usuário que irá mudar o papel deve possuir a permissão de executar a ação. E caso a ação seja ':make_creator', o usuário que irá mudar o papel receberá o papel de 'admin', pois só deve existir um usuário criador do evento.

Para remover um participante de um evento, deve-se invocar o método 'remove_participant', definido na classe evento. Esse método possui a seguinte assinatura:

```
remove_participant(remover, participant)
```

Código 6.37 – Método para mudar o papel de um usuário

É necessário passar como parâmetro o usuário que irá remover e o usuário que será removido do evento.

Para adicionar uma rota no evento deve-se chamar o método 'add_route', definido na classe evento. Esse método possui a seguinte assinatura:

```
add_route(user, route)
```

Código 6.38 – Método para mudar o papel de um usuário

É necessário passar como parâmetro o usuário que irá adicionar a rota no evento e a rota a ser adicionada.

Para ver todos os usuários do evento, que confirmaram a participação, ou não, basta chamar o método 'users' da classe evento, com a seguinte assinatura:

```
users(confirmed = true, role = "all")
```

Código 6.39 – Método para mudar o papel de um usuário

O parâmetro 'confirmed' é responsável por determinar se o retorno deve ser dos usuários que confirmaram participação ou não no evento. O parâmetro 'role' define o tipo do papel dos usuários no evento, sendo esse valor 'all' para todos os papéis.

O módulo de agenda possui a funcionalidade de verificar a disponibilidade de um determinado grupo de usuários para marcação de um evento. Nesse problema, têm-se os *slots* de horários e os usuários como elementos chave. Esses elementos devem ser relacionados entre si para se atingir o objetivo esperado, sendo que um usuário só pode se relacionar a um *slot* e vice-versa. Portanto, esse arranjo pode ser entendido como um grafo bipartido. A seguir é apresentado um exemplo de um problema comum em que a sua solução faz uso de um grafo bipartido.

Neste problema, tem-se um conjunto de n funcionários e m posições em uma empresa. Cada funcionário está qualificado para uma ou mais posições. Deve-se atribuir uma posição para cada funcionário, de modo que cada funcionário ocupe exatamente uma posição. Este problema pode apresentar como dados, adicionalmente, uma função que atribui a cada funcionário um valor numérico, correspondente à sua eficiência para ocupar determinada posição na empresa. Com esta restrição, deve-se encontrar uma alocação que maximize a eficiência total dos funcionários (FIGUEIREDO; SZWARCFITER, 1999).

No problema apresentado, que é um típico problema de atribuição, deve-se alocar cada funcionário para uma determinada posição. Porém, no problema de compatibilidade de horários, deve-se verificar a disponibilidade de um usuário para fazer a sua ligação com um determinado *slot*. A principal diferença é que, em um *slot*, pode-se alocar múltiplos usuários.

O problema de atribuição dos funcionários pode possuir também, um atributo que indica o nível de eficiência de cada funcionário para uma posição específica, o que ocorre também no problema do melhor horário, onde cada usuário pode ter um peso de importância para a marcação de um determinado evento. Outra diferença das duas soluções é que, no problema de atribuição de funcionários, deve-se encontrar uma combinação ótima para maximizar a eficiência dos funcionários. Desse modo, deve-se selecionar as arestas que possibilitem esse resultado (FIGUEIREDO; SZWARCFITER, 1999). No problema de marcação de horários deve se encontrar o *slot* com o maior peso. Sendo assim, deve-se obter o vértice que possui a maior somatório dos pesos de suas arestas.

Dessa forma, o grafo de *slots* e usuários ficou com a seguinte configuração: existem vértices de *slots* e usuários que possuem relação entre si através de arestas, e cada uma dessas arestas, pode possuir um peso associado. O algoritmo constrói todas as arestas do grafo analisando as disponibilidades de horário dos usuários no intervalo de tempo de cada *slot*. Caso o usuário possua disponibilidade, a aresta é criada relacionando os usuários com o *slot*. Ao final, pode-se obter o maior peso de um *slot* analisando suas arestas.

Na Figura 26, pode-se observar um exemplo de como o grafo bipartido é montado. Neste exemplo, serão analisados os horários de 08:00 às 13:00 horas, e o tamanho do *slot* é de uma hora. Foram considerados três usuários, o primeiro não possui horário disponível em sua agenda entre 11:00 e 13:00 horas, e sua participação no evento tem o peso igual a um. O segundo usuário possui uma disponibilidade de 09:00 às 11:00 horas, e o seu peso é igual a dois. O usuário três possui disponibilidade em sua agende de 08:00 às 09:00 horas e de 11:00 às 13:00 horas, a sua participação no evento possui um peso igual a três. Dese modo, é possível verificar que o *slot* que possui uma maior quantidade de pesos é o *slot* de 08:00 às 09:00 horas, totalizando um peso igual a quatro.

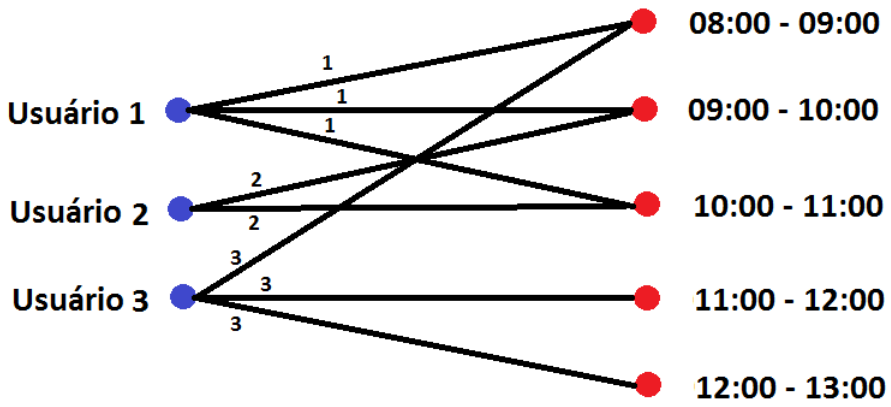


Figura 26 – Exemplo de um grafo bipartido para o problema de melhor horário

O peso máximo assumido para os usuários é definido na variável ‘max_weight_schedule’ no *initializer* ‘social_framework.rb’. Existe também o valor ‘:fixed’ para peso dos usuários. Esse valor representa que é obrigatório que o usuário esteja presente no evento que se deseja marcar.

A verificação do melhor horário para a marcação de eventos foi implementada através do método ‘verify_availabilities’ da classe ‘ScheduleContext’, com a seguinte assinatura:

```
verify_availabilities(users, start_day, finish_day, start_hour = Time.parse
    ("00:00"), finish_hour = Time.parse("23:59"), slots_size =
    SocialFramework.slots_size)
```

Código 6.40 – Método para mudar o papel de um usuário

O parâmetro 'users' representa os usuários que terão suas agendas verificadas, no intuito de encontrar o melhor horário para uma possível marcação de evento. Este parâmetro pode ser um *array*, onde nenhum usuário possui peso no evento ou pode ser uma *hash*, onde a chave é o usuário e o valor é o peso de sua presença no evento.

O parâmetro 'start_day' indica a partir de qual dia o evento poderá ser marcado, e o parâmetro 'finish_day' indica o dia limite para a marcação do evento. O parâmetro 'start_hour' e o parâmetro 'finish_hour' definem o horário de um dia em que o evento poderá ocorrer. Por exemplo pode-se definir o horário comercial de 08:00 às 18:00. Caso estes valores não sejam passados como parâmetros, será considerado que o evento poderá ser marcado em qualquer horário.

O parâmetro 'slots_size' indica o tempo de duração dos *slots*. Caso não seja passado como parâmetro, é utilizado o tempo padrão, definido no arquivo configuração e com duração de 1 hora.

A quantidade de *slots* que serão adicionados no grafo depende do intervalo de tempo em que se deseja analisar e do tempo de duração de cada *slot*.

Para criar as arestas entre os usuários e os *slots*, é verificado se o usuário possui disponibilidade em sua agenda no intervalo de tempo que o *slot* possui. Caso o usuário possua disponibilidade para esse *slot*, é criada uma aresta entre o usuário e o *slot*, e o peso do usuário é adicionado ao peso do *slot*. O método retorna os *slots* ordenados decrescentemente pelo peso.

6.3.3 Módulo de Rotas

Este módulo fornece a lógica para trabalhar com rotas. É utilizada a API do Google Maps[3] para fornecer alguns recursos para este módulo, como construir uma rota e obter uma localização a partir de uma latitude e uma longitude.

Para um usuário criar uma rota, deve-se utilizar o método 'create_route', presente na classe de usuário. A seguir, pode-se visualizar a sua assinatura:

```
create_route(title, distance, locations, mode_of_travel = "driving",
    accepted_deviation = 0)
```

Código 6.41 – Método para criar uma rota

Os parâmetros são o título da rota, a distância que a rota possui e as localizações, sendo esse último parâmetro um *array* de *hashs* com todos os pontos de latitude e longitude para construir rota, a seguir é ilustrado um exemplo do parâmetro *locations*:

```
locations = [{latitude: -15.792740000000, longitude: -47.876360000000},
```

[3] <https://developers.google.com/maps/?hl=pt-br>

```
{latitude: -15.792520000000, longitude: -47.876900000000}]
```

Código 6.42 – Exemplo de localizações

O parâmetro ‘mode_of_travel’ representa qual o meio de transporte que será utilizado para percorrer a rota, podendo ser ‘driving’, ‘bycicling’, ‘walking’ ou ‘transit’, o valor padrão é ‘walking’ para viajar de carro. Além disso, existe também o parâmetro ‘accepted_deviation’ com valor padrão igual a zero. Esse parâmetro diz respeito a quanto uma rota pode ter a sua distância alterada, é usado na comparação de rotas.

Este módulo também fornece um recurso para verificar a compatibilidade entre duas rotas. Para isso, deve-se chamar o método ‘compare_routes’ presente na classe ‘RouteContext’.

```
compare_routes(principal_route, secondary_route)
```

Código 6.43 – Método para comparar rotas

Este método é utilizado para verificar se duas rotas podem ser conectadas em apenas uma rota. O parâmetro ‘principal_route’ representa a rota que será alterada para auxiliar a rota secundária representada pelo parâmetro ‘secondary_route’, se possível. Quando a rota principal não pode ser alterada, pode-se tentar alterar a rota secundária para atingir o objetivo. O método utiliza os valores de desvio máximo das rotas para poder verificar se as rotas podem ser desviadas para atender os objetivos esperados.

O método *compare_routes* retorna um mapa com as informações de compatibilidade, como pode ser visualizado a seguir:

```
{compatible: false, principal_route: {deviation: :none,
        distance: 0}, secondary_route: {deviation: :none, distance: 0}}
```

Código 6.44 – Retorno do método que compara as rotas

A chave ‘compatible’ é falsa quando as rotas são incompatíveis ou verdadeira quando são compatíveis. O desvio representa onde a rota poderá ser desviada, podendo ser ‘none’, ‘both’, ‘origin’ ou ‘destiny’. No caso, ‘distance’ é o valor da distância total que a rota possui com o desvio incluso.

As rotas também podem ser relacionadas aos eventos. Para fazer o relacionamento, deve-se invocar o método ‘add_route’, presente na classe de evento.

```
add_route(user, route)
```

Código 6.45 – Retorno do método para adicionar uma rota em um evento

O primeiro parâmetro é o usuário que tentará adicionar a rota no evento, sendo necessário que o usuário tenha a permissão ‘add_route’ no evento (as permissões para executar as ações nos eventos estão definidas no *initialize* ‘social_framework.rb’). O se-

gundo parâmetro é a rota que será adicionada no evento. Quando uma rota é adicionada a um evento, todos os usuários neste evento recebem essa rota.

6.4 Hotspots

O SocialFramework possui três pontos de *Hotspots* principais. Nesses pontos foram implementados três padrões de projeto para auxiliar na estruturação dos elementos do *framework*. Além dos pontos mencionados, o *framework* é extensível, e permite ao usuário mudar qualquer comportamento padrão implementado através de heranças e sobrescrita de métodos.

O primeiro Hotspot definido para o *framework* está presente nas classes de modelo. Como mencionada nas sessões Configuração das Modelos e Configuração das Migrations, é mostrada a possibilidade de se estender uma classe de modelo ou mudar uma *migration* adicionando ou removendo novos atributos para a classe. Porém, existe um problema, quando se estende uma classe de modelo para sobrescrever algum método padrão, essas alterações não serão refletidas para todos os pontos do *framework* onde aquele método é usado. Foi utilizado uma adaptação do Padrão Factory Method para resolver este problema. Essa adaptação se deu pelo fato de não ser necessário a criação das classes abstratas, pois as classes que serão instanciadas pela fábrica já representam o comportamento padrão do *framework* e para a fábrica instanciá-las só é necessário nome da classe. Com isso, agora é possível, usando-se a classe *ModelFabric*, dizer qual classe de modelo deve ser construída para ser utilizada. Dessa forma, caso um usuário do *framework* estenda uma classe de modelo qualquer, sobrescrevendo um de seus métodos, este poderá simplesmente chamar o método estático *get_class*, passando o nome de sua nova classe. E essa classe passará a ser usada pelo *framework*. A seguir, é apresentado um modelo de uso desse método para facilitar o entendimento.

```
ModelFabric.get_class("NovaClasseModelo")
```

Código 6.46 – Método para construir nova classe de modelo

Existe outro facilitador desenvolvido no *framework* para trabalhar com o padrão mencionado. Os nomes de todas as classes de modelo estão definidos no *initializer* 'social_framework.rb', sendo assim, quando um desenvolvedor cria uma nova classe de modelo que estende de alguma *model* padrão, basta que mude o nome da respectiva classe nesse arquivo e tudo estará funcionando. Dessa forma, a classe *ModelFabric* se torna transparente para um usuário do *framework*, pois está sendo chamada nos métodos internos, e basta fazer a alteração de nome no arquivo 'social_framework.rb', não se esquecendo, que ao se tratar da classe de usuário, que o nome dessa também deve ser alterado no arquivo 'routes.rb' para o Devise saber com qual classe de usuário estará trabalhando.

A seguir na figura 27, pode ser visto o diagrama das classes construído para aplicação do padrão.

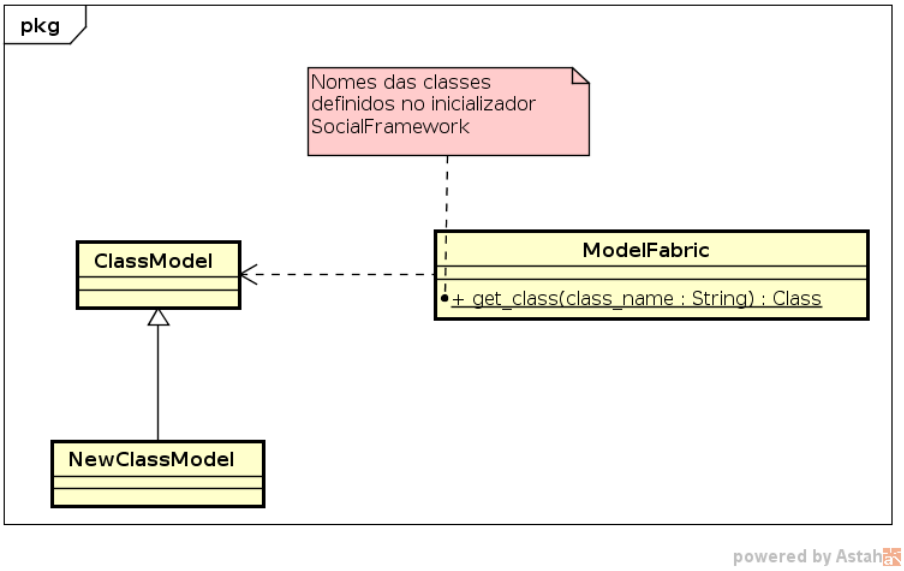


Figura 27 – Classes no SocialFramework para aplicação do Factory Method

Outro padrão de projeto de criação utilizado foi o Padrão Abstract Factory. Esse padrão foi utilizado para as classes que fazem parte da construção dos grafos, os quais foram utilizados na rede de usuários e na conciliação de agendas para marcação de um horário comum. As classes são *Vertex* e *Edge*. Para cada um delas, existe uma classe abstrata com as assinaturas dos métodos necessários. Também existe a classe abstrata *ElementsFactory* que possui os métodos para criação de *Vertex* e *Edge*.

Para cada classe citada anteriormente, foi desenvolvida uma classe concreta que implementa todos os métodos abstratos com o comportamento padrão esperado para eles. As classes são *VertexDefault*, *EdgeDefault* e *ElementsFactoryDefault* que constrói as duas classes anteriores.

Um desenvolvedor com o intuito de alterar qualquer uma das classes *Vertex* ou *Edge* deve estender essas classes e sobrescrever seus métodos conforme sua necessidade. Deve-se também estender a classe *ElementsFactory* e sobrescrever os métodos de criação, passando suas novas classes. Nas classes que fazem o uso dessas, deve ser passado nas instâncias o novo *ElementsFactory* criado e, assim, as novas classes desenvolvidas passarão a ser utilizadas.

O diagrama com as classes construídas no *framework* para aplicação do padrão *Abstract Factory* pode ser visualizado na Figura 28.

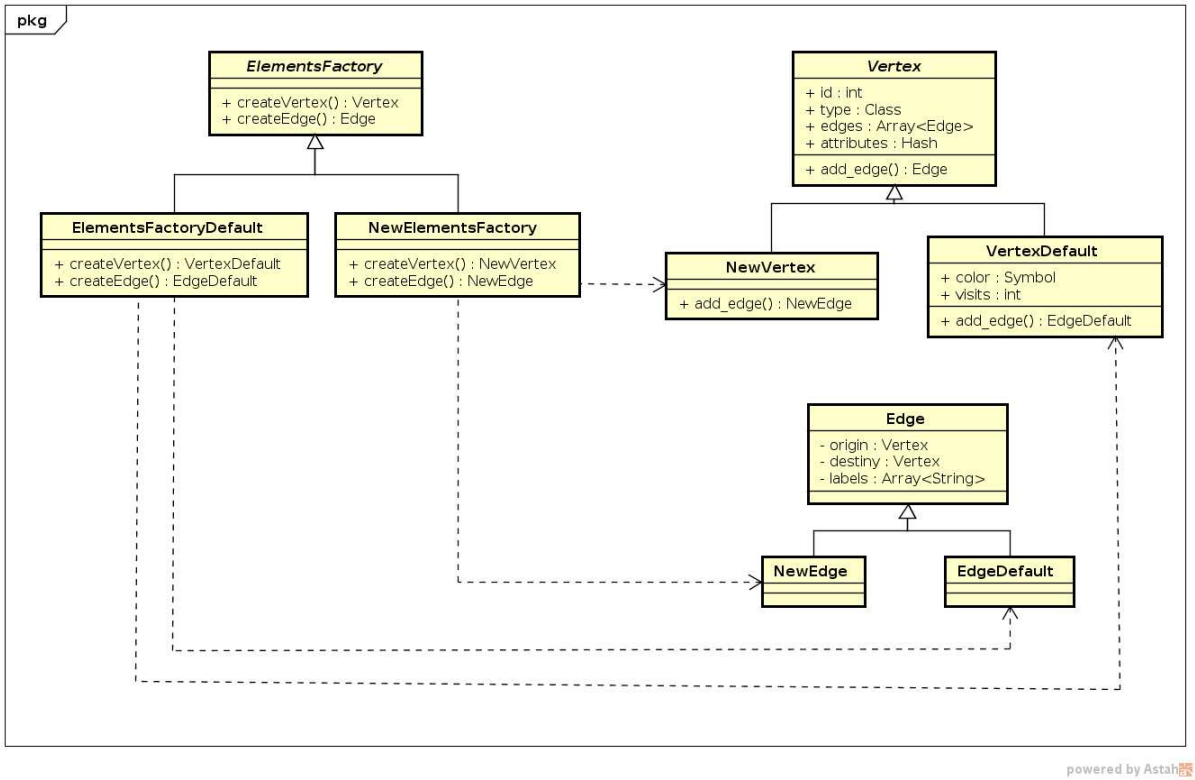


Figura 28 – Classes no SocialFramework para aplicação do Abstract Factory

No diagrama, é mostrado também um exemplo de como seria feita a adição de novas classes que mudem a implementação padrão desenvolvida no *framework*.

Por fim, foi utilizado também o Padrão Strategy para cada um dos três módulos do SocialFramework. Para cada um dos módulos, existe uma classe abstrata de estratégia que define todos os métodos públicos do módulo, e uma classe de contexto que será utilizada pelo usuário do *framework*. Essa classe de contexto recebe o tipo de estratégia que será utilizado e o *ElementsFactory* quando for alterado.

Por padrão, o SocialFramework já implementa uma classe de estratégia concreta para cada um dos módulos, e recebe essas classes no construtor de cada classe de contexto. Porém, é simples para que um desenvolvedor altere a estratégia padrão utilizada, basta que esse crie sua nova classe que estenda a classe abstrata de estratégia do módulo e sobrescreva seus métodos, conforme suas necessidades. Adicionalmente, na instanciação da classe de contexto do módulo, deve ser passada a sua nova estratégia, construída através do construtor. Com isso, toda a nova implementação passará a ser usada, sem alteração real alguma sendo feita pelo *framework*.

Os métodos públicos de cada módulo do SocialFramework, que devem ser implementados na mudança de uma estratégia, são descritos detalhadamente nas subseções dos Módulos do SocialFramework. As classes de contexto de cada módulo são: *GraphContext* para o módulo de usuário, *ScheduleContext* para o módulo de agenda, o e *RouteContext* para o módulo de rotas.

A seguir, é apresentada na Figura 29 como está a disposição de classes para os três módulos do SocialFramework.

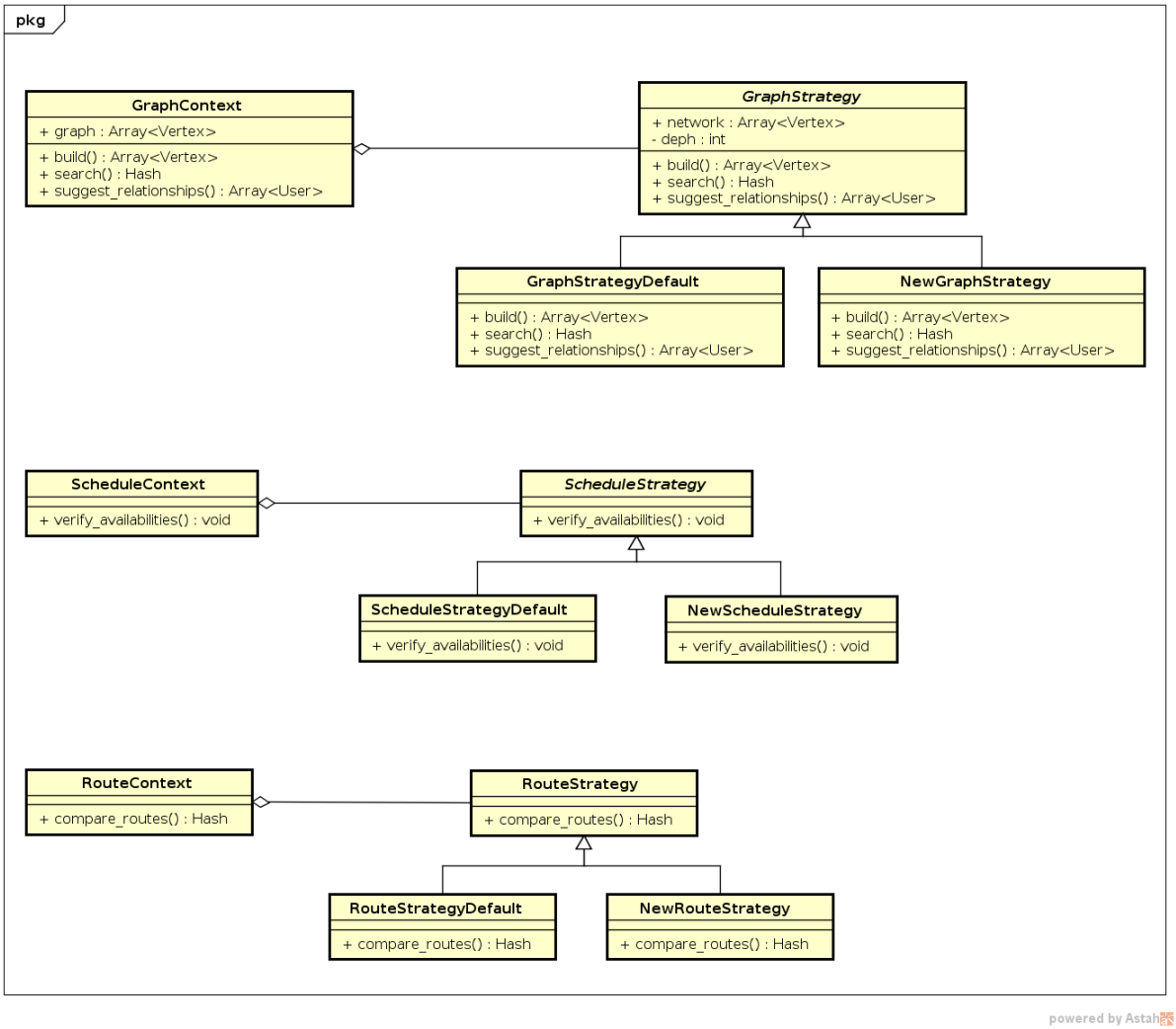


Figura 29 – Classes no SocialFramework para aplicação do Strategy

Em seguida, é apresentado um exemplo de uso quando se altera a estratégia de um módulo.

```
ScheduleContext.new(NovaEstrategiaDesenvolvida)
```

Código 6.47 – Instanciação do *ScheduleContext* quando se muda a estratégia

Com isso, ao se chamar o método *verify_availabilities* dessa classe, estará chamando de fato o método desenvolvido na classe 'NovaEstrategiaDesenvolvida'.

Um exemplo de instanciação, com alteração da estratégia e também da fábrica de vértices e arestas, pode ser visualizado abaixo.

```
ScheduleContext.new(NovaEstrategiaDesenvolvida, NovaFabricaDeElementos)
```

Código 6.48 – Instanciação do *ScheduleContext* quando foram mudadas a estratégia e a fábrica

Nesse caso, nenhuma das classes padrão do SocialFramework, desenvolvidas para serem usadas no módulo de agenda, estarão sendo usadas, e sim, as novas classes implementadas pelo usuário do *framework*.

6.5 Resumo do Capítulo

Este capítulo apresentou as funcionalidades desenvolvidas para o SocialFramework, passando por toda a sua interface pública e apresentando cada comportamento. Também foi possível conhecer mais a fundo cada módulo que foi proposto e como foram implementados, além dos *Hotspots* criados que trabalham como facilitadores para alterações que se tornem necessárias nas diversas instanciações possíveis do *framework*. A partir dos padrões de projeto implementados nos *Hotspots*, gerou-se uma maior capacidade de reutilização e manutenção do *framework*.

7 Resultados

Neste capítulo, serão apresentados os principais resultados obtidos referentes ao desenvolvimento do SocialFramework. Esses resultados foram obtidos através da análise da cobertura do código fonte, da complexidade dos algoritmos utilizados e da qualidade do código.

7.1 Testes Unitários

Durante o desenvolvimento do SocialFramework foi elaborada uma suíte de testes para tentar garantir que cada ponto do código funcione da devida maneira. Desse modo, o desenvolvedor pode inserir novos códigos, módulos ou alterar um código existente, com mais tranquilidade e sabendo que o restante do código irá funcionar adequadamente. Caso algum resultado esperado seja diferente do obtido, a suíte de testes irá apontar a falha.

Para a realização dos testes, foi utilizada a *gem* do Rspec[1]. Essa *gem* é amplamente utilizada pela comunidade do Rails. E para a visualização do relatório de testes, foi utilizado o Code Climate[2], que é uma ferramenta *online* que realiza uma análise estática do código fonte. Atualmente, são suportadas as linguagens: Ruby, Python, JavaScript e PHP.

No início do desenvolvimento, foi estabelecida uma meta de que se deveria testar no mínimo 90% do código fonte. Essa meta foi cumprida, obtendo-se 98,8% de cobertura do código. Porém essa métrica não corresponde à realidade, pois o arquivo "engine.rb", que teve sua cobertura de 47.83%, é um arquivo de configuração do *framework* e não deveria ser testado, e os arquivos "elements_factory.rb", "elements_factory_default.rb" e "graph_elements.rb" tiveram a cobertura, respectivamente, de 60%, 80% e 84%, diminuindo a cobertura real do código fonte. Entretanto, esses arquivos possuem classes "abstratas" e deveriam ser desconsiderados. Porém, em Ruby, não existe o conceito de classe abstrata. Olsen em seu livro (OLSEN, 2007), apresenta uma abordagem de se obter esse conceito. Um exemplo dessa abordagem pode ser observado, conforme consta no 7.1:

```
1 # Define abstract methods to Vertex
2 class Edge
3   attr_accessor :origin, :destiny, :labels
4
5   # Constructor to Edge
6   # ======= Params:
7   # +origin+:: +Vertex+ relationship origin
```

[1] <http://rspec.info/>
[2] <https://codeclimate.com/>

```
8   # +destiny+:: +Vertex+ relationship destiny
9   # Returns NotImplementedError
10  def initialize origin, destiny
11    raise 'Must implement method in subclass'
12  end
13 end
```

Código 7.1 – Classe abstrata

A seguir, são apresentados, na Figura 30 cada arquivo e sua respectiva cobertura.

All Files (98.83% covered at 31.85 hits/line)

52 files in total. 2832 relevant lines. 2799 lines covered and 33 lines missed

Search:

File	% covered
lib/social_framework/engine.rb	47.83 %
lib/social_framework/graphs/elements_factory.rb	60.0 %
lib/social_framework/graphs/elements_factory_default.rb	80.0 %
lib/social_framework/graphs/graph_elements.rb	85.0 %
spec/dummy/config/application.rb	90.0 %
app/models/social_framework/user.rb	95.95 %
lib/social_framework.rb	97.83 %
app/helpers/social_framework/network_helper.rb	97.85 %
app/helpers/social_framework/schedule_helper.rb	98.61 %
app/helpers/social_framework/route_helper.rb	99.28 %
spec/helpers/social_framework/route_helper_spec.rb	99.52 %
app/controllers/users/registrations_controller.rb	100.0 %
app/controllers/users/sessions_controller.rb	100.0 %
app/models/social_framework/edge.rb	100.0 %
app/models/social_framework/event.rb	100.0 %
app/models/social_framework/location.rb	100.0 %
app/models/social_framework/participant_event.rb	100.0 %
app/models/social_framework/route.rb	100.0 %
app/models/social_framework/schedule.rb	100.0 %
config/initializers/devise.rb	100.0 %
config/routes.rb	100.0 %
lib/generators/social_framework/install_generator.rb	100.0 %
lib/generators/social_framework/install_migrations_generator.rb	100.0 %
lib/generators/social_framework/views_generator.rb	100.0 %
lib/social_framework/fabrics/model_fabric.rb	100.0 %

Figura 30 – Cobertura de teste

7.2 Relatório de Desempenho

Após a verificação do correto fluxo que o *framework* executa, foram realizados testes de desempenho do mesmo, analisando o tempo de execução e o consumo de memória dos principais algoritmos desenvolvidos. Não foram realizados esses teste no módulo de rotas, pois os seus métodos fazem somente requisições de rotas à API do Google Maps e comparações sobre essas rotas.

Para se obter cada valor, foram feitas cinco medições, e foi realizada a média aritmética dos valores.

7.2.1 Módulo de Usuário

Para o módulo de usuário foram, realizados testes com 10 mil usuários, onde a quantidade de relacionamentos que os usuários possuíam em cada teste variou entre 100 relacionamentos, 200 relacionamentos, 300 relacionamentos, 400 relacionamentos e 500 relacionamentos. Para todos esses casos, cada usuário estava presente em 10 eventos.

7.2.1.1 Tempo de Execução

Para o método "build", que é responsável por construir o grafo a partir de um usuário raiz, foi especificado que o grafo deveria ser construído com três níveis de profundidade. Foi obtido o gráfico da Figura 31 como resultado.

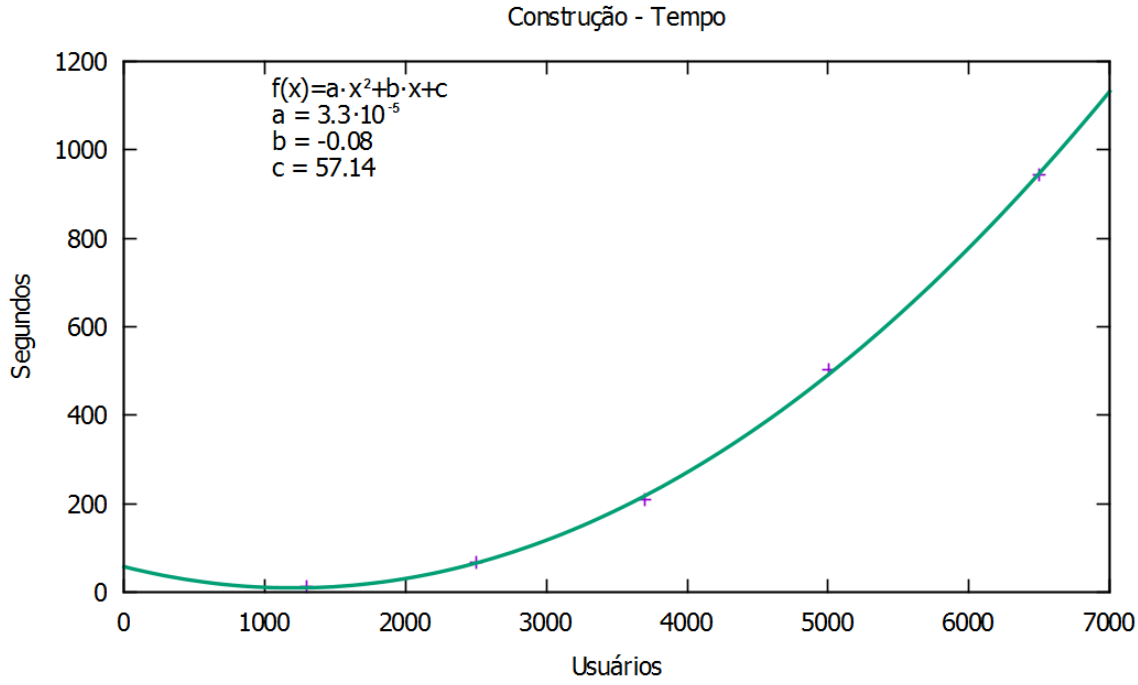


Figura 31 – Construção

Para o método "search", que é responsável por fazer pesquisas no grafo, foram feitos dois tipos de pesquisas. A pesquisa padrão é uma pesquisa na qual serão encontrados os

vértices até o segundo nível do grafo. O resultado desta pesquisa pode ser observado na Figura 32.

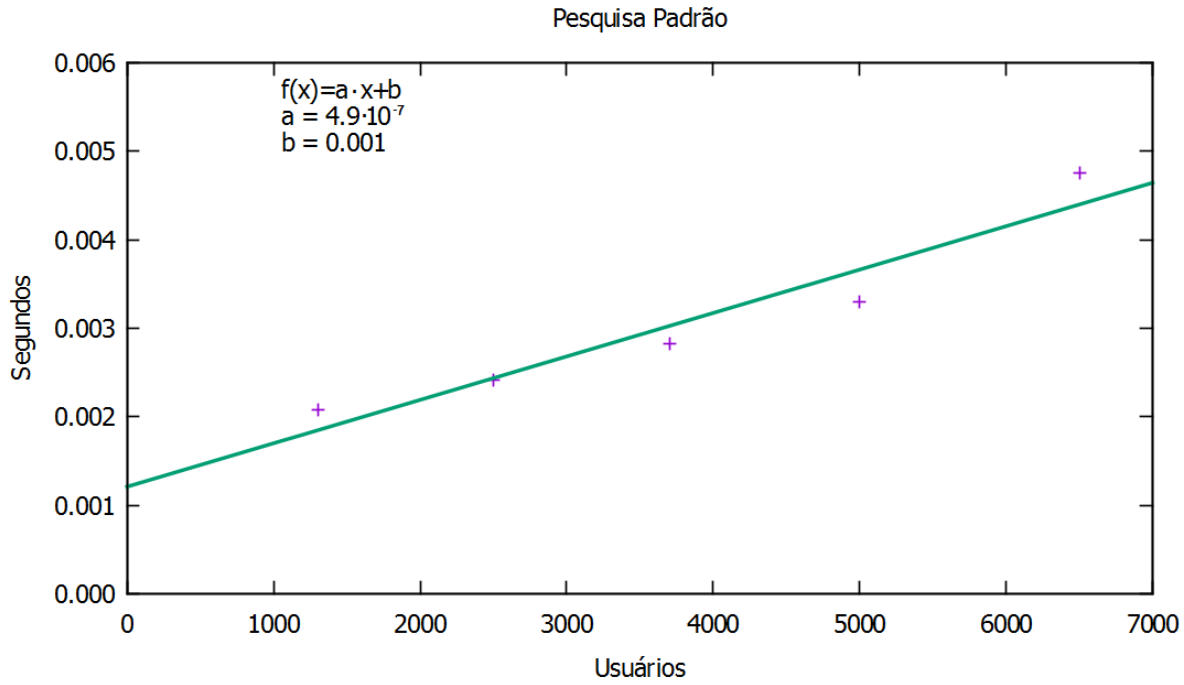


Figura 32 – Pesquisa padrão

A pesquisa no terceiro nível é uma pesquisa que irá percorrer até o último nível do grafo. Pode-se visualizar o resultado dessa pesquisa na Figura 33.

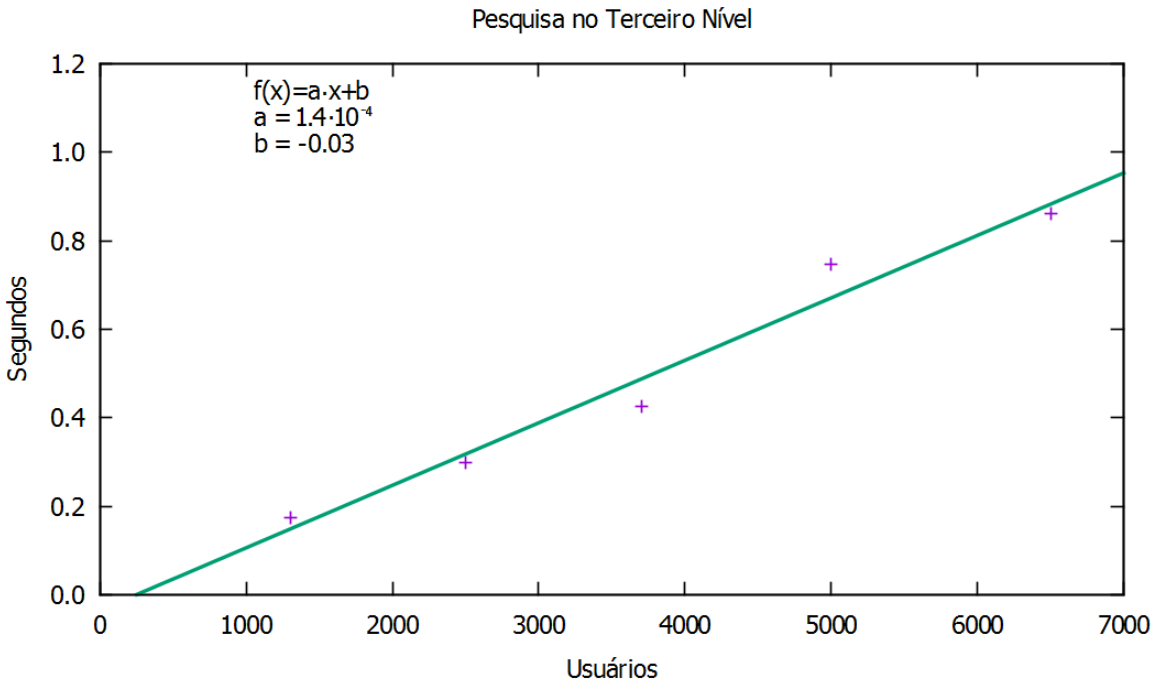


Figura 33 – Pesquisa no terceiro nível

Para o método "suggest_relationships", foi utilizado o parâmetro padrão para a quantidade de relacionamento para realizar a sugestão, que é cinco. Os tempos de execução para esse método podem ser visualizados na Figura 34.

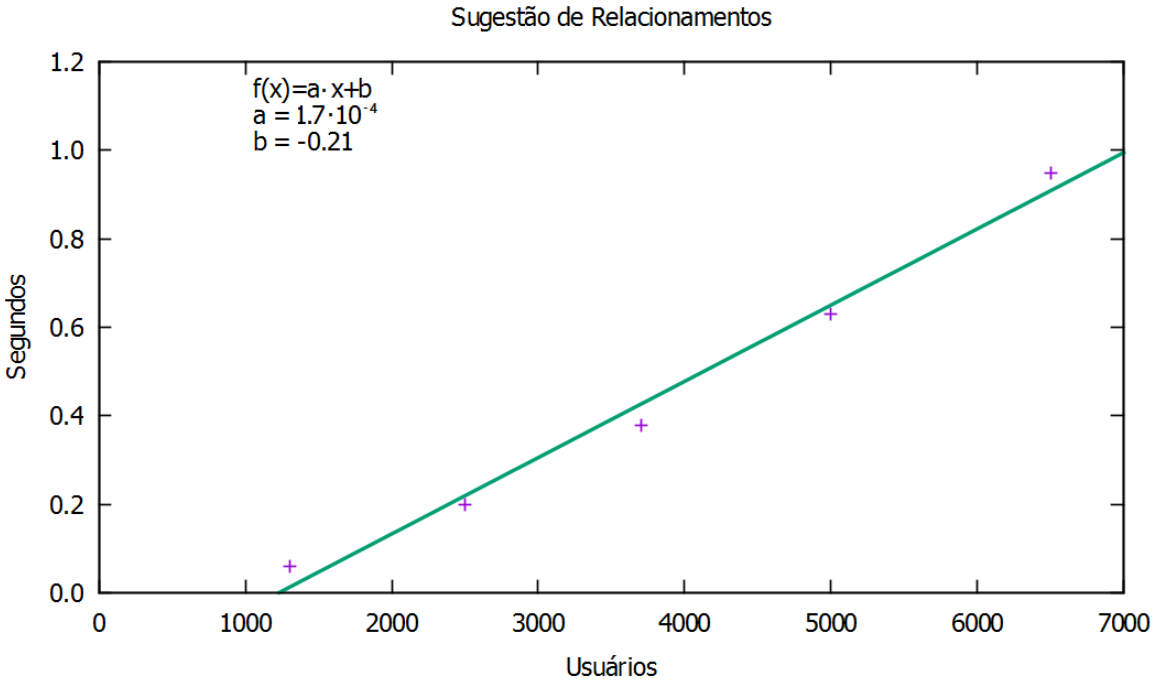


Figura 34 – Sugestão de relacionamentos

7.2.1.2 Consumo de Memória

O consumo de memória que o método "build" utilizou pode ser visualizado na Figura 35.

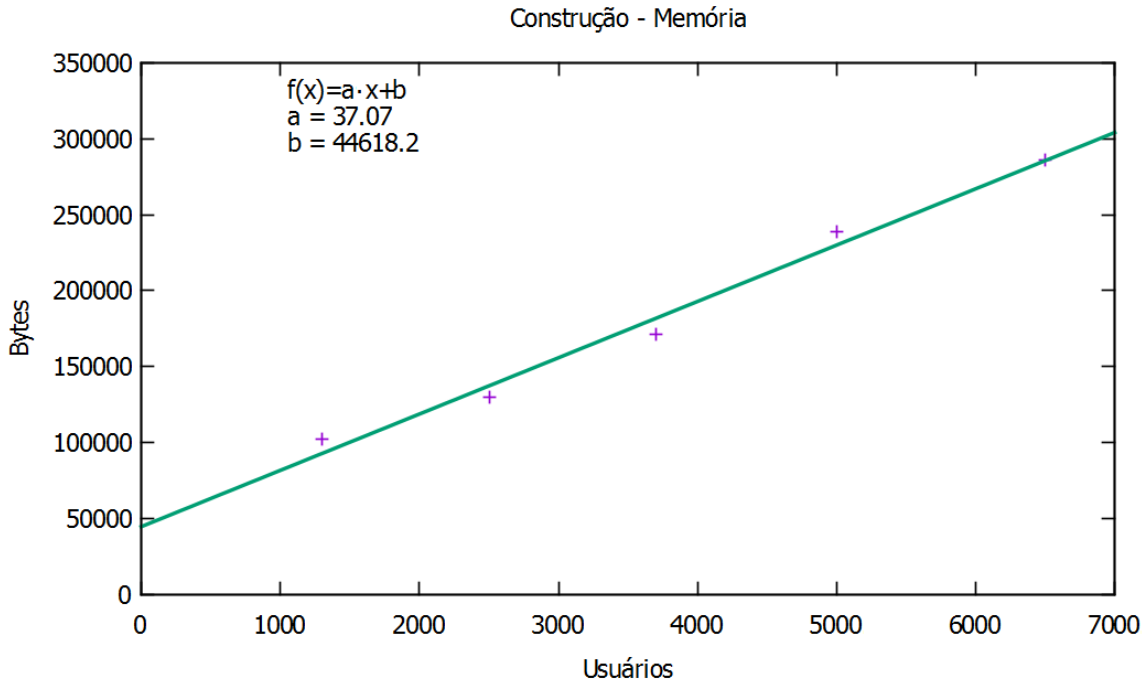


Figura 35 – Consumo de memória do método *build*

Os métodos "search" e "suggest_relationships", que tiveram seus tempos de execução apresentados na seção anterior não tiveram o seu consumo de memória medido, pois esses apenas utilizam a instância do grafo gerado pelo método "build".

7.2.2 Módulo de Agenda

No módulo de agenda, o método que teve o seu desempenho medido foi o "verify_availabilities", o qual verifica o melhor horário para um evento levando em consideração um conjunto de usuários. Para tal, fui utilizado o *slot* com o tamanho de 1 hora e considerou-se que os eventos poderiam ser marcados no horário de 08:00 às 12:00.

Para esse método, devem ser consideradas duas variáveis que podem influenciar o seu desempenho, que são o número de usuários e o intervalo de tempo. Portanto, foram feitas duas baterias de testes. Cada bateria de teste tinha uma dessas variáveis fixa. Já a outra era alterada.

7.2.2.1 Intervalo de Tempo Fixo

Com o intervalo de tempo fixo, foram feitos testes com o número de usuários variando a partir dos valores: 10, 50, 100, 300 e 500.

A seguir, é apresentado, na Figura 36, o tempo de execução do método "verify_availabilities" com o intervalo de tempo de 31 dias.

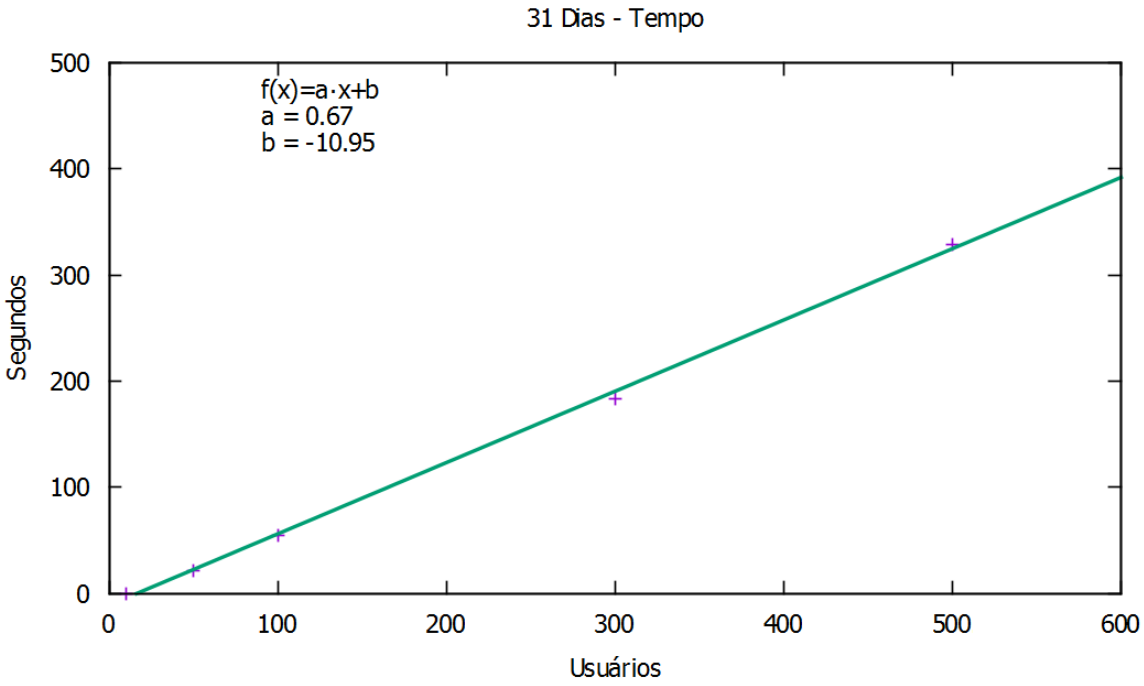


Figura 36 – Tempo de execução com disponibilidade de 31 dias

Na Figura 37, pode-se visualizar a quantidade de memória consumida com um intervalo de tempo de 31 dias.

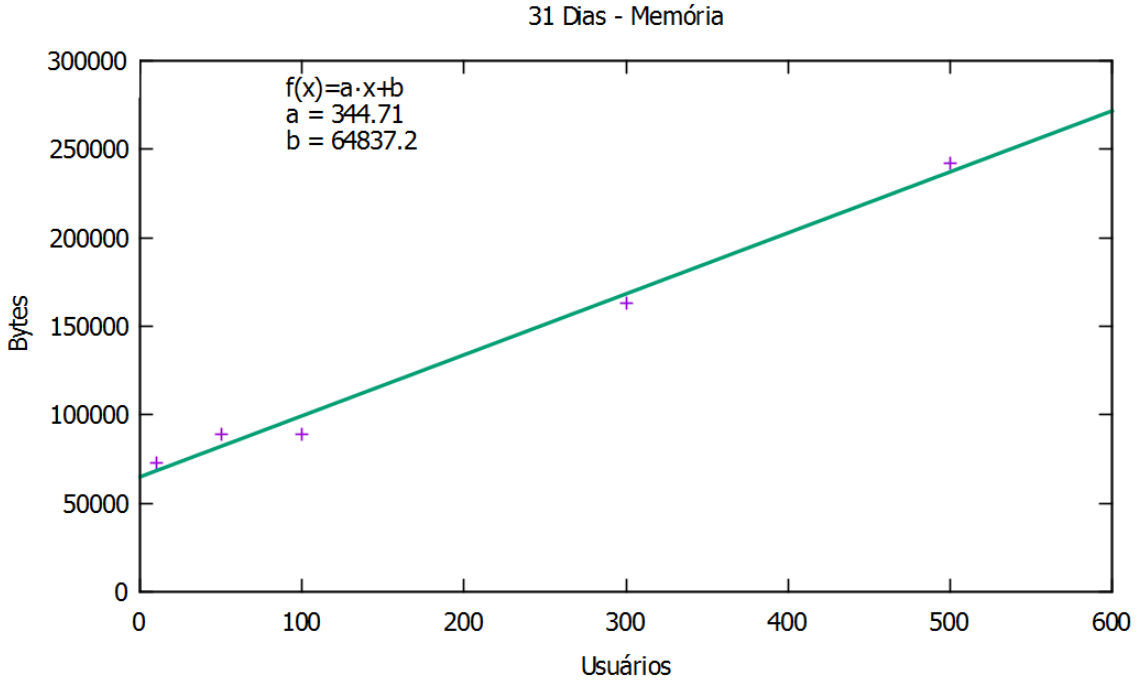


Figura 37 – Consumo de memória com disponibilidade de 31 dias

7.2.2.2 Número de Usuários Fixo

Com o número de usuários fixo, foram realizados testes com o intervalo de tempo variando entre 7, 15, 31, 45 e 60 dias.

É possível observar na Figura 38 o tempo de execução quando o número de usuários é fixo em 500 e o intervalo de tempo é variável.

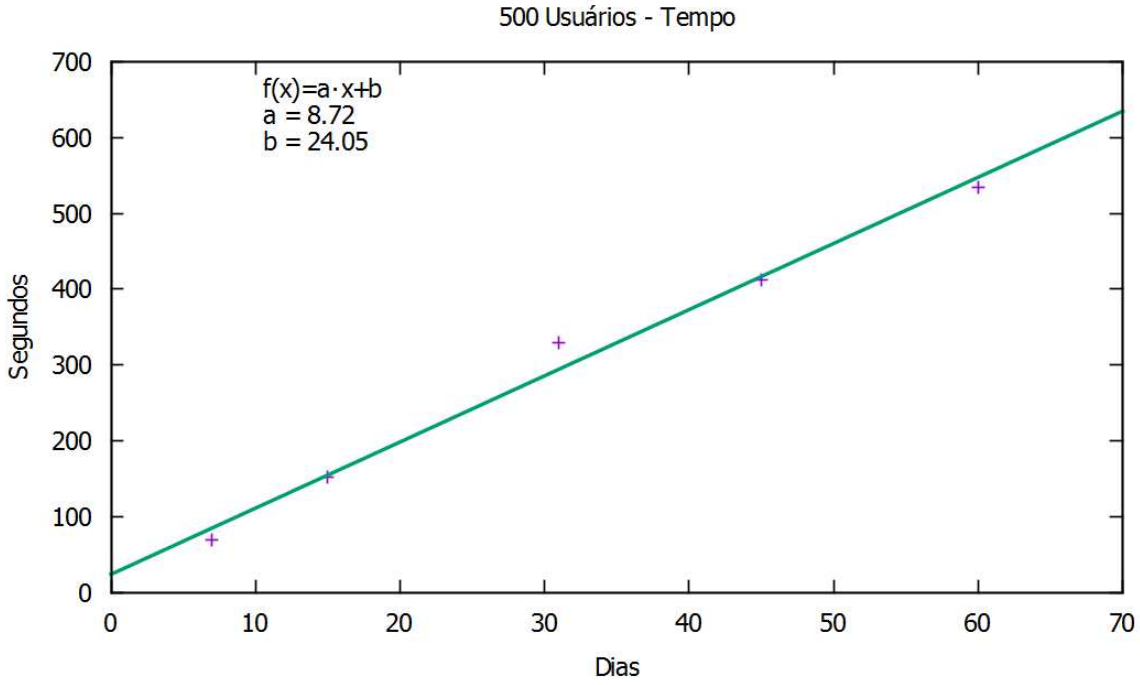


Figura 38 – Tempo de execução para 500 usuários

A quantidade de memória consumida com 500 usuários fixos, pode ser observada na Figura 39.

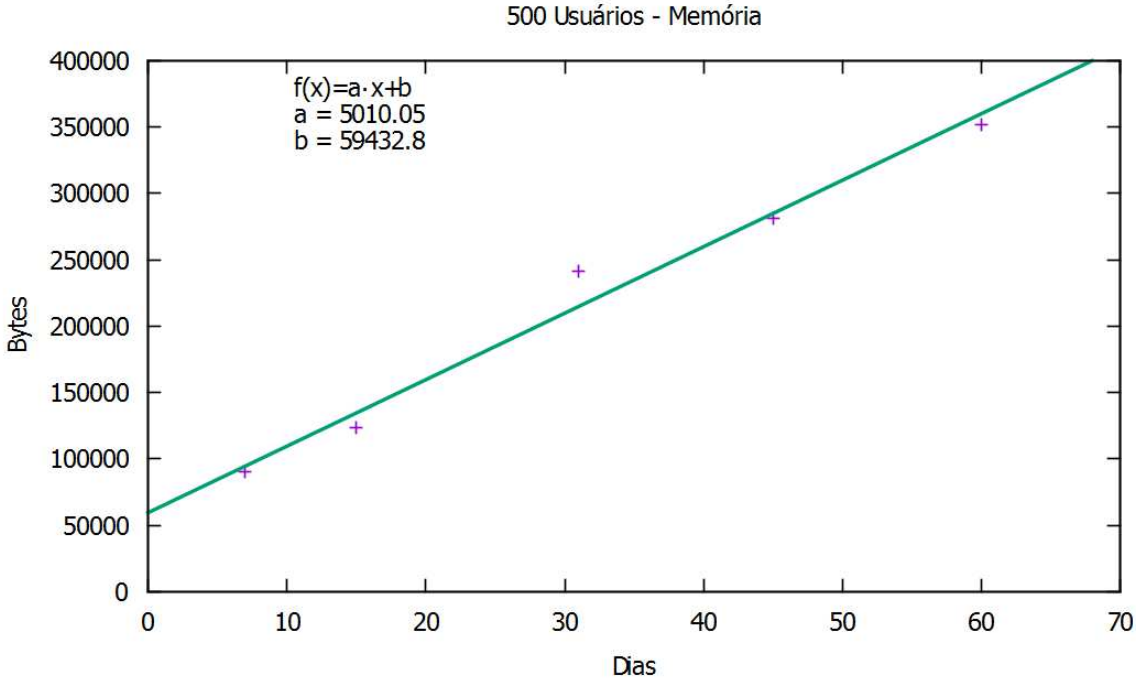


Figura 39 – Consumo de memória para 500 usuários

7.2.3 Complexidade

A partir dos gráficos apresentados anteriormente e da pesquisa realizada e documentada na seção Complexidade de algoritmos, pôde-se avaliar a complexidade dos algoritmos utilizados. Considera-se "n" como a quantidade de usuários.

O tempo para a construção do grafo têm sua função aproximada a uma complexidade $O(n^2)$. Isso se dá, devido ao algoritmo usar o BFS que têm uma complexidade O(n) e ser necessário verificar, no momento de inserção de um vértice no grafo, se o vértice já não foi adicionado, essa verificação também tem a complexidade O(n), resultando assim, na complexidade final mencionada. Já o tempo no algoritmo de pesquisa é linear, pois esse utiliza puramente o BFS, sendo assim, sua complexidade é O(n).

A complexidade de exeução para a sugestão de relacionamentos é O(n) já que esse algoritmo usa puramente o DFS. Vale ressaltar que, nesse caso, apenas três níveis do grafo são considerados.

No algoritmo de complexidade têm-se a complexidade como O(n * m), onde 'n' representa a quantidade de usuários e 'm' a quantidade de *slots*.

O consumo de memória de todos os algoritmos que foram medidos tiveram os seus gráficos aproximados a uma reta. Desse modo pode dizer que esses algoritmos possuem uma complexidade de consumo de memória de O(n).

7.3 Qualidade do SocialFramework

Essa seção procurará exibir os resultados quanto à qualidade do código fonte do *framework*. A partir da análise de código fonte realizada pela ferramenta Code Climate, foi possível obter as métricas e realizar uma análise qualitativa dos resultados obtidos.

Medir e monitorar a qualidade do software é fundamental. Mesmo uma ótima suíte de testes pode produzir informações apenas sobre características externas, não refletindo qualidades como manutenibilidade, modularidade, flexibilidade e simplicidade (FILHO, 2013).

Filho, em (FILHO, 2013), afirma ainda que nem tudo é relevante para ser controlado. Portanto, não se deve procurar medir tudo, mas escolher bem as métricas para avaliar e controlar o projeto. Desse modo, as métricas que foram monitoradas durante o projeto foram:

1. **Complexidade**: Essa métrica fornece uma análise da complexidade e do esforço gasto em um trecho de código. Para tanto, foram utilizadas as seguintes métricas:

   a) **Complexidade ciclomática**: é calculada pela quantidade de caminhos que é possível percorrer no método. Dessa forma, sempre começa em 1, pois todo método tem pelo menos um caminho possível. Para cada estrutura condicional ou repetitiva(*if*, *unless*, *elsif*, *while*, *until*, *for*, *when*, *rescue*), é adicionado mais 1 para a complexidade;

   b) **Tamanho do código**: que é a quantidade de linhas que o código possui, desconsiderando comentários e espaços em branco;

   c) **Complexidade ABC**: o cálculo da complexidade é baseado em três pontos:

      i. **Assignment**: qualquer atribuição explícita de um dado para uma variável, por exemplo: = *= /= %= += «= »= &= |= >= ++ —, entre outros;

      ii. **Branch**: quando um caminho fora do escopo do programa é feito, por exemplo, uma chamada de método de instância de um objeto, um operador *new* instanciando uma classe, etc;

      iii. **Condition**: qualquer teste lógico/booleano, == != <= >= < > *else case default try catch* ? e condicionais unários.

2. **Duplicação de código**: analisa a similaridade de trechos de código na aplicação. Para cada nível de similaridade, é atribuído um peso diferente, indo de similar até idêntico, resultando no número que é a somatória de todos os arquivos. Espaços em branco, nomes de variáveis, métodos e classes são ignorados pela ferramenta;

3. **Estilo**: verifica se o projeto utiliza o guia de estilo descrito pela comunidade Rails. Aponta onde estão as práticas não adequadas relacionadas, relacionadas a padrões e convenções, as quais não estão sendo respeitadas na aplicação, e

4. **Segurança**: utiliza um *fork* do Brakeman na versão 2.6.2 para identificar as vulnerabilidades do projeto Rails.

A partir dessas métricas, o Code Climate realiza o cálculo do GPA (*grade point average*) que é uma espécie de média de todos os arquivos do projeto. O GPA possui um intervalo de 0 a 4 pontos, levando em consideração as notas dos arquivos, que varia de 'A' a 'F'. Desse modo, no início do projeto, foi estabelecido uma meta de 3 pontos. Essa meta foi atingida com êxito, obtendo-se 3.45 pontos, como pode ser observado na Figura 40.

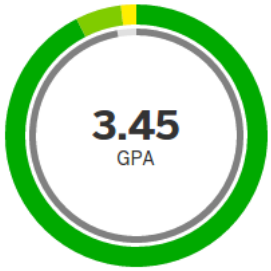


Figura 40 – GPA

Esse valor do GPA foi obtido de acordo com a Tabela 2, que ilustra a quantidade de arquivos com as notas de 'A' até 'F'.

Tabela 2 – Nota de cada arquivo

Nota A	Nota B	Nota C	Nota D	Nota E	Nota F
50	2	1	0	0	0

7.4 SocialBike

O SocialBike é a rede social desenvolvida para fazer uso do SocialFramework e, dessa forma, aplicá-lo em um uso real para comprovação de seus recursos. É uma rede voltada para ciclistas que auxilia os mesmos a, principalmente, marcarem eventos em comum para andar de bicicleta e/ou verificar rotas de outros usuários que coincidam com as suas. Nessa rede, foram desenvolvidas funcionalidades que perpassam pelos três módulos do SocialFramework, Usuários, Agenda e Rotas.

Para o módulo de usuários, foram desenvolvidas funcionalidades gerais de redes sociais, com cadastro de usuários, *login*, relacionamentos e pesquisas. A seguir, são apresentadas uma sequência de imagens do SocialBike, com as telas dessas funcionalidades.

A Figura 41 apresenta a tela de cadastro de usuários. Quando um usuário se cadastra na rede, esse é automaticamente logado na mesma.

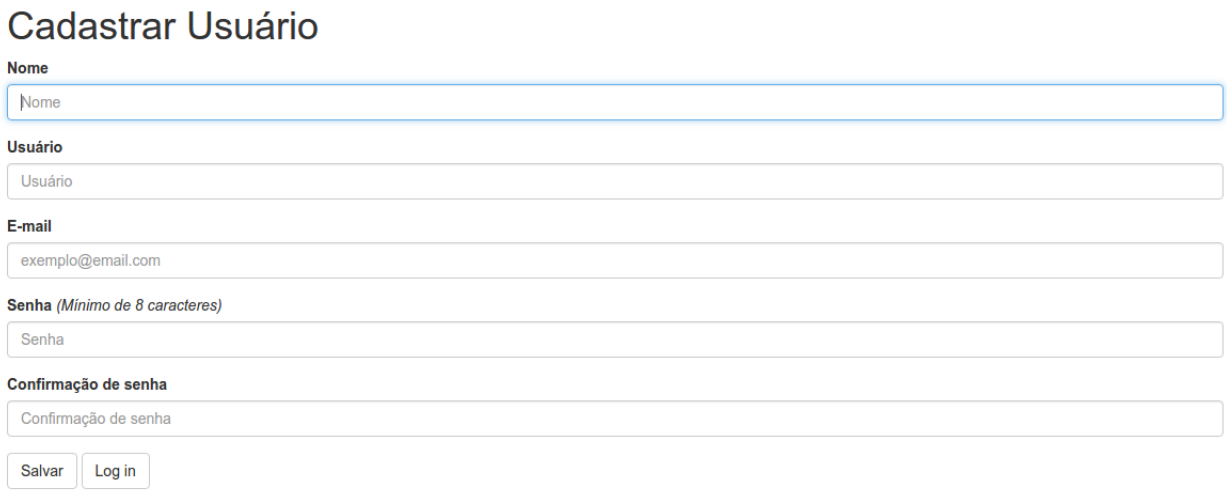


Figura 41 – Tela de cadastro de usuários

A Figura 42 apresenta a tela para entrar na rede social.

Figura 42 – Tela de *login*

A Figura 43 mostra a tela principal da rede social, com as listas dos amigos e sugestões de amizades para o usuário logado, além de um campo para realização de pesquisas.

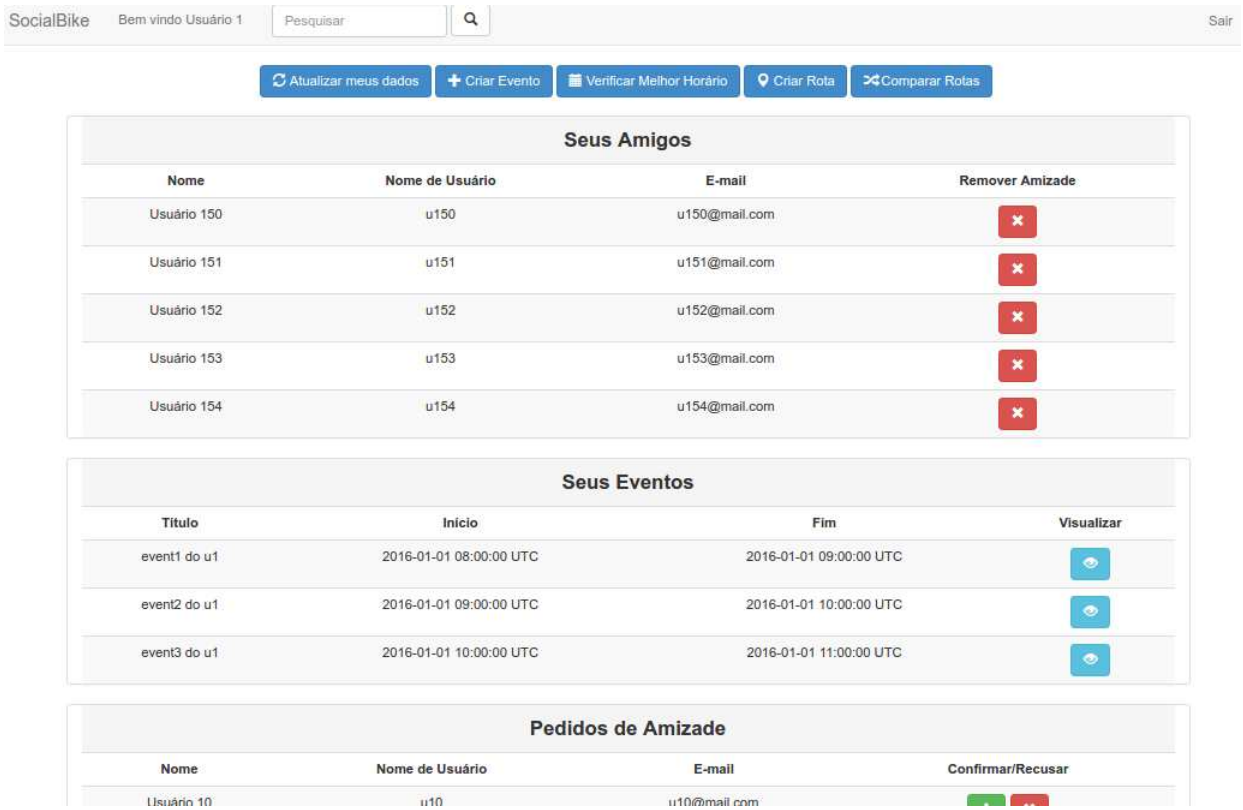


Figura 43 – Tela principal

A Figura 44 mostra os resultados retornados quando um usuário faz uma pesquisa. Ao se realizar uma pesquisa, é apresentado o botão “Continuar”, para que o usuário continue retornando mais resultados da pesquisa realizada. Nesse caso a tabela com os resultados é incrementada.

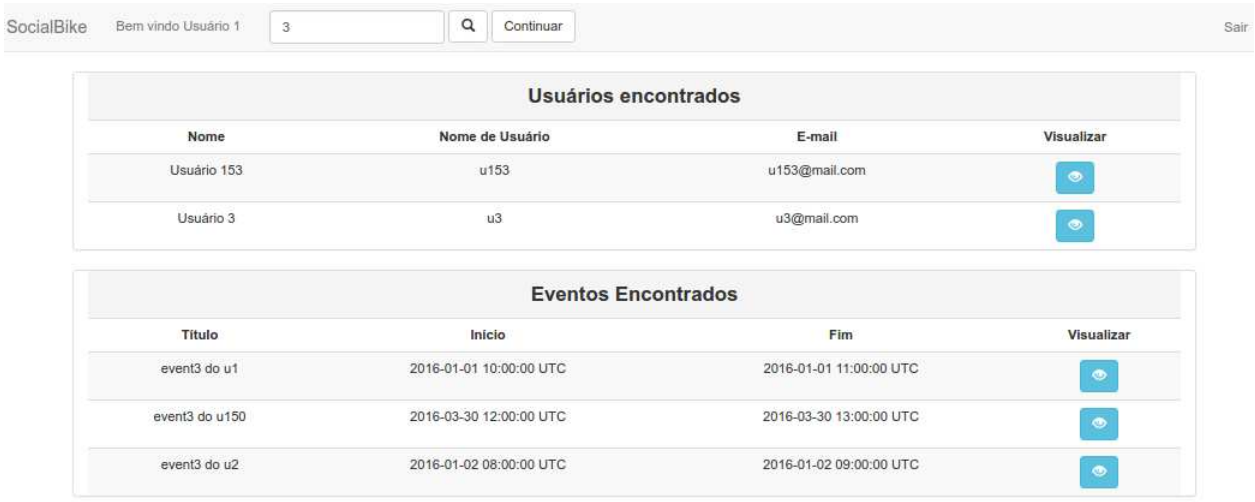


Figura 44 – Resultados da pesquisa

Na sequência, as Figuras 45 e 46 mostram, respectivamente, as telas para criação de eventos e a visualização desses eventos, com a possibilidade de convidar usuários.

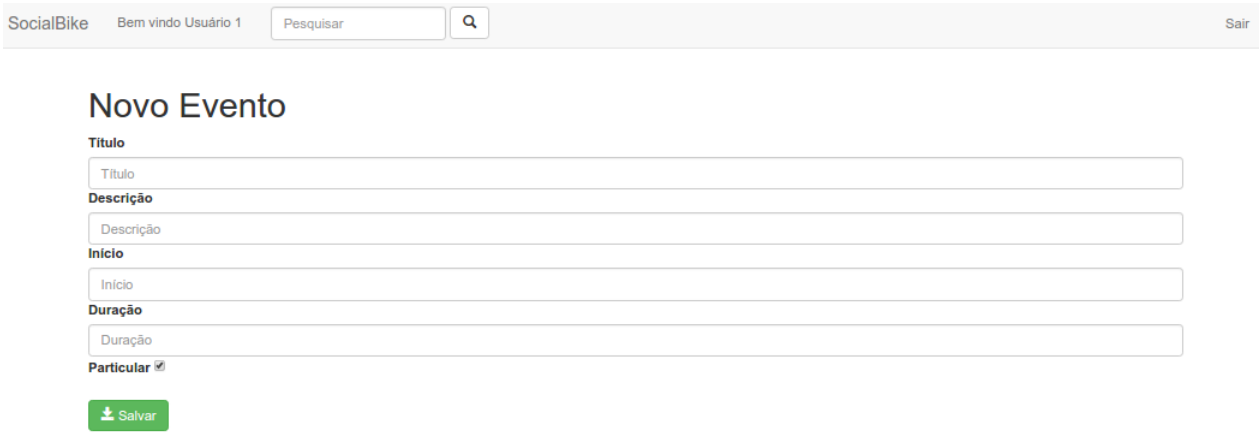


Figura 45 – Criar Evento

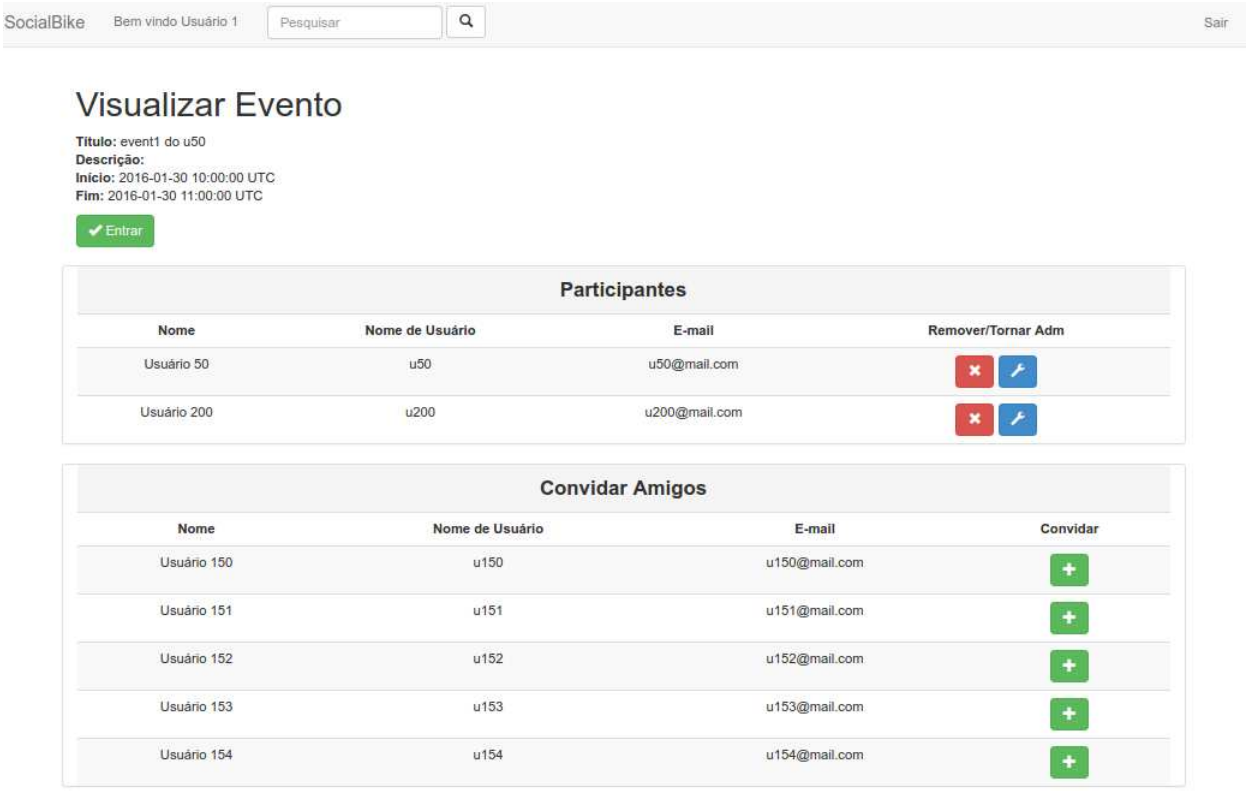


Figura 46 – Visualizar Evento

A tela apresentada na Figura 47 é usada para verificar a compatibilidade de horários para marcação de um evento em um determinado período de tempo. Têm-se a lista de amigos do usuário logado para que este possa adicioná-los juntamente com seus pesos para realizar a verificação. Outros atributos que devem ser passados são os já mencionados antes, os quais são necessários para a execução do algoritmo.

Figura 47 – Verificar compatibilidade de horários

As Figuras 48 e 49 mostram, respectivamente, as telas para criação e visualização de rotas.

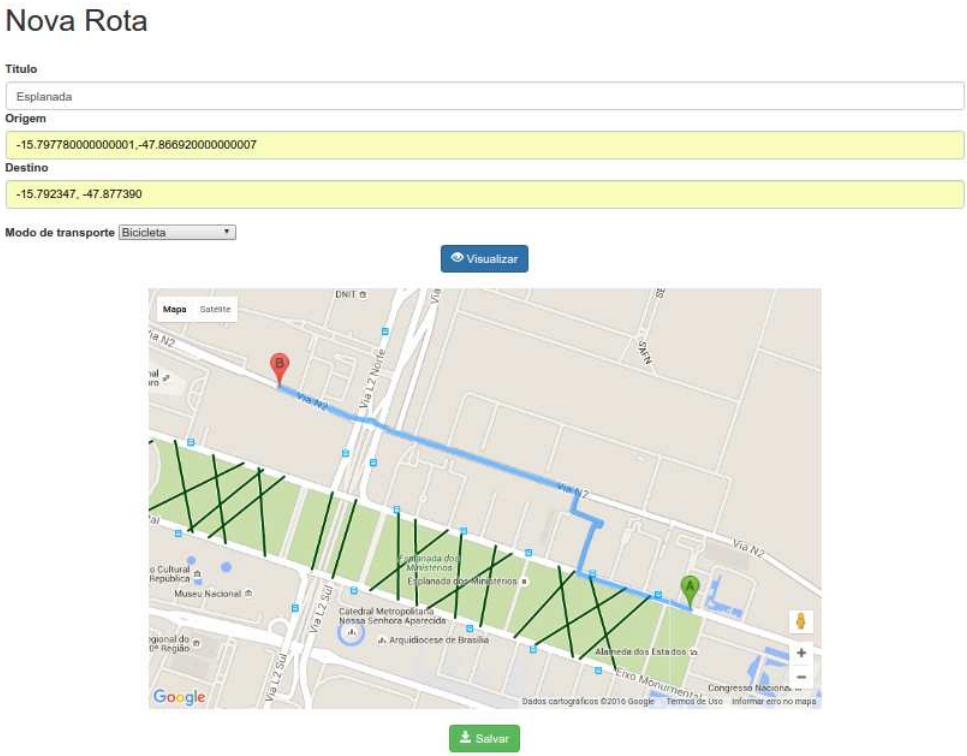


Figura 48 – Cadastrar Rota

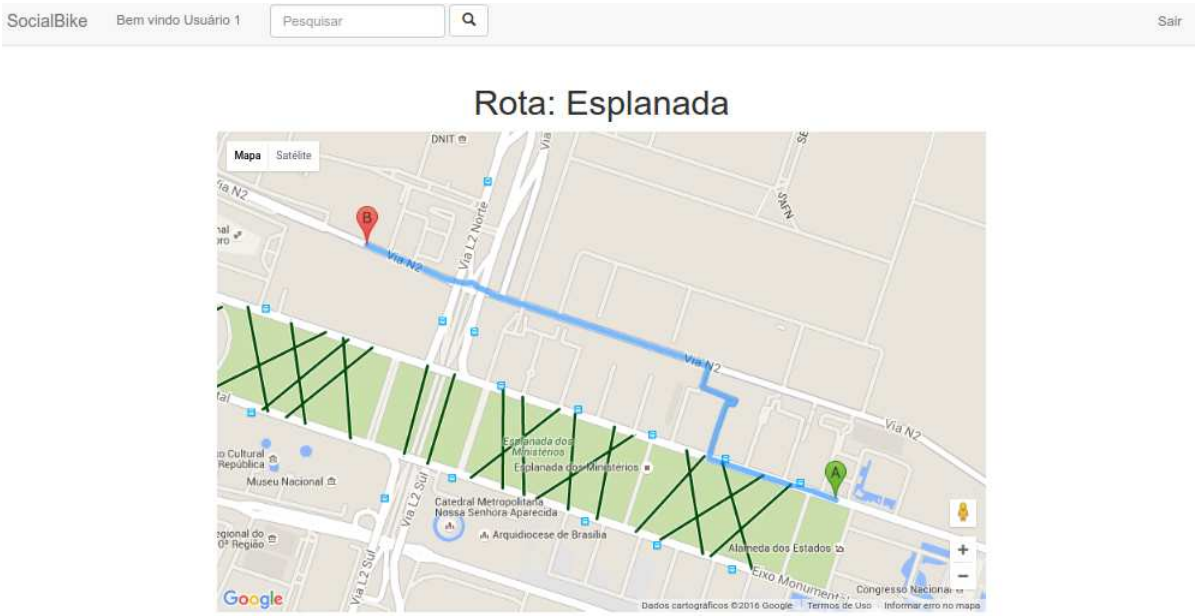


Figura 49 – Visualizar Rota

Por fim, têm-se as Figuras 50 e 51. Nessas imagens, pode-se ver um formulário para realizar a comparação entre duas rotas e o resultado dessa comparação, em que se percebe que as rotas são compatíveis e qual a nova distância total para a rota principal, que será alterada para atender a rota secundária.

Figura 50 – Comparar rotas

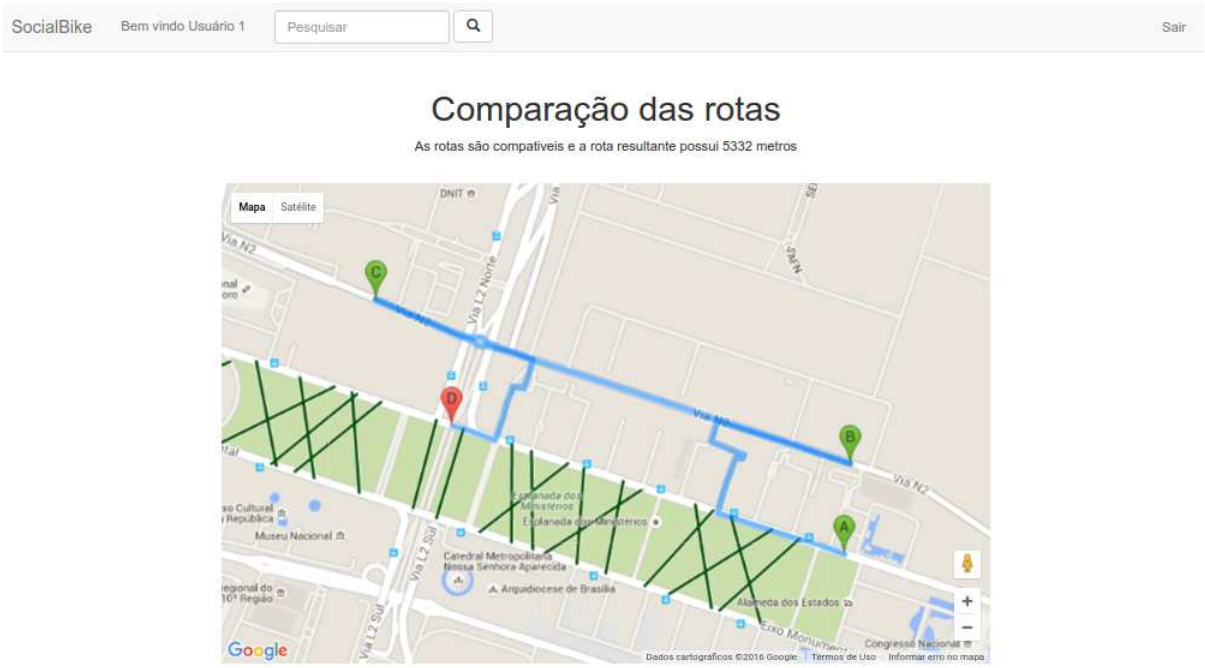


Figura 51 – Resultado da comparação

O código da SocialBike pode ser acessado através do endereço <https://github.com/TCC-SocialNetwork/social_bike>.

7.5 Resumo do Capítulo

Este capítulo foi responsável por apresentar os principais resultados obtidos ao longo do processo de desenvolvimento, tanto do SocialFramewok quanto da rede social SocialBike.

Foram apresentados alguns relatórios contendo os resultados. Um dos relatórios apresentados foi o relatório de testes. Nesse, constam a cobertura de código. Adicionalmente, foram apresentados os principais relatórios de desempenho, os quais mostram como os algoritmos se comportaram em complexidade e uso de memória. Por fim, têm-se os relatórios de qualidades, nos quais constam os resultados obtidos - utilizando o Code Climate - visando aferir as principais medidas de qualidade de código.

Além disso, foram apresentadas a rede social SocialBike e suas funcionalidades, as quais foram todas desenvolvidas a partir dos recursos do SocialFramework para uma melhor análise de uso do mesmo.

8 Conclusão

Este trabalho teve como objetivo o desenvolvimento do SocialFramework, que é um *framework* para desenvolvimento de redes sociais, porém, é focado em um nicho específico de rotas e agendas.

No decorrer do desenvolvimento da aplicação, foram utilizados padrões de projetos, os princípios das técnicas de programação, desenvolvimento de testes, monitoramento da análise estática do código, além de outros princípios da engenharia de software. A intenção foi procurar obter um *framework* orientado por critérios de qualidade, no caso, facilidade de manutenção e evolução de software.

Neste trabalho, foi levantada a seguinte questão de pesquisa: "É possível oferecer um *framework* que auxilie no desenvolvimento de redes sociais, disponibilizando recursos gerais de relacionamentos e específicos de definições de rotas e agenda, proporcionando ao desenvolvedor facilidade ao lidar com preocupações intrínsecas desse contexto?" Com base nos resultados apresentados e na instanciação de uma rede social que possuía como base o *framework* desenvolvido, pode-se concluir que a resposta para essa questão é: Sim, o SocialFramework atingiu os objetivos estabelecidos.

Verificou-se que todos os objetivos esperados foram atingidos com o desenvolvimento do SocialFramework e da rede social SocialBike. A seguir, são apresentadas algumas sugestões para trabalhos futuros, em especial, aqueles que partem dos incrementos que foram desenvolvidos até o momento.

8.1 Trabalhos Futuros

A seguir, serão apresentados os principais pontos que deverão ser evoluídos para a continuidade do *framework*.

1. Implementar, no módulo de usuários, suporte para grupos;
2. Desenvolver suporte para postagens de usuários;
3. Inserir, no módulo de usuários, suporte para *chat*;
4. Evoluir o módulo de agenda para oferecer suporte à repetição de eventos;
5. Integração da agenda do *framework* com a agenda do Google.

Referências

ALBUQUERQUE, J. Análise de complexidade de algoritmos. 2004. Citado na página 38.

ALEXANDER, C. *The Timeless Way of Building*. New York: Oxford University Press, 1979. Hardcover. ISBN 0195024028. Citado na página 42.

ALMEIDA, R. R. de. Model-view-controller (mvc). 2006. Disponível em: <http://www.dsc.ufcg.edu.br/~jacques/cursos/map/html/arqu/mvc/mvc.htm>. Citado na página 42.

ALONSO, G. et al. *Web Services: Concepts, Architectures and Applications*. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2010. ISBN 3642078885, 9783642078880. Citado na página 122.

BONDY, J.-A.; MURTY, U. S. R. *Graph theory*. New York, London: Springer, 2007. (Graduate texts in mathematics). OHX. ISBN 978-1-8462-8969-9. Citado 5 vezes nas páginas 34, 36, 37, 114 e 115.

BOYD, D. M.; ELLISON, N. B. Social network sites: Definition, history, and scholarship. *Journal of Computer-Mediated Communication*, v. 13, n. 1, 2007. Citado na página 27.

BRASSARD, G.; BRATLEY, P. *Algorithmics: Theory &Amp; Practice*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988. ISBN 0-13-023243-2. Citado 2 vezes nas páginas 119 e 120.

CAPRA, F. *Conexões Ocultas, as*. [S.l.]: Cultrix, 2002. ISBN 9788531607486. Citado na página 33.

CORMEN, T. H. et al. *Introduction to Algorithms*. 2nd. ed. [S.l.]: McGraw-Hill Higher Education, 2001. ISBN 0070131511. Citado 3 vezes nas páginas 117, 118 e 119.

COSTA, P. P. da. Teoria de grafos e suas aplicações. 2011. Citado 5 vezes nas páginas 36, 37, 113, 114 e 115.

DIESTEL, R. *Graph Theory*. [S.l.]: Springe, 1997. (Graduate Texts in Mathematics, 173). Citado 3 vezes nas páginas 36, 114 e 115.

EMARKETER. Brazil's social audience keeps growing, as new web users join in. 2013. Disponível em: <http://www.emarketer.com/Article/Brazils-Social-Audience-Keeps-Growing-New-Web-Users-Join/1010003>. Citado na página 27.

FAYAD, M.; SCHMIDT, D. C. Object-oriented application frameworks. *Commun. ACM*, ACM, New York, NY, USA, v. 40, n. 10, p. 32–38, out. 1997. ISSN 0001-0782. Citado 3 vezes nas páginas 39, 41 e 42.

FIGUEIREDO, C. M. H. d.; SZWARCFITER, J. L. Emparelhamento em grafos, algoritmos e complexidade. Instituto de Matemática, UFRJ, 1999. Citado na página 77.

FILHO, C. M. d. O. Kalibro: interpretação de métricas de código-fonte. 2013. Citado na página 96.

FILHO, O. F. F. Serviços semânticos: Uma abordagem restful. Escola Politécnica da Universidade de São Paulo, 2009. Citado 2 vezes nas páginas 122 e 123.

FRANCA, L. P. A.; STAA, A. V. Geradores de artefatos: Implementação e instanciação de frameworks. Departamento de Informática, PUC-Rio, 2001. Citado na página 62.

GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0-201-63361-2. Citado 7 vezes nas páginas 42, 44, 123, 124, 125, 126 e 127.

GIL, A. C. *Como elaborar projetos de pesquisa*. [S.l.]: Atlas, 2010. ISBN 9788522458233. Citado na página 130.

GOULAR, E. E. Análise de redes sociais: aplicação nos estudos de transferência da informação. Ciência da Informação, Brasília, 2001. Citado na página 28.

HANNEMAN, R. A.; RIDDLE, M. Introduction to social network methods. Riverside, CA: University of California, 2005. Disponível em: <http://faculty.ucr.edu/~hanneman/nettext/>. Citado na página 28.

JOHNSON, R. E. Frameworks = (components + patterns). *Commun. ACM*, ACM, New York, NY, USA, v. 40, n. 10, p. 39–42, out. 1997. ISSN 0001-0782. Citado 3 vezes nas páginas 39, 40 e 42.

JOHNSON, R. E.; FOOTE, B. Designing reusable classes. *Journal of Object-Oriented Programming*, v. 1, n. 2, p. 22–35, June/July 1988. Disponível em: <http://www.laputan.org/drc.html>. Citado na página 40.

JúNIOR, W. M. P. Análise de algoritmos. 2014. Citado na página 38.

KESWANI, R.; JOSHI, S.; JATAIN, A. Software reuse in practice. In: *Proceedings of the 2014 Fourth International Conference on Advanced Computing & Communication Technologies*. Washington, DC, USA: IEEE Computer Society, 2014. (ACCT '14), p. 159–162. ISBN 978-1-4799-4910-6. Disponível em: <http://dx.doi.org/10.1109/ACCT.2014.57>. Citado na página 29.

KOCHE, J. *Fundamentos de metodologia científica: teoria da ciência e prática da pesquisa*. [S.l.]: Vozes, 1997. ISBN 9788532618047. Citado na página 129.

KRISTENSEN, N. R.; MADSEN, H.; JøRGENSEN, S. B. A method for systematic improvement of stochastic grey-box models. *Computers & Chemical Engineering*, v. 28, n. 8, p. 1431–1449, 2004. Disponível em: <http://dblp.uni-trier.de/db/journals/cce/cce28.html#KristensenMJ04>. Citado na página 41.

KROTH, E. Arquitetura de software para reuso de componentes. Universidade Federal do Rio Grande do Sul, RS, BR, 2000. Citado na página 41.

KRUEGER, C. W. Software reuse. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 24, n. 2, p. 131–183, jun. 1992. ISSN 0360-0300. Citado 2 vezes nas páginas 29 e 39.

LIMA, T. M. P. Uma abordagem socio-técnica para apoiar modelagem e análise de ecossistemas de software. 2015. Disponível em: <http://monografias.poli.ufrj.br/monografias/monopoli10014249.pdf>. Citado na página 29.

LUCRéDIO, D. Uma abordagem orientada a modelos para reutilização de software. INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO UNIVERSIDADE DE SÃO PAULO, 2009. Citado na página 39.

MAFRA, S. N.; TRAVASSOS, G. H. Estudos primários e secundários apoiando a busca por evidência em engenharia de software. UFRJ: Rio de Janeiro, 2006. Citado 2 vezes nas páginas 130 e 131.

MALTA, G. H. S. Grafos no ensino médio: uma inserção possível. 2008. Citado na página 35.

MARTELETO, R. M. Mídias sociais: uma contribuição de análise. EDIPUCRS: Editora Universitária da PUCRS, 2014. Citado 2 vezes nas páginas 28 e 34.

MILGRAM, S. The small world problem. *Psychology Today*, v. 67, n. 1, p. 61–67, 1967. Citado na página 34.

MORESI, E. Metodologia da pesquisa. Universidade Católica de Brasília - Brasília DF, 2003. Citado na página 129.

NETTO, P. O. B. *Grafos: Teorias, Modelos, Algoritmos*. [S.l.: s.n.], 2012. ISBN 9788573936346. Citado na página 113.

NEWCOMER, E.; LOMOW, G. *Understanding SOA with Web Services (Independent Technology Guides)*. [S.l.]: Addison-Wesley Professional, 2004. ISBN 0321180860. Citado 2 vezes nas páginas 121 e 122.

NEWMAN, M. *Networks: An Introduction*. New York, NY, USA: Oxford University Press, Inc., 2010. ISBN 0199206651, 9780199206650. Citado na página 33.

OLSEN, R. *Design Patterns in Ruby (Addison-Wesley Professional Ruby Series)*. 1. ed. [S.l.]: Addison-Wesley Professional, 2007. ISBN 0321490452, 9780321490452. Citado na página 87.

ORE, O. *Graphs and their uses*. New York: Random House, 1963. (New mathematical library). Citado 3 vezes nas páginas 34, 35 e 113.

PAUTASSO, C. Restful web service composition with bpel for rest. *Data Knowl. Eng.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 68, n. 9, p. 851–866, set. 2009. ISSN 0169-023X. Citado na página 123.

ROSENBERG, F. et al. Composing restful services and collaborative workflows: A lightweight approach. *IEEE Internet Computing*, v. 12, n. 5, 2008. Disponível em: <http://dblp.uni-trier.de/db/journals/internet/internet12.html#RosenbergCDK08>. Citado na página 122.

SCOTT, J. P.; CARRINGTON, P. J. *The SAGE Handbook of Social Network Analysis*. [S.l.]: Sage Publications Ltd., 2011. ISBN 1847873952, 9781847873958. Citado na página 28.

SHALLOWAY, A.; TROTT, J. *Design Patterns Explained: A New Perspective on Object-Oriented Design (2Nd Edition) (Software Patterns Series)*. [S.l.]: Addison-Wesley Professional, 2004. ISBN 0321247140. Citado 2 vezes nas páginas 42 e 43.

SILVA, R. P. e. Suporte ao desenvolvimento e uso de frameworks e componente. Universidade Federal do Rio Grande do Sul, RS, BR, 2000. Citado na página 41.

SOCIAL, W. A. Digital, social & mobile worldwide in 2015. 2015. Disponível em: <http://wearesocial.com/uk/special-reports/digital-social-mobile-worldwide-2015>. Citado 2 vezes nas páginas 29 e 30.

SODRE, M. Antropológica do espelho: Uma teoria da comunicação linear e em rede. Editora Vozes, Petrópolis, RJ, v. 3, n. 4, 2002. Citado na página 33.

STOJANOVIC, Z.; DAHANAYAKE, A. *Service-oriented Software System Engineering Challenges And Practices*. Hershey, PA, USA: IGI Global, 2005. ISBN 1591404274. Citado na página 121.

TAFNER, E. P.; SILVA, R. Apostila de metodologia científica. 2007. Citado 3 vezes nas páginas 129, 130 e 131.

TOMAE, M. I.; ALCARA, A. R.; CHIARA, I. G. Das redes sociais à inovação. Ci. Inf., Brasília, 2005. Citado 3 vezes nas páginas 29, 33 e 34.

TRAVASSOS, G. H. *Introdução à engenharia de software experimental*. UFRJ, 2002. (RT-ES-590/02). Disponível em: <http://www.ufpa.br/cdesouza/teaching/topes/4-ES-Experimental.pdf>. Citado na página 129.

VICTORINO, M.; BRäSCHER, M. Organização da informação e do conhecimento, engenharia de software e arquitetura orientada a serviços: uma abordagem holística para o desenvolvimento de sistemas de informação computadorizados. DataGramaZero - Revista de Ciência da Informação - v.10 n.3, 2009. Citado 2 vezes nas páginas 121 e 122.

WASSERMAN, S.; FAUST, K. *Social network analysis: Methods and applications*. [S.l.]: Cambridge university press, 1994. Citado na página 33.

WELLMAN, B. Are personal communities local? a dumptarian reconsideration. *Social Networks*, v. 18, n. 4, p. 347–354, 1996. Citado na página 34.

WILSON, G. et al. *Best Practices for Scientific Computing*. 2012. Disponível em: <http://arxiv.org/abs/1210.0530>. Citado na página 51.

WOHLIN, C. et al. *Experimentation in Software Engineering: An Introduction*. Norwell, MA, USA: Kluwer Academic Publishers, 2000. ISBN 0-7923-8682-5. Citado na página 129.

Apêndices

APÊNDICE A – Grafos

A.1 Um Pouco mais de História

Graças à resolução dada por Euler, mais tarde muitos outros problemas importantes, para o desenvolvimento da Matemática Aplicada, foram possíveis de serem modelados. Um desses modelos são as relações de amizade, de hierarquia, de trabalho. Netto (NETTO, 2012), aponta grafos como um auxílio para o estudo de problemas envolvendo inter-relacionamento de elementos (em química orgânica, eletricidade, organização, transporte, psicossociologia). Na verdade, grafos modelam diversas situações e muitas delas não quantificáveis.

Conforme Ore (ORE, 1963), o matemático irlandês William Hamilton, em 1859, inventou um jogo chamado "*The Icosian Game*", com um peculiar enigma envolvendo um dodecaedro, em que cada um dos 20 vértices foram nomeados com nomes de cidades importantes. O objetivo do jogo era, utilizando as 30 arestas do dodecaedro, passar por cada uma das cidades apenas uma vez, começando e terminando na mesma cidade. Um exemplo do grafo pode ser visualizado na figura 52.

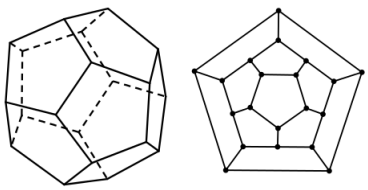


Figura 52 – Grafo hamiltoniano (ORE, 1963)

Apesar da simples formulação, o problema admite muitos caminhos como resposta. No problema de Hamilton, temos uma diferença significativa em relação ao problema de Euler. Encontrar um caminho euleriano significa encontrar um caminho que passe por todas as arestas do grafo uma única vez, podendo ser aberto ou fechado. Nos caminhos hamiltonianos, cada vértice é visitado uma única vez. O problema fica mais complexo com tal condição (COSTA, 2011).

A.2 Outras Definições

Um grafo é finito se tanto o seu conjunto de vértices, quanto o seu conjunto de arestas são finitos. Um grafo sem vértices (e, portanto, sem arestas) é o grafo nulo. Qualquer grafo apenas com um vértice é referido como trivial. Todos os outros grafos são não-triviais (COSTA, 2011).

Um grafo é simples se não tem *loops* ou arestas paralelas (DIESTEL, 1997), como exemplificado na figura 53.

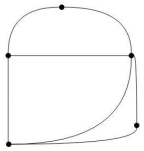


Figura 53 – Exemplo de grafo simples (COSTA, 2011)

Certos tipos de grafos podem desempenhar papéis proeminentes na teoria dos grafos. Um grafo conexo é um grafo simples no qual quaisquer dois vértices são ligados por um caminho. Um grafo é vazio quando não há dois vértices adjacentes (isto é, o conjunto de arestas é vazio). Um grafo é bipartido se o seu conjunto de vértices pode ser particionado em dois subconjuntos X e Y para que cada aresta tem um fim em X e um fim em Y; uma tal partição $(X,\ Y)$ é chamada uma bipartição do grafo, e X e Y suas partes. Pode-se denotar um grafo bipartido G com bipartição $(X,\ Y)$ por $G[X,\ Y]$. Se $G[X,\ Y]$ é simples e todos os vértices de X estão associados a cada vértice em Y, então G é chamado de um grafo bipartido completo. Uma estrela é um grafo bipartido completo $G[X,\ Y]$ com $|X| = 1$ ou $|Y| = 1$ (DIESTEL, 1997). A figura 54 ilustra estes tipos de grafos, sendo o grafo "A" é um grafo conexo, o grafo "B" é um grafo vazio e o grafo "C" é um grafo bipartido completo.

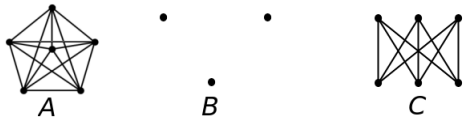


Figura 54 – Tipos de grafos (DIESTEL, 1997)

Um caminho é um grafo simples cujos vértices podem ser dispostos em uma sequência linear. De tal forma que dois vértices são adjacentes se forem consecutivos na sequência, e não adjacentes caso não forem consecutivos (BONDY; MURTY, 2007). Dessa forma, diz-se que um vértice é alcançável a partir de outro, se houver um caminho levando o primeiro vértice ao último (COSTA, 2011). A figura 55 apresenta um caminho (arestas em vermelho) do vértice "a" até o vértice "e".

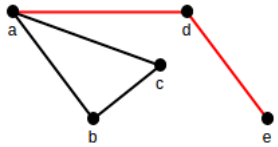


Figura 55 – Caminho (COSTA, 2011)

Do mesmo modo, um ciclo é um grafo simples cujos vértices podem ser dispostos em uma sequência cíclica de tal maneira que dois vértices são adjacentes se forem consecutivos

na sequência. O comprimento de um caminho ou de um ciclo é o número de suas arestas (COSTA, 2011). É possível observar na figura 56 alguns exemplos de grafos com ciclo.

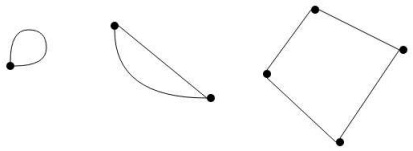


Figura 56 – Exemplo de grafos com ciclos (COSTA, 2011)

Um grafo é conectado se, para cada partição de seus vértices definido em dois conjuntos X e Y não vazios, existe uma aresta com uma extremidade em X e uma extremidade em Y; caso contrário, o grafo é desconectado. Em outras palavras, um grafo é desconectado se o conjunto de vértices pode ser particionado em dois subconjuntos não vazios X e Y e que nenhuma aresta tem uma extremidade em X e a outra extremidade em Y. É instrutivo comparar esta definição com a de um grafo bipartido. Os exemplos de grafos conectados e desconectados são apresentados na figura 57, onde o grafo "X" e o grafo "Y" são dois grafos distintos conectados. Porém se fossem trados como um único grafo, este seria um grafo desconectado (BONDY; MURTY, 2007).

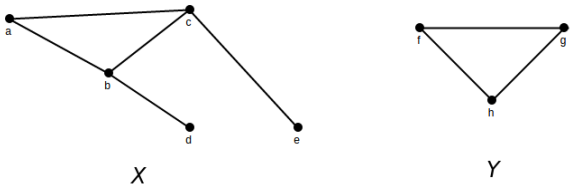


Figura 57 – Exemplo de grafos conectados e desconectados (BONDY; MURTY, 2007)

O grau de um vértice v em um grafo G, designado por $d_G(v)$, é o número de arestas de G que incidem em v; para cada *loop* é contado duas arestas. Em particular, se G é um grafo simples, $d_G(v)$ é o número de vizinhos de v em G. Um vértice de grau zero é chamado um vértice isolado. Denominamos por $\delta(G)$ e $\Delta(G)$ mínimo e máximo graus dos vértices de G, e por $d(G)$, o seu grau médio, $\frac{1}{n}\sum_{v\in V} d(v)$ (DIESTEL, 1997).

APÊNDICE B – Algoritmos

B.1 Busca em Grafos

A busca em grafos refere-se ao método de explorar o grafo, ou seja, obter um método sistemático para percorrer seus vértices e arestas. Para isto, há dois principais algoritmos, busca em profundidade (*DFS - Depth First Search*) e busca em largura (*BFS - Breadth First Search*). Em ambos os casos parte-se de um vértice qualquer.

B.1.1 DFS

A ideia por trás do *DFS* é alcançar "mais fundo" no grafo quanto possível. A ordem em que os vértices são visitados depende da ordem em que os vizinhos de cada vértice aparecem na lista de adjacência (CORMEN et al., 2001). A seguir é exemplificado um fluxo do funcionamento do DFS.

Considere as Figuras a seguir para a exemplificação do DFS.

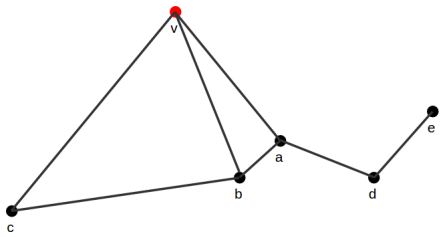


Figura 58 – Exemplo DFS etapa 1 (CORMEN et al., 2001)

A busca irá partir do vértice arbitrário v, chamado de vértice raiz. A partir do vértice raiz é possível percorrer três arestas: (v, a), (v, b) e (v, c). A aresta a ser seguida será (v, a).

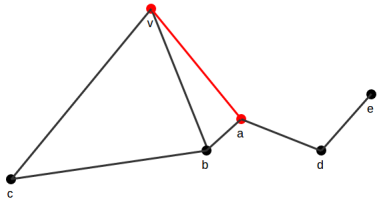


Figura 59 – Exemplo DFS etapa 2 (CORMEN et al., 2001)

O vértice a possui três arestas, que são: $(a,\ v)$, $(a,\ b)$ e $(a,\ d)$. Como o vértice v já foi visitado, a aresta a ser seguida será $(a,\ b)$.

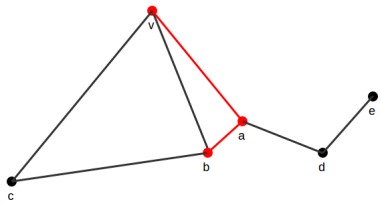


Figura 60 – Exemplo DFS etapa 3 (CORMEN et al., 2001)

A partir de b, é possível escolher as seguintes arestas: $(b,\ v)$, $(b,\ a)$ e $(b,\ c)$. Porém como os vértices v e a já foram visitados, a opção restante será a aresta $(b,\ c)$.

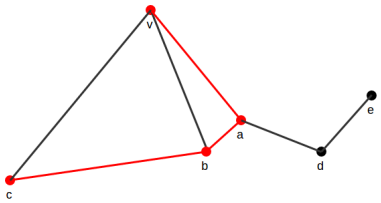


Figura 61 – Exemplo DFS etapa 4 (CORMEN et al., 2001)

Ao alcançar c, há duas possibilidades: $(c,\ v)$ e $(c,\ b)$. Porém ambos os vértices v e b já são conhecidos. Neste caso não há para onde se aprofundar. Entretanto, ainda existem vértices não descobertos. Nesse caso, deve-se voltar até o vértice b, verificando se há alguma aresta que leva a um vértice ainda não visitado. Caso ocorra tal situação, deve-se voltar novamente pelo caminho percorrido, chegando ao vértice a. Em a, a aresta $(a,\ d)$ leva a um vértice ainda não descoberto, portanto, esse caminho deve ser tomado.

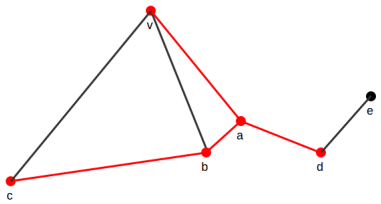


Figura 62 – Exemplo DFS etapa 5 (CORMEN et al., 2001)

Em d, há dois caminhos a seguir: $(d,\ a)$ e $(d,\ e)$. Porém a única aresta que leva a um vértice não visitado é $(d,\ e)$. Esta deverá ser seguida.

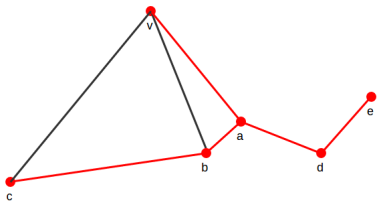


Figura 63 – Exemplo DFS etapa 6 (CORMEN et al., 2001)

Ao alcançar o vértice e, não existem vértices não visitados, mesmo na volta no caminho. Portanto, o percurso realizado pelo *DFS* pode ser observado na figura 64, o qual é uma árvore.

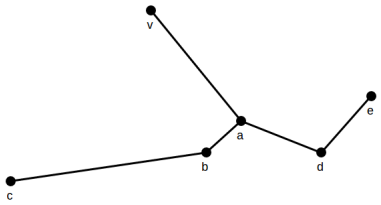


Figura 64 – Percurso do DFS (CORMEN et al., 2001)

B.1.2 BFS

O *DFS*, a partir de um vértice v, tenta visitar algum vizinho deste, em seguida, um vizinho deste vizinho, e assim por diante. O *BFS*, a partir de um vértice v, visita todos os vizinhos de v. Quando visitado todos os vizinhos de um vértice, os vizinhos destes vizinhos serão visitados (BRASSARD; BRATLEY, 1988).

A Figura 65 exemplifica o algoritmo da busca *BFS*, onde as letras significam o estado da busca e a ordem de cada estado. Os vértices marcados como vermelho representam que estes já foram visitados; os vértices marcados como azul representam que estes vértices serão os próximos a serem visitados.

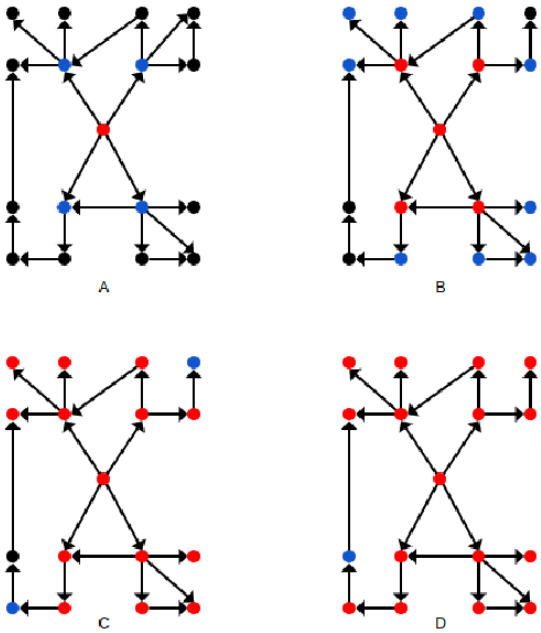


Figura 65 – Percurso do BFS (BRASSARD; BRATLEY, 1988)

APÊNDICE C – Reutilização de Software

C.1 Serviços

De uma maneira geral, serviços são atividades providas e realizadas por uma máquina ou um humano. No desenvolvimento de software, serviços são considerados um bloco de construção reutilizável que oferece uma funcionalidade particular. A noção de reusabilidade apresentada lembra a reutilização antes comentada no conceito de componentes. De fato, essas duas abordagens estão próximas, pois representam uma ideia de desenvolvimento comum, onde blocos de software são construídos para serem usados por atores diferentes e em outros locais (STOJANOVIC; DAHANAYAKE, 2005).

Com o advento dos serviços, surgiu um novo termo, a "Orientação a serviços". Essa ocorre quando alguns processos são modelados e construídos como unidades bem definidas e formam serviços. Esses são encapsulados em componentes de software para serem usados por outras aplicações, conferindo a ideia de um sistema provedor (o sistema que provê o serviço) e um sistema consumidor (que consome os serviços prestados) (VICTORINO; BRäSCHER, 2009).

Se todas as aplicações usassem a mesma interface de programação e o mesmo protocolo de interoperabilidade, todo trabalho de software seria mais simples. Essa é a premissa que trouxe o desenvolvimento orientado a serviços para o mundo da tecnologia da informação. Quando se desenvolve dessa forma, os serviços tornam-se a base para a criação de novas estratégias de solução (NEWCOMER; LOMOW, 2004).

Apesar das abordagens de orientação a serviços e orientação a componentes serem bem similares, no livro (STOJANOVIC; DAHANAYAKE, 2005), são apresentadas algumas diferenças entre as mesmas, que são:

- o tempo de integração. Na orientação baseada em componentes, a integração com a aplicação desenvolvida ocorre no momento da construção; enquanto, na orientação a serviços, a integração pode ocorrer antes ou durante a execução, pois apenas as descrições dos serviços estão disponíveis no momento da execução;

- nos serviços, o foco é a descoberta, existindo uma ênfase mais forte na separação entre a descrição do serviço e a implementação. Nos componentes, é a composição, onde as partes são integradas diretamente na aplicação;

- serviços são mais voltados para tarefas dinâmicas, ao contrário dos componentes que são mais voltados para pontos estáticos. Porém, esse comportamento dinâmico

pode ser feito também em componentes, e;

- a orientação a componentes atribui maior responsabilidade para o ambiente e a execução da própria aplicação. Os serviços não são necessariamente dessa forma, pois podem estar presentes em outras máquinas e outros servidores externos.

A seguir serão apresentadas duas formas conhecidas e já consolidadas de implementações de serviços, os *Web Services* e os *Serviços RESTful*.

C.1.1 Web Services

Os serviços são informatizados por tecnologias interoperáveis, que são capazes de se comunicar entre si, e isso independe da plataforma e da linguagem de programação utilizadas. Dentre essas tecnologias, os "*Web Services*" (serviços web) se destacam. Esses serviços fornecem um modo padronizado de integrar aplicativos web, e assim, organizações podem se comunicar sem que uma tenha conhecimento abrangente dos sistemas da outra (VICTORINO; BRäSCHER, 2009).

Um "*Web Service*" é uma noção abstrata que deve ser implementada por um agente concreto, sendo esse agente um pedaço de software que envia e recebe mensagens. O serviço é o recurso caracterizado pela funcionalidade abstrata que é servida.

O XML hoje é amplamente utilizado quase como o protocolo HTTP, e faz parte da solução para aplicação de "*Web Services*". Normalmente, usado para realizar a formatação dos dados (ALONSO et al., 2010).

Simplificando, os "*Web Services*" possuem três características e tecnologias principais que são usadas para desenvolvê-los: o XML, que é a especificação sobre a qual estes são construídos; o WSDL, que provê as interfaces fundamentais que serão usadas, e SOAP, que é responsável pela comunicação com outros serviços. Pode-se dizer que um "*Web Service*" deve fornecer interfaces de uso simples e se comunicar facilmente com outros serviços (NEWCOMER; LOMOW, 2004).

C.1.2 Serviços RESTFul

REST é um estilo arquitetural para sistemas hipermídia distribuídos, reunindo um grupo de critérios que serão incorporados ao projeto de aplicações distribuídas. Os Serviços RESTful são serviços web que seguem os critérios REST (FILHO, 2009).

O uso de serviços RESTFul permitiu grande impulso no desenvolvimento de aplicações distribuídas baseadas em padrões WEB tradicionais. Esses serviços são leves, práticos e fáceis de serem integrados em várias aplicações (ROSENBERG et al., 2008).

Os serviços da web estão a cada dia buscando soluções mais simples e mais leves para compor suas funcionalidades. Os serviços RESTFul introduzem um novo tipo de abstração que busca oferecer as funcionalidades já existentes dos serviços da *web* comuns. Porém, de uma forma mais leve e simples (PAUTASSO, 2009).

De acordo com (FILHO, 2009), os serviços RESTFul possuem cinco conceitos importantes, que são descritos a seguir:

- **recurso:** é uma abstração ou conceito relevante para o domínio tratado. Os recursos podem ser concretos ou abstratos, e têm-se maior flexibilidade para definição dos mesmos. Um serviço pode ter mais de um recurso sem afetar a sua qualidade;
- **representação:** é a representação de um recurso, apresentando informações sobre o mesmo. Dependendo do serviço e do nível de abstração, são mostradas mais ou menos informações dos recursos. A representação é uma serialização do recurso em uma linguagem específica, as mais usadas são: XML, XHTML e JSON. Um serviço pode mostrar mais de um tipo de serialização de seus recursos, basta que as requisições informem qual o formato desejado que o serviço irá apresentar os dados na serialização correta;
- **identificador uniforme:** cada recurso deve ter ao menos um identificador para localização do mesmo. Se não existe um identificador uniforme, não se pode considerar que um objeto é um recurso. Pode-se ter ilimitados identificadores para um mesmo recurso.
- **interface unificada:** A interface unificada diz respeito a todos os serviços utilizarem o mesmo protocolo, no caso, o HTTP. Dessa forma, um cliente que conheça esse protocolo pode facilmente utilizar os métodos que são proporcionados por ele para fazer requisições nos serviços disponíveis, e;
- **escopo de execução:** neste caso, o serviço RESTFul também se utiliza do protocolo HTTP, porém, diz respeito aos parâmetros que devem ser enviados junto da requisição para que o serviço possa tratar o recurso corretamente. Pode ser, por exemplo, a identificação de um objeto em um conjunto de recursos.

C.2 Padrão Abstract Factory

A seguir, têm-se uma breve descrição do padrão "*Abstract Factory*", como foi definido em (GAMMA et al., 1995).

Esse padrão fornece uma estrutura para criação de famílias de objetos relacionados sem a necessidade de definir suas classes concretas. A figura 66 apresenta o modelo do "*Abstract Factory*".

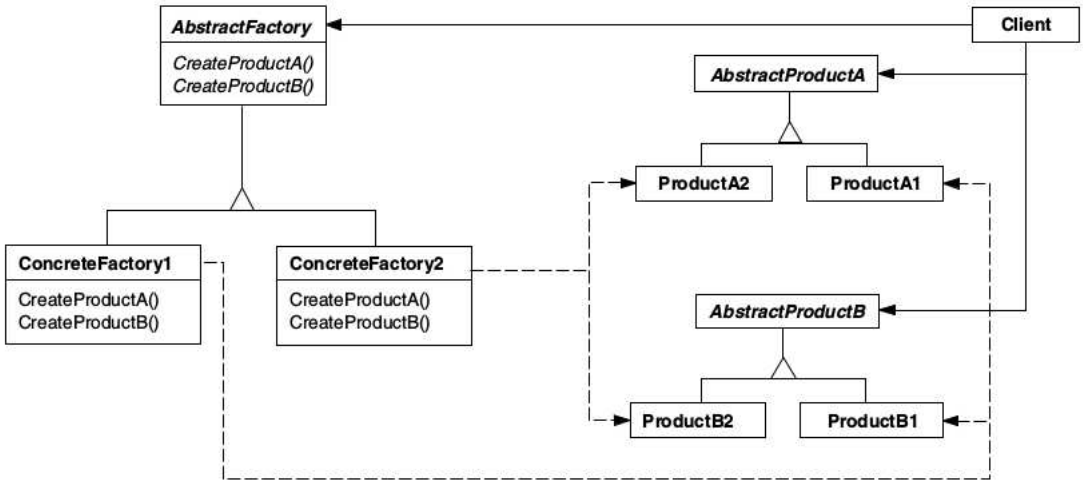


Figura 66 – Modelo genérico do Abstract Factory (GAMMA et al., 1995)

O modelo da figura apresenta cinco tipos de classes. "*Abstract Factory*", "*Concrete Factory*", "*Abstract Product*", "*Product*" e "*Client*".

- ***Abstract Factory:*** Faz declarações de interfaces para criação de quaisquer produtos;
- ***Concrete Factory:*** Essas classes já estão focadas no tipo de produtos que vão criar e implementam os métodos abstratos para essa criação;
- ***Abstract Product:*** Classes abstratas que declaram interfaces para um determinado tipo de produto que deverá ser criado;
- ***Product:*** Classes que representam o próprio produto que deverá ser criado, são as classes que são chamadas pelos métodos de criação presentes nas "*Concrete Factory*";
- ***Client:*** É a classe que representa quem irá fazer as chamadas aos métodos de criação. Não é necessário que conheça de fato as classes concretas de produto, pois apenas faz uso das interfaces declaradas em "*Abstract Factory*" e "*Abstract Product*"; a primeira para criar os produtos, e a segunda para usá-los.

Este padrão oferece algumas vantagens e desvantagens que são apresentadas a seguir:

- **Isolamento de classes concretas:** O cliente pode trabalhar com as criações dos produtos sem necessariamente conhecer as classes concretas que existem por traz, pois este trabalha apenas com as interfaces abstratas providas.

- **Fácil troca de famílias de produtos:** Basta trocar qual é a classe concreta que deverá ser usada que todo o comportamento dos produtos irá se alterar de acordo com essa classe. Isso pode ser feito facilmente no momento de instanciação da fábrica.
- **Harmonia entre produtos:** Como o padrão permite aos clientes trabalharem apenas com uma família por vez, fica fácil alcançar harmonia, pois todos os produtos da família estão de alguma forma relacionados.
- **Suporte a novos tipos de produtos é difícil:** Como a interface do "*Abstract Factory*", no início, cria uma quantidade fixa de produtos para serem implementados, alterar isso fica difícil, pois é necessário mexer na classe principal e criar as subclasses concernentes.

C.3 Padrão Factory Method

Um outro padrão de criação bem parecido com o Padrão Abstract Factory, porém, diferentemente do anterior que cria famílias de objetos, o *Factory Method* é responsável por criar um único objeto (GAMMA et al., 1995). A figura 67 apresenta o modelo de classes desse padrão:

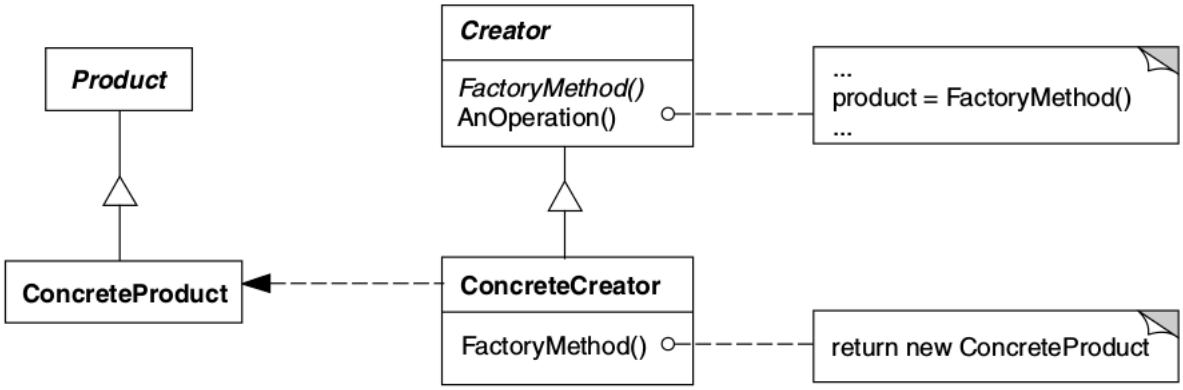


Figura 67 – Modelo genérico do Factory Method (GAMMA et al., 1995)

Com esse padrão fica fácil e prático fazer a alteração de uma sub-classe concreta, uma vez que, a aplicação passa a trabalhar com a interface de "*Product*", o código pode trabalhar com quaisquer classes "*ConcrectProduct*" definidas pelo usuário.

As classes mostradas no modelo são descritas a seguir:

- ***Product***: Interface de objetos que o método de fábrica irá criar;
- ***ConcretProduct***: Implementação da interface "*Product*";
- ***Creator***: Interface com método para criação de um "*Product*";
- ***ConcrectCreator***: Implementação da interface "*Creator*", reimplementa o método para criação de um "*ConcrectProduct*".

C.4 Padrão Strategy

O padrão "*Strategy*" foi desenvolvido para auxiliar desenvolvedores a trabalhar com famílias de algoritmos que podem variar independentemente dos clientes que os usam (GAMMA et al., 1995). Com esse padrão é possível mudar facilmente o comportamento de um dado algoritmo apenas trocando a classe que é usada na implementação do mesmo. A seguir é apresentado o modelo de classes desse padrão na figura 68.

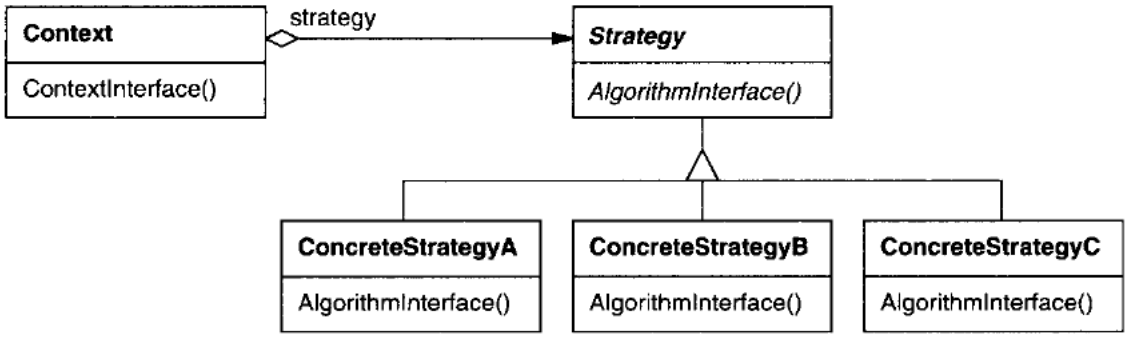


Figura 68 – Modelo genérico do Strategy (GAMMA et al., 1995)

As classes do modelo são descritas a seguir.

- ***Strategy***: Interface comum para todos os algoritmos que serão suportados.
- ***ConcretStrategy***: Implementação da interface "*Strategy*" definindo cada método com um comportamento esperado para um determinado cliente;
- ***Context***: Essa classe faz uso da interface "*Strategy*" chamando cada um de seus métodos através de um objeto "*ConcretStrategy*" que é definido, sendo assim, o comportamento esperado será retornado de acordo com o "*ConcretStrategy*" que é passado.

APÊNDICE D – Classificações de metodologias

Metodologias específicas colaboram para estabelecer diretrizes e boas práticas na condução do trabalho, conferindo padronização, noções de pesquisa científica, dentre outras contribuições (WOHLIN et al., 2000).

A metodologia pode ser entendida como um conjunto de etapas a serem realizadas na condução do processo investigativo em um determinado contexto. É a metodologia que define os passos que serão seguidos para realização das pesquisas, escolha do tema, planejamento da investigação, desenvolvimento, coleta e análise dos dados, análise dos resultados e conclusões a respeito das lições aprendidas (MORESI, 2003).

Pesquisar significa identificar uma dúvida que necessite ser esclarecida, construir e executar o processo que apresente a solução desta, quando não há teorias que a expliquem ou quando as teorias que existem não estão aptas para fazê-lo (KOCHE, 1997). A seguir, serão apresentadas as formas de se classificar uma pesquisa.

D.1 Classificação da pesquisa

- Do ponto de vista da natureza da pesquisa, esta pode ser:
  - **Pesquisa Básica:** Possui o objetivo de gerar novos conhecimentos para a ciência. Neste tipo de pesquisa não é obrigatório que o conhecimento gere um uso prático (TAFNER; SILVA, 2007).
  - **Pesquisa Aplicada:** Visa gerar uma maior compreensão para assuntos práticos dirigidos à solução de problemas específicos (TAFNER; SILVA, 2007).
- Do ponto de vista da forma de abordagem do problema,tem-se:
  - **Pesquisa Quantitativa:** O estudo quantitativo considera que tudo pode ser quantificável, ou seja, que os números podem ser classificados, gerando informações ao analisá-los, através da análise estatística (TRAVASSOS, 2002).
  - **Pesquisa Qualitativa:** O estudo qualitativo está relacionado à pesquisa sobre os objetos, quando os resultados são apresentados em termos naturais, usando conjuntos nebulosos, porcentagens de satisfação, dentre outras formas de classificar algo subjetivo (TRAVASSOS, 2002).
- Do ponto de vista de seus objetivos, tem-se:

  - **Pesquisa Exploratória:** Esta pesquisa tem como objetivo proporcionar maior familiaridade com o problema, possibilitando o aprimoramento de ideias ou a descoberta de intuições (GIL, 2010).
  - **Pesquisa Descritiva:** Seu principal objetivo é a descrição das características de determinada população ou fenômeno ou, então, o estabelecimento de relações entre variáveis (GIL, 2010).
  - **Pesquisa Explicativa:** Tem como preocupação central identificar os fatores que determinam ou que contribuem para a ocorrência dos fenômenos. Esse é o tipo de pesquisa que mais aprofunda o conhecimento da realidade, porque explica a razão, o porquê das coisas (GIL, 2010).

- Do ponto de vista dos procedimentos técnicos, pode ser:
  - **Pesquisa Bibliográfica:** Visa encontrar as fontes primárias e secundárias e os materiais científicos e tecnológicos necessários para a realização do trabalho científico ou técnico-científico. Muitos estudos fazem uso do levantamento bibliográfico ou são desenvolvidas exclusivamente por fontes bibliográficas. Sua principal vantagem é possibilitar ao investigador a cobertura de uma gama de acontecimentos muito ampla (TAFNER; SILVA, 2007).
  - **Pesquisa Documental:** Assemelha-se à pesquisa bibliográfica. Porém, esta é realizada a partir de materiais que não receberam tratamento analítico. Por exemplo, reportagens de jornal, cartas, contratos, diários, filmes, fotografias e gravações. A pesquisa documental pode ser realizada ainda através de documentos de segunda mão, que de alguma forma já foram analisados. Por exemplo, relatórios de empresas e tabelas estatísticas (GIL, 2010).
  - **Levantamento:** É uma investigação realizada em retrospecto, que em seguida, mediante análise quantitativa, chega às conclusões correspondentes aos dados coletados. O levantamento feito com informações de todos os integrantes do universo da pesquisa origina um censo. (MAFRA; TRAVASSOS, 2006).
  - **Estudo de Caso:** São estudos conduzidos com o propósito de se investigar uma entidade ou um fenômeno dentro de um espaço de tempo específico. Estes são usados principalmente para a monitoração de atributos presentes em projetos, atividades ou atribuições. Durante a sua condução, dados são coletados e analisados estatisticamente de forma a permitir a avaliação de um determinado atributo ou do relacionamento entre diferentes atributos. (MAFRA; TRAVASSOS, 2006)
  - **Pesquisa-Ação:** É realizada em conjunto com uma ação ou com a resolução de um problema coletivo, visando definir o campo de investigação, as expectativas dos interessados e o tipo de auxílio que estes poderão exercer ao longo

do processo de pesquisa. Esta pesquisa implica no contato direto com o campo de estudo, envolvendo o reconhecimento visual do local, consulta a documentos diversos e a discussão com os envolvidos na pesquisa. A abordagem dos problemas dos grupos investigados na pesquisa-ação é mais qualitativa do que quantitativa (TAFNER; SILVA, 2007).

– **Pesquisa Participante:** A intenção é obter um maior conhecimento sobre o grupo. O grupo investigado tem ciência da finalidade, dos objetivos da pesquisa e da identidade do pesquisador. A pesquisa participante permite a observação das ações no próprio momento em que ocorrem (TAFNER; SILVA, 2007).

– **Pesquisa Experimental:** É conduzida quando deseja-se obter um maior controle da situação, ao se manipular as variáveis envolvidas no estudo de forma direta, sistemática e precisa. A pesquisa experimental necessita de previsão de relações entre as variáveis a serem estudadas, como também o seu controle. O objetivo é manipular uma ou mais variáveis e controlar todas as outras variáveis em um valor fixo. O efeito da manipulação das variáveis é então medido e, baseado nessa medição, análises estatísticas são conduzidas. A condução de experimentos reais é rara em Engenharia de Software, devido à dificuldade de se alocar os participantes do estudo a diferentes tratamentos de forma aleatória. Nessas situações, tais estudos denominam-se quasi-experimentos (MAFRA; TRAVASSOS, 2006).

– **Pesquisa Ex-Post-Facto:** Realizada quando o experimento se dá depois dos fatos. Neste caso, o pesquisador não tem controle sobre as variáveis. Esta pesquisa difere da pesquisa experimental pelo fato de o fenômeno ocorrer naturalmente sem que o pesquisador tenha controle sobre ele, ou seja, o pesquisador passa a ser um mero observador do acontecimento (TAFNER; SILVA, 2007).

APÊNDICE E – Prova de Conceito

Foi desenvolvida uma pequena aplicação para implementação de alguns conceitos discutidos neste trabalho.

De modo geral, a aplicação desenvolvida implementa uma rede de usuários ligados entre si, formando um grafo. A solução contêm as classes *User*, *Edge* e *Graph*. A classe *Edge* é usada para fazer as ligações entre as entidades. A classe *Graph* representa a própria rede com todos os usuários. A classe de *User* pode ser vista como os vértices do grafo. Foram implementadas as funcionalidades de relacionamento descritas nas figuras 18 e 19, além de uma varredura dos usuários presentes no grafo pelos algoritmos de busca BFS, *Breadth-First Search*, e DFS, *Depth-First Search*.

A Figura 69 apresenta o modelo das classes desenvolvidas na aplicação de prova de conceito e seus respectivos relacionamentos.

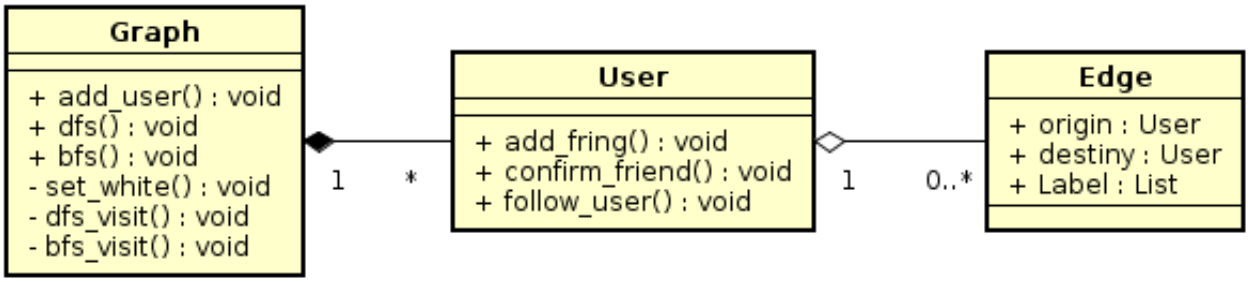


Figura 69 – Classes da prova de conceito

Fica exemplificado o relacionamento entre as classes *Graph* e *User* na forma de uma composição, que indica que um usuário não pode existir sem estar presente na rede. O relacionamento entre usuários, como pode ser visto na Figura 69, é feito através das arestas que possuem: (i) um atributo *origin*, que representa de qual usuário parte o relacionamento e (ii) um atributo *destiny*, que representa a qual usuário é indicado o relacionamento criado. Cada aresta possui ainda uma lista de *labels* que representam os nomes dos relacionamentos entre os usuários.

Pode-se ver ainda na Figura 69 a apresentação dos métodos construídos. A classe *User* possui os métodos responsáveis por estabelecer os relacionamentos entre usuários, conforme apresentado nas figuras 18 e 19. O método *"follow_user"* é responsável por criar um relacionamento unidirecional, que representa seguir um usuário. O método *"add_friend"* é responsável por relacionamentos bidirecionais. Este último, por sua vez, necessita ainda ser complementado pelo método *"confirm_friend"*, pois, ao se invocar o método *"add_friend"* cria-se uma aresta que possui o *label* amigo e pendente, o que indica

que o relacionamento ainda não está completo. Ao se invocar o método *"confirm_friend"*, outra aresta é criada no outro sentido e o *label* pendente é removido, completando-se assim, o relacionamento entre os dois usuários.

A classe *Graph* possui o método *"add_user"*, que é chamado sempre na criação de um novo usuário, visando adicionar este a rede. O método *"set_white"* tem como objetivo definir a cor de todos os usuários no grafo para branco. Esse atributo é usado ao se trabalhar com os recursos de buscas no grafo. Colorir o nó com a cor branca significa que este ainda não foi visitado. O método *"dfs"* implementa o algoritmo de busca DFS e faz uma varredura no grafo. O mesmo acontece para o método *"bfs"* que implementa a varredura de acordo com o algoritmo BFS. Os métodos *"dfs_visit"* e *"bfs_visit"* são métodos auxiliares que visitam o nó corrente e definem o próximo nó que será visitado de acordo com as regras definidas nos algoritmos.

A abordagem de lista de adjacência foi utilizada na implementação do grafo. Como as arestas trabalhadas possuem outras informações além da representação das ligações entre os nós, essa abordagem ocupa menos espaço em memória. Isso ocorre pois não existe representação de arestas que não estejam presentes no grafo. Além da redução da memória gasta, o tempo gasto para listar todos os nós que fazem ligação a um outro nó é reduzido, quando se usa lista de adjacência, pois basta realizar a leitura da lista para se obter essas informações. Já na matriz de adjacência e incidência, é necessário realizar uma busca em uma linha da matriz para verificar todas as ligações que um nó possui. O código fonte dessa aplicação pode ser encontrado em <https://github.com/TCC-SocialNetwork/concept-test>.

APÊNDICE F – Sprints

Neste capítulo serão apresentadas todas as estórias de usuário que foram definidas e desenvolvidas ao longo da construção do SocialFramework e SocialBike. O período de desenvolvimento compreende as datas de 07/03/2016 a 21/06/2016. Durante esse período foram estabelecidas oito *sprints* com duas semanas cada uma, com exceção da última que ficou com apenas uma semana.

A seguir serão apresentadas tabelas com as respectivas estórias, pontuação e status. Pode-se verificar que todas as estórias estão completas e foram desenvolvidas para os sistemas propostos.

Tabela 3 – Sprint 1 - Período de 07/03 a 19/03

Id	**Estória**	**Pontos**	**Status**
1	Como desenvolvedor gostaria de fornecer cadastro de usuários para ser usado em aplicações que utilizem o framework.	3	Completa
2	Como desenvolvedor gostaria de fornecer autenticação de usuários para ser usada em aplicações que utilizem o framework.	5	Completa
3	Como desenvolvedor gostaria de fornecer funcionalidades para relacionamento entre usuários para ser usada em aplicações que utilizem o framework.	13	Completa
4	Como desenvolvedor gostaria que o framework monte uma rede com os usuários cadastrados apresentando todos os relacionamentos e vínculos entre os mesmos para facilitar consultas.	13	Completa

Tabela 4 – Sprint 2 - Período de 20/03 a 03/03

Id	Estória	Pontos	Status
5	Como desenvolvedor gostaria de fornecer pesquisas para encontrar algum elemento presente na rede social.	13	Completa
6	O framework deve ser capaz de sugerir amigos a um usuário para que este possa aumentar sua rede de relacionamentos.	13	Completa
8	Como desenvolvedor gostaria que o framework forneça scripts que auxiliem no uso do mesmo para suporte nas configurações e gerações de arquivos de acordo com um template definido.	5	Completa
9	Como desenvolvedor gostaria de fornecer um script que gere as Views de usuários para acesso das suas funcionalidades.	3	Completa

Tabela 5 – Sprint 3 - Período de 20/03 a 17/04

Id	Estória	Pontos	Status
7	Como desenvolvedor gostaria que o Framework possibilite ao usuário a adição ou remoção de atributos das classes de modelo para mudar conforme as suas necessidades.	8	Completa
12	Como desenvolvedor gostaria que o framework possibilite, na montagem do grafo, passar quais serão os atributos que os vértices terão para facilitar em pesquisas futuras.	3	Completa
13	Como desenvolvedor gostaria que o framework possibilite a continuação de pesquisas no Grafo mesmo após ter retornado a quantidade padrão de resultados para visualização de mais resultados.	13	Completa
14	Como desenvolvedor gostaria de fazer algumas refatorações para garantir a integridade do Módulo de Usuários.	3	Completa
21	Como desenvolvedor gostaria que o Framework possibilite a definição de um período para buscar o melhor horário para marcação de um evento entre um conjunto de usuários.	8	Completa

Tabela 6 – Sprint 4 - Período de 18/04 a 01/05

Id	Estória	Pontos	Status
10	Como desenvolvedor gostaria de fornecer uma documentação do framework para facilitar o uso do mesmo.	3	Completa
11	BUG: Como desenvolvedor gostaria de especificar relacionamentos como unidirecionais ou bidirecionais para detalhar melhor.	5	Completa
15	Como desenvolvedor gostaria que o Framework possibilite a qualquer usuário checar a sua disponibilidade antes de aceitar participar ou não de um evento para maior controle.	3	Completa
16	Como desenvolvedor gostaria que o Framework possibilite tornar membros como “Administradores” em um evento para que possam gerenciar e/ou convidar mais pessoas para o evento em questão.	3	Completa
18	Como desenvolvedor gostaria que o Framework possibilite verificar o horário que contemple o maior peso de um conjunto de usuários para marcação de um evento.	20	Completa

Tabela 7 – Sprint 5 - Período de 02/05 a 15/05

Id	Estória	Pontos	Status
17	Como desenvolvedor gostaria que o Framework possibilite convidar qualquer um dos meus relacionamentos para um determinado evento para um melhor controle.	3	Completa
19	Como desenvolvedor gostaria que o Framework possibilite definir usuários 'Fixos' em um evento que deverá ser marcado para garantir que esses usuários poderão estar presentes.	13	Completa
20	Como desenvolvedor gostaria que o Framework possibilite a definição de um peso para os usuários para auxiliar no casamento de horários buscando o maior peso.	3	Completa
24	Como desenvolvedor gostaria que o Framework possibilite aos usuários a criação de agendas para um maior controle de suas participações em qualquer evento.	1	Completa
25	Como desenvolvedor gostaria que o Framework realizasse pesquisa em eventos levando em consideração os usuários presentes na rede de um usuário para aumentar os elementos das pesquisas.	8	Completa
26	Como desenvolvedor gostaria que o Framework possibilite a qualquer usuário participar de um evento público para facilitar a divulgação do mesmo.	2	Completa
27	Como desenvolvedor gostaria que o Framework possibilite ao usuário criador de um evento remover o mesmo para cancelamento.	2	Completa

Tabela 8 – Sprint 6 - Período de 16/05 a 29/05

Id	Estória	Pontos	Status
22	Como desenvolvedor gostaria que o Framework possibilite a criação de eventos definindo a data, hora e duração quando for o caso para anunciar o mesmo a outros usuários.	8	Completa
28	Como desenvolvedor gostaria que o Framework possibilite ao usuário Criador e aos Administradores de um evento remover convidados para manter um melhor controle.	2	Completa
29	Como desenvolvedor gostaria de fornecer uma documentação do Módulo de Agenda para auxiliar no uso do mesmo.	3	Completa
30	Como desenvolvedor gostaria que o Framework possa salvar rotas para os usuários para que possam comparar com outros.	3	Completa
32	Como desenvolvedor gostaria que o Framework possibilite relacionar rotas com um evento para verificar os horários.	5	Completa
35	Como desenvolvedor gostaria que o Framework verifique se duas rotas são compatíveis para sugerir aos usuários.	5	Completa
36	Como desenvolvedor gostaria que o Framework sugira uma pequena alteração de rotas para se adequar a rotas de outros usuários.	5	Completa
37	Como desenvolvedor gostaria de fornecer uma documentação do Módulo de Rotas para auxiliar no uso do mesmo.	3	Completa

Tabela 9 – Sprint 7 - Período de 30/05 a 12/06

Id	Estória	Pontos	Status
34	Como desenvolvedor gostaria que o Framework calcule a menor distância entre dois pontos dentro de um raio pré-definido para obter os pontos com a menor distância.	8	Completa
40	BUG: Como desenvolvedor gostaria de ajustar retorno da pesquisa quando já está finalizada para que o retorno seja consistente.	1	Completa
41	Como desenvolvedor gostaria de adicionar o Padrão Strategy para os helpers do framework para criar um ponto flexível no framework.	5	Completa
42	Como desenvolvedor gostaria de adicionar o padrão Abstract Factory para os Vértices e Arestas para criar um ponto flexível no framework.	3	Completa
43	Como desenvolvedor gostaria de colocar os atributos para construção dos vértices no grafo no arquivo de inicialização do social framework.	1	Completa
45	Como desenvolvedor gostaria de adicionar o padrão factory method para as classes de modelo para criar um ponto flexível no framework.	3	Completa
46	Como desenvolvedor gostaria de criar método em Evento que retorne os usuários confirmados ou não do evento para saber os usuários que realmente estão participando do evento.	2	Completa
47	Como desenvolvedor gostaria de tornar método “execute_action?” do evento público para que seja possível a sua utilização fora de sua classe.	2	Completa

Tabela 10 – Sprint 8 - SocialBike - Período de 13/06 a 21/06

Id	Estória	Pontos	Status
1	Como usuário gostaria de me registrar para ter acesso a rede social.	2	Completa
2	Como usuário gostaria de me relacionar com outros usuários rede para interagir com a minha rede de contatos.	2	Completa
4	Como usuário gostaria de criar eventos para outros usuários participarem.	3	Completa
5	Como usuário gostaria de pesquisar outros usuários e eventos na rede para me relacionar.	3	Completa
6	Como usuário gostaria de ver usuários sugeridos para facilitar meus relacionamentos.	2	Completa
7	ABERTA: Como usuário gostaria de ver um horário em que o maior peso de um conjunto de usuários possa participar para marcar um evento.	3	Completa
8	Como usuário gostaria de criar uma rota para melhor controle dos caminhos percorridos.	8	Completa
9	ABERTA: Como usuário gostaria de definir um horário de partida para uma rota para melhor controle dos meus compromissos.	3	Completa
10	ABERTA: Como usuário gostaria de verificar se algum usuário em um grupo tem uma rota compatível com uma rota específica.	5	Completa