UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

EDUARDO MATTOS DA SILVA MELLO

Gitabs: Uma extensão ao sistema Git para gestão de projetos

Trabalho de Graduação

Prof. Dr. Dante Augusto Barone
Orientador

Porto Alegre, dezembro de 2013.

AGRADECIMENTOS

Agradeço aos meus pais, Oberon e Regina e a minha irmã, Daniela, pelo incentivo, apoio ao longo de todo o curso. A minha namorada, Ivana Rebeschini, que tem sido minha grande amiga e companheira nos últimos anos, além da paciência e apoio incondicional.

Ao meu irmão Andres, e novamente meu pai, Oberon e minha namorada Ivana, pela importante ajuda ao revisar meu trabalho.

Aos meu sócios Alvaro, Diego, Pedro e Samir pela paciência e compreensão durante meus períodos de aula nos últimos dois anos.

Aos profissionais que estiveram ao meu lado e mantiveram os projetos rodando enquanto finalizava o curso.

Ao Prof. Dante e ao Vinícius Woloszyn pela orientação ao longo do desenvolvimento deste trabalho.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS 6

LISTA DE FIGURAS 7

LISTA DE TABELAS 8

RESUMO 9

ABSTRACT 10

1 INTRODUÇÃO 11

1.1. Objetivo 12

1.2. Estrutura do trabalho 13

2 DOCUMENTOS E FERRAMENTAS EM GESTÃO DE PROJETOS 14

3 CONTROLE DE VERSÃO E O SISTEMA GIT 20

4 IMPLEMENTAÇÃO 25

4.1 Metodologia 25

4.1.1 Desenvolvimento Dirigido a Testes 25

4.1.2 Desenvolvimento Dirigido a Comportamento 26

4.1.3 Definição de metodologia 28

4.2 Definição de tecnologias 30

4.3 Análise de requisitos 30

4.4 Criação de meta-ramificações 31

4.5 Criação de meta-dados 34

4.6 Execução de tarefas 36

4.7 Entrega de tarefas 39

4.8 Uma camada para interação com o sistema Git 41

5 RESULTADOS 44

6 CONCLUSÃO..... 46

6.1 Possibilidades futuras..... Error! Bookmark not defined.

REFERÊNCIAS 48

APÊNDICE A: FUNCIONALIDADES NA FERRAMENTA *CUCUMBER*..... 50

APÊNDICE B: COMPORTAMENTOS NA FERRAMENTA *MINITEST* 59

LISTA DE ABREVIATURAS E SIGLAS

TI	Tecnologia da Informação
TDD	Test-driven Development
BDD	Behavior-driven Development
VCS	Version Control System
CVS	Concurrent Version System
XP	Extreme Programming
ATDD	Automated Test-driven Development
ANSI	American National Standards Institute
JSON	JavaScript Object Notation
REST	Representational state transfer
API	Application Programming Interface

LISTA DE FIGURAS

Figura 2.1: Modelo em cascata e iterativo.....15
Figura 2.2: Modelos por prototipação e espiral.....16
Figura 2.3: Um cartão de histórias.....17
Figura 2.4: Fluxo do processo *Scrum*.....18
Figura 3.1: Um exemplo de ramificação.....21
Figura 3.2: Um repositório Git com uma ramificação.....22
Figura 3.3: Um repositório Git com duas ramificações.....23
Figura 4.1: ciclo BDD.....29
Figura 4.2: Definição do cenário no *Cucumber* para criação de meta-ramificação.....32
Figura 4.3: Testes para criação de meta-ramificação no *Minitest*.....33
Figura 4.4: Código da aplicação para o comando metabranch.....34
Figura 4.5: Definição de cenário na ferramenta Cucumber para inserção de meta-dados.....35
Figura 4.6: Testes para criação de meta-dados no *Minitest*.....36
Figura 4.7: Descrição de cenário no Cucumber para execução de uma tarefa.....37
Figura 4.8: Testes de aceitação para execução de tarefa no *Minitest*.....38
Figura 4.9: Descrição de cenário no Cucumber para entrega de uma tarefa.....40
Figura 4.10: Testes de aceitação para entrega de tarefas no *Minitest*.....40
Figura 4.11: Resultado de refatoração.....42
Figura 4.12: Diagrama de classes final da biblioteca *Gitabs*.....43

LISTA DE TABELAS

Tabela 2.1: Uso e preferências de ferramentas para métodos ágeis.....19
Tabela 2.2: Uso especifico de ferramenta de gerenciamento de projetos ágeis.....19
Tabela 4.1: Funcionalidade ‘Gerente de Projeto manipula meta-ramificações’.....31
Tabela 4.2: Funcionalidade ‘Colaborador manipula meta-dados’.....35
Tabela 4.3: Funcionalidade ‘Colaborador executa tarefa’.....37
Tabela 4.4: Funcionalidade ‘Colaborador entrega tarefa’.....39
Tabela 5.1: Resultados da ferramenta *Cucumber*.....44
Tabela 5.2: Resultados da ferramenta Minitest.....45

RESUMO

O objetivo deste trabalho é desenvolver uma biblioteca que possa ser utilizada por aplicações que auxiliem a gestão de projetos. Sua construção se deu de forma genérica possibilitando a construção de diferentes abordagens em gestão de projetos e até ferramentas mais amplas de armazenamento de dados, como banco de dados baseado em documentos.

Atualmente a gestão eficiente de um projeto é determinante para o sucesso do mesmo. Sendo assim, gerentes utilizam diversas metodologias e ferramentas para auxiliá-lo nas suas tarefas ao longo de um projeto. Quaisquer que sejam as medidas adotadas por um gerente, sempre haverá necessidade de comunicação com o time de desenvolvimento. Este time, por sua vez, utiliza sistemas para controle de versão de seu código. Estes sistemas funcionam como um banco de dados para manter um histórico de modificações de um projeto. Desta forma, as informações necessárias para a execução do projeto (escopo, tarefas, requisitos, diagramas, entre outros) estão desvinculadas do código que foi produzido a partir delas. O presente trabalho apresenta uma extensão para o sistema de controle de versões *Git* que possibilite utilizá-lo como uma ferramenta para gestão de projetos.

Palavras-Chave: gerência de projetos, sistema de controle de versões, git, BDD, ruby.

Gitabs: a Git extension for Project management

ABSTRACT

The goal of this project is to build a library that serves as a platform for new project management tools. The library was built in a way that it suits different project management approaches and even broader applications, like document based databases may be built using it.

Nowadays, managing a Project efficiently is very important for its success. Therefore, managers uses different methodologies and tools to help them on executing the necessary tasks along the project. No matter what methodology or tool was chosen by the manager, there will always be the need to communicate with the development team. This team utilizes a control version system to keep track on how its code evolved. Then, the necessary information for the project execution (scope, tasks, requirements, and diagrams, among others) are disconnected from the produced code that were generated because of them. This work presents an extension for the version control system, Git, so it can be used as a project management tool.

Keywords: project management, version control system, git, BDD, ruby.

1 INTRODUÇÃO

Qualquer projeto que seja desenvolvido por uma equipe gera, ao longo do seu curso de vida, uma série de informações - como escopo de um projeto e requisitos mínimos, diagramas - externas ao produto gerado. Estas informações variam dependendo do projeto ou da equipe mas, na sua essência, existem para guiar o desenvolvimento do projeto e documentar o produto final. Em engenharia de software podemos encontrar muitas formas de abordar a gestão de um projeto mas, mesmo que elas apresentem significativas diferenças sobre o que e como deve-se documentar um projeto, invariavelmente surgirão informações vitais que precisarão ser persistidas.

Geralmente estas informações existem em ferramentas externas (físicas ou digitais) ao sistema desenvolvido e a equipe deve relacioná-las manualmente ao que foi produzido. Isso, naturalmente, pode gerar um desgaste no projeto – com informações desatualizadas – ou na equipe, sacrificando um tempo excessivo para manutenção destas ferramentas.

Neste cenário, surgiram diversas ferramentas para auxiliar na gestão de um projeto. Entre as quais podemos citar as de controle de versão de que auxiliam a manutenção de um projeto e o processo de desenvolvimento. Com elas é possível efetuar atualizações paralelas de um mesmo arquivo, além de armazenar um histórico de entregas por parte dos colaboradores envolvidos. Entre estas ferramentas podemos citar o sistema de controle de versões *Git*, que tem sido uma das mais aplicadas nos últimos anos, Este apresenta todas capacidades esperada de um sistema com este propósito como controle de versões de um arquivo e capacidades de fusão de arquivos. Além disso possui uma implementação que facilita a criação de ramificações e funciona de forma distribuída (CHACON, 2009).

Sistemas de controle de versão, hoje em dia, é imprescindível em projetos de

desenvolvimento de software (MASON, 2010). Mesmo assim, carecem de sistemas complementares para determinação de tarefas, metas e controle de erros. Alguns sites, como o *GitHub*[1], procuram oferecer estas capacidades junto as suas funcionalidades principais. Porém, apresentam uma estrutura de dados pouco flexível do ponto de vista do usuário. Além disso, dependem de uma gestão manual entre o repositório e informações adicionais. Existem ainda, alguns projetos, como *ticgit-ng*[2], para controle de erros direto nos repositórios *Git* mas com estrutura especifica e imutável.

Sendo assim, mesmo com o significativo avanço recente, tanto em metodologias como em ferramentas, ainda é possível uma maior aproximação entre a gestão e a execução de projetos. Neste contexto, uma solução prática que unifique estes dois universos seria uma peça valiosa para um maior controle durante o andamento de um projeto.

1.1 Objetivo

O objetivo deste trabalho é a implementação de uma biblioteca de operações que capacitem o sistema *Git* a unificar informações externas a um projeto.

Esta biblioteca se chamará *Gitabs* e tem, para os fins propostos, quatro requisitos: a definição dos tipos de dados a serem armazenados, a inserção de dados, a execução de tarefas e a entrega de tarefas. Sendo assim, um gerente de projetos poderá registrar, junto ao repositório de um projeto, informações externas ao conjunto de arquivos do software. Já um colaborador poderá executar tarefas e entregá-las partindo de uma informação inserida pelo gerente de projetos.

O sistema *Git*, em seu funcionamento padrão, organiza as alterações dos arquivos em forma de um grafo acíclico dirigido e a biblioteca *Gitabs* se valerá disso para possibilitar uma visualização global do projeto. Isso facilitará a recuperação de informações para análise continua ou posterior de um projeto.

---

[1] *GitHub*, http://www.github.com, é uma rede social de compartilhamento de código. Mais de 4 milhões de usuários colaboram em diferentes projetos utilizando sua hospedagem de repositórios *Git*.

[2] *ticigit-ng*, https://github.com/jeffWelling/ticgit, é um projeto que possibilita a criação de tíquetes de erros junto ao código em determinado repositório.

Por fim, a biblioteca *Gitabs* pretende ser uma base para construção de ferramentas para gestão de projetos. Desta forma, para validar o funcionamento da biblioteca, será desenvolvida uma aplicação que apresentará uma interface para as funcionalidades propostas pelo trabalho.

1.2 Estrutura do trabalho

O trabalho tem a seguinte estrutura: após a introdução, apresentando o objetivo do trabalho, com o capítulo atual, o segundo e terceiro capítulos apresentam uma contextualização em gerência de projetos e em sistemas de controle de versão, respectivamente, sendo que este último aprofunda-se no funcionamento do sistema *Git*.

O capítulo 4, apresenta conceitos sobre Desenvolvimento dirigido a testes (do inglês, TDD) e Desenvolvimento dirigido a comportamento (do inglês, BDD), juntamente com uma justificativa para a escolha da metodologia. Ainda no capitulo 4, é apresentado como cada requisito do trabalho foi abordado através do uso da metodologia escolhida.

Já o capitulo 5, apresenta resultados que visam mostrar o que foi atingido com o desenvolvimento do projeto. Por fim, o capitulo 6 conclui o trabalho e apresenta possibilidades futuras.

2 DOCUMENTOS E FERRAMENTAS EM GESTÃO DE PROJETOS

Segundo o Project Management Institute, "Gerenciamento de Projetos[...] é a aplicação de conhecimentos, habilidades e técnicas para a execução de projetos de forma efetiva e eficaz" (INSTITUTE, 2013). Alguns dos principais objetivos para gestão de projetos são: entrega do software dentro do prazo estabelecido, administração do custo conforme orçamento, atender as expectativas do cliente e administrar o ânimo do time de desenvolvimento (SOMMERVILLE, 2010).

Portanto, o gerente de projeto, deve ser uma pessoa com experiência e capacidade para não perder o controle de nenhuma destas variáveis. O que, por vezes, acaba ocorrendo. De fato, o Standish Group, formado em 1985, analisa casos reais em TI aponta que 18% dos projetos foram terminados antes de sua conclusão e que 43% foram entregues atrasado, acima do orçamento ou incompletos (MANIFESTO, 2013).

Segundo Pressman (2010), um gerenciamento efetivo tem foco nos 4 P's: pessoas, produto, processo e projeto. No que diz respeito a pessoas, a engenharia de um software é um empreendimento essencialmente humano e é possível identificar alguns papéis: *Stakeholders*, Gerentes de projetos e o time de desenvolvimento. Desta forma, podemos identificar um dos problemas para casos de insucesso: problemas de coordenação e comunicação entre as partes envolvidas (PRESSMAN, 2010). No âmbito formal, isto ocorre por meio de "comunicação escrita, reuniões estruturadas e de outros canais de comunicação relativamente não interativos e impessoais" (KRAUL apud PRESSMAN, 2010, p. 573).

Quanto ao produto, é necessário ter controle sobre ele e o que ele necessita fazer. Para que isto seja atingido, se faz necessário a definição de um escopo. Este escopo deve descrever de forma clara, para todos interessados, o contexto em que o software estará inserido, os objetivos a serem atingidos com o desenvolvimento do software, restrições e limitações, e fatores de mitigação (PRESSMAN, 2010).

Antes que o último P, o projeto, possa ser iniciado efetivamente, deve ser escolhido pelo time um modelo de processo que seja mais apropriado para a situação. Atualmente, existem diversas opções que foram desenvolvidas ao longo das últimas décadas. Em situações mais lineares e que os requisitos para solução de um problema são bem compreendidos é possível aplicar um modelo em cascata, onde cada etapa depende diretamente do fim da anterior. Para projetos com um escopo indefinido, procura-se aplicar um modelo incremental. Neste modelo, são aplicadas sequências lineares uma após a outra (PRESSMAN, 2010). Ambos apresentam três etapas essencialmente de documentação: Comunicação, com levantamento de necessidades; Planejamento, com estimativas e cronogramas; e Modelagem, com análise e projeto.

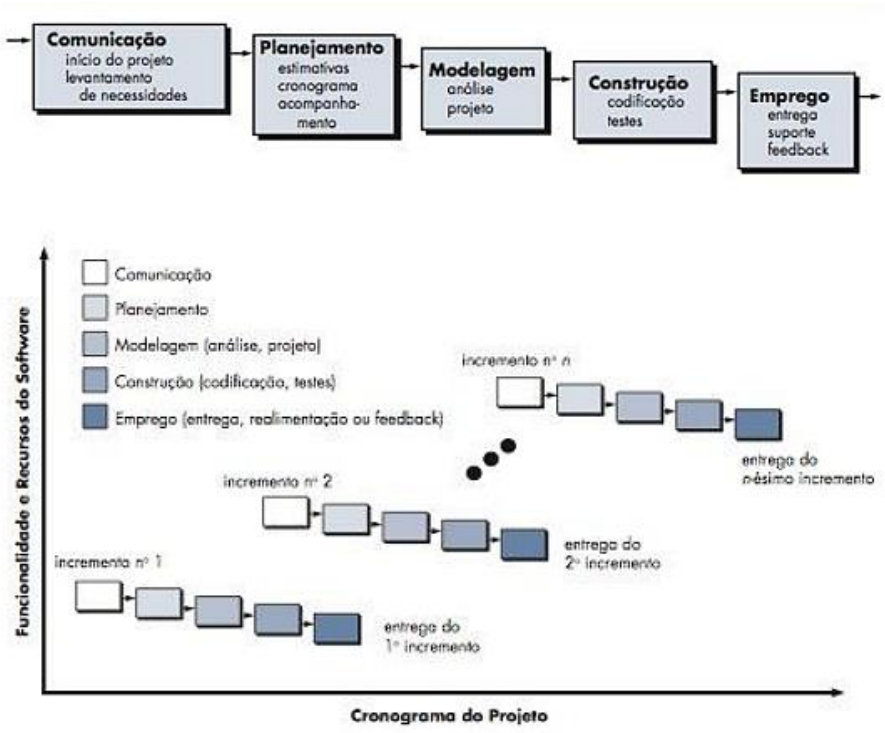


Figura 2.1: Modelo em cascata e iterativo. (PRESSMAN, 2010).

Alguns projetos, mesmo com uma versão entregue, mantém-se vivos com o acréscimo de novas funcionalidades e revisão de funcionalidades existentes. Para estes casos recomenda-se o uso de um modelo evolucionário. Estes podem ser vistos a partir do paradigma de prototipação, onde é desenvolvido um rápido projeto para auxiliar nas definições de projeto, ou no modelo espiral – que procura casar conceitos iterativos com o controle do modelo em cascata (PRESSMAN, 2010).

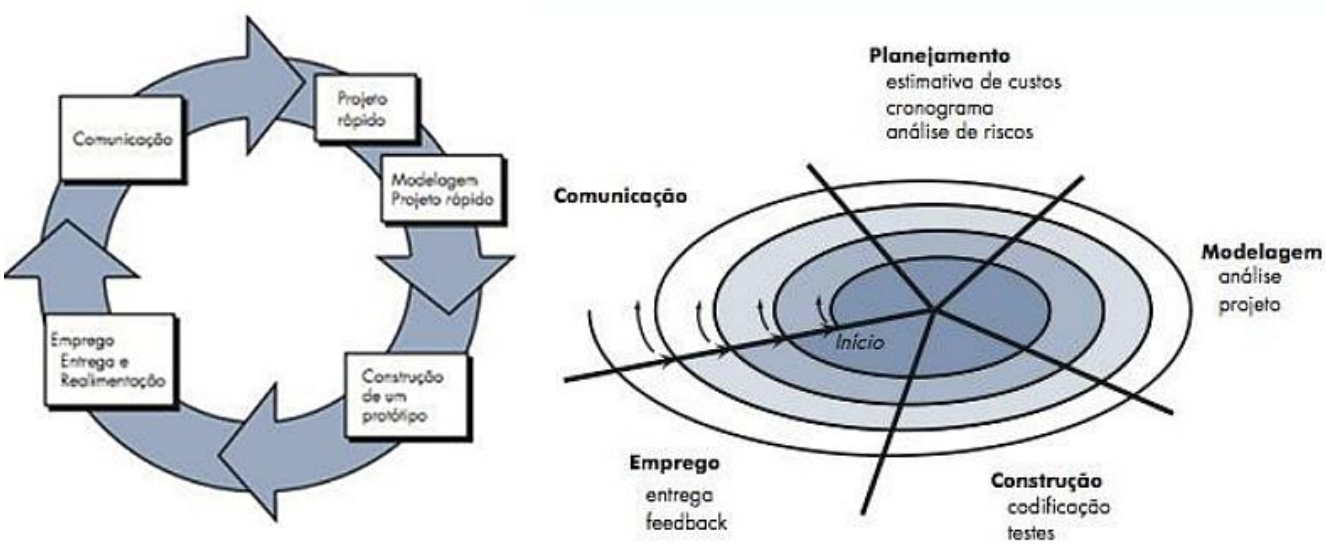


Figura 2.2: Modelos por prototipação e espiral (PRESSMAN, 2010).

Sommerville (2010), ainda aponta engenharia de software orientada a reuso, onde, a partir de uma especificação de software requerida, busca-se um componente que se adeque melhor ao problema proposto. Este componente é então analisado e alterado, se houver necessidade, para posterior integração com o software. Esta prática é bastante comum hoje em dia e muitos componentes estão sendo lançados com licenças de software livre através de *sites* na Internet.

Em 2001, foi escrito o *Agile Manifesto* que, de forma sucinta, propôs valorizar mais as pessoas e as interações entre elas, o software, a colaboração com o cliente e a resposta a mudanças (BECK, 2013). Sua intenção era compartilhar melhores maneiras de desenvolver programas de computador e a partir disso houve o surgimento de novas metodologias.

Uma destas metodologias é conhecida por *Extreme Programming* (XP), que segundo Beck (2004) "é uma maneira divertida, cientifica, previsível, flexível, de baixo risco, eficiente e leve de desenvolver um software". Beck nomeou a metodologia como *Extreme Programming* (em português, Programação Extrema) pois ela sugere que se leve tudo que é considerado bom em desenvolvimento de software ao seu máximo. Deve se aplicar programação em pares como uma forma constante de revisão de código; testes automáticos aplicados constantemente; refatoração; integração continua; iterações mais curtas possíveis (Beck, 2004). Ainda assim, em o Jogo do Planejamento, Beck (2004) sugere o uso de um cartão de histórias, onde a equipe deve escrever um caso de uso do software e se comprometer com sub-tarefas e seus prazos. Assim, é possível verificar que existe uma informação comum aos interessados do projeto que deve ser unificada de forma a guiar as diferentes etapas do projeto.

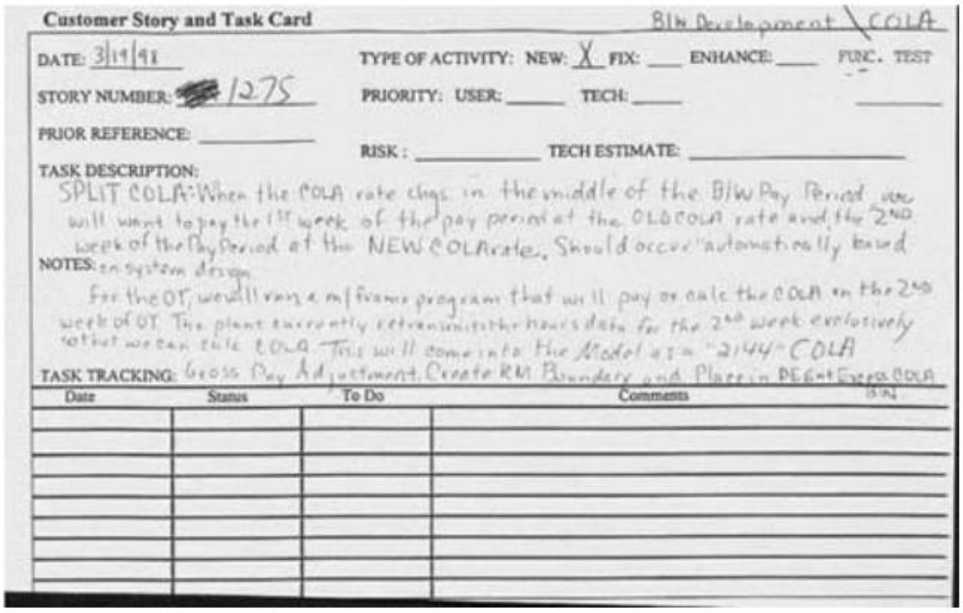


Figura 2.3: Um cartão de histórias (BECK, 2004)

Recentemente podemos verificar a grande aceitação do método *Scrum*, numa pesquisa feita pela empresa *VersionOne*, entre 4.048 pessoas de diferentes áreas de desenvolvimento de software, 72% utilizam *Scrum* ou uma variante (VERSIONONE, 2013). Esta metodologia teve sua primeira apresentação em 1995, antes do *Agile Manifesto*, mas é consistente com o mesmo (PRESSMAN, 2011). De acordo com Schwaber (2004), o mínimo para se iniciar um projeto *Scrum* consiste de uma visão, que justifica o propósito do projeto seu objetivo, e um *Product Backlog*, onde são

listados os requisitos do projeto. Posteriormente, o projeto é divido em *sprints*, cada qual com seu próprio *backlog*, derivado do *Product Backlog*. A figura 2.4 apresenta o fluxo do processo Scrum.

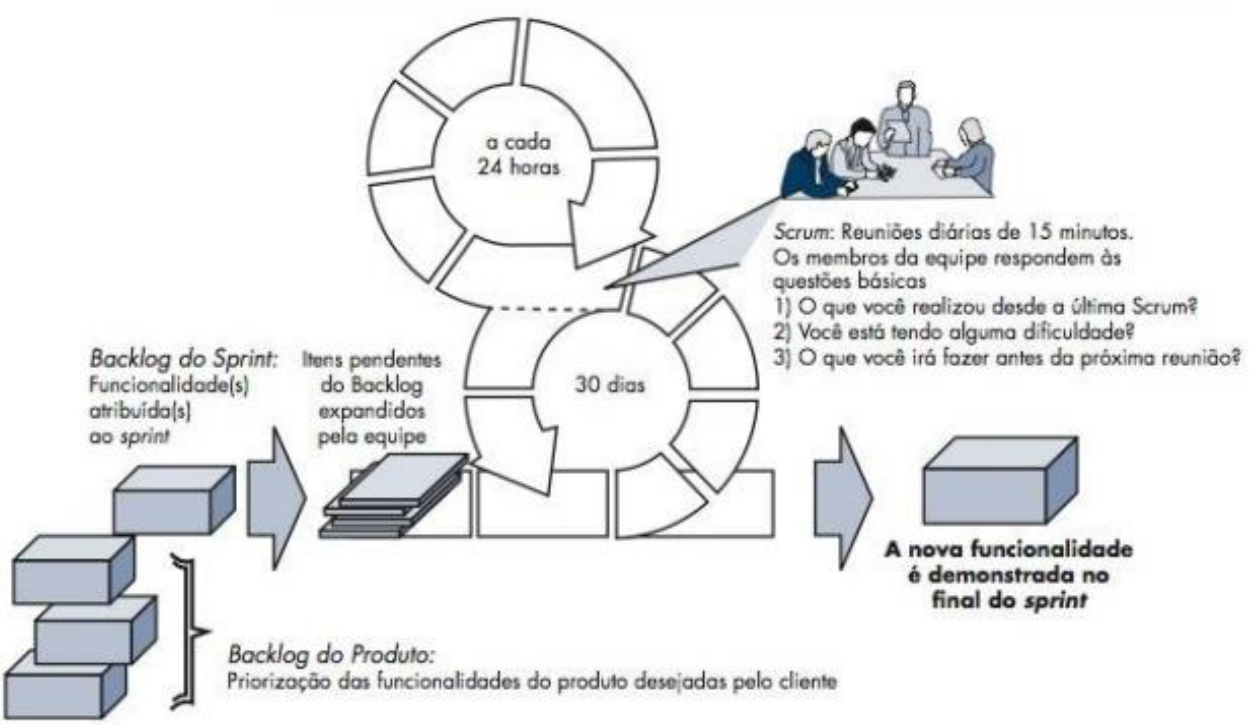


Figura 2.4: Fluxo do processo *Scrum* (PRESSMAN, 2011)

Esta revisão de alguns dos principais métodos apontam um certo antagonismo entre os métodos clássicos e os métodos ágeis, mas nenhum deles foge de formalizar informações. Alguns podem ser bastante densos com levantamento de requisitos, escopo, planejamento, cronogramas, tarefas, outros bastante sucintos, apenas com histórias de usuários. Portanto, muitos gerentes de projetos e times de desenvolvimento de software utilizam ferramentas que auxiliam no controle destas informações ao longo do projeto. Estas ferramentas ajudam a manter estas informações centralizadas e atualizadas guiando os passos do projeto.

As tabela 2.1 e 2.2 apresenta dados da pesquisa da VersionOne (2013) e a partir delas é possível verificar que não só esta documentação é presente, como ela se encontra espalhada por diversas ferramentas, sejam elas digitais ou analógicas. O resultado variado demonstra que os times consultados buscam diferentes formas de gerir seus projetos, o que pode ser interpretado como uma carência por uma forma mais consistente de documentar e acompanhar seus projetos.

Tabela 2.1: Uso e preferências de ferramentas para métodos ágeis

Ferramenta	Usam atualmente	Planejam usar	Gostariam de usar	Não precisam
Controlador de erros	83%	5%	7%	5%
Wikis	72%	7%	10%	11%
Planilha de dados	69%	2%	3%	26%
Ferramenta de gerenciamento de projetos ágeis	60%	11%	18%	11%
Ferramenta de gerenciamento de projetos tradicional	49%	4%	5%	42%

Fonte: VERSIONONE, 2013. p 15.

Tabela 2.2: Uso especifico de ferramenta de gerenciamento de projetos ágeis

Ferramenta	Uso (múltipla escolha era possível)
Excel	69%
Microsoft Project	48%
VersionOne	36%
JIRA/Greenhopper	33%
HP Quality Center	24%
Microsoft TFS	22%
Bugzilla	22%
In-House/Homegrown	20%

Fonte: VERSIONONE, 2013. p 16.

3 CONTROLE DE VERSÃO E O SISTEMA GIT

Um sistema de controle de versão (VCS) grava alterações a um conjunto de arquivos possibilitando a recuperação de versões anteriores (CHACON, 2009). A utilização de um VCS apresenta algumas vantagens: capacidade de regressar o projeto à momentos anteriores, possibilitando a reversão de eventuais erros; múltiplos colaboradores trabalhando em um mesmo código sem interferência; manutenção de um histórico de mudanças como uma forma de entender alterações no projeto; múltiplas versões simultâneas de um mesmo projeto (MASON, 2006).

Existem alguns softwares que atendem esta especificação e um dos primeiros a se tornar a ferramenta de escolha por muitos anos foi o *Concurrent Versions System* (CVS). Este possibilita operações através de rede, facilitando o trabalho de programadores dispersos geograficamente. Entretanto, suas falhas mostraram ser um grande desafio para solução e isso abriu espaço para uma outra ferramenta, o *Subversion* (PILATO, 2008).

Tanto o CVS quanto o *Subversion,* operam de modo centralizado, ou seja, existe um servidor central que mantém uma cópia mestre do repositório e possibilita a criação de uma cópia no computador de cada usuário. Este processo de cópia é conhecido como *checkout* e garante que o desenvolvedor tenha a estrutura do repositório central espelhados em sua máquina (MASON, 2006).

Após a execução de algumas mudanças nos arquivos em cópia local pode-se tornar necessário salvá-los de volta ao repositório central - este processo se chama *commiting*. Outros colaboradores podem estar trabalhando no mesmo projeto e acabarão fazendo *commits* próprios. Estas alterações estarão no repositório central, mas não atingirão as cópias locais dos outros

colaboradores a não ser que eles atualizem, no processo conhecido como *update*, sua cópia local em relação a versão atual do repositório central (MASON, 2006).

Estes repositórios, inicialmente, apresentam uma base de códigos única, entretanto é possível que ao longo de um produto que surja a necessidade de divergir do código inicial para a criação de outra versão. Isto deve ocorrer sem parar a linha inicial de desenvolvimento. Para tanto, existe a possibilidade da criação de ramificações (*branches)* - linhas de desenvolvimento que existem independentemente de outras, mas compartilham de um passado comum (PILATO, 2008).

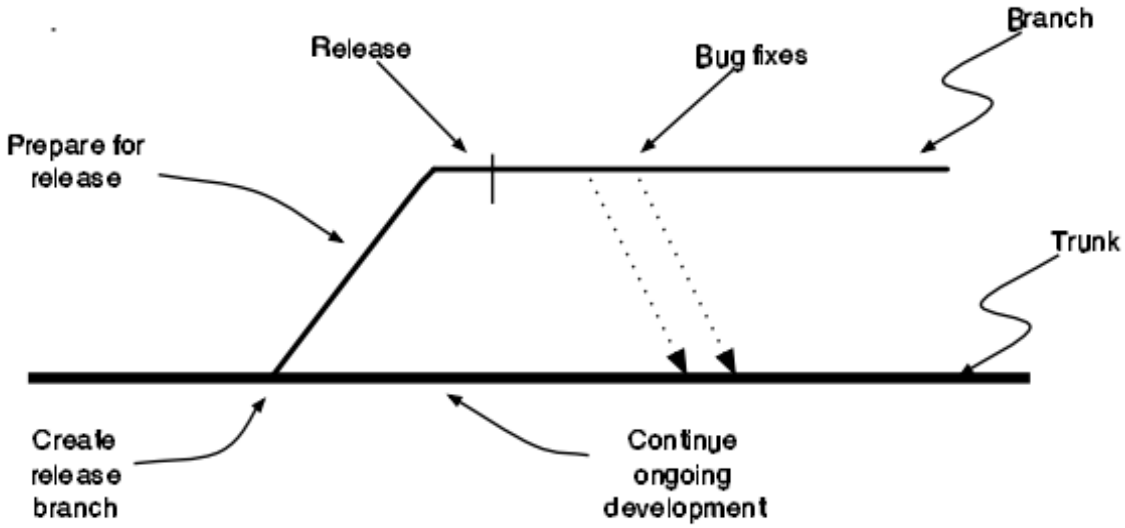


Figura 3.1: Uma exemplo de ramificação. (MASON, 2006)

Muitas vezes, ao término do desenvolvimento de uma funcionalidade ou uma nova versão, deseja-se copiar as alterações a linha inicial de desenvolvimento. Esta replicação das mudanças de uma ramificação para outra é conhecido como *merging* (PILATO, 2008).

Mason (2006) defende que seja evitado a criação de muitas ramificações. O autor alega que pode ser tentador testar diversas possibilidades em diferentes ramificações mas que pode ter custos significativos ao proceder com o *merge* de diferentes ramificações.

Em um caminho diferente aparece a ferramenta *Git*, Swicegood (2008), afirma que a criação de ramificações é uma arte e não uma ciência e sugere que seja usado inúmeras ramificações para correções de erros, experimentos ou novas funcionalidades. Em geral, estas ramificações são temporárias sendo excluídas logo que fundidas de volta a sua linha inicial. Isto é tão marcante que, "algumas pessoas se referem ao modelo de ramificação em *Git* como sua característica 'matadora', e que certamente o destaca na comunidade de VCS.[...] Ao contrário de muitos outros VCSs, o *Git*

incentiva um fluxo de trabalho no qual se fazem *branches* e *merges* com frequência, até mesmo várias vezes ao dia" (CHACON, 2009, p. 43).

O sistema *Git* apresenta esta característica pois internamente ele tem algumas diferenças em relação a outros VCSs. Chacon (2009) esclarece:

> Conceitualmente, a maior parte dos outros sistemas armazena informação como uma lista de mudanças por arquivo. Esses sistemas (CVS, Subversion, Perforce, Bazaar, etc.) tratam a informação que mantém como um conjunto de arquivos e as mudanças feitas a cada arquivo ao longo do tempo [...]ao invés disso, o Git considera que os dados são como um conjunto de snapshots (captura de algo em um determinado instante, como em uma foto) de um mini-sistema de arquivos.

Desta maneira, o sistema *Git*, para criar uma nova ramificação precisa apenas criar um novo ponteiro para o *snapshot* desejado. A figura 3.2 apresenta um grafo que mostra uma ramificação chamada *master* apontando para um determinado *commit.* Este, por sua vez, tem ligação com seu *snapshot* e um *commit* pai.

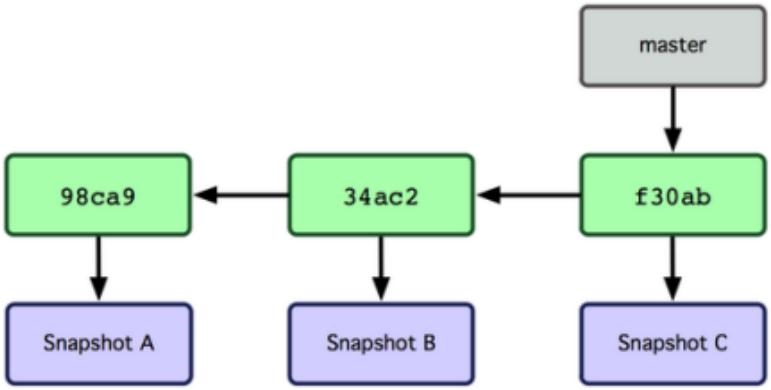


Figura 3.2: Um repositório Git com uma ramificação (CHACON, 2009)

Ao adicionar uma nova ramificação ficaríamos com um repositório estruturado como a figura 3.3 que demonstra que esta nova ramificação, *testing*, é apenas uma referência ao mesmo *commit* de *master*. Na verdade, esta referência é apenas um arquivo com nome *testing* e conteúdo "f30ab" criado no sistema de arquivos do *Git*.

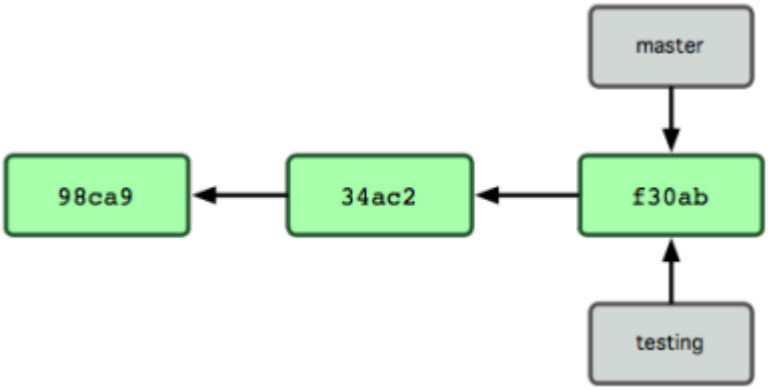


Figura 3.3: Um repositório Git com duas ramificações (CHACON, 2009)

Ao conhecer melhor a implementação do sistema, descobre-se que o *Git* é um sistema de arquivos de conteúdo endereçável. Isto é verificado pelo uso de um mecanismo de armazenamento chave-valor, ou seja, para qualquer valor armazenado o *Git* retorna com uma chave capaz de recuperar o valor desejado. O valor armazenado em forma de objeto chamado *blob* (CHACON, 2009). Swicegood (2008) incentiva que se use o Git para armazenar qualquer coisa que você precise para trabalhar em seu projeto. Ele cita ainda que outros itens comuns, como arquivos exemplos de configuração, documentação e imagens sejam incluídos em seu repositório.

Objetos do tipo *blob* não apresentam nenhuma informação adicional sobre o arquivo como o nome ou diretório em que está contido, portanto o *Git* faz uso de um segundo objeto, chamado objeto árvore. Este corresponde, de forma simplificada, a diretórios em um sistema *UNIX*. Uma árvore é composta por uma ou mais entradas, com um ponteiro para um objeto *blob* ou uma sub-árvore (CHACON, 2009).

Por fim, o *Git* apresenta o objeto de *commit*. Este objeto aponta para uma árvore previamente criada, o seu *snapshot*, em face dos arquivos escolhidos para versionamento. Além disso, ele guarda informações de quem o criou, quando foi criado e, opcionalmente, uma mensagem referente a sua criação. Um objeto de *commit* pode, ainda, apontar para outro, criado anteriormente, estabelecendo assim uma ordem temporal entre eles (CHACON, 2009).

Conforme foi mostrado nas imagens 3.2 e 3.3, o *Git* possui arquivos chamados referências que facilitam a identificação de ramificações. Estas referências podem também serem usadas para criar *tags* que marcam *commits* importantes em um respositório (CHACON, 2009).

Outra diferença importante na implementação do sistema é sua natureza distribuída. Ao contrário de sistemas como *Subversion*, o *Git* não necessita de um repositório central. Cada indivíduo mantém uma cópia completa do projeto e pode adicionar um ou mais servidores remotos e pode atualizá-los quando julgar necessário e não em toda a operação de *commit* (SWICEGOOD, 2008). Isso proporciona a utilização de diferentes fluxos de trabalho. Alguns se assemelham aos sistemas anteriores com um repositório comum a todos colaboradores; outros apresentam hierarquias de repositórios com gerentes de integração de código (CHACON, 2009).

O sistema *Git*, inicialmente, foi um conjunto de ferramentas, ou uma biblioteca, para um VCS mais do que um sistema VCS amigável e, por conta disso, apresenta comandos mais baixo nível que os comumente utilizados como *checkout*, *commit*, *branch* e *merge* (CHACON, 2009).

Chacon (2009) explica que os comandos disponíveis são, usualmente, separados em dois grupos: Comandos de "porcelana", que contém os comandos de alto nível e os comandos de "encanamento" para os comandos de baixo nível. Mais adiante o autor também acrescenta que "estes comandos não são destinados a ser usados manualmente na linha de comando, mas sim para serem usados como blocos de construção para novas ferramentas e scripts personalizados" (CHACON, 2009, p. 205).

Portanto, podemos verificar a importância que VCS têm para controle de projetos e, com mais detalhes, que o sistema *Git* apresenta um avanço importante para VCS, através de sua abordagem para armazenamento de dados, que o torna em um sistema de arquivos com ferramentas poderosas, além de um simples VCS (CHACON, 2009). Além disso é possível verificar que o desenvolvimento de novas funcionalidades para o sistema é algo incentivado através dos comandos de baixo-nível.

4 IMPLEMENTAÇÃO

Este capítulo detalha a implementação da biblioteca *Gitabs* e da aplicação que é utilizada para validação posterior. É apresentado, inicialmente, a metodologia utilizada para o desenvolvimento projeto e as tecnologias escolhidas para isto.

Também é realizada uma análise do problema e apresentada uma explicação de como cada um dos requisitos do trabalho foi abordado. Para cada requisito, é explicado como a aplicação da metodologia escolhida atingiu o resultado esperado. Isto é exemplificado através de um cenário de aceitação escrito a partir da funcionalidade requerida.

Outros cenários e suas especificações podem ser encontrados nos Apêndices e todos seguiram a mesma abordagem explicada neste capítulo.

4.1 Metodologia

Para a implementação da biblioteca e sua aplicação foi escolhido a metodologia de Desenvolvimento Dirigido a Comportamento (BDD). Proposto por Astels (2013), BDD é uma metodologia derivada de Desenvolvimento Dirigido a Testes (TDD), que foi apresentada inicialmente pelo mesmo autor de XP, Kent Beck, em (BECK, 1999).

Para um completo entendimento da metodologia BDD é necessário uma explicação sobre TDD que é feita na próxima seção.

4.1.1 Desenvolvimento Dirigido a Testes

TDD segue duas regras básicas: escreva código apenas se você tiver um teste falhando e

elimine duplicatas. Estas duas regras trazem algumas implicações técnicas: o design do software é orgânico, com rápidos retornos a partir dos testes escritos; cada desenvolvedor deve escrever seus próprios testes; o ambiente de desenvolvimento deve fornecer respostas rápidas a pequenas mudanças; e o design do código deve consistir de muitos componentes altamente coesos e fracamente acoplados, de forma a facilitar os testes (BECK, 2002).

Em uma abordagem mais prática, Beck (2002), fornece um conjunto de tarefas de programação que é conhecida como *Red/Green/Refactor*. *Red*, trata de escrever um pequeno teste que falhe. *Green*, consiste em escrever um código que passe o teste, sem preocupações com o código gerado. Por fim, *Refactor*, fala sobre eliminar qualquer duplicata gerada para passar o teste. Estas etapas ocorrem em sequência e são constantemente repetidas até que se atinja a situação esperada.

A etapa de refatoração busca aprimorar o código inicial aceito pelo teste. Refatoração "é o processo de mudar o sistema de um software de tal maneira que não altere o comportamento externo do seu código mas apresente melhoras na sua estrutura externa" (FOWLER, 1999, p 9).

De acordo com Fowler (1999), refatoração melhora o design de seu software; facilita seu entendimento e a procura por erros; e ajuda a programar mais rápido. Através de uma série de padrões, chamados *Code Smells*, Fowler (1999) apresenta possíveis conjuntos de código que podem sofrer refatoração juntamente com uma série de métodos de refatoração, que eliminam estes padrões.

4.1.2 Desenvolvimento Dirigido a Comportamento

Em 2005, Astels (2013) cunhou a expressão *Behavior-driven Development* (Desenvolvimento Orientado a Comportamento). Ele explica que com o surgimento de TDD muitas empresas investiram em treinamento mas poucos profissionais realmente entenderam a metodologia por completo. Os desenvolvedores estavam focando mais em testes e verificações do que em comportamento e especificações, que é o verdadeiro valor por trás deste método. Por fim, Astels (2013) aponta que é necessário uma troca na forma como se vê a metodologia e seu

vocabulário do que na técnica em si e que portanto novos frameworks, focados no comportamento de um sistema, deveriam ser desenvolvidos.

Desta forma, BDD foca nas interações entre usuários e programas de computadores e entre diferentes componentes dentro do sistema e é neste contexto que North (2013), propõe o padrão *given-when-then* para especificação de cenários. Com este padrão é possível estabelecer um entendimento comum entre analistas de negócio, testadores e desenvolvedores (CHELIMSKY, 2010). O mesmo é suficientemente flexível para que um analista possa usá-lo e é estruturado o bastante para transformá-lo em testes automatizados. Ele define um contexto na primeira clausula, *Given.* Em seguida apresenta uma alteração deste contexto para a segunda cláusula, *When*, e é aceito caso esta sequência resulte em algo previsto pela cláusula final, *Then*. Cada uma destas cláusulas podem ser expandidas com o uso da cláusula adicional, *And* (North, 2006).

Solis explica que "existem poucos estudos publicados sobre BDD, [...] e que isso pode ser um reflexo da visão original de BDD como uma pequena e simples mudança das práticas existentes de TDD" (2011, p. 1) mas ajuda a estabelecer as principais características deste método.

O centro de BDD é o conceito de linguagem oblíqua. A criação de uma linguagem oblíqua, cuja estrutura seja proveniente do domínio, é importante pois deve ser usada ao longo do ciclo de desenvolvimento. Neste sentido, o padrão *given-when-then* possibilita a clara visualização de uma funcionalidade que o sistema deve suportar e por que ela deve ser suportada (SOLIS, 2011).

A análise de um sistema, em BDD, procura identificar os comportamentos esperados do mesmo. Estes comportamentos são derivados dos objetivos do projeto e são detalhados em funcionalidades do sistema, que indicam o que deve ser feito para atingir o objetivo do sistema (SOLIS, 2011).

BDD se vale de automação de testes da aceitação proveniente de *Automated Test-driven Development* (ATDD). Um teste de aceitação, neste contexto, é uma especificação executável do comportamento do sistema e que verifica o comportamento dos objetos ao invés de seus estados. Desta forma, ATDD é uma parte integral da implementação em uma abordagem BDD (SOLIS, 2011).

Em sintonia com os valores ágeis, BDD sugere que parte da documentação do sistema seja entregue pelo próprio código. Isso implica em cuidados ao escolher nome para métodos, classes e testes já que eles devem descrever o comportamento dos objetos (SOLIS, 2011).

Chelimsky (2010), apresenta um ciclo de BDD em duas camadas. Este ciclo propõe o uso de dois ciclos *red/green/refactor*: um para descrever o comportamento de aplicações e outro para descrever o comportamento de objetos.

Na descrição de histórias para comportamento de aplicações, Chelimsky (2010), define que, uma história consiste de um título, uma narrativa detalhando melhor a história e critérios de aceitação que definem quando que o objetivo foi atingido. Mais especificamente, em BDD, os critérios de aceitação são apresentados em forma de cenários que são compostos de passos (CHELIMSKY, 2010).

4.1.3 Definição de metodologia

Tendo em vista a natureza do presente projeto, com uma divisão entre um conjunto de operações fornecidas pela biblioteca e a necessidade de uma aplicação que seja uma interface com o usuário, a metodologia BDD proposta por Chelimsky (2010) - e ilustrada na figura 4.1 - guiará o desenvolvimento subsequente do projeto.

O primeiro ciclo tem foco no aplicação e parte das possíveis interações entre um usuário e as operações desejadas. Inicialmente, escreve-se um cenário que deve ter um passo de validação. Este passo é um teste de aceitação para um certo comportamento. Com este passo falhando, são escritos exemplos de comportamento e é dado continuidade ao ciclo *Red/Green/Refactor*. Com os exemplos sendo aceitos é feita uma refatoração ao código. Neste momento, o passo escrito inicialmente deve ser aceito e é possível uma nova refatoração do código. Com isto, o ciclo é reiniciado até que todos os cenários são escritos e aceitos.

Deverão ser escolhidas ferramentas capazes de rodar os testes de aceitação para cada um dos ciclos. O resultado destes testes serão utilizados para validar a implementação com relação as funcionalidades entregues.

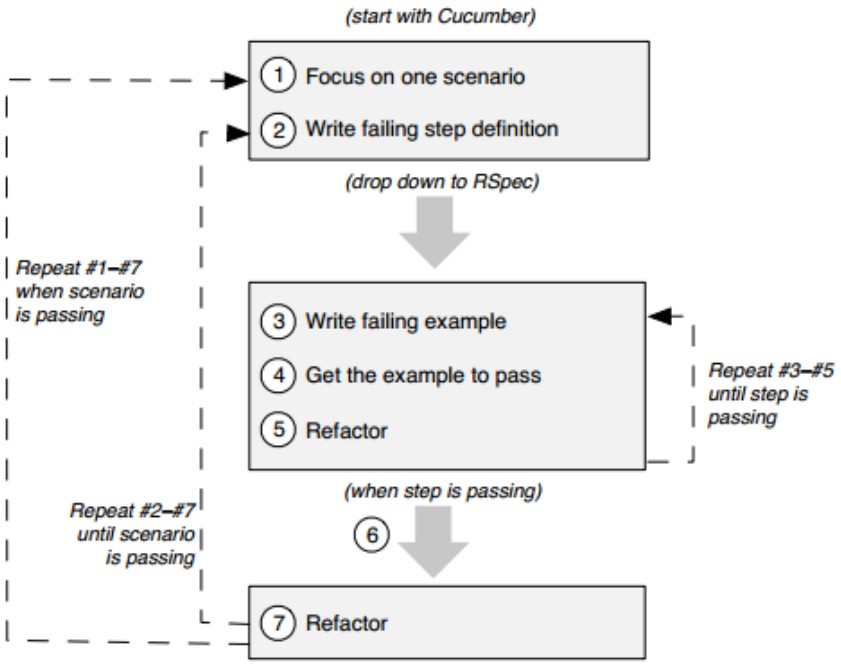


Figura 4.1: ciclo BDD (CHELIMSKY, 2010).

4.2 Definição de tecnologias

Para a implementação da biblioteca foi escolhido a linguagem *Ruby* por apresentar opções maduras nos pontos críticos do projeto: interação com sistema *Git* e validação de testes.

A interação com o sistema *Git* exigiu um profundo estudo do funcionamento de seus comandos de "encanamento" e da biblioteca *Rugged*. A biblioteca *Rugged*, desenvolvida em *Ruby*, fornece uma ligação ao *Libgit2* que é, por sua vez, uma implementação em C ANSI dos métodos base do *Git*.

Chelimsky (2010) sugere para seu ciclo BDD o uso das ferramentas *Cucumber* [3]e *RSpec*[4] para criação dos testes de aceitação. Entretanto, por questões técnicas, foi utilizado a ferramenta *Minitest* ao invés de *RSpec*. A ferramenta *Minitest* apresenta capacidades semelhantes ao *RSpec*, mas a *Minitest*, além de ser padrão da linguagem *Ruby*, possui métodos nativos para captura de saída de terminais de comando importante para interação com o sistema *Git.*

4.3 Análise de requisitos

A análise de requisitos iniciou valendo-se dos requisitos determinados no objetivo do trabalho. A partir deles, foi possível detalhar as funcionalidades da aplicação e diferentes cenários, com o uso do padrão *given-when-then* que foi proposto por Astels, no início do BDD (ASTELS, 2013).

A ferramenta *Cucumber* facilita esta etapa pois, através dela, é possível atender os conceitos de linguagem obliqua estabelecidos por BDD (SOLIS, 2011) . Em um primeiro momento, devido

[3] *Cucumber* é uma ferramenta que permite a especificação de softwares através de uma linguagem de domínio específico.

[4] *RSpec* é uma ferramenta para linguagem Ruby que foi desenvolvida a partir do surgimento do BDD e procura aprimorar TDD.

à natureza da metodologia, existiu apenas um detalhamento da funcionalidade, enquanto os testes foram adicionados, testados e refatorados respeitando o ciclo definido na seção anterior.

Para fins de organização do texto, as narrativas definidas nesta etapa de análise de requisitos assim como os testes de aceitação serão abordados nas próximas seções.

4.4 Criação de meta-ramificações

Para que a definição dos tipos de dados a serem armazenados seja possível, foi necessário a criação do conceito de meta-ramificações. Meta-ramificações são ramificações do sistema *Git* com o uso prioritário para armazenamento de meta-dados. Estes, por sua vez, são os dados externos ao projeto e que serão inseridos com o uso da biblioteca.

O armazenamento destes dados será feito usando estruturas JSON[5] e cada meta-ramificação recebe um arquivo em JSON-*Schema*, que define um contrato de como o dado JSON deve ser construído (IETF, 2013). Estas definições culminaram no detalhamento descrito na tabela abaixo:

Tabela 4.1: Funcionalidade 'Gerente de Projeto manipula meta-ramificações'

Um gerente de projetos deve conseguir criar uma meta-ramificação em um determinado repositório de tal forma que ele possa armazenar meta-dados nele.
Para criar uma meta-ramificação é necessário fornecer um arquivo JSON-*Schema* válido para que possa validar futuros meta-dados.
Uma meta-ramificação não deve ter uma relação inicial com qualquer outra relação.
Quando uma meta-ramificação foi recém criada, *Gitabs* manipula o histórico de *commits* para que esta tenha apenas um arquivo: o JSON-*Schema* fornecido.
O arquivo JSON-*Schema* é renomeado para o nome de sua meta-ramificação e sua extensão é alterada para *.schema* de forma a identificá-lo.

[5] JSON (JavaScript Object Notation) é um formato para troca de dados extremamente leve e independente de implementação.

Com esta expressão da funcionalidade, foi dado início ao ciclo *red/green/refactor*. Um dos cenários mais importantes para esta funcionalidade é o que contempla a criação bem-sucedida de uma meta-ramificação. A figura 4.2 apresenta uma parte do arquivo que define os cenários desta funcionalidade. Entre as linhas 39 e 59 é feita a descrição do contexto do teste. Nesta descrição é colocado que deve ser verificado se o usuário está em um diretório que tenha um repositório git e fornece um arquivo JSON, cujo conteúdo é um JSON-Schema válido. Isto é feito, por que a ferramenta *Cucumber*, ao rodar os testes, simula um sistema de arquivos próprio para não afetar o sistema de arquivos do desenvolvedor.

Com este contexto validado, o teste então roda o comando de criação de meta-ramificações no passo *When*. Este comando é válido com 4 asserções: o programa deve confirmar a criação da meta-ramificação; o programa deve carregar a meta-ramificação recém criada com sucesso; e esta nova meta-ramificação não deve ter nenhum arquivo de meta-dado.

```
38      Scenario: run metabranch command with file option
39              Given I am on a directory with a git repository
40              And a file named "assets/json-schema/user-schema.json" with:
41                      """
42                      {
43                              "$schema": "http://json-schema.org/draft-04/schema#",
44                              "title": "User",
45                              "description": "A User",
46                              "type": "object",
47                              "properties": {
48                                              "name": {
49                                                      "description": "The user name",
50                                                      "type": "string"
51                                              },
52                                              "e-mail": {
53                                                      "description": "The user e-mail",
54                                                      "type": "string"
55                                              }
56                              },
57                              "required": ["name", "e-mail"]
58                      }
59                      """
60              When I run `gitabs metabranch users-meta -f assets/json-schema/user-schema.json`
61              Then the output should contain "Metabranch created"
62              And I run `gitabs metabranch users-meta`
63              And the output should contain "Loaded metabranch 'users-meta'"
64              And I run `gitabs metabranch users-meta -s`
65              And the output should contain "0 metadata records"
```

Figura 4.2: Definição do cenário no *Cucumber* para criação de meta-ramificação

Com os testes para a aplicação escritos, a implementação avançou no ciclo de desenvolvimento e passou a tratar do comportamento da biblioteca. A figura 4.4 demonstra os testes relacionados à criação de uma meta-ramificação descritos para a ferramenta *Minitest.* Cada bloco *it...end* apresenta um exemplo de como a ferramenta deve se comportar e guiaram a criação do código da figura 4.3.

```
describe "a new metabranch is created" do

        it "should have two valid arguments" do
                mb = Gitabs::Metabranch.new('users-meta', @assets_path + '/json-schema/user-schema.json')
                mb.name.wont_be_nil
                mb.file.wont_be_nil
        end

        it "should have a valid json-schema file" do
                mb = Gitabs::Metabranch.new('users-meta', @assets_path + '/json/invalid.json')
                mb.valid?.must_equal false
        end

        it "should be parentless" do
                mb = Gitabs::Metabranch.new('users-meta', @assets_path + '/json-schema/user-schema.json')
                mb.branch.tip.parents.count.must_equal 0

        end

        it "should have its json-schema file on head commit" do
                mb = Gitabs::Metabranch.new('users-meta', @assets_path + '/json-schema/user-schema.json')
                head = mb.branch.tip
                head.tree.first[:name].must_match /.*schema*./
        end

        it "should have a single file on head commit" do
                mb = Gitabs::Metabranch.new('users-meta', @assets_path + '/json-schema/user-schema.json')
                head = mb.branch.tip
                head.tree.count.must_equal 1
        end
end
```

Figura 4.3: Testes para criação de meta-ramificação no *Minitest.*

Esta especificação, deu início ao ciclo TDD que resultou em um comando da aplicação para criação de meta-ramificações. A figura 4.4 apresenta um secção do código que atende as necessidades descritas.

```
9         desc "metabranch [name] ", "Use this command to create and edit metabranches"
10        option         :file,
11                        :aliases => "f",
12                        :desc => "path to json-schema file"
13        option         :size,
14                        :aliases => "s",
15                        :desc => "show how many metadata records this metabranch have",
16                        :banner => ""
17
18        def metabranch(name)
19
20                file = options[:file] ? File.absolute_path(options[:file]) : nil
21                size = options[:size]
22
23                treat_metabranch_options(name, file, size)
24        end
```

Figura 4.4: Código da aplicação para o comando *metabranch*

É possível identificar neste código que o comando *metabranch* tem três argumentos. O primeiro e obrigatório é o argumento *name*, na definição do método *metabranch*, na linha 18. Ele é usado para identificar a meta-ramficação a ser criada. Os outros argumentos, *file* e *size* são opcionais e determinam o comportamento do método. Isto é verificado nas linhas 10 e 13 da figura 4.4. Se for fornecido um caminho de um arquivo para o argumento *file* o método cria uma nova meta-ramficação com este arquivo. Caso o argumento *size* seja fornecido como entrada, então o método apenas retorna o total de meta-dados que a ramificação possui no momento.

O código demonstrado na figura 4.4 é resultado do ciclo de refatoração após a validação dos testes descritos nas ferramentas *Cucumber* e *Minitest.*

4.5 Criação de meta-dados

É desejado que, ao definir uma estrutura de dados, seja possível a inserção de diferentes informações. Com isso, foi descrito na tabela 4.2 uma nova funcionalidade:

Tabela 4.2: Funcionalidade ‘Colaborador manipula meta-dados’

Um colaborador deve poder manipular meta-dados em uma determinada meta-ramificação de tal forma que ele possa registrar informações sobre o projeto.
Para criar um meta-dado, é preciso existir uma meta-ramificação e um arquivo json válido para aquela meta-ramificação.
O arquivo fornecido terá sua extensão alterada para *.data*, de forma a identificá-lo.

Na figura 4.5 é proposto um teste para inserção de um dado do tipo usuário em uma meta-ramificação que registra usuários. Com a cláusula *Given*, é descrito um contexto onde o teste se encontra em um diretório com um repositório *Git* e que este repositório possui uma meta-ramificação *users-meta*. Além disso, ele fornece um arquivo JSON válido com dados de um usuário para fins de teste.

Na cláusula *When* aparece um evento referente a inserção de meta-dados. Este, quando executa, altera o estado corrente e seu resultado então é verificado na clásula *Then*, finalizado assim a descrição oblíqua de um teste na ferramenta *Cucumber*.

```
@one-metabranch
Scenario: add valid metadata on current branch
        Given I am on a directory with a git repository
        And this repository has a metabranch named 'users-meta'
        And a file named "assets/json/john-doe.json" with:
        """
        {
                "name": "John Doe",
                "e-mail": "john@doe.com"
        }
        """
        When I run `gitabs metadata john-doe -f assets/json/john-doe.json`
        Then the output should contain "Metadata created"
```

Figura 4.5: Definição de cenário na ferramenta *Cucumber* para inserção de meta-dados

Com este cenário escrito, foi possível avançar para o segundo ciclo *red/green/refactor* que teve seus testes definidos conforme a figura 4.6. Estes testes procuram assegurar que a biblioteca não procede em casos onde o arquivo JSON fornecido é inválido de alguma forma ou se a ramificação em que o usuário se encontra não é uma meta-ramificação.

```
before(:each) do
        Gitabs::Metabranch.new('users-meta', @assets_path + '/json-schema/user-schema.json')
end
describe "add metadata" do
        it "fail for a invalid json file" do
                proc { Gitabs::Metadata.new("john-doe", @assets_path + "/json/invalid.json") }.must_raise(RuntimeError)
        end
        it "fails for a valid json file without required fields" do
                proc { Gitabs::Metadata.new("john-doe", @assets_path + "/json/john-doe-required-problem.json") }.must_raise(RuntimeError)
        end
        it "fails for a json file with fields other than required by schema" do
                proc { Gitabs::Metadata.new("john-doe", @assets_path + "/json/john-doe-required-problem.json") }.must_raise(RuntimeError)
        end
        it "fails if current branch is not a metabranch" do
                system('git checkout -q master')
                proc { Gitabs::Metadata.new("jonh-doe", @assets_path + "/json/john-doe.json")}.must_raise(RuntimeError)
        end
        it "succeeds for a valid json file if its accepted by metabranch schema" do
                md = Gitabs::Metadata.new("john-doe", @assets_path + "/json/john-doe.json")
                md.data.wont_be_nil
        end
end
```

Figura 4.6: Testes para criação de meta-dados no *Minitest*

4.6 Execução de tarefas

Com isto, a biblioteca encontra-se pronta para implementar a execução de uma tarefa. A funcionalidade foi aprofundada e pode ser vista na tabela 4.3. Para um completo entendimento desta funcionalidade é preciso separar os conceitos de ramificações de trabalho e ramificações de tarefa.

Uma ramificação de tarefa é onde o colaborador deverá inserir arquivos gerados a partir de uma determinada solicitação feita através de um meta-dado. Este pedido deverá ser feito através de uma inserção de meta-dado enquanto a adição dos novos arquivos é feita respeitando-se o histórico do repositório. Para tanto, ao criar uma ramificação de tarefa com a função de execução, deve ser fornecido uma ramificação para posterior *merge* da ramificação de tarefa.

Tabela 4.3: Funcionalidade ‘Colaborador executa tarefa’

Um colaborador deve poder escolher um meta-dado, em uma meta-ramificação, e relacioná-lo com uma ramificação de trabalho de forma a executar uma nova tarefa.
Esta operação deve resultar em uma terceira ramificação, a ramificação de tarefas, com dois pais: a meta-ramificação e a ramificação de trabalho.
Seu *snapshot* deve listar apenas os arquivos existentes na ramificação de trabalho escolhida.

Mais uma vez o ciclo BDD, com sua abordagem de fora para dentro, foi aplicado na construção do sistema que corresponda ao comportamento esperado por ele. As figuras 4.7 e 4.8 mostram os testes de aceitação para a aplicação e para a biblioteca, respectivamente.

É importante colocar que, a partir de um capacidade da ferramenta *Cucumber*, o cenário escolhido é executado em um ambiente previamente preparado. Este ambiente contempla um repositório *Git* com uma meta-ramificação chamada *task-meta* que tem seu JSON-*Schema* modelado para armazenar tarefas de trabalho. Além disso, a meta-ramificação *task-meta* possui um meta-dado válido chamado *landing-page* que faz a descrição de uma tarefa.

Inicialmente, visando passar os testes o mais rápido possível, o código para atender esta funcionalidade foi colocado junto a classe *Metadata*, criada no ciclo BDD anterior. Com a etapa de refatoração foi identificado um acúmulo de responsabilidades nesta classe e o Método de Extração de Classes (FOWLER, 1999) foi aplicado dando origem a mais uma classe para o tratamento de tarefas, a classe *Task*.

```
@task-meta
Scenario: create task branch
        Given I am on a directory with a git repository
        And current branch is 'task-meta'
        When I run `gitabs execute landing-page -w master`
        Then the output should contain "new task branch 'landing-page' created"
```

Figura 4.7: Descrição de cenário no *Cucumber* para execução de uma tarefa

```
describe "succeeds if work branch exists" do
        let(:md) { Gitabs::Metadata.new("landing-page", @assets_path + "/json/landing-page.json") }
        let(:task) { Gitabs::Task.new(md) }
        before(:each) do
                task.execute('master')
        end

        it "should create a branch after the metadata" do
                output = capture_subprocess_io { system('git rev-parse --abbrev-ref HEAD')}.join ''
                output.must_match 'landing-page'
        end


        it "should have the metabranch as parent" do
                parent_list         = capture_subprocess_io { system('git rev-list --parents -n 1 HEAD')}.join ''
                metabranch_hash        = md.metabranch.branch.tip.oid

                parent_list.must_match metabranch_hash
        end

        it "should have the work branch as parent" do
                parent_list = capture_subprocess_io { system('git rev-list --parents -n 1 HEAD')}.join ''
                master_hash = capture_subprocess_io { system('git show-ref --hash master')}.join ''

                parent_list.must_match master_hash
        end

        it "should have the files from work branch" do
                task_files = capture_subprocess_io { system('git ls-files')}.join ''
                system('git checkout -q master')
                master_files = capture_subprocess_io { system('git ls-files')}.join ''
                task_files.must_match master_files
        end

        it "shouldn't have files from metabranch" do
                task_files = capture_subprocess_io { system('git ls-files')}.join ''
                system('git checkout -q task-meta')
                metabranch_files = capture_subprocess_io { system('git ls-files')}.join ''
                task_files.wont_match metabranch_files
        end

        it "should have a tag after metadata point to its HEAD" do
                output = capture_subprocess_io { system('git describe --tags --abbrev=0 landing-page')}.join ''
                output.must_match "task-meta.landing-page"
        end

end
```

Figura 4.8: Testes de aceitação para execução de tarefa no *Minitest*

4.7 Entrega de tarefas

O passo final do fluxo esperado pela biblioteca *Gitabs* é a entrega de um trabalho finalizado. Em uma situação onde meta-ramificações foram criadas e devidamente alimentadas com dados, quando um colaborador executa um tarefa e a finaliza é necessário um processo, como o de *commit* do sistema *Git*, que registre o término da tarefa junto a meta-ramificação escolhida e que os arquivos de trabalho sejam mesclados a ramificação de trabalho apontada início da execução. A tabela 4.4 apresenta a formalização desta funcionalidade:

Tabela 4.4: Funcionalidade 'Colaborador entrega tarefa'

Um colaborador deve poder entregar uma tarefa de tal forma que ele registre, junto a seu meta-dado relacionado, o fim da execução da mesma.
Os arquivos da ramificação de tarefa devem ser mesclados de volta a ramificação de trabalho.
A meta-ramficação envolvida não pode ter sua árvore alterada por esta operação.

Para demonstrar o ciclo de implementação aplicado, foi escolhido o cenário onde a entrega de tarefa é bem sucedida. Este cenário, a exemplo do cenário usado para explicar a Execução de tarefas, possui um ambiente previamente preparado pela ferramenta *Cucumber*.

Este ambiente é o resultado do cenário do teste da seção anterior - um repositório Git, com uma meta-ramificação *task-meta* e uma ramificação de tarefa criado a partir de um meta-dado chamado *landing-page* - com a adição de um arquivo que simula o trabalho produzido pelo colaborador. O trecho de código ilustrados nas figuras 4.9 e 4.10 demonstram os testes escritos para este cenário.

Com os ciclos TDD foi possível escrever código suficiente para atender as funcionalidades requeridas. Com isto, a implementação da biblioteca encontra-se atendendo os testes propostos para as funcionalidades esperadas e deve passar por uma grande refatoração (FOWLER, 1999).

```
@ready-to-submit
Scenario: run submit command with 1 argument
        When I run `gitabs submit -m "this work is done"`
        Then the output should contain "Task submitted"
```

Figura 4.9: Descrição de cenário no *Cucumber* para entrega de uma tarefa

```
describe "succeeds if work branch exists" do
        let(:md) { Gitabs::Metadata.new("landing-page", @assets_path + "/json/landing-page.json") }
        let(:task) { Gitabs::Task.new(md) }
        before(:each) do
                task.execute('master')
        end

        it "should create a branch after the metadata" do
                output = capture_subprocess_io { system('git rev-parse --abbrev-ref HEAD')}.join ''
                output.must_match 'landing-page'
        end


        it "should have the metabranch as parent" do
                parent_list         = capture_subprocess_io { system('git rev-list --parents -n 1 HEAD')}.join ''
                metabranch_hash        = md.metabranch.branch.tip.oid

                parent_list.must_match metabranch_hash
        end

        it "should have the work branch as parent" do
                parent_list = capture_subprocess_io { system('git rev-list --parents -n 1 HEAD')}.join ''
                master_hash = capture_subprocess_io { system('git show-ref --hash master')}.join ''

                parent_list.must_match master_hash
        end

        it "should have the files from work branch" do
                task_files = capture_subprocess_io { system('git ls-files')}.join ''
                system('git checkout -q master')
                master_files = capture_subprocess_io { system('git ls-files')}.join ''
                task_files.must_match master_files
        end

        it "shouldn't have files from metabranch" do
                task_files = capture_subprocess_io { system('git ls-files')}.join ''
                system('git checkout -q task-meta')
                metabranch_files = capture_subprocess_io { system('git ls-files')}.join ''
                task_files.wont_match metabranch_files
        end

        it "should have a tag after metadata point to its HEAD" do
                output = capture_subprocess_io { system('git describe --tags --abbrev=0 landing-page')}.join ''
                output.must_match "task-meta.landing-page"
        end

end
```

Figura 4.10: Testes de aceitação para entrega de tarefas no *Minitest*

4.8 Uma camada para interação com o sistema Git

Uma consequência da metodologia aplicada no desenvolvimento foi a constante evolução do design do software. Tendo as funcionalidades previstas com testes passando, as classes da biblioteca passaram a acumular métodos quem atribuíam diferentes responsabilidades a elas.

Como no caso da criação da classe *Task*, foi possível observar em todas as classes da biblioteca, mais uma vez, um acúmulo de atribuições. Estes objetos armazenavam dados em memória sobre o estado deles e cuidavam da lógica em torno destes dados. Esta lógica, por sua vez, acabava por interagir diretamente com o sistema *Git*.

Assim, foram criadas 4 novas classes para lidar diretamente com o sistema Git. Com elas, é possível verificar o padrão *Data Mapper*, que sugere uma camada de mapeamento para o movimento de dados entre objetos e uma base de dados (FOWLER, 2002). A criação destas classes foi conduzida utilizando o Método de Extração de Classes (FOWLER, 1999). A figura 4.9 apresenta os códigos antes e depois da refatoração para o método *execute* da classe *Task* enquanto a figura 4.10 apresenta o diagrama de classes resultante deste processo.

O diagrama ilustrado na figura 4.10 apresenta as três classes que abstraem os conceitos de meta-ramificações, meta-dados e tarefas. Estas classes foram construídas através da aplicação da metodologia proposta e são responsáveis por manter e validar os objetos que compõe as capacidades da biblioteca. Em comunicação direta com estas classes existem as classes que foram criadas através da refatoração implantada ao final da implementação. Estas classes interagem diretamente com o sistema *Git*, utilizando a biblioteca *Rugged* e comandos baixo-nível do próprio sistema *Git*.

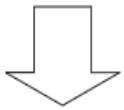
A linguagem Ruby apresenta o conceito de módulos. Estes módulos podem ser incluídos em objetos ou como métodos estáticos de classes. A camada de mapeamento foi definida com este recurso e são incluídas diretamente nos objetos necessários. Esse mapeamento foi realizado assim visando simplificar o código, que é um dos objetivos da metodologia adotada.

```
module Gitabs
        class Task
                include TaskController
                attr_reader :metadata

                def execute(workbranch)
                        load_metabranch
                        raise 'work branch not found' unless `git branch`.include?(workbranch)

                        `git mktree </dev/null`
                        emptycommit = `git commit-tree -p HEAD -p #{workbranch} #{workbranch}^{tree} -m 'create #{@name}'
                        `git checkout -q -b #{@name} #{emptycommit}`
                        `git tag #{@metabranch.name}.#{@name}`

                end
        end
end
```



```
module Gitabs
        class Task
                include TaskController
                attr_reader :metadata
                def execute(workbranch)
                        raise 'work branch not found' unless is_branch?(workbranch)
                        create_taskbranch(workbranch)
                end
        end
end
```

```
module Gitabs
        module TaskMapper
                include GitMapper
                def create_taskbranch(taskbranch)
                        `git mktree </dev/null`
                        emptycommit = `git commit-tree -p HEAD -p #{taskbranch} #{taskbranch}^{tree} -m
                                'create task branch for #{@metadata.metabranch.name}.#{@metadata.name}'`
                        `git checkout -q -b #{@metadata.name} #{emptycommit}`
                        `git tag #{@metadata.metabranch.name}.#{@metadata.name}`
                end
        end
end
```

Figura 4.11: Resultado de refatoração

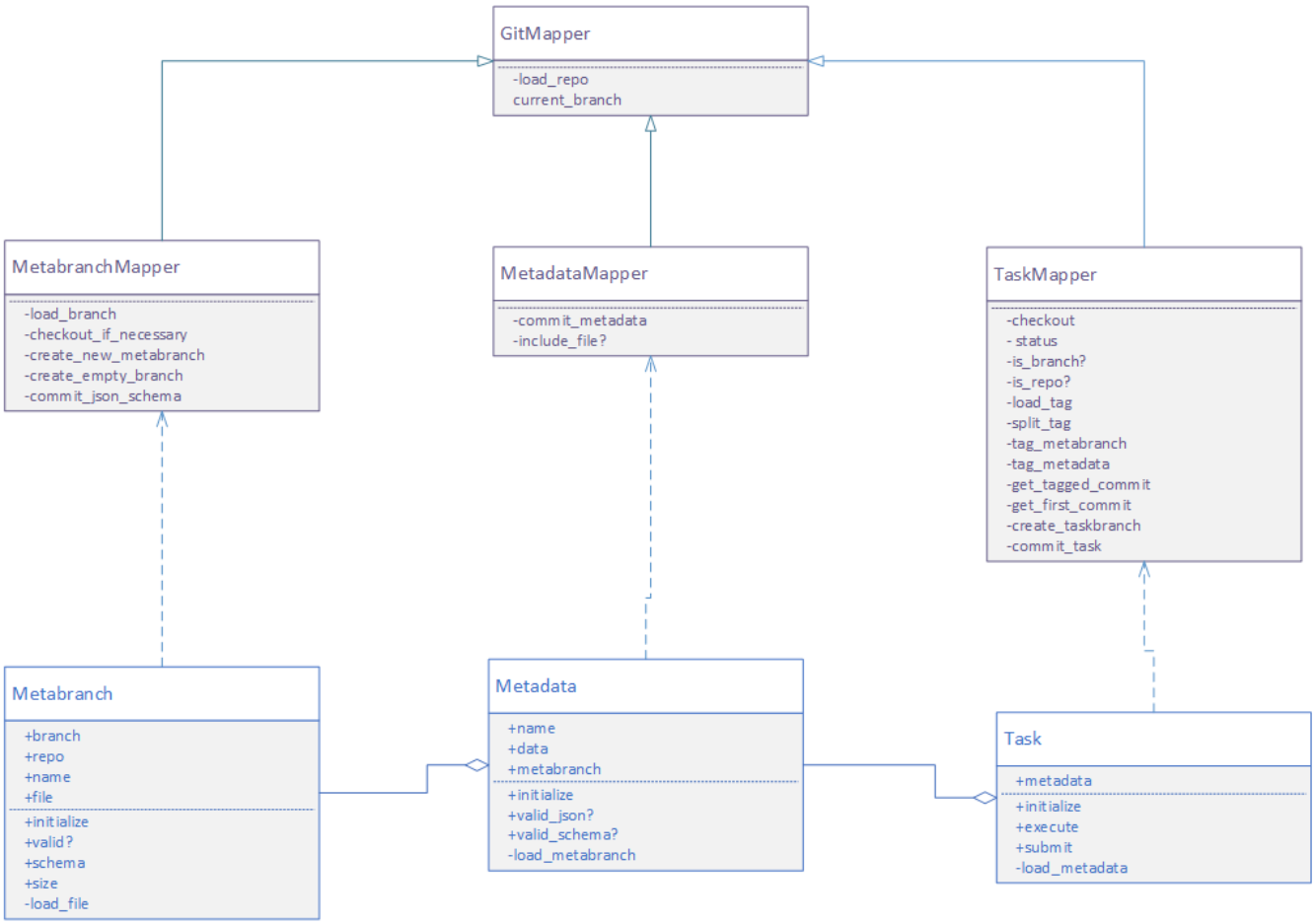


Imagem 4.12: Diagrama de classes final da biblioteca *Gitabs*.

5 RESULTADOS

Tanto a ferramenta *Cucumber* quanto a ferrmanenta *Minitest* possuem formas de executar os testes descritos ao longo dos ciclos BDD. A cada novo cenário repetia-se o ciclo *red/green/refactor* que, com suas informações acerca do funcionamento do sistema, guiou cada decisão do projeto em fase de implementação.

Conforme os testes definidos no apêndice A, a tabela 5.1 apresenta o resultado para cada funcionalidade descrita pelo *Cucumber*.

Tabela 5.1: Resultados da ferramenta *Cucumber.*

Funcionalidade	Cenários (aceitos)	Passos (aceitos)	Tempo de execução
Gerente de projetos manipula meta-ramificação	7 (7)	21 (21)	3,761s
Colaborador manipula meta-dado	6 (6)	23 (23)	3,497s
Colaborador executa tarefa	7 (7)	21 (21)	4,108s
Colaborador entrega tarefa	4 (4)	8 (8)	4,364s

Para as interações entre os objetos foram definidos testes de aceitação usando a ferramenta *Minitest.* A definição dos testes pode ser vista no apêndice B, e a tabela 5.2 apresenta os resultado para cada classe da biblioteca.

Tabela 5.2: Resultados da ferramenta *Minitest.*

Classe	Testes	Verificações	Falhas	Erros	Ignorados	Tempo
Metabranch	9	13	0	0	0	0.476s
Metadata	5	5	0	0	0	0.357s
Task	17	28	0	0	0	2.177s

6 CONCLUSÃO

O presente trabalho apresentou o objetivo de desenvolver uma biblioteca para unificar a gestão de projeto ao desenvolvimento dele. Através de quatro requisitos, foi possível definir um escopo para o projeto que guiou a implementação e a partir dele foi possível manter o controle da situação do projeto, uma vez que os testes de aceitação descritos são considerados como um sinal de que a implementação corresponde ao comportamento esperado.

Antes de que fosse iniciado a implementação da biblioteca e da aplicação foi feito um estudo sobre gerenciamento de projetos que procurou identificar diferentes métodos e suas formas de documentação. Além disso, verificou-se que equipes de desenvolvimento de software utilizam uma ou mais ferramenta de controle em seus projetos.

Sabendo que o controle de versão para desenvolvimento de projetos em engenharia de software é uma necessidade, foi realizado um aprofundamento do sistema *Git*. A partir disso e, de seus comandos de "encanamento", tornou-se possível entender que o sistema é capaz de expandir suas funções indo além do controle de versão de arquivos.

A metodologia escolhida provou-se eficiente, pois com ela foi possível sair do objetivo do projeto para uma aplicação funcional rapidamente. Através dos ciclos TDD o código atingiu uma maturidade ao longo do tempo apresentando facilidade em leitura e identificação de erros. Além disso, os testes definidos em código servirão como testes de regressão em situações futuras.

As escolhas feitas para armazenamento de dados mostraram-se acertadas tendo em vista a existência de diferentes formatações de equipe, processos e necessidades. Cada time pode utilizar a ferramenta implementada ou construir uma nova interface e armazenar as informações que julgarem necessário.

Além disso, é importante apontar que, devido a estrutura de dados flexível que foi definida para atender as necessidades do trabalho, foi dado início é um sistema que pode ser encarado de forma mais genérica e ser aplicado em outros contextos, onde a necessidade de armazenamento de dados temporais seja necessário.

A implementação da biblioteca foi publicada, desde o seu início, em um repositório público no site *GitHub*. Este repositório pode ser acessado em https://github.com/eduardomello/gitabs - possibilitando que qualquer usuário cadastrado ao site possa copiar seu código e abrir um linha de desenvolvimento própria ou utilizá-lo em outros projetos.

É possível acrescentar comandos a biblioteca que possam dar sustentação a um banco de dados não-relacional baseado em documentos de texto.

A criação de um *REST API*[6] parece um caminho natural para suportar aplicações web com robustez. A própria forma de armazenamento em JSON já mostra que a biblioteca é eficiente para servir a este propósito, uma vez que JSON é um ótimo candidato para troca de dados entre duas aplicações.

Em relação ao sistema *Git*, é possível aprofundar em situações onde a resolução de conflitos em operações de *merge* se fazem necessários. Também pode-se aprofundar nos impactos do uso da ferramenta em um fluxo completamente distribuído.

[6] *REST*, ou *Representational State Transfer*, é uma arquitetura muito utilizada no desenvolvimento de *Web Services.*

REFERÊNCIAS

MASON, M. **Pragmatic Version Control using Subversion.** 2a. Edição. Estados Unidos da América: The Pragmatic Bookshelf. 2006.

INSTITUTE, P.M. **O que é Gerenciamento de Projetos?** Disponível em <http://brasil.pmi.org/brazil/AboutUS/WhatIsProjectManagement.aspx>. Acesso em novembro de 2013.

SOMMERVILLE, I. **Software engineering.** 9ª Edição. Estados Unidos da América: Addison-Wesley, 2010.

GROUP, T. S. **Chaos Manifesto 2013: Think Big, Act Small.** Boston: The Standish Group International Inc. 2013.

PRESSMAN, R. S. **Software Engineering: A Practioner's Approach.** 7ª Edição. Estados Unidos da América: McGraw-Hill, 2010.

BECK, K., et al., **Manifesto for Agile Software Development**. Disponível em <http://www.agilemanifesto.org>. Acesso em novembro de 2013.

BECK, K. **Extreme Programming Explained: Embrace Change.** 1ª Edição. Estados Unidos da América: Addison-Wesley, 1999.

SCHWABER, K; SUTHERLAND. J. **The Scrum Guide**. Disponível em <https://www.scrum.org/Portals/0/Documents/Scrum%20Guides/2013/Scrum-Guide.pdf#zoom=100>. Acesso em novembro de 2013.

SCHWABER, K. **Agile Project Management with Scrum.** Estados Unidos da América: Microsoft Press, 2004.

VERSIONONE. **7th Annual State of Agile Development Survey.** Disponível em <http://www.versionone.com/state-of-agile-survey-results/>. Visualizado em novembro de 2013.

CHACON, S. *Pro Git***.** 2009.

PILATO, C. M.; FITZPATRICK, B. W.; COLLINS-SUSSMAN, B. **Version Control with Subversion For Subversion.** 2ª Edição. Estados Unidos da América: O'Reilly Media: 2008.

SWICEGOOD, T. **Pragmatic Version Control Using Git.** Estados Unidos da América: The Pragmatic Bookshelf. 2008.

ASTELS, D. **A New Look At Test Driven Development**. Disponível em <http://techblog.daveastels.com/files/BDD_Intro.pdf>. Acesso em novembro de 2013.

BECK, K. **Test-Driven Development By Example.** Estados Unidos da América: Addison-

Wesley, 2002.

FOWLER, M. **Refactoring: Improving the design of existing code**. Estados Unidos da América: Addison Wesley, 1999.

CHELIMSKY, D. **The RSpec Book: Behaviour-Driven Development with RSpec, Cucumber, and Friends.** Estados Unidos da América: The Pragmatic Programmers, 2010.

NORTH, D. **Introducing BDD.** Disponível em <http://dannorth.net/introducing-bdd/>. Acesso em novembro de 2013.

INTERNET ENGINEERING TASK FORCE. **JSON Schema: core definitions and terminology json-schema-core.** Disponível em <http://json-schema.org/latest/json-schema-core.html>. Acesso em 20 de novembro de 2013.

SOLIS, C.; XIAOFENG W. A Study of the Characteristics of Behaviour Driven Development. **Software Engineering and Advanced Applications (SEAA),** p.383-387, set. 2011.

FOWLER, M. **Patterns of Enterprise Application Architecture.** Estados Unidos da América: Addison-Wesley, 2002.

APÊNDICE A: FUNCIONALIDADES NA FERRAMENTA *CUCUMBER*

Feature: Project Manager manipulates a meta-branch

A project manager must be able to create a metabranch on some repository so he can start storing metadata on it. To create a metabranch you need to provide a valid json-schema file which will validate future metadata. A metabranch shouldn't have any parent commits from other branches. When a metabranch was just created, gitabs manipulate the commit history in order to have a single file: the json-schema provided in its creation. The json schema will be renamed to the metabranch name and its extension changed to .schema in order to indetify it.

Scenario: list metabranch command help

When I run `gitabs help metabranch`

Then the output should contain "Use this command to create and edit metabranches"

Scenario: run metabranch command with 0 arguments

When I run `gitabs metabranch`

Then the output should contain "ERROR"

@one-metabranch

Scenario: try to load metabranch (1 argument) and it exists

When I run `gitabs metabranch users-meta`

Then the output should contain "Loaded metabranch 'users-meta'"

```
Scenario: try to load metabranch (1 argument) and it doesn't exist
    When I run `gitabs metabranch users-meta`
    Then the output should contain "Metabranch doesn't exist"

Scenario: run metabranch command with file and size option
    When I run `gitabs metabranch users-meta -f assets/json-schema/user-schema.json
-s`
    Then the output should contain "ERROR"
Scenario: run metabranch command with file option
    Given I am on a directory with a git repository
    And a file named "assets/json-schema/user-schema.json" with:
        """
        {
            "$schema": "http://json-schema.org/draft-04/schema#",
            "title": "User",
            "description": "A User",
            "type": "object",
            "properties": {
                    "name": {
                        "description": "The user name",
                        "type": "string"
                    },
                    "e-mail": {
                        "description": "The user e-mail",
```

```
                                        "type": "string"
                                    }
                },
                "required": ["name", "e-mail"]
        }
        """
```

When I run `gitabs metabranch users-meta -f assets/json-schema/user-schema.json`

Then the output should contain "Metabranch created"

And I run `gitabs metabranch users-meta`

And the output should contain "Loaded metabranch 'users-meta'"

And I run `gitabs metabranch users-meta -s`

And the output should contain "0 metadata records"

Scenario: run metabranch command with invalid json-schema as file option

Given I am on a directory with a git repository

When I run `gitabs metabranch users-meta -f assets/json/invalid.json`

Then the output should contain "Invalid JSON-Schema"

Feature: Project Manager manipulates a meta-branch

A project manager must be able to create a metabranch on some repository so he can start storing metadata on it. To create a metabranch you need to provide a valid json-schema file which will validate future metadata. A metabranch shouldn't have any parent commits from other branches. When a metabranch was just created, gitabs manipulate the commit history in order to have a single file: the json-schema provided in its creation. The json schema will be renamed to the metabranch name and its extension changed to .schema in order to I identify it.

```
Scenario: list metabranch command help
    When I run `gitabs help metabranch`
    Then the output should contain "Use this command to create and edit metabranches"

Scenario: run metabranch command with 0 arguments
    When I run `gitabs metabranch`
    Then the output should contain "ERROR"

@one-metabranch
Scenario: try to load metabranch (1 argument) and it exists
    When I run `gitabs metabranch users-meta`
    Then the output should contain "Loaded metabranch 'users-meta'"

Scenario: try to load metabranch (1 argument) and it doesn't exist
    When I run `gitabs metabranch users-meta`
    Then the output should contain "Metabranch doesn't exist"

Scenario: run metabranch command with file and size option
    When I run `gitabs metabranch users-meta -f assets/json-schema/user-schema.json
-s`
    Then the output should contain "ERROR"

Scenario: run metabranch command with file option
    Given I am on a directory with a git repository
```

And a file named "assets/json-schema/user-schema.json" with:

```
"""
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "User",
    "description": "A User",
    "type": "object",
    "properties": {
        "name": {
            "description": "The user name",
            "type": "string"
        },
        "e-mail": {
            "description": "The user e-mail",
            "type": "string"
        }
    },
    "required": ["name", "e-mail"]
}
"""
```

When I run `gitabs metabranch users-meta -f assets/json-schema/user-schema.json`

Then the output should contain "Metabranch created"

And I run `gitabs metabranch users-meta`

And the output should contain "Loaded metabranch 'users-meta'"

And I run `gitabs metabranch users-meta -s`

And the output should contain "0 metadata records"

Scenario: run metabranch command with invalid json-schema as file option

Given I am on a directory with a git repository

When I run `gitabs metabranch users-meta -f assets/json/invalid.json`

Then the output should contain "Invalid JSON-Schema"

Feature: Collaborator executes task

A collaborator must be able to select a metadata, on a metabranch, and relate it to a work branch in order to execute a new task. This operation must result on a third branch, with both the selected metabranch and work branch as parents. It's tree object must list the same objects from the selected work branch.

Scenario: list execute command help

When I run `gitabs help execute`

Then the output should contain "Use this command to execute some work out of a certain metadata"

Scenario: run execute command with 0 arguments

When I run `gitabs execute`

Then the output should contain "No value provided for required options"

Scenario: run execute command with name argument

When I run `gitabs execute task`

Then the output should contain "No value provided for required options"

Scenario: run execute command with -w flag only

When I run `gitabs execute -w branch-name`

Then the output should contain "ERROR"

Scenario: run execute but there is no metadata on current branch

Given I am on a directory with a git repository

And the current branch is not 'task-meta'

When I run `gitabs execute landing-page -w master`

Then the output should contain "Provided branch is not a metabranch"

@task-meta

Scenario: run execute but no branch exists for -w flag

Given I am on a directory with a git repository

And current branch is 'task-meta'

And the branch 'void-branch' does not exist

When I run `gitabs execute landing-page -w void-branch`

Then the output should contain "work branch not found"

@task-meta

Scenario: create task branch

```
    Given I am on a directory with a git repository
    And current branch is 'task-meta'
    When I run `gitabs execute landing-page -w master`
    Then the output should contain "new task branch 'landing-page' created"
```

```
Feature: Collaborator submits task

    A collaborator must be able to submit a task branch so that it registers on its related metadata
the end of the execution. The files from task branch must be merged back on the work branch and
the metabranch must not have its tree altered.

    Scenario: list submit command help
        When I run `gitabs help submit`
        Then the output should contain "Use this command to submit a finished work"

    Scenario: run submit command with 0 arguments
        When I run `gitabs submit`
        Then the output should contain "No value provided for required options"

    @execute-task
    Scenario: run submit command with nothing to commit
        When I run `gitabs submit -m "this work is done"`
        Then the output should contain "Nothing to commit"
```

```
@ready-to-submit

Scenario: run submit command with 1 argument

    When I run `gitabs submit -m "this work is done"`

    Then the output should contain "Task submitted"
```

APÊNDICE B: COMPORTAMENTOS NA FERRAMENTA *MINITEST*

Gitabs::Metabranch::#initialize::a new metabranch is created

0001 should have two valid arguments

0002 should have a valid json-schema file

0003 should be parentless

0004 should have its json-schema file on head commit

0005 should have a single file on head commit

Gitabs::Metabranch::#initialize::try to load a metabranch

0001 should load branch if metabranch exists

0002 should load schema if metabranch exists

0003 should fail if metabranch doesn't exists

Gitabs::Metabranch::#size

0001 should return metadata total

Gitabs::Metadata::#initialize::add metadata

0001 fail for a invalid json file

0002 fails for a valid json file without required fields

0003 fails for a json file with fields other than required by schema

0004 fails if current branch is not a metabranch

0005 succeeds for a valid json file if its accepted by metabranch schema

Gitabs::Task::#initialize::failure situations

0001 should be on a repository

0002 should fail with invalid metadata

Gitabs::Task::#initialize::successful load situations

0001 should load if no metadata is provided

Gitabs::Task::#execute

0001 fails if work branch doesn't exist

Gitabs::Task::#execute::succeeds if work branch exists

0001 should create a branch after the metadata

0002 should have the metabranch as parent

0003 should have the work branch as parent

0004 should have the files from work branch

0005 shouldn't have files from metabranch

0006 should have a tag after metadata point to its HEAD

Gitabs::Task::#submit::failing situations

0001 fails if no argument is provided

0002 fails if no message is provided

0003 should fail if nothing to stage

Gitabs::Task::#submit::on successful submit

0001 should merge back into work branch

0002 metabranch should know task was submitted

0003 metabranch should keep tree of files

0004 task branch must be deleted