UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS FLORIANÓPOLIS

Christian Silva de Pieri

O impacto que as ferramentas de Bug Tracking Report têm sobre o ambiente de desenvolvimento e teste ágil

Florianópolis
2019

Christian Silva de Pieri

O impacto que as ferramentas de Bug Tracking Report têm sobre o ambiente de desenvolvimento e teste ágil

Relatório Final, referente ao Trabalho de Conclusão de Curso submetido à Universidade Federal de Santa Catarina, como requisito necessário para obtenção do grau de Bacharel em Ciências da Computação

Florianópolis, julho de 2019

UNIVERSIDADE FEDERAL DE SANTA CATARINA

Christian Silva de Pieri

Esta Monografia foi julgada adequada, para a obtenção do título de Bacharel em Ciências da Computação, sendo aprovada em sua forma final pela banca examinadora:

Orientador: Prof. Dr. Raul Wazlawick
Universidade Federal de Santa Catarina - UFSC

Prof. Dr. Jean Hauck
Universidade Federal de Santa Catarina - UFSC

Leonardo Mayer de Souza
Laboratório Bridge - UFSC

Florianópolis, 14 de julho de 2019

Agradecimentos

Agradeço a todos os meus amigos e familiares pela ajuda fornecida. Em especial àqueles presentes, diariamente, na minha vida por compreender e auxiliar na execução deste trabalho final e conclusão do curso. Agradeço também ao Laboratório Bridge, por fazer-me sentir integrante desta grande família e aceitar ser o local de aplicação do meu estudo final.

"A ferrovia que leva ao sucesso
é construída em cima de um solo de humildade,
com pesados trilhos chamados erros que
somente são fixados numa linha reta
com maciços pregos de perseverança.
(Eduardo Siqueira Filho)

Resumo

O foco principal deste trabalho de conclusão de curso foi fazer um estudo de caso com equipes ágeis, a fim de entender melhor as melhores ferramentas de Bug Tracking Report e o *feedback*, seja positivo ou negativo, que elas podem dar a um ambiente de desenvolvimento e de teste ágil. Foram realizados dois tipos de pesquisa: a bibliográfica sistemática e a pesquisa exploratória em campo no laboratório. O projeto e-SUS AB possui seis diferentes equipes ágeis multidisciplinares, que compreendem testadores, desenvolvedores, analistas de sistemas e *designers*. Cada equipe é também auto-gerenciável, ou seja, possui liberdade para trabalhar da melhor forma que decidir dentro do desenvolvimento ágil. Diversas ferramentas são utilizadas pelas equipes. O estudo se deu em cima da bibliografia, metodologias e testes ágeis com o uso destas ferramentas, em busca de uma maior compreensão se existe ou não uma melhor ferramenta de bug tracking report que a equipe se adapte com mais excelência. Buscou-se entender quais as melhores funcionalidades existentes nessas ferramentas e quais delas são essenciais do ponto de vista do desenvolvimento e do teste ágil.

Palavras-chave: Bug Track System, Teste Ágil, Metodologia Ágil.

Abstract

The main focus in this work is to do a case study with agile teams in order to know about the best Bug Tracking Report tools and its feedback, whether positive or negative, that they can give to an agile development and testing environment.There will be two major research areas: the systematic bibliography and exploratory field research in the laboratory. The e-SUS AB project has 6 different multidisciplinary teams, including testers, developers and software analysts. Each team is also self-manageable, that is, they have freedom to decide how to work, repecting the agile development philosofy. Several tools are used by teams. The study will be based on the bibliography, methodologies and agile tests with the use of these tools, in search of a greater understanding if there is a better tool of bug tracking report that the team adapts with more excellence. The most searched parameters was understant what is the bests existing functionalities in these tools and which of them are essential from the point of view of development and agile test.

Keywords: Bug Track System, Agile Test, Agile Tools.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

BTS	Bug Tracking System
BTR	Bug Tracking Report
AB	Atenção Básica
EA	Equipe Ágil
TI	Tecnologia da Informação
SUS	Sistema Único de Saúde
BI	*Business Inteligence*
RNI	Registro Nacional de Implantes
$SISMOB$	Sistema de Monitoramento de Obras
IA	Inteligência Artificial
PR	Pull Request

Sumário

1 Introdução

Não há dúvida que a tecnologia se faz presente e é de extrema importância na sociedade atual. Dada a gama de atividades cotidianas que podem ser automatizadas por *softwares*, como programas para cálculos matemáticos sendo realizados em alguns segundos; troca de mensagens instantâneas ao invés de cartas; monitoramento e *feedback* de sensores; abandono de pilhas de folhas e documentos para dar lugar a um pequeno e compacto *hard disk* (HD); dentre muitas outras opções que existem atualmente. Com o passar do tempo, as tecnologias e *frameworks* de desenvolvimento evoluíram, porém qualidade do software foi deixada de lado, valorizando-se muito mais a velocidade e resultado, independente de como esse fosse entregue.

Como afirma [Wazlawick 2013], antigamente, se esperava que a tarefa de programar fosse considerada quase que perfeita. Era esperado que os programadores construíssem programas de boa qualidade logo na primeira entrega. Testar a aplicação desenvolvida, era um castigo: um trabalho árduo, ingrato e indesejado.

Com o passar do tempo, quando se teve ideia de que o *software* necessita ter qualidade, essa ideia foi mudando e dando espaço para a atividade de teste, que hoje é considerada extremamente importante e faz parte do ciclo de desenvolvimento dele. Uma aplicação não testada ou até mesmo pouco testada pode levar a um erro catastrófico; ou a situação em que um paciente é submetido a um tratamento sensível a resposta de um equipamento que faz uso de um programa não testado, quais as probabilidades de um mau funcionamento causarem danos a esta pessoa?

Juntamente a esta ideia de haver sempre mais qualidade nas aplicações desenvolvidas, os processos de teste são avaliados e discutidos, sobretudo os modelos ágeis incorporam essa disciplina do desenvolvimento como uma atividade crítica, como afirma [Wazlawick 2013]. Para tal, existem ferramentas criadas com o objetivo de, ao encontrar um *bug* ou uma inconsistência, o profissional de qualidade poder reportar, de modo formal e rastreável, este erro que aconteceu no programa.

Ao avaliar estas ferramentas individualmente, deve-se ter em mente alguns tópicos para serem levados em consideração:

a) como se dá um bom *report* de *bug*? Como efetivamente escrever um bom passo de teste de um erro que ocorreu para ser corrigido depois? Seja este pela própria pessoa que o reportou ou algum outro desenvolvedor.

b) entendidas as necessidades que um *report* precisa atender, como deve-se avaliar e elencar os campos, funcionalidades e componentes que um *Bug Track System* deve possuir?

c) estas ferramentas podem e devem ser aperfeiçoadas para se adequarem a organização a qual estão atendendo, mas quais melhorias são necessárias e como fazê-las?

d) duas tarefas referenciando um mesmo problema são extremamente comuns em projetos de grande porte e com um vasto *backlog* de tarefas a serem corrigidas e desenvolvidas. Como, então, identificar duplicação de tarefas e ter o rastreio de problemas recorrentes?

Atualmente, segundo [Wazlawick 2013], o desenvolvimento ágil de software – que é uma metodologia de software que providencia uma estrutura conceitual para reger projetos de engenharia de software, é utilizado por diversas empresas e equipes. Interligado ao desenvolvimento de software tem-se o teste de software, elemento importante no ciclo de vida de um sistema e usado para garantir a qualidade final de um determinado produto. No desenvolvimento do software ágil, ao passo que o desenvolvedor programa um determinado algoritmo, módulo, classe, etc., o analista de qualidade faz o teste do mesmo. Ao final do processo, se erros (bugs) forem encontrados, eles são reportados em alguma ferramenta de armazenamento, chamadas de Bug Tracking System, e o programador, assim, consegue saber qual problema aconteceu e em qual parte do sistema, para que ele possa corrigir.

1.1 Laboratório Bridge

No Laboratório Bridge, que é um laboratório ligado à Universidade Federal de Santa Catarina e que atua na pesquisa e desenvolvimento de soluções tecnológicas, conectando governo e cidadãos, foi feita toda a pesquisa proposta neste trabalho abrangendo as equipes ágeis e o ferramental que seus colaboradores utilizam no dia-a-dia. O laboratório conta, atualmente, com 5 grandes projetos, que são:

- e-SUS - Atenção Básica, que é uma estratégia de informatização das unidades básicas de saúde (UBS) de todos os municípios em território nacional;
- Redesign e-SUS, visando uma melhor forma de apresentação de dados, tecnologias mais recentes e de ponta no mercado, alinhado a readequação de regras de negócio, o projeto e-SUS AB tem sido todo refeito, juntamente ao *framework* e ao *design system* com as características visuais do laboratório;
- Sismob - Sistema de Monitoramento de Obras, sistema que permite o monitoramento das obras de engenharia e infraestrutura de diversos estabelecimentos de saúde financiados pelo Ministério da Saúde. Projeto este que foi finalizado e entregue no ano de 2019;
- RNI - Registro Nacional de Implantes, sistema desenvolvido em parceria com a ANVISA e sociedades médicas, para rastreio e controle de qualidade de componentes

implantáveis em pacientes; e o

- BI - Business Inteligence, que é uma plataforma para consulta consolidada de dados dos atendimentos efetuados no sistema e-SUS AB, com objetivos de mensurar métricas dos atendimentos nas UBS.

O laboratório possui quinze equipes ágeis distribuídas nos projetos, sendo seis delas do projeto e-SUS AB, ao qual o autor já esteve inserido em uma destas equipes. Cada equipe é auto-organizável, autogerenciável e composta por colaboradores de forma interdisciplinar. Normalmente esta distribuição se dá aos pares, ou seja, dois programadores, dois testadores e dois analistas de requisitos. Esta formação não é uma regra geral, algumas equipes, por exemplo, contam com 3 programadores e 1 analista de requisitos. Existe também um núcleo especializado em *design*, composto apenas por profissionais da área. E outro núcleo de qualidade, com profissionais que desempenham o papel do usuário final do sistema e reportam erros, defeitos de usabilidade e inconsistências entre sistema e documentação. Esta é a última barreira para testes e descobrimento de *bugs* antes do sistema ir para produção.

O laboratório não segue a risca um determinado modelo ágil para gestão, planejamento e desenvolvimento de *software*, mas é fortemente inclinado para o *Scrum* (apresentado no capítulo 3, de embasamento teórico, na seção 3.3.2.1), com o catalizador Kanban (apresentado na seção seguinte à do Scrum).

1.2 Problemática

Nas equipes do projeto e-SUS AB, por elas serem autogerenciáveis, existem vários *softwares* para *report* de *bugs* e *backlogs* de tarefas. Cada equipe pode escolher e fazer uso da ferramenta que mais se adéqua à sua realidade. Em um levantamento informal realizado pelo autor, apenas perguntando aos colaboradores do projeto que ferramentas eles utilizam, foram identificadas as seguintes:

- [Github issues]: ferramenta localizada no próprio GitHub, onde todo o código das aplicações é armazenado, onde é possível apontar os problemas, criar *milestones*, definir prioridades e atribuições dos responsáveis por cada tarefa;
- [Planilhas Google]: ferramenta de uso geral para análise de dados e cálculos matemáticos, pode ser alterada para receber tarefas, criar células com nome dos responsáveis e *status* de cada atividade;
- [Redmine]: ferramenta de *software* livre, criada para gerenciamento de projetos, sendo esta a principal fonte formal de *report* de *bugs* pela equipe de qualidade às demais equipes do projeto;

- [Trello]: ferramenta *web* que disponibiliza cartões de afazeres e atividades, possui ideia similar aos *post its*, tem integração com GitHub, Redmine, dentre outras aplicações;

- [Asana]: também web e similar ao funcionamento do Trello, é uma ferramenta que ajuda no gerenciamento de projetos, rastreio e organização de equipes;

Foi identificado, através de perguntas informais aos membros das equipes, que recentemente fizeram a troca do *Bug Track System* utilizado para algum outro, obteve-se melhoria na produtividade e menor recorrência de um mesmo erro ou tarefa; o que motivou a execução deste trabalho de conclusão de curso. Outra motivação, foi o descontentamento da equipe com a ferramenta de *BTS* utilizada pela mesma, buscando sempre melhorias de processo e de produtividade. A partir de então teve-se a ideia de um estudo mais profundo sobre as ferramentas de *report* de *bugs*, como elas auxiliam no processo de desenvolvimento e teste ágil e o impacto que elas causam neste ambiente.

1.3 Objetivos

1.3.1 Objetivo Geral

Realizar um estudo de caso com foco nas equipes multidisciplinares do laboratório, a fim de saber como as equipes interagem com as ferramentas de *Bug Tracking System e Report*, o dia-a-dia delas com estas ferramentas e o *feedback* que elas proporcionam para cada equipe. São feitas avaliações e comparações destas ferramentas conforme a literatura e a realidade da organização. Verificam-se as limitações de cada uma delas, bem como possíveis melhorias para atender melhor os usuários, e elencar seus pontos positivos, os quais fazem as equipes optarem por essas ferramentas hoje.

1.3.2 Objetivos Específicos

É avaliado se existe ou não uma melhor ferramenta que se adéqua a cada particularidade da equipe e se esta equipe tem sua produtividade aumentada com o uso da ferramenta. Além disso, após a aplicação da pesquisa com base nos tópicos descritos por [Blair 2004], se terá uma maior noção das limitações de cada ferramenta. Busca-se, também, por propostas de melhorias nas ferramentas e estas podem ser levadas até as equipes ágeis para que possam aplicá-las e melhorar seus fluxos de trabalho. Outro fator importante a ser estudado, é se as equipes levam em consideração princípios ágeis tais como comunicação e se os consideram mais importantes do que o uso de ferramentas e processos.

1.4 Organização

O presente trabalho está organizado da seguinte forma:

- Capítulo 2: metodologias definidas e utilizadas para o desenvolvimento deste Trabalho de Conclusão de Curso: como se deu a pesquisa para trabalhos correlatos e a pesquisa interna organizacional para coleta de dados;
- Capítulo 3: embasamento teórico, para maior obtenção de conhecimento e clareza sobre os tópicos que serão apresentados no trabalho;
- Capítulo 4: revisão e estudo dos trabalhos correlatos, bem como a comparação de tópicos relevantes ao tema de acordo com as mais variadas referências elencadas;
- Capítulo 5: desenvolvimento - entrevistas, coleta e processamento de dados;
- Capítulo 6: conclusões retiradas após o término da pesquisa;
- Capítulo 7: possíveis trabalhos futuros com base no que foi concluído.

2 *Metodologia*

Neste capítulo serão apresentadas as duas metodologias de pesquisa utilizadas na construção deste trabalho. A primeira é em busca de conteúdo na literatura, ao passo que a segunda é uma pesquisa em forma de entrevista dentro do ambiente do laboratório Bridge.

2.1 Tipos de pesquisa

2.1.1 Bibliográfica Sistemática

Conforme [Kitchenham 2004], fazer uma revisão sistemática de uma literatura é identificar, avaliar e interpretar todas as pesquisas relevantes a um determinado tópico. Existem muitas razões para realizar este tipo de exame, como verificar novos parâmetros para a pesquisa em foco; identificar falhas na pesquisa atual e sugerir ideias; resumir evidências e elencar benefícios.

De fato, este tipo de pesquisa demanda um pouco mais de tempo e esforço comparado as revisões tradicionais. Porém, tendo uma metodologia bem definida, fica mais difícil fazer com que os resultados sejam tendenciosos. Além disso, caso as informações dos estudos apresentem resultados consistentes, as revisões sistemáticas fornecem evidências robustas.

Ao focar a atenção para o conteúdo deste trabalho, o autor tem bem definida a estratégia de busca que é detectar o máximo de informações possíveis e relevantes da literatura; critérios bem definidos e explícitos para uma ferramenta de *Bug Track System* ser considerada boa, tais como usabilidade e aclamar todas as características que um reporte de teste exige. Além disto, estes critérios são documentados para que os leitores possam avaliar o rigor e a completude dos mesmos.

2.1.2 Exploratória de Campo

Ao passo em que a pesquisa bibliográfica sistemática elenca a relevância de trabalhos correlatos, de um determinado tópico presentes na literatura, a pesquisa exploratória faz uma investigação sobre o assunto. "Pequisa exploratória é quando a pesquisa se encontra na fase preliminar, tem como finalidade proporcionar mais informações sobre o assunto que vamos investigar, possibilitando sua definição e seu delineamento, isto é, facilitar a delimitação do tema da pesquisa; orientar a fixação dos objetivos e a formulação das hipóteses ou descobrir um novo tipo de enfoque para o assunto. Assume, em geral, as

formas de pesquisas bibliográficas e estudos de caso." [Prodanov Cleber Cristiano; Freitas 2013]. São feitas entrevistas com pessoas que interagem com o objeto de estudo da pesquisa, ou seja, possuem experiência prática com as ferramentas descritas no capítulo anterior e possibilitam análise de exemplos que incitam a compreensão.

3 Fundamentação Teórica

Neste capítulo, os assuntos tratados são os que embasam o estudo com base na literatura da Engenharia de *Software*; mais especificadamente da parte de qualidade e de processos de *software*. Inicialmente, tem-se bem definido os conceitos sobre qualidade e teste de *software*, além de uma breve história a seu respeito, bem como os problemas que sua ausência ou pouca presença podem ocasionar. Depois, são caracterizados diferentes tipos de testes (todos realizados pelas equipes de qualidade dentro dos projetos) e uma seção separada para tratar sobre falha, defeito, erro. Além disso, é falado sobre verificação e validação. E, por fim, há uma seção explicativa sobre metodologias ágeis.

3.1 Qualidade de software

Conforme [Koscianski e Soares 2007] contam em seu livro, o termo qualidade é muito antigo. Não fora por ela, os egípcios, há milhares de anos atrás, não teriam feito construções - ditas pirâmides, tão perfeitas e precisas que fascinam a humanidade até os dias de hoje.

Importante salientar também a revolução industrial, outra referência para o termo qualidade na história da humanidade. Tiveram diversas mudanças na economia e automação de processos, criação de novas fábricas e, com elas, alta concorrência, o que estimula a melhoria contínua.

Em 1920 houve o surgimento dos controles de produções. Com base neles, ficava mais fácil e possível garantir que peças ou produtos separadamente tivessem em conformidade com suas especificações, mesmo que produzidos em larga escala. 20 anos depois, órgãos de certificação e controle surgiram, tais como a ISO (*International Stardalization Organization*, a ASQC *American for Quality Control* e no Brasil, a ABNT (Associação Brasileira de Normas Técnicas).

Na área de produção de *software* não foi muito diferente, especialmente no período pós-guerra, onde os computadores digitais estavam mais acessíveis para um maior número de pessoas, e não apenas para uso restrito de militares e acadêmicos; portanto, a qualidade dos programas se mostrava extremamente importante. Um dos grandes problemas da época, era que com o rápido avanço da tecnologia não haviam escolas ou cursos para a profissão de programador; na verdade, a profissão nem sequer existia: as primeiras pessoas a trabalharem com isso aprendiam e exercitavam a atividade de forma empírica. Segundo [Wazlawick 2013], aceita-se que o termo Engenharia de *Software* tenha sido utilizado desde 1950 e que a primeira conferência sobre o assunto foi a da OTAN, na

Alemanha, em 1968.

Nos anos 2000, organizações demandavam *softwares* de pequeno e médio porte, o que originou o surgimento de soluções mais simples e das metodologias ágeis. O processo agora era desburocratizado e adequado para equipes pequenas e competentes. Alinhado a isso, com a *internet* sendo mais facilmente acessada pela população, *softwares* comerciais, de tempo real ou embutidos necessitavam cada vez mais de qualidade, onde foi dado uma maior importância para testes, processos e programa final, ou seja, o entregável requisitado pelo cliente.

3.1.1 Controle de qualidade ou garantia de qualidade?

Um assunto não tão complicado, mas que ainda gera muitas dúvidas e equívocos; e até mesmo pessoas qualificadas se confundem com esses dois conceitos: controlar a qualidade e realizar a garantia da qualidade. [Bartié 2002] em seu livro Garantia da Qualidade de Software, explicita bem a diferença entre os dois.

O primeiro, basicamente, é feito quando se está testando (checando) uma entrega, uma tarefa, uma funcionalidade ou um artefato de um determinado projeto. Verifica-se se o *software* está atendendo os padrões de qualidade exigidos e especificados. Por exemplo, um programa que tem um requisito não funcional de desempenho, onde uma página da aplicação deve ser carregada em 2 segundos: checar se esta página carrega em 2 segundos é realizar o controle da qualidade. Portanto, como aponta o autor: "é um processo que se concentra no monitoramento e desempenho dos resultados do projeto para determinar se ele está atendendo aos padrões de qualidade no processo de desenvolvimento".

Já o segundo, engloba os processos aos quais o projeto será submetido para garantir o adequado desenvolvimento do produto. É checado se os processos estabelecidos nos projetos são de fato seguidos. Por exemplo, é definido que será necessário o sistema ter uma documentação e ela deve ser validada mensalmente com o cliente. Ou que o ciclo de desenvolvimento de uma funcionalidade se inicie sempre com o analista de requisitos junto ao cliente, passe para desenvolvimento e teste dentro da equipe ágil e, por fim, se encerre na equipe de qualidade. Então, checar se esse processo está sendo executado é realizar a garantia da qualidade.

Além disso, faz parte da garantia da qualidade fazer com que estes processos sejam melhorados, já que eles não devem ser estáticos e devem ser continuamente enriquecidos. Dentro destes processos, existem sub-processos que podem ser aperfeiçoados, tais como o foco deste trabalho: as ferramentas de *Bug Tracking Report* que são utilizadas no dia a dia das equipes ágeis do projeto e-SUS AB e como elas são utilizadas nas coletas e relatos dos erros encontrados; melhorar a comunicação inter e intra-equipes; validar casos de teste; entre outros. Portanto, é um conceito que diz que, ao ser executado este determinado

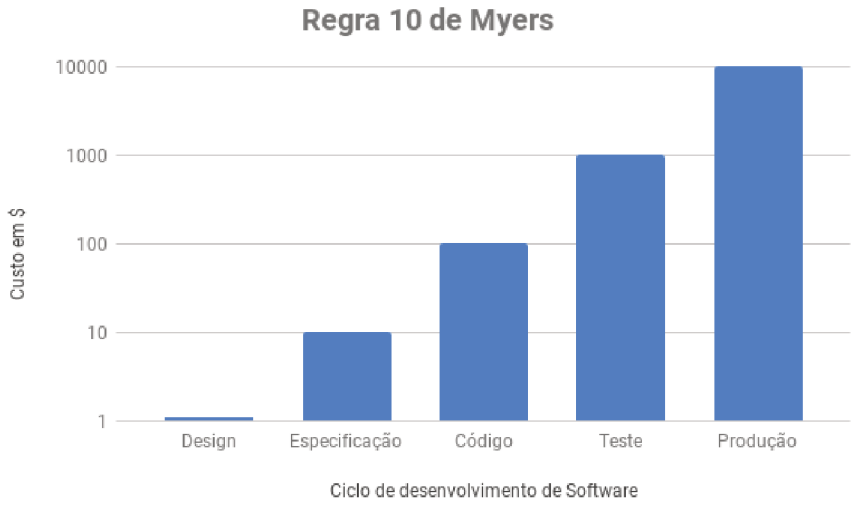
processo, ao final da entrega, haverá um produto com a qualidade especificada inicialmente.

3.2 Teste

Teste de *software* nada mais é do que "... o processo de executar um programa com objetivo de encontrar erros..."[Myers, 1979]. Se dada uma especificação, padrão ou convenção, uma determinada parte do sistema não está de acordo com o esperado dizemos que ali há um defeito. E, segundo [Koscianski e Soares 2007], o objetivo do teste é encontrar defeitos; e, a partir disto, ajustar o sistema para que funcione corretamente.

Quanto antes for identificado e corrigido um problema no *software*, menos recursos sua correção custará. É o que aponta a regra 10 de Myers (Figura 1), explicada por [Filho e Rios 2003]: "Os defeitos detectados quando o sistema ainda está nos seus estágios iniciais de desenvolvimento, podem ser 1000 ou mais vezes mais baratos do que aqueles corrigidos quando o sistema já está em produção"; ou seja, o custo para corrigir os defeitos aumenta exponencialmente dentro do ciclo de vida do projeto.

Figura 1 – Regra de 10 Myers



Fonte: o autor, 2019.

Ainda sobre o teste de *software*, [Wazlawick 2013] indica que testar um programa não é uma tarefa simples. Pode ser que elaborar um determinado caso de teste, ou seja, o passo a passo para se fazer o teste de uma funcionalidade, seja mais difícil e custoso do que produzir o próprio *software*. Portanto, é necessário que haja controle, sistematização e que a atividade tenha resultados efetivos e previsíveis. Não "testar apenas por testar"(modo *ad hoc* explicado a seguir), mas realizar a função com foco e direção nos pontos mais frágeis e suscetíveis a erros da aplicação.

3.2.1 História do teste de software

Em 1980, quando a indústria de desenvolvimento de *software* começou a dar mais importância aos requisitos e suas análises, bem como foi aumentado o esforço para com a integração das diversas partes que formavam o sistema, é que o teste de *software* começou a ser tratado como um processo formal, evoluindo até os dias atuais. Nas décadas anteriores, como afirmam [Rios e Moreira 2006], o desenvolvimento era voltado para codificação e testes unitários, não despendendo muito tempo na integração dos programas e testes de sistema. O teste era tido apenas como uma prova para o cliente (usuário final), que os produtos funcionavam; não sendo, assim, alinhados com o fluxo de desenvolvimento desses sistemas.

Atualmente, com o rápido ascender da tecnologia, juntamente ao advento da *internet*, a abrangência e complexidade das aplicações aumentou significativamente; juntamente, a exigência dos usuários por um produto de qualidade. Os autores declaram que um *site* fora do ar, situação esta ocasionada por um defeito no programa, pode comprometer a imagem da organização e causar sérios transtornos ao negócio. Portanto, a atividade de teste tornou-se cada vez mais especializada e requisitada, uma vez que fatores como segurança, usabilidade e performance se tornaram expressivos no ponto de vista de qualidade de uma aplicação.

[Koscianski e Soares 2007] trazem em seu livro dois exemplos de catástrofes que aconteceram, e em decorrência dos defeitos de *software* que levaram a quebra do sistema, milhões de dólares foram perdidos pelas empresas e até mesmo vidas humanas: o caso Ariane 501 e o Therac-25.

No dia 4 de junho de 1996, foi lançado o foguete Ariane 5. Passados 40 segundos do seu lançamento, o foguete desviou da rota e se autodestruiu em uma explosão. O que aconteceu foi uma falha proveniente de um trecho de código copiado do projeto anterior: o Ariane 4; que, aliás, não era necessário no atual projeto, dado o comportamento diferente dos dois foguetes. O problema facilmente seria corrigido através de um comando *if*, para tratar a exceção lançada quando foi tentado converter um número em ponto flutuante de 64 *bits* para um inteiro de 16 *bits*. Ou seja, um número muito grande para ser tratado em apenas 16 *bits*. A partir daí, o sistema de referência inercial (SRI) do foguete não forneceu dados corretos, fazendo o foguete cortar o ar de forma incorreta: o foguete desintegrou-se em virtude da alta carga aerodinâmica aplicada nele.

O caso do Therac-25 foi bem mais dramático, pois levou a morte de uma paciente por alta exposição de radiação. Ele era uma máquina de terapia radiológica e sua promessa era dar mais segurança na operação por meio de *software*. Mas o projeto continha diversos erros, inclusive de usabilidade, onde as mensagens de pane não eram exatamente claras o suficiente para os operadores do equipamento. O que ocorreu com a paciente foi que ao

fazer uma radiografia, a máquina apresentou erro de posicionamento horizontal e disse que nenhuma radiação havia sido emitida. O operador tentou colocá-la novamente em funcionamento e não conseguiu. Tentou por mais de 5 vezes, até a máquina precisar ser reiniciada. Ao todo, seis pacientes foram vítimas do Therac-25.

3.2.2 Tipos de teste

Os tipos de teste abaixo, com exceção do item *c*, têm como referência o livro de [Wazlawick 2013].

a) Teste caixa preta: é uma técnica que pode ser aplicada em todas as quatro fases de teste (descritas na próxima subseção), e consiste em avaliar as saídas dadas às diferentes entradas submetidas, sem entrar em contato com a estrutura do programa. O ideal para um teste é que todas as entradas possíveis sejam mapeadas e testadas, mas nem sempre isso é possível. Portanto, quanto mais casos o teste abranger, mais rico ele será. Tendo isso em vista, esse tipo de teste faz com que o testador monte variados casos de teste, direcionados para os pontos mais frágeis da aplicação. Uma boa prática, é fazer os testes baseados nos requisitos do sistema, assim, pode-se dizer que as funcionalidades especificadas foram acompanhadas e observadas em busca de erros.

b) Teste caixa branca: partindo da perspectiva interna do programa, os casos de teste são criados conforme o código da aplicação. Os testes são baseados na implementação, onde o testador tem acesso a todas as instruções dos métodos. Por se tratar de uma técnica um pouco mais complexa, necessita de um profissional mais capacitado, o que a torna mais custosa. Porém, ela possui a vantagem de que com a estrutura interna em mãos, servindo de referência, é mais fácil oferecer valores de entrada úteis para o teste. Essa abordagem é comumente usada para testes de unidade, mas também pode ser usada em testes de integração, de regressão e de sistema.

c) Teste caixa cinza: embora não haja uma referência bibliográfica para tal tipo, comenta-se muito sobre. Ele funciona com a mescla das duas técnicas descritas acima. Ou seja, o testador possui acesso aos algoritmos do sistema e executa os testes conforme a caixa-preta. É possível, por exemplo, olhar o código para verificar limites superiores e inferiores que o sistema aceita, além de mensagens de erro, para criar os casos de teste com bases nessas entradas e saídas esperadas.

3.2.3 Níveis de teste

Para exemplificar os níveis de testes existentes, [Delamaro, Jino e Maldonado 2017] os explicam em seu livro.

a) Teste de unidade: é a modalidade de teste de mais baixa escala, sendo aplicada diretamente nos componentes do código criado, isoladamente, tendo em vista garantir que

as funcionalidades implementadas correspondam às especificações do sistema.

b) Teste de integração: é executado em diversas partes do programa, em forma de combinações, para verificar se juntos funcionam de maneira correta. Pode-se ter testes de integração a nível de módulos, clientes e servidores, aplicações diferentes e partes do código.

c) Teste de sistema: normalmente executado pela equipe de testes, visa testar o sistema como um todo (ou uma parte dele separada em módulos). Neste nível de teste, a operação é controlada e realizada de forma normal, simulando todas as funcionalidades presentes e dando foco ao que ocorrerá no ambiente de produção. Aqui também são realizados testes de estresse, carga e desempenho.

d) Testes de aceite - *alfa* e *piloto*: são testes realizados diretamente pelos usuários (conduzidos pela equipe responsável pelos testes) finais da aplicação. Aqui, o cliente verifica se a solução dada realmente atende suas necessidades, objetivos de negócio e requisitos contratados. Neste processo são dadas atenções a usabilidade e a funcionalidade. Testes *alfa* e *piloto* são subtipos dos testes de aceite. O teste *alfa* é um teste restrito a poucos usuários e em ambiente controlado. Ele é realizado entre o término de desenvolvimento de uma aplicação e sua entrega, e é feito sob observação dos desenvolvedores. Já a fase de *piloto* é quando a aplicação é liberada para homologação junto ao cliente e usuários parceiros, sem que haja observação direta dos desenvolvedores e no ambiente onde o cliente vai instalar e executar a aplicação. Eventuais problemas encontrados, sugestões de melhorias ou adaptações são coletadas e devidamente ajustadas antes do sistema entrar em produção.

3.2.4 Outros tipos de teste

Também existem outros tipos de teste, estes são exemplificados por [Delamaro, Jino e Maldonado 2017].

a) Teste de regressão: consiste em retestar todo o sistema, assim que novas funcionalidades são adicionadas ou modificações são feitas. Tudo que já estava funcionando deve permanecer em pleno estado de execução. Normalmente, tem-se um conjunto de casos de teste e passos a serem seguidos a fim de verificar se as mudanças realizadas não danificam o que já está acertado.

b) Teste de carga/resistência/estresse (não funcionais): são testes de performance e são utilizados para avaliar confiabilidade e estabilidade de um sistema. Verifica, especialmente, operações que são frequentemente utilizadas a fim de identificar gargalos e possíveis travamentos ou baixas de desempenho. O teste de carga, por exemplo, avalia o comportamento do sistema em termos de tempo para dados, transações ou requisições que o sistema tipicamente teria. Já o teste de estresse é mais extremo. Levando o sistema ao

limite máximo, é possível ver se ele se comportará bem em momentos anormais de carga de trabalho. E o teste de resistência verifica se o programa mantém o seu funcionamento, de forma constante, quando submetido a uma carga de trabalho constante, por um longo período de tempo. Assim, pode-se aferir se há perdas ou degradação do uso de memória e de processamento.

c) Teste *back-to-back*: faz quando o mesmo teste é executado em duas ou mais versões diferentes do sistema e compara-se o resultado.

d) Teste de usabilidade: dado o ponto de vista do usuário, é verificado a facilidade no uso por parte deles. É um teste que deve ser muito prezado, principalmente em aplicações *Web*, visto que elas podem possuir muitos botões, campos e navegação entre páginas. As telas são avaliadas conforme sua clareza de entendimento, fácil e rápido acesso a informação, mensagens de ajuda de contexto e explicativas de possíveis erros. Fortemente subjetivo, por se tratar da opinião do usuário, não é recomendado que o próprio programador faça esse tipo de teste, pois ele já está com o "vício"no sistema por tê-lo desenvolvido.

e) Teste funcional: verifica se os requisitos descritos e especificados no contrato foram de fato implementados e de maneira correta, ou seja, testa as funcionalidades do sistema em si e se estão em pleno estado de funcionamento. Aqui, por exemplo, faz-se os testes de valores extremos e regras de negócio.

f) Teste de interoperabilidade: são testes de integração feitos com outros sistemas ou ambientes. Por exemplo, se o *software* deve consumir ou fornecer informação para outra aplicação, quais ações devem ser tomadas para estabelecer perfeita comunicação entre eles; e, caso ela não possa ser firmada, quais ações ou mensagens de erro devem ser tomadas para informar ao usuário o que aconteceu.

g) Teste de verificação *Web*: testes feitos com foco em explorar *links* que possam ter se tornado inválidos ou quebrados com o tempo (principalmente os que levam a outros *sites*), arquivos ou páginas que jamais conseguirão ser navegados, acesso lento a recursos ou interações entre páginas e componentes.

h) Teste *ad hoc*: é um teste feito sem planejamento ou documentação. Puramente, utiliza-se o sistema em foco, sem elaborar casos de teste, tampouco uma porta de entrada e um resultado esperado. Cliques, preenchimento de formulários e navegações são feitas com intuito de explorar o programa e fazer com que "estoure"algum erro. Por se tratar de um teste rápido e sem rumo definido, é comum que sejam achadas inconsistências; porém, sem rastros e casos de teste determinados, muitas vezes é difícil reproduzir o teste quando se encontra um erro.

i) Teste exploratório: é uma modalidade de teste que deve ser considerada complementar aos testes planejados. O testador não tem informações detalhadas sobre o que vai testar, mas, com base na sua experiência, realiza a atividade de forma intuitiva, empírica,

criativa e investigativa sobre os componentes do sistema, a fim de encontrar um erro ou uma inconsistência. Literalmente, explora e aprende sobre a aplicação, enquanto força cenários que potencialmente podem levar a erros.

3.2.5 Falha, defeito e erro

Embora essas palavras lembrem significados parecidos, falha, defeito e erro não são sinônimos entre si; e cada ideia representa um conceito diferente, muitas vezes gerando confusão e até equívocos. [Wazlawick 2013] cita em seu livro que:

a) Falha: é tudo aquilo que condiz com o não funcionamento do *software*, podendo ou não, ter sido gerado por um defeito no mesmo;

b) Defeito: codificação feita de forma incorreta no programa. Pode ser uma linha, uma função ou um conjunto de blocos incorretos que levam a um erro;

c) Erro: dada a observação entre resultado esperado durante o uso do sistema e o resultado obtido, o erro é a diferença vista.

[Koscianski e Soares 2007] trazem-nos um exemplo prático da diferença entre eles. Por exemplo, no seguinte trecho de código escrito abaixo:

$$a = receberValorFuncaoExterna();$$

$$c = b/a;$$

Pode haver um problema na segunda linha, caso o valor da variável a seja zero ou nulo, o que acarretaria em um *crash* do sistema e um problema de execução, pois é sabido que não se pode ter divisão por zero. Pode-se dizer, então, que esta linha apresenta defeito ou é defeituosa. Uma outra possibilidade de defeito, neste caso, é ele estar presente na função que se encarrega de produzir o valor de a, um engano cometido pelo programador responsável por aquela função.

O resultado errado provocado pelo defeito na segunda linha é dito ser uma falha. Porém, mesmo existindo defeitos, pode ser que as falhas nunca venham à tona. Imaginemos que o defeito esteja dentro da função *receberValorFuncaoExterna()* e que ela pode retornar um valor igual a zero, mas que isso nunca tenha acontecido. É uma potencial falha, porém nem sempre visível. Como dito anteriormente, a falha nem sempre é proveniente de um defeito, ela também pode ser ocasionada por outros fatores como corrupção de base de dados, invasão de endereçamento de memória e problemas com *hardware*.

3.2.6 Verificação, validação e teste

a) Verificação: é a parte que trata de analisar para visualizar se o programa foi realmente feito como o especificado nos requisitos, se está livre de defeitos e possui

características de qualidade;

b) Validação: avalia se o que foi desenvolvido, depois de pronto, atende a real necessidade dos usuários;

c) Teste: atividade responsável para realizar os exames de verificação e validação.

Conforme [Wazlawick 2013], o processo de verificação se preocupa em responder a seguinte pergunta: "estamos fazendo do jeito certo?". Enquanto a pergunta para a parte de validação é: "estamos fazendo a coisa certa?". Ambas são examinadas pelos testes de *software*, com base nas especificações já descritas acima; sendo a maior parte dos níveis e tipos de testes destinados a fase de verificação, enquanto os testes de aceite fazem a parte da etapa de validação.

3.3 Modelos de processo

Metodologias foram criadas para ajudar no processo de desenvolvimento de um *software*. Elas descrevem, de forma documentada, o passo-a-passo dentro da produção. Várias destas metodologias foram criadas e podemos dividir o estudo em dois tipos: as prescritivas (do modelo clássico) e as ágeis (aqui exemplificadas por *Scrum* e *Kanban*). As ágeis, sobre as clássicas, afirmam garantir melhorias de produção e de qualidade.

Dentre todos os métodos, elenca-se quatro fases que devem estar presente neles, independente de tipo ou fundamentação:

- Especificação;
- Projeto e implementação;
- Validação;
- Evolução.

Na fase de **especificação**, devem ser definidas todas as funcionalidades e características do produto. A fase de **projeto e implementação**, diz respeito a como o projeto é produzido de acordo com suas especificações; propondo modelos com base em diagramas, para posteriormente serem codificados. A fase de **validação** trata sobre revisão de código e requisitos, bem como a parte de testes para certificar que tudo esta funcionando seguindo o que fora especificado. E, por fim, a fase de **evolução** que corresponde a manutenibilidade e adaptação do *software* conforme demanda e necessidade de mudanças.

Segundo [Koscianski e Soares 2007], muitas organizações desenvolvem seus *softwares* sem seguir nenhum tipo de processo. [Wazlawick 2013] chama esse método de *anti-modelo* ou *Codificar e Consertar*. Em resumo, esse modelo consiste em perguntar ao cliente o que

ele quer que seja desenvolvido; implementar alguma coisa; mostrar uma versão preliminar ao cliente e ficar corrigindo ela até que ele seja satisfeito; realizar testes finais e correções de erros (que serão inevitáveis); e por fim, entregar o produto. O resultado dessa falta de sistematização, infalivelmente, faz com que a entrega do *software* desenvolvido seja atrasada e dificultosa, de forma com que a qualidade final do produto seja prejudicada. Pode-se, inclusive, inviabilizar futuras melhorias e evoluções do sistema.

Não há um modelo que seja melhor que o outro. Projetos diferentes têm necessidades diferentes. A escolha do processo a ser seguido, pelo engenheiro de *software*, deve ser bem pensada e adequada à realidade do projeto. Do contrário, pode-se ter uma equipe frustrada, situações de retrabalho ou retrocesso no processo de desenvolvimento.

3.3.1 Modelo prescritivo

Antigamente, o desenvolvimento de *software* era feito de forma muito diferente do atual. Uma vez que não existiam ferramentas para auxiliar o programador, tais como analisadores e depuradores de código, o programa era todo planejado e documentado antes de ser implementado. Além disso, o acesso as máquinas era muito limitado, o que tornava a realização de alterações em código muito custosas. A principal metodologia utilizada era a de modelo clássico, também conhecida como cascata.

O modelo em cascata, determina que a produção seja feita em etapas sequenciais. Sempre ao término de uma etapa, tem-se associado uma documentação que deve ser aprovada para prosseguir à próxima etapa no ciclo de desenvolvimento. Nesta metodologia, depois da fase de desenvolvimento, não se prevê mais mudanças nas especificações; fato que o torna muito limitado para ser utilizado apenas em projetos que os requisitos são certamente estáveis.

Seguindo a mesma regra 10 de *Meyers*, o custo para mudanças quando o produto já está pronto pode ser cem vezes mais caro do que quando foram levantados os requisitos. Inclusive, se, durante o teste de uma funcionalidade for verificado que é necessário incluir um campo novo no sistema, essa funcionalidade será incluída no código fonte final da aplicação, mas não será voltado na fase de documentação e incluso lá. Essa prática, de não alterar modelos já feitos, pode acarretar em dificuldades para produzir futuras alterações, uma vez que o sistema não estará consistente com sua documentação.

3.3.2 Modelos ágeis

Como apontam [Wazlawick 2013] e [Koscianski e Soares 2007], também conhecida por *processos leves*, a metodologia ágil teve sua popularidade no ano de 2001, quando 17 especialistas da área de processos de desenvolvimento de *software* se reuniram para discutir princípios que diversos métodos ágeis compartilhavam em comum. Na ocasião,

houve o estabelecimento do *Manifesto ágil*, que diz o seguinte:

a) *Indivíduos e interações estão acima de processos e ferramentas*: processos bem estruturados não adiantam e não têm serventia se pessoas não os seguem;

b) *Software funcionando está acima de documentação compreensível*: de nada vale uma rica documentação, se o *software* não atende e satisfaz os requisitos do cliente;

c) *Colaboração do cliente está acima de negociação de contrato*: a prioridade é satisfazer o cliente, com entregas rápidas, contínuas e sistemas com valor de uso;

d) *Responder às mudanças está acima de seguir um plano*: mudanças são extremamente bem-vindas, inclusive no final do projeto, e a mudança é vista como um diferencial competitivo para o cliente.

É interessante salientar que os processos, ferramentas, documentação, contratos e planos ainda são considerados importantes, bem discutidos, valorizados e também participam do princípio de melhoria contínua. Porém, eles só tem sentido e valor depois de indivíduos, colaboração do cliente, reposta a mudanças e *software* funcionando, forem colocados em primeiro lugar. Conforme exemplificado na Tabela 1 mostrada a seguir:

Tabela 1 – Conceitos chave do Manifesto ágil

(+) indivíduos e interações	processos e ferramentas
(+) software executável	documentação
(+) colaboração do cliente	negociação de contratos
(+) respostas rápidas a mudanças	seguir planos

Dentre as mais diversas metodologias ágeis existentes, as mais conhecidas são *Scrum*, *Kanban*, *XP - Extreme Programming* e *Crystal Clear*, sendo as duas primeiras explicadas nas próximas subseções, por serem as metodologias adotadas no laboratório *Bridge*, local onde este trabalho está sendo realizado.

3.3.2.1 Scrum

No *Scrum*, as funcionalidades a serem implementadas são mantidas em uma lista e se faz um planejamento juntamente ao dono do produto para saber quais funcionalidades serão priorizadas. Assim a equipe seleciona a atividade que será capaz de implementar durante uma *Sprint*, que, segundo [Wazlawick 2013] é um dos conceitos mais importantes deste modelo ágil. Elas são ciclos tipicamente mensais, mas podem variar de 15 a 30 dias. Ao final de cada *Sprint* são apresentados os resultados. A cada dia, a equipe faz uma pequena e rápida reunião, chamada de *Daily Meeting*, com objetivo de disseminar conhecimento, identificar empecilhos e alinhar todos os colaboradores.

Em geral, o recomendado é que cada time seja composto de 6 a 10 pessoas. [Wazlawick 2013] afirma ainda que "No caso de projetos muito grandes, é possível aplicar

o conceito de *Scrum of Scrums*, em que vários Scrum *teams* trabalham em paralelo e cada um contribui com uma pessoa para a formação do *Scrum of Scrums*, quando então as várias equipes são sincronizadas.". Conceito este que é aplicado semanalmente no projeto *e-Sus AB*: todas as equipes do projeto se reúnem por cerca de 15 minutos e são apontadas as principais tarefas desenvolvidas individualmente por elas ao longo da semana, dando ênfase nos impactos que essas tarefas possam causar no trabalho de outros times.

3.3.2.2 Kanban

O *Kanban* tem se tornado uma extensão de muitos modelos ágeis tradicionais utilizados nas empresas atualmente, pois possui foco persistente no fluxo e no contexto de produção, sendo considerada menos prescritiva quando se comparado ao ágil.

Kanban, de origem japonesa, tem significado "cartão visual"e é um sistema utilizado a mais de cinco décadas pela fabricante automotiva *Toyota*, para visualmente controlar e equilibrar suas linhas de produção. Atualmente, o termo vem se relacionando muito com as áreas de tecnologia da informação (TI), auxiliando no processo de desenvolvimento de *software*.

Esse método presa em visualizar o trabalho em andamento, ou melhor, ver passo a passo do trabalho e sua cadeia de valor; bem como medir e gerenciar o fluxo para que se possa tomar decisões e ter noções de consequências após tomadas de ações.

Parte do pressuposto de que o princípio da melhoria contínua é de responsabilidade de todos e ajuda a identificar pontos e oportunidades de melhorias.

Ele também limita o trabalho em progresso ou, em inglês, *WIP - (Work In Progress)*, restringindo o total de trabalho permitido para cada estágio, de forma que nunca se possa colocar mais trabalho no sistema do que sua capacidade de processá-lo e terminá-lo. Ou seja, é necessário que se complete todo o trabalho iniciado antes de se começar um novo. Dessa forma, as funcionalidades não são realizadas conforme previsões ou demandas, mas sim com base na capacidade do sistema de executá-las.

[Boeg 2010] em seu guia sobre *Kanban*, explica que tanto *Kanban*, quanto *Scrum* (ou qualquer outra metodologia ágil) não são opostos entre si, tampouco um é melhor que o outro. Por funcionar como um agente de mudanças ou um catalisador para condução de mudanças, o *Kanban* necessita de um ponto de partida. Muitos projetos são bem-sucedidos iniciando com *Scrum* e utilizando *Kanban* para impulsionar futuras mudanças.

4 Revisão dos Trabalhos Correlatos

Como dito no capítulo 2, existem dois tipos de metodologias de pesquisa que foram aplicadas neste trabalho: a bibliográfica sistemática e a de campo. Neste presente capítulo, será apresentada a primeira; e, no capítulo de desenvolvimento, será mostrado a pesquisa de campo e seus resultados.

Para a pesquisa de trabalhos correlatos foi escolhida a ferramenta de busca e de base de dados *Google Scholar*. É uma ferramenta desenvolvida pela própria *Google*, que está em funcionamento desde 2004, e permite pesquisar trabalhos acadêmicos, literatura escolar, periódicos e artigos científicos. A escolha dela deu-se por ser uma base mundial, internacionalmente utilizada e conhecida.

Os termos de busca colocados no campo de pesquisa são os que compõe as palavras-chave deste trabalho: *bug track system, bug reports, improving report tools* e todos eles foram colocados na língua inglesa, embora alguns resultados apareceram em língua portuguesa.

Dessa pesquisa inicial, foram trazidos aproximadamente 990 resultados. Neste presente momento, os trabalhos são selecionados conforme seus títulos e pertinência quanto ao tema em questão, restando apenas 100 de todo o monte. Não foram levados em consideração períodos como data de escrita ou de publicação. Após isso, um novo peneiramento foi feito, novamente em cima do título, elencando apenas 28 dos 100 já selecionados. Desses 28, todos os *abstracts* foram lidos, com a ideia de se obter mais clareza (do que apenas lendo o título), sobre o que se tratava tal trabalho. Aqui, foram criados seis macro grupos de estudo, e, de acordo com o trabalho, ele era colocado dentro de algum destes grupos. São eles:

- Aperfeiçoamento de *BTS* (3 ocorrências);
- Avaliação de *BTS* (3 ocorrências);
- *BTS* em relação a duplicação de tarefas (4 ocorrências);
- O que faz um bom reporte de *bug* e como configurá-lo em um *BTS* (2 ocorrências);
- Quem é o responsável designado para corrigir um determinado *bug* (3 ocorrências);
- Quanto a tempo e severidade de *bugs* (4 ocorrências).

Os demais 9 trabalhos que não entraram em nenhum desses 6 grupos após a leitura dos seus resumos, foram descartados, tendo como base o critério de exclusão de não fazer parte do escopo central do desenvolvimento do trabalho.

Destes 19 trabalhos restantes, todos foram lidos na íntegra, e apenas 1 de cada grupo foi escolhido para ser trabalho correlato. Aqui, também, as seções de **tempo e severidade de *bugs*** e **quem deve corrigir o *bug*** foram retiradas, por fugirem do escopo.

Por conseguinte, os 4 trabalhos restantes são descritos a seguir. Começando por como fazer um bom reporte de *bug*, algo que deve ser levado em consideração, pois, não importa ferramenta, o que for escrito no reporte pelo usuário deve ser entendível para qualquer outra pessoa da equipe, cliente ou que estará lendo o problema relatado. Depois é falado como se avaliam essas ferramentas, com base em um guia. Este guia traz uma série de tópicos e funcionalidades que os *BTS* deveriam ter para serem considerados boas ferramentas. Após isso, tem-se uma seção para falar sobre aperfeiçoamento dessas ferramentas, porque, dentro de um ambiente de desenvolvimento ágil, todo o processo de desenvolvimento deve ser melhorado continuamente. E por fim, como duplicações de tarefas podem causar prejuízos ou benefícios e como elas podem alterar o ciclo de vida de um projeto.

4.1 Como fazer um bom reporte de *bug*

[Bettenburg et al. 2008] afirmam que os registros de erros são vitais para qualquer desenvolvimento de *software*. Mas ao passo que os programas precisam ter qualidade, os reportes de *bugs* também precisam ter o mínimo de detalhe necessário para o programador encontrar a falha no código que está ocasionando tal erro. São encontradas descrições de tarefas tais como "A página está muito desajeitada. (Mozilla Bug #109242)"; o que faz com que identificação e resolução do problema sejam mais demoradas e custosas.

Neste trabalho foram entrevistados desenvolvedores e reportadores da Apache, do Eclipse e do Mozilla. Ao final da pesquisa, os maiores problemas com os reportes de erros evidenciados pelos desenvolvedores foram os com pouca ou incompleta informação, juntamente com erros nos passos de como reproduzir (sendo estes considerados os problemas mais severos), casos de teste e duplicação de tarefas. Versionamentos incorretos, comportamentos observados e esperados, e fluência na linguagem do reporte também foram salientados por eles.

Um bom reporte de *bug* precisa ser direto e ter a garantia de que a pessoa que irá ler entenderá tudo que ali está escrito. Precisa ter com detalhes todo o passo-a-passo necessário para reproduzir o erro, e, sobretudo, nada mais além disso. É desejável, por exemplo, que ele possua itemização: estruturar os passos em itens e não apenas em um texto corrido que prejudica a leitura; palavras-chaves bem definidas: que categorizam de antemão o problema ocorrido, o local e o motivo; exemplos de códigos e se possível, o local onde o erro ocorre: diminui e muito o tempo de procura da linha no código por parte do

programador; *stack traces*: identificam exatamente o ponto de execução do programa onde o erro pode ter acontecido; capturas de tela: em geral, uma imagem do ocorrido ou um vídeo explicativo do passo-a-passo ajudam a entender como o erro aconteceu; legibilidade: sem dúvidas a forma como o reporte é escrito e o estilo de escrita do documento contam para a boa qualidade do reporte.

4.2 Avaliando *Bug Track Systems*

Tendo em vista que uma organização é composta por variadas disciplinas de profissionais, tais como: testadores; programadores; analistas; dentre outros, é interessante pensar que cada um destes papeis têm visões diferentes sobre os reportes de *bugs*. O programador é mais objetivo e quer saber onde efetivamente o código está com a falha. O analista tem o olhar mais crítico e verificará que aquilo que o cliente quer não é o que o programa está fazendo. Já o testador identificará a inconsistência entre o que foi pedido pelo analista e o que foi feito pelo programador.

Segundo [Blair 2004], o primeiro passo é identificar e entender cada papel, bem como suas responsabilidades no uso do sistema e processo de reporte de erros. Depois, responder algumas perguntas, tais como:

a) que tipo de informação será necessária reportar?

b) como se dão os passos de resolução no processo como um todo? Quem são os responsáveis por escrever e resolver cada reporte?

c) quais são as métricas utilizadas para cada reporte?

d) usuários diferentes necessitam de tipos de acessos diferentes?

Uma vez identificadas as respostas dessas perguntas, tem-se uma lista de características a se levar em consideração:

- Adaptabilidade (nem sempre é desejável que o *BTS* sirva apenas para reportar problemas. Muitas vezes essas ferramentas também servem como um repositório de *backlogs* ou até mesmo ordens de compra e venda; portanto, é necessário que a ferramenta se adapte e consiga se moldar para atender os anseios da organização. Campos, notificações, fluxo de trabalho: tudo isso deve ser adaptável. É importante que a ferramenta possua *templates* para diferentes tipos de *issues* tais como: tarefas de suporte, casos de teste, e claro, reportes de *bugs*);

- Rastreabilidade (a ferramenta mantém toda a linha de modificações feitas em uma determinada tarefa, seja para auditoria, possível futura revisão ou em uma recorrência de *bug* identificar mais rapidamente onde ocorreu. É importante também que a ferramenta indique quem realizou as alterações e o que essa pessoa alterou com

clareza. Por exemplo, o que é mais claro e detalhado: simplesmente dizer "Artefato modificado"ou "Prioridade alterada para Urgente após conversa com o gerente do projeto"? Certamente, a segunda opção aliada a um bom sistema de notificações causará maior impacto e comoção da equipe);

- Integração com o controle de versionamento de código (um ponto que vale a pena ressaltar é a capacidade dessas ferramentas fazerem um *link* entre o *bug* e o código fonte do programa; ou seja, ter uma forma do documento que descreve o erro estar ligado ou relacionado com o código, podendo assim ter um maior rastreio e saber onde efetivamente mexer para correção do problema);

- Campos customizáveis (é importante ter essa *feature* no *BTS*, porque a partir dela pode-se coletar as informações realmente necessárias de acordo com o modelo de reporte de *bug*; e nada mais além disto, evitando o excesso dispensável de dados irrelevantes. Por exemplo se uma ferramenta provê classificação de *bugs* por prioridades é interessante que ela permita modificar estes níveis de acordo com a utilização do usuário. Telas customizáveis, conjuntos de campos e a forma como eles aparecem para os usuários também são levados em conta);

- Usabilidade (assim como qualquer outro programa, a *interface* precisa ser consistente e de fácil navegação, amigável, simples e limpa. Por exemplo, deve ser fácil e rápido adicionar um novo *bug* na ferramenta; as *features* do programa são de fácil acesso (com poucos cliques de mouse); as mensagens presentes na tela são entendíveis);

- Notificações (são de fato *features* diretamente relacionadas com o fluxo de trabalho e ajudam no gerenciamento dos *bugs*. Elas podem manter os usuários informados sobre mudanças tais como na prioridade dos problemas encontrados, novos *bugs* adicionados, uma nova pessoa ser designada a correção ou algum *status* alterado. É interessante que a ferramenta possua uma forma automática de envio de *e-mails* indicando as alterações e que também seja de fácil customização para indicar os ocorridos);

- Segurança (dependendo do cenário, é preciso que alguns grupos de usuários tenham permissões diferentes de outros. Nem todos são aptos a fazerem modificações como inclusão, edição, visualização ou customizações. Algumas funcionalidades devem ser apenas superficiais. Um repositório comunitário, por exemplo, não deveria permitir que um usuário comum tenha privilégio de exclusão de *issues*; visto que todo o trabalho produzido por outros poderia ser descartado por um erro ou até mesmo de forma maliciosa);

- Métricas (é útil que um *BTS* provenha, de forma rápida e organizada, informações e relatórios, com métricas detalhadas para tomadas de decisão. Por exemplo: número

de *issues* criadas ou corrigidas, organizadas por data e prioridade; sinalizar *bugs* que ainda não foram corrigidos e sua funcionalidade é requerida na próxima entrega; ter uma biblioteca de reportes predefinidos; listar os *trending bugs*, os mais acessados ou recorrentes);

- Fluxo de trabalho (sobretudo, além das características desejáveis elencadas acima, uma ferramenta para reporte de *bugs* necessita se enquadrar nos fluxos e processos de trabalho utilizados na organização. Ele está presente no ambiente de desenvolvimento ágil para otimizar o processo como um todo e ajudar a gerenciar e resolver os erros de forma mais clara e concisa. É importante indicar aqui também os estados que uma determinada *issue* pode assumir, como: nova, em desenvolvimento, em testes, fechada, duplicada, rejeitada, dentre outras. O *BTS* deve ser capaz de rastrear as informações de alteração dessa tarefa, como uma transição de *nova* para *em desenvolvimento* e posteriormente para *em testes*; e o profissional que as fez. Pode-se, inclusive, como ideia, juntamente à característica de segurança supracitada permitir que apenas o grupo responsável pela qualidade altere a tarefa de *em testes* para *fechada* ou *corrigida*).

4.3 Aperfeiçoando *Bug Track Systems*

Em conversa informal com alguns desenvolvedores do laboratório, foi questionado a eles o que os testadores podem escrever nas tarefas que os ajudaria mais a descobrir a origem de um *bug* e posteriormente corrigi-lo. O mais indicado foi informar aonde efetivamente o erro aconteceu, como por exemplo no *breadcrumb* X, componente ou campo Y. Também é desejável um passo-a-passo de como reproduzir o erro. Este passo-a-passo pode ser um caso de teste, mais técnico, juntamente de capturas e de gravações de tela para auxiliar. *Logs* de erros e comportamentos observados e esperados também são pontos desejados na hora da criação das *issues*.

Frente a isso, [Zimmermann et al. 2009], em pesquisa da *Microsoft*, elencou 4 pilares a serem melhorados pelos *Bug Track Systems*: ferramental, informacional, processual e a nível de usuário; e centrado no pilar de informação, conduziu o estudo com *machine learning* como proposta de aprimoramento a um *BTS*.

Do ponto de vista ferramental, o autor afirma que para melhorar um *BTS* pode-se automatizar alguns processos, tais como: localizar a *stack trace* e adicioná-la ao reporte, reproduzir os passos do usuário, gerar casos de teste; reduzindo assim a carga de informação desnecessária colocada nas tarefas.

Para melhorar o pilar informacional, o foco é direcionado para a informação dada pelo usuário reportador. Alguns *BTS* contam com um *feedback* em tempo real, que dão noção à pessoa que está escrevendo a tarefa de quão bom está o seu reporte; outros

conduzem a escrever da melhor forma possível para o desenvolvedor entender onde está o problema.

Já a parte de processos, tem foco na administração; como por exemplo triagem de *bugs*, relatórios de problemas encontrados e corrigidos, automatização de para qual desenvolvedor a tarefa deve ser direcionada.

Por fim, aprimorar o pilar centrado no usuário, aproximando o reportador do desenvolvedor. Os programadores podem auxiliar e treinar os usuários para escreverem tarefas mais concisas e técnicas, detalhando a informação que realmente interessa para resolver o *bug*.

Voltando para a parte informacional, algumas perguntas devem ser respondidas tais como: qual o componente que apresentou defeito? Em qual *build*? Sobre o que é o *bug*? Qual o passo-a-passo para reproduzi-lo? Essas são todas perguntas que se encaixam para qualquer reporte de erro. Portanto, a proposta foi criar um sistema inteligente que automaticamente faz perguntas relevantes ao usuário, garantindo assim todas as informações necessárias para abrir a tarefa. A seleção de quais perguntas fazer não seria estática e não seguiria uma ordem, dependendo de cada resposta escolhida, novas ações serão tomadas, e novas perguntas aparecerão. Ao final, teria-se uma melhor descrição do *bug*. Inicialmente as perguntas seriam de forma geral, como pedir capturas de tela, qual *build* ocorreu o problema, inclusão de *stack traces*. Depois mais aprofundadas, como componentes e campos em específico.

A partir de reporte de *bugs* previamente criados, seria construído modelos de *machine learning*, que conduziriam a seleção das questões e faria a predição do local da origem do defeito com base nas respostas.

4.4 Duplicação de tarefas: um problema recorrente

Em grandes projetos, especialmente, aqueles que contam com um amplo número de colaboradores, é complicado manter um repositório central com todos os *bugs* encontrados esperando para correção, rastreio dos antigos erros e garantir que as tarefas que estão com situação de *abertas* não são duplicadas.

O repositório no *Redmine* do projeto e-Sus AB tem hoje mais de 11 mil tarefas criadas. Destas, aproximadamente, 970 estão com situação de aberta. E desde o dia 07 de agosto de 2014, foram criadas aproximadamente 75 tarefas duplicadas; isto é 1,5 tarefa duplicada por mês de trabalho. O processo de criação de novas tarefas é um pouco custoso, visto que quando um problema é encontrado, primeiro checa-se no repositório se já há alguma tarefa para aquilo. Isto leva tempo, uma vez que a pesquisa é manual por meio de palavras-chave no título da tarefa.

[Bettenburg et al. 2008] elenca 5 razões para uma pessoa duplicar uma tarefa:

- Usuários preguiçosos ou sem experiência (afirma que alguns usuário não procuram se outras pessoas já tiveram um problema parecido e simplesmente abrem uma nova tarefa, ou eles não possuem experiência suficiente em reporte de *bugs*);
- Ferramenta de pesquisa fraca ou ineficiente (quando a busca do *BTS* não retorna ou retorna dados que não correspondem exatamente com o que o usuário quis pesquisar sobre o erro que ele deseja reportar. Neste caso, ao visualizar que não existe nenhuma *issue* a qual ele encontrou presente na base de dados, ele criará uma nova);
- Mais de uma falha ocasionada pelo mesmo defeito (nem sempre é visível que duas falhas distintas, ou seja, problemas na execução, são originadas de único defeito - um erro no código. Inclusive, um mesmo usuário pode abrir 2 tarefas neste caso, e ao resolver um deles, o outro poderá ser resolvido também);
- Duplicação intencional (acontecem mais em repositórios de *bugs* abertos a comunidade, que, depois que um usuário abre uma tarefa e não obtém resposta, intencionalmente ele abre outra igual em busca de correspondência);
- Duplicação acidental (quando os usuários reportadores clicam mais de uma vez no botão de criar tarefa, e elas são criadas acidentalmente mais de uma vez. Normalmente elas aparecem como criadas pela mesma pessoa e no mesmo horário, com mesmo título, descrição e características).

O autor afirma ainda que nem sempre duplicações de tarefas são prejudiciais. Alguns desenvolvedores consideram que certas *issues*, ao serem duplicadas, recebem informações adicionais sobre o problema, o que ajuda a diagnosticar e resolver o problema mais rapidamente. Outro ponto, é que quanto mais pessoas reportam um mesmo erro, mais ele tende a se tornar importante, pois mais usuários estão se deparando com ele nos seus fluxos de trabalho e teste. Ainda assim, neste último caso, seria mais interessante, ao invés de duplicar a tarefa, colocar um botão de ocorrências. Sempre que um novo usuário se deparar com o *bug*, clicará neste botão e ficará salvo que uma nova ocorrência aconteceu.

5 Pesquisa de campo

Neste capítulo será apresentada toda a pesquisa de campo. Desde quais as perguntas elaboradas para as entrevistas até seus resultados. Aqui também são mostradas as configurações de cada equipe ágil, ou seja, por quais especialidades seus membros possuem; e quais ferramentas utilizavam ou ainda utilizam no seu dia a dia.

5.1 Dado o local de aplicação

A pesquisa de campo, foi aplicada no Laboratório Bridge, que é um laboratório da Universidade Federal de Santa Catarina (UFSC), onde o autor deste trabalho ocupa a função de estagiário em qualidade de *software*.

5.2 Espaço de experiência

O ambiente conta com aproximadamente 120 colaboradores multidisciplinares alocados em um dos três projetos em andamento dentro do laboratório. E, mesmo retirando os profissionais que não fazem parte de times ágeis, como os do administrativo, limpeza e gestão, ainda seriam muitas pessoas para aplicar a pesquisa e coletar informações. Portanto, o escopo das entrevistas foi reduzido para apenas o líder de cada uma das equipes ágeis.

Não há problema em ouvir apenas um colaborador por equipe, no caso os líderes, pois dentro do laboratório, os líderes refletem os anseios dos demais, ou seja, a resposta do líder de usar uma determinada ferramenta não se dá por decisão única, mas sim, porque em conversa com a equipe todos optaram por ela e chegaram a um consenso que ela era a melhor de se utilizar.

5.3 Composição das equipes ágeis

Nesta seção estão descritas as composições, ou seja, a disciplina dos membros das equipes ágeis que compõe o projeto e-SUS AB. Percebe-se que nenhuma possui o padrão [2, 2, 2] - dois programadores, dois testadores e dois analistas; tampouco possuem quantidades iguais de membros.

Tabela 2 – Composição das equipes

Equipe	Programadores	Testadores	Analistas	Designers
A	3	2	1	0
B	3	2	1	0
C	3	2	1	0
D	2	2	1	1
E	2	2	1	0
F	4	1	1	2

5.4 Quanto ao processo de pesquisa

A pesquisa de campo se deu a partir de uma conversa com cada entrevistado e foi direcionada pelo autor deste trabalho com base em dez perguntas, são elas:

- 1) Qual(is) ferramenta(s) de BTS a equipe utiliza no dia a dia? Se usam mais de uma, porque o fazem? Há limitação em alguma das ferramentas sendo necessário o uso de outra?
- 2) Utilizavam alguma outra ferramenta antes desta? Qual? O que motivou a mudança? O que melhorou?
- 3) Por que utilizam tal(is) ferramenta(s)?
- 4) Quais os pontos positivos da ferramenta atual?
- 5) A ferramenta atual possui alguma limitação? Tanto do ponto de vista da equipe (para o seu uso corrente no dia a dia) ou que os usuários acham que ela possa conter no geral.
- 6) Sobre melhorias na ferramenta atual:
  - a) Quais pontos já existentes da ferramenta poderiam ser melhorados?
  - b) E quais funcionalidades poderiam ser adicionadas para melhorá-la?
- 7) Todos da sua equipe utilizam essa ferramenta? Se não, por quê?
- 8) O que você pensa sobre o *Redmine*?
- 9) O que você acha sobre duplicação de tarefas? É um problema?
- 10) Dê uma nota de 1 a 5 quanto a cada um dos tópicos a seguir: adaptabilidade, rastreabilidade, integração com versionamento de código, campos customizáveis, usabilidade, notificações, segurança, métricas e fluxo de trabalho.

- Notas sobre as perguntas

- Na questão 8 foi perguntado sobre a ferramenta *Redmine*, pois ela é a ferramenta central, utilizada por todos os projetos e colaboradores do laboratório. Também é por ela que o cliente pode acompanhar a evolução das tarefas e, se desejar, abrir tarefas entregáveis, sugerir mudanças, dispor opiniões e comentários para desenvolvimento.

- Os tópicos da questão 10 foram pensados e retirados do guia disposto por [Blair 2004]. O guia foi colocado e exemplificado como um trabalho correlato, descrito no capítulo 4, que trata sobre os trabalhos correlatos.

5.5 Resultados das entrevistas

Nesta seção é explicado, de forma dissertativa, as respostas de cada uma das equipes, após a conversa (em forma de entrevista), dirigida com base nas nove primeiras perguntas da seção anterior. Aqui são descritas as percepções de cada uma com base no seu dia a dia de trabalho e levantadas questões relevantes com base nas respostas delas.

5.5.1 Sobre quais ferramentas cada equipe usa

Tabela 3 – Ferramentas utilizadas atualmente por cada equipe

Equipe	Github	Trello	Asana	Quadro físico	Planilhas Google
A	X				
B	X				
C	X				
D	X				
E			X	X	
F	X				

Pela tabela 3, podemos perceber que praticamente todas as equipes do projeto e-SUS AB fazem uso da ferramenta Github. Apenas uma equipe, não o utiliza e, em vez dele, faz uso do conjunto Asana e quadro físico. Isso se deu porque a maioria das equipes decidiu migrar para o Github com base nas funcionalidades que ele provê, principalmente alta integração com o código e centralização em uma única ferramenta. As mudanças e os motivos delas são acompanhados na próxima pergunta (subseção), onde é perguntado qual ferramenta era utilizada antes e o porquê da troca. Quando se idealizou fazer este trabalho de conclusão de curso, as equipes faziam uso das ferramentas descritas na subseção 5.5.2; portanto, um uso bem heterogêneo de ferramentas. Com o passar do tempo e durante o desenvolvimento deste trabalho, as equipes migraram suas ferramentas.

A única equipe (E) que alegou não fazer uso do Github disse que, antigamente, fazia uso apenas do Asana para tudo: planejamento e *backlog* de tarefas e *sprints*, desenvolvimento de atividades e reportes de *bugs* encontrados no sistema. Porém, ela sofreu uma reestruturação no seu processo de trabalho e, hoje em dia, como faz o planejamento semanal e implantou o *daily meeting*, torna-se mais saudável e visível o uso do quadro físico. Tudo se deu a partir de um experimento no uso das duas ferramentas que deu certo e a equipe encontrou uma melhor forma de trabalhar. Foi dito também que o Asana era de certa forma meio complicado de se usar, já o quadro físico era possível mexer durante a reunião todos os dias nas tarefas, assim, todos sabem tudo o que os outros membros da equipe estão fazendo ou vão fazer. As tarefas não são replicadas nas duas ferramentas e o quadro físico funciona como uma extensão do Asana. O segundo guarda as tarefas macro e definições com o cliente e supervisor de desenvolvimento, já o primeiro tem tarefas menores e o andamento delas dentro do fluxo da equipe. Por exemplo, no Asana é dito formalmente que uma funcionalidade terá quatro características e no quadro físico são colocadas essas quatro características em forma de tarefas palpáveis em diferentes *post-its*.

A equipe D disse que acha interessante o quadro físico pela forma como ele aproxima o time, ao passo que artefatos físicos são mais palpáveis e ajudam na comunicação interna. Aponta ainda que: "(...) não fazemos uso do quadro por preferir a ideia do Github de centralizar tudo numa única ferramenta, mas talvez o físico se sobressaia ao Github por causa da interação dentro do time.".

5.5.2 Ferramentas anteriores às atuais e motivação para a mudança

Tabela 4 – Ferramentas utilizadas anteriormente por cada equipe

Equipe	Github	Trello	Asana	Quadro físico	Planilhas Google
A		X			
B					X
C		X			
D		X			
E			X	X	
F	X				

Anteriormente ao uso do Github, as equipes A, C e D faziam uso da ferramenta Trello, no passo que a equipe B usava a ferramenta Planilhas Google para reportar suas tarefas. As equipes E e F, desde sua criação, sempre utilizaram as ferramentas descritas na seção anterior.

Todas as equipes que migraram de ferramenta disseram que os principais motivos foram por questões de centralização em uma única plataforma, integração com o código e automatização de processos. As equipes A e C informaram que por falta de automatização

nas suas ferramentas, elas facilmente ficavam desatualizadas. Por exemplo, por muitas vezes se esquecia de atualizar o *status* de desenvolvimento de uma tarefa no seu *board*. "Estávamos trabalhando em uma tarefa e aconteceram algumas vezes dela ser integrada e acabar ficando o *card* esquecido lá na ferramenta. Hoje não, com a automatização não precisa mais disso. Ele (o Github) manda sozinho de *doing* para *done*", afirma uma das equipes.

Outros pontos indicados pelas equipes foram a maior facilidade no uso e melhoria nos filtros de pesquisa da ferramenta atual perante a anterior, por se tratar dela ser mais completa, simples e otimizada. A equipe B disse que: "Basicamente conseguimos fazer tudo hoje no Github que fazíamos na planilha, só que de forma muito mais melhorada e otimizada. Agora conseguimos filtrar e organizar melhor as *issues*. Outro ponto foi a automatização e conseguir referenciar tudo dentro de tudo.".

A equipe E disse que já pensou em fazer uso do Github, mas que ele pecava em alguns pontos, tais como não ter um visual agradável e não prover a funcionalidade de criação de sub-níveis dentro de uma mesma tarefa, que é muito utilizado pela equipe e que é presente no Asana. E completa dizendo que: "Certamente, não usamos todo o potencial da ferramenta, mas também há a questão histórica. Funcionamos muito bem assim e precisamos de um motivo muito bom para mudar. Já tivemos várias discussões para migrar, mas ainda não vimos vantagem.".

5.5.3 Principais motivos para uso das ferramentas e seus pontos positivos

Os usuários da ferramenta Github alegaram que ela possui um módulo chamado de *Issues*. Neste módulo é possível reportar e descrever tarefas de forma muito parecida como já era feito anteriormente ao se abrir ou revisar um *Pull Request*. Ele tem a mesma interface, fluxo e identificadores que os *PRs*. Isso torna possível referenciar recursos diretamente entre eles, por exemplo fechar uma determinada tarefa ao integrar um código que fazia referência a ela.

O fato de tudo estar presente dentro de um único sistema de forma centralizada e unificada foi relatado por todos os entrevistados, em conjunto a parte de automatização que o coloca à frente das demais ferramentas. Além disso, ele possui menos burocracia, mais facilidade de uso e de aprendizado. Outro ponto indicado por duas equipes foi a parte de integração que ele tem com outros programas ou *plugins*. Na parte de desenvolvimento, por exemplo, existem várias ferramentas de teste e de *build* contínuo, que indicam dentro do *PR* se há alguma anomalia e tudo isso pode ser configurado para o código não ser integrado até que tudo seja corrigido.

Já a equipe que faz uso de Asana e quadro físico, diz que a principal característica

da segunda ferramenta é a aproximação do time e o forte impulsionamento que ele provê para a comunicação interna dar certo. Ele é altamente customizável e os membros podem mexer manualmente nas tarefas. Ele tem ajudado muito no *daily meeting* e aponta ainda que pelo fato das pessoas retirarem um tempo ficando paradas em frente ao quadro, há uma maior absorção dos afazeres durante a semana. Já o Asana possui várias características que a equipe utiliza diariamente e que não encontraram em outra ferramenta, tais como a criação de sub-níveis dentro de uma mesma tarefa. Por exemplo, uma mesma tarefa pode ter o nível de *teste* e dentro do *teste* ter sub-níveis de *teste automatizado*, *testes manuais* e *casos de teste*. Outro ponto forte dito foi a possibilidade de criar *checkboxes*, *checklists*, rápidos cliques para marcar itens como completos.

5.5.4 Limitações das ferramentas

Para a ferramenta Asana, a limitação mais discutida foi a falta de integração direta com o Github para fazer o versionamento de código e também a falta de automatização que ela tem, comparando com a outra ferramenta. Essa seria uma funcionalidade que ajudaria muito a equipe e, ela disse ainda que acha que esse é o principal motivo dos outros times estarem migrando de plataforma. Além disso, o Asana provia uma forte limitação para a equipe, conforme explicado, para realização dos *daily meetings*. Motivo esse pelo qual adotaram o quadro físico. Durante a reunião diária, eles precisariam fazer uso de um projetor, por exemplo, para verificar as informações presentes no Asana.

Já quando questionados sobre o quadro físico, apontaram que a maior limitação é a de segurança, uma vez que ele não tem ligação com nada. Qualquer pessoa com acesso a sala onde o quadro está pode alterar ou omitir as informações presentes nele. Ele também pode possuir limitações físicas, em questões de tamanho. Mas a equipe alegou que isso não chega a ser um problema, por ele ser grande nunca precisaram utilizar todo o espaço.

A ferramenta Github recebeu bastantes e variadas críticas sobre seu funcionamento. As mais comentadas foram sobre não dar suporte para múltiplos times ou quando muitos usuários necessitam utilizar a ferramenta ao mesmo tempo. A equipe D disse, por exemplo, que acha muito bom o funcionamento da ferramenta para 1 único time. Quando se coloca um segundo time dentro do mesmo repositório já fica mais complicado; e, acima de dois times, pesa pela desorganização e caos que isso pode causar. Disse que: "É complicado gerenciar muitas *issues*. Todas elas vão aparecer na mesma listagem para todas as equipes. Tenho medo de ficar insustentável, até porque nunca tivemos essa experiência". E completou comparando com o Redmine, pois tem medo que aconteça de ficar muitas tarefas perdidas, que ninguém sabe que elas ainda existem ou não sabem o que fazer com elas.

Outro ponto levantado pelas equipes A e C, ainda relacionado com a interação entre os times, foi de que quando eles desejam fechar uma tarefa dentro do próprio repositório, fazendo uso da automatização provida por ela, mas abrindo um *Pull Request* para outro

repositório externo, é necessário que a pessoa responsável por aceitar o PR no segundo repositório tenha privilégio de administrador do primeiro. Disse que é complicado ter que dar esse tipo de privilégio para pessoas de fora do time apenas para utilizarem dessa funcionalidade. Acredita ainda, que o Github tem essa política por fatores de segurança, mas não vê isso com tanta criticidade para ter que funcionar dessa maneira. O líder da equipe alegou que: "Já participei de outras equipes que por causa dessa limitação, eles não adotaram essa ferramenta. Essa parte toda de automatização do Github é muito ferida quando se trata de repositórios distintos".

A equipe F, que utiliza o Github *issues* desde que foi criado, também apontou que a automatização é limitada e eles gostariam que existissem mais opções. Eles fazem uso do módulo *projects* do Github. Essa funcionalidade é bem parecida com os *cards* do Trello e do Asana, porém possui automatização e pode ser relacionado com *issues* e *Pull Requests*. Quando alguma alteração for feita neles, um *card* do *board* é movido para a aba de *doing* (tarefas em desenvolvimento) para *done* (tarefas completadas), por exemplo. Mas nem sempre o fluxo da equipe funciona assim. Eles possuem outra coluna dentro da etapa de desenvolvimento, e gostariam que o *card* fosse movido por dentro dela também, o que hoje não é possível. Outro ponto dito pela equipe foi que quando eles abrem um PR, é necessário que pelo menos duas pessoas revisem e aprovem-no para que ele seja integrado. O que acontece com a automatização, atualmente, é que se uma única pessoa aprovar, ele já troca o *status* de em desenvolvimento para feito. Eles gostariam, então, que houvesse mais configurações de automatização, para customizar essa parte e indicar que o *card* só trocará de coluna de forma automática se n pessoas aprovarem as alterações no código.

A equipe B disse que não vê nenhuma limitação na sua ferramenta atual que impeça seu uso. Apontaram que pelo time ter uma ótima comunicação, não importa a ferramenta que usem, o processo ainda vai funcionar. Relembraram que quando usavam as Planilhas do Google, possuíam a limitação de filtrar tarefas de forma eficiente; uma vez que era necessário utilizar a ferramenta de pesquisa do próprio navegador (comando $ctrl + f$) para encontrar algo na página. E isso era ruim, pois eles precisavam digitar exatamente o que desejavam, uma vez que não existe um algorítimo de busca como das ferramentas mais complexas. Porém, como estavam sempre conversando, o fluxo de trabalho era bom e produtivo.

Um ponto levantado pela equipe que eles não tinham problema anteriormente e que têm hoje com o uso do Github, é que o histórico, principalmente na parte de evolução das *issues* pode acabar ficando muito grande e atrapalhar quando necessitarem saber algo dentro dela. Citaram o exemplo de quando adicionam ou retiram *labels* da tarefa, o histórico é mantido. Gostariam de poder filtrar para ler apenas comentários ou trocar a forma de apresentação, para que o mais recente fique em cima. Atualmente a linha do tempo de uma tarefa no Github é da mais antiga primeiro para a mais recente no final da

página.

5.5.5 Melhorias discutidas

Muitas melhorias foram sugeridas pelas equipes, de acordo com suas necessidades de uso diário e também de acordo com as limitações apontadas por elas anteriormente.

A equipe A, exemplificando, disse que gostaria que ficasse mais claro alguns funcionalidades que o Github possui, mas que ficam escondidas. Muitas vezes eles precisam recorrer a documentação da ferramenta para saber como proceder. Outra situação de melhoria discutido foi a questão dos filtros, que, do ponto de vista deles, possui sintaxe própria e não amigável para usuários não tão avançados. Por exemplo, hoje, a ferramenta de comunicação das equipes ágeis com o cliente e equipes de qualidade e suporte do projeto é o Redmine; e, caso todo o laboratório migrasse para o Github, tanto o cliente, quanto alguns membros da equipe de suporte (que não são da área de tecnologia) teriam que aprender a usar, o novo filtro.

Uma outra melhoria apontada por essa equipe, juntamente da equipe F, é sobre adequações no processo de automatização de tarefas e *Pull Requests*: como explicado na seção anterior, hoje é de certa forma limitado e essas equipes gostariam de ter mais customizações e liberdade para automatizar melhor seus processos internos.

Já a equipe C, alertou sobre algumas funcionalidades que estão presentes na ferramenta Trello, mas não estão presentes no Github, que é a possibilidade de adicionar pequenos compontens nos *cards* e nas tarefas com apenas um clique. Esses componentes podem ser *checklists*, listas e anotações. Hoje, eles precisam fazer uso de *Markdown* para criar essas customizações nas suas *issues*. Outra questão levantada que existe no Trello e não no Github é a possibilidade de atribuir *labels* diretamente dentro de um *card*. Atualmente, é necessário sair da página, criar a *label* e depois adicioná-la manualmente no *card*.

A equipe D disse que, no momento, é essencial para a ferramenta e também para o laboratório, que tenha alguma forma de organizar, visualizar e separar as *labels*, os *cards* e *projects* do Github por times. Realmente, implantar a noção de times nesse sistema, que hoje não existe. No presente momento, ela conta apenas com usuários independentes e é necessário contornar isso, para a realidade atual do projeto, com a criação de vários repositórios (diferentes para cada equipe ágil) que abrem PR para um principal.

Dando continuidade nas melhorias para a ferramenta Github, a equipe B contribuiu dizendo que gostaria que a parte de visualização de uma tarefa fosse mais organizada e customizada de acordo com o que eles necessitavam ver no momento. Atualmente, quando se abre uma *issue*, é fornecido o título dela e uma descrição opcional. Conforme vão se adicionando artefatos nela, ela vai aumentando seu tamanho de forma progressiva para

baixo. De modo que o mais antigo aparece no topo e o mais recente no final da página. Ao adicionar *labels*, descrições, comentários e referências na tarefa, o seu corpo só tende a crescer. Muitas dessas informações podem ser irrelevantes, na hora de verificar algo dentro da *issue*, caso ela tenha tomado proporções gigantes. A equipe então relatou que seria interessante a ferramenta prover um filtro, para ocultar ou mostrar determinados tópicos. Por exemplo apresentar apenas os comentários, ocultando as outras modificações. Outra melhoria sugerida foi a de poder ordenar a *timeline*, ao invés de ser de forma cronológica (da mais antiga para a mais recente), ser da forma contrária (da mais recente para a mais antiga); mostrando o mais atual primeiro (em cima).

E por fim, a equipe E, que faz uso das ferramentas Asana e Quadro físico apontou melhorias pontuais nos dois casos. Para o quadro, eles disseram que seria interessante se os *post-its* já viessem pré-preparados para receber as tarefas de acordo com a realidade deles. Hoje, eles usam um papel totalmente em branco e, nele, desenham pequenas caixas para colocar data de entrada e saída do quadro e outras customizações padronizadas pela equipe. Para a ferramenta virtual, eles disseram que gostariam que ela tivesse interação com o Github, porque hoje não possui.

Em pesquisa, redigindo esse trabalho, o autor verificou que de fato não existe interação nativa da ferramenta Asana com o Github, mas encontrou uma ferramenta terceira chamada [Unito]. Com ela é possível não só sincronizar as duas ferramentas em questão como também outras ferramentas existentes no mercado: Trello, Jira, Gitlab, Bitbucket, Basecamp e Wrike. Dessas descritas comportadas pelo Unito, apenas o Trello está presente no escopo deste trabalho, por ser relevante para as equipes do laboratório.

O Unito consegue conectar e sincronizar as ferramentas e, a partir disso, é possível aplicar ações que serão direcionadas para as demais ferramentas tais como fechar tarefas, atualizar títulos e descrições, atribuir um colaborador à *issue*, adicionar *tags* e *labels*, fazer *upload* de arquivos e comentários. Ele tem como desvantagem ser uma ferramenta paga, mas possui uma versão de *free trial* para teste. Constatado tudo isso, foi então apresentado a equipe essa ferramenta, que, por sua vez, comprometeu-se a avaliar e possivelmente implementar seu uso no dia a dia.

5.5.6 Sobre os usuários

Todas as equipes afirmaram que todos os seus integrantes, sem exceções utilizam as ferramentas. As equipes que possuem profissionais da área de *design*, afirmam ainda que nem sempre essas pessoas participam ativamente do fluxo no dia a dia. Mas sempre que há uma atividade que os compete, eles fazem uso em seu fluxo de trabalho.

5.5.7 Ferramenta central do projeto: Redmine

Nessa parte da entrevista houve bastante discussão e convergência das equipes nas suas respostas. Todas apontaram diversos problemas que o Redmine possui hoje em dia, tais como *bugs*, deficiência de filtros, falta de integração com código e a forma burocrática com a que ele trabalha.

Embora ele apresente todas essas dificuldades de uso, foi unânime entre todos que ele é "o mal necessário", hoje, na realidade do Laboratório Bridge. Com ele há a organização necessária, principalmente para comunicação externa às equipes ágeis. Através dele é que é feita a comunicação de reportes de *bugs* encontrados pela equipe de qualidade, chamados técnicos abertos pela equipe de suporte em atendimento aos municípios, onde é realizada a triagem e direcionado as tarefas; e, ainda, o cliente comenta diretamente nas tarefas e atribui novas funcionalidades a serem desenvolvidas no sistema.

As equipes sempre buscam diminuir o número de ferramentas utilizadas no dia a dia, mas como o Github ainda não provê uma interação satisfatória intertimes, atualmente, a ferramenta mais viável é o Redmine. Além disso, mesmo que com percalços, ele funciona muito bem e cumpre o objetivo.

Os pontos positivos encontrados nele são a centralização de tarefas em uma única base que ele provê, burocracia (necessária padronização para que haja entendimento de todos), altamente customizável e de código aberto e funcionamento aceitável na realidade atual. Já os pontos negativos são pequenos *bugs* que impedem de pré-definir filtros, os filtros em si serem pobres de customizações e a falta de integração com o código da aplicação. Hoje, por exemplo, é necessário copiar o *link* do código e fazer upload manual do documento de *pdf* da documentação dentro de uma tarefa.

5.5.8 Duplicação de tarefas: um problema do passado

Essa questão acabou sendo interessante não pelo fato da duplicação de tarefas ser um problema ou não. Mas sim da limitação de processo contida hoje no dia a dia do laboratório Bridge, especialmente no projeto e-SUS AB onde a pesquisa foi aplicada.

Todas as equipes apontaram que não veem a duplicação de tarefas como um problema, ou ainda, não existem muitas recorrências delas, sobretudo internas, atualmente no fluxo de trabalho. Em contrapartida, um ponto abordado por todos foi que, ao abrir uma tarefa sobre um determinado problema encontrado, a pessoa precisa antes fazer uma busca por esse problema na base do Redmine, pra verificar se ele já foi encontrado por outra pessoa e reportado lá. Essa atividade de pesquisa, por vezes, é mais custosa e demorada do que o tempo que levaria para reportar a própria tarefa, como aponta uma das equipes: "Coloca aí na conta: tempo para achar o *bug*, tempo para reportar o *bug* da melhor forma possível para outras pessoas entenderem, e ainda ter que gastar tempo procurando na

base de dados se aquele *bug* já foi encontrado antes e reportado, isso é muito ruim. Às vezes o tempo de fazer isso tudo é menor do que o tempo de ficar procurando por uma tarefa, pra ver se ela já existe.".

As equipes indicaram ainda, que uma forma de mitigar essa limitação, é a ferramenta prover de uma boa funcionalidade de filtro e de pesquisa. "O usuário nunca vai escrever perfeitamente o que ele quer. A ferramenta que for mais eficiente nesse sentido é a que menor terá duplicação de tarefas.", completa uma das equipes. Com base nisso, eles disseram que os filtros e a forma de pesquisa do Redmine é ruim e contra-produtiva. A quantidade de tarefas que estão presentes na base é um gravame: mais de 11 mil e delas, 9% estão com situação de aberta. Um exemplo de pesquisa realizado agora: pesquisado por "cns do cidadão", retornou 467 resultados, sendo 39 abertas. Deve ser olhado título a título qual se parece mais com o problema encontrado e, as que mais se enquadrarem, ler as descrições e validar se o caso já foi reportado ou não.

Na equipe F, que trata de readequar o sistema para novas tecnologias e, portanto, está sendo feito todo o redesign do produto, juntamente a análise e refatoração do código, foi dito que apenas a analista faz uso do Redmine. Ela verifica apenas tarefas que ainda estão abertas e que deverão ser corrigidas no novo sistema. A equipe em si não chega a reportar tarefas para o Redmine. Em contrapartida, eles alegaram que já tiveram recorrências de tarefas duplicadas no início, dentro do Github. Eles disseram que a ferramenta não provê uma forma eficiente de categorizar a *issue* e a forma como eles criavam tarefas contribuía para casos de duplicação.

Era feito uma tarefa macro e, dentro dela, várias tarefas pequenas em forma de *checklists*. Isso era ruim, porque a tarefa macro demorava a ser fechada (necessitava que todos os *checklists* fossem preenchidos). E caso isso ocorresse, às vezes, eles desmembravam a tarefa em tarefas menores, separadas e diferentes. Nesse processo poderia ocorrer de ficar uma ou outra duplicada. Para contornar o problema, eles decidiram escrever os títulos de forma padronizada. Hoje, então, colocam entre colchetes, no início do título o módulo que a tarefa se refere e depois sim o relato do problema. Assim eles conseguem filtrar mais fácil o módulo que ele ocorre. Alertam ainda que foi uma decisão da equipe pela padronização, a ferramenta fornece apenas o campo de busca.

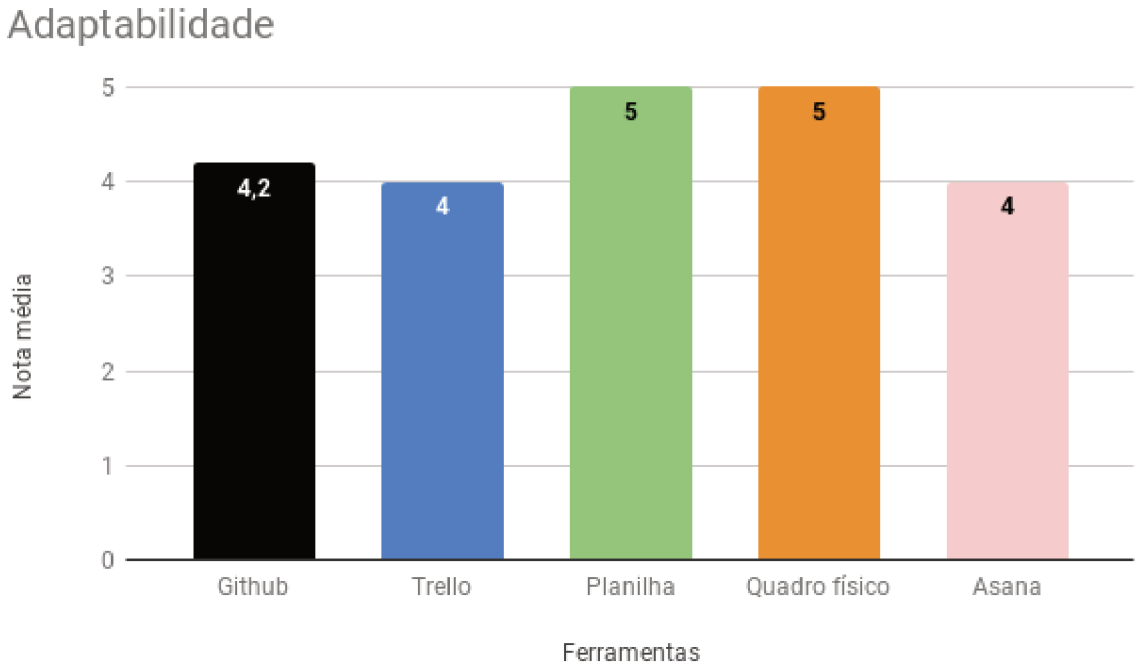
5.6 Discussão sobre os tópicos do guia de Blair

Esta seção é comparativa e compreende as notas da pergunta de número 10 com os entrevistados. Cada subseção apresenta um dos referidos tópicos do guia de [Blair 2004]. Os tópicos foram explicados e detalhados para os entrevistados, os quais deveriam atribuir uma nota de 1 a 5, sendo 1 para muito ruim e desparelho com a realidade da equipe e 5 para muito bom e completamente incluso no dia a dia do processo de trabalho. Os

participantes eram questionados sobre o porquê da atribuição daquela nota, realizando isso em forma de uma pequena explicação. A nota foi obtida através de média simples de cada uma das respostas dos entrevistados.

5.6.1 Adaptabilidade

Figura 2 – Adaptabilidade

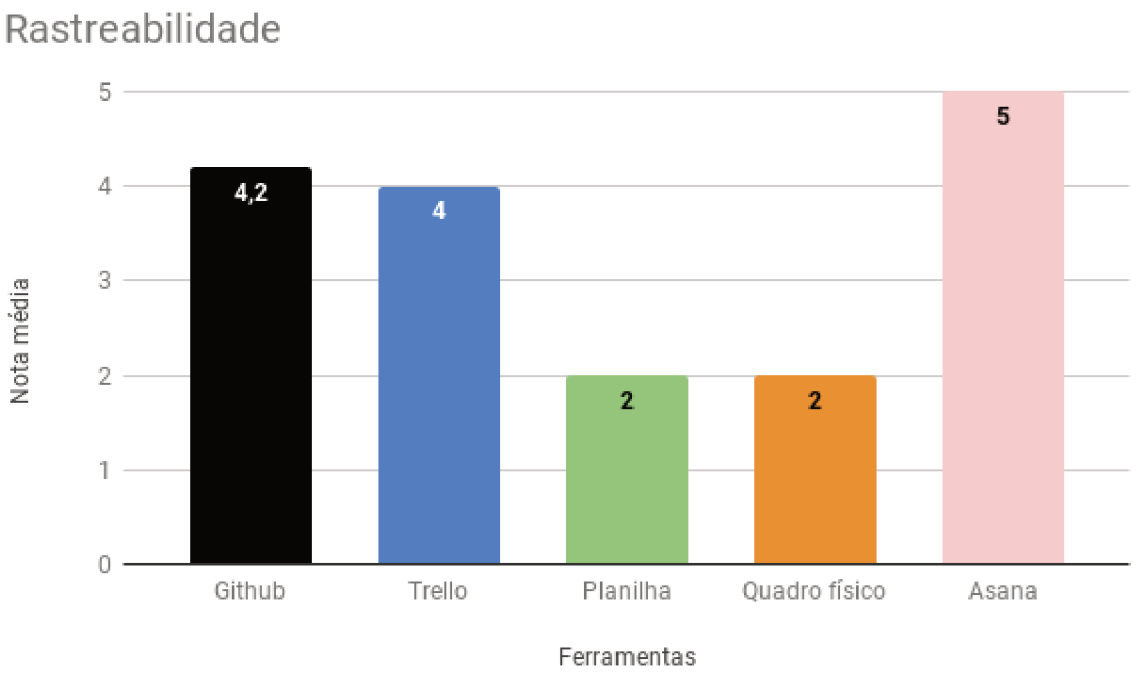


Fonte: o autor, 2019.

A notas médias presentes neste tópico foram bem altas, sempre acima de 4 como se pode perceber no gráfico. Isso se deve pelo fato das equipes serem independentes, autogerenciáveis e, portanto, poderem escolher as ferramentas que mais se adaptam às suas realidades. Nos três casos que não se obteve nota máxima, as principais causas foram a falta de determinadas automatizações e também a forma de organização dos *boards* e das tarefas quando uma outra equipe está envolvida no mesmo repositório. Essas ferramentas não possuem funcionalidades adequadas para lidar com o quesito intertimes, portanto os entrevistados se sentiam limitados nas suas configurações caso outras equipes precisassem fazer uso da mesma ferramenta.

5.6.2 Rastreabilidade

Figura 3 – Rastreabilidade



Fonte: o autor, 2019.

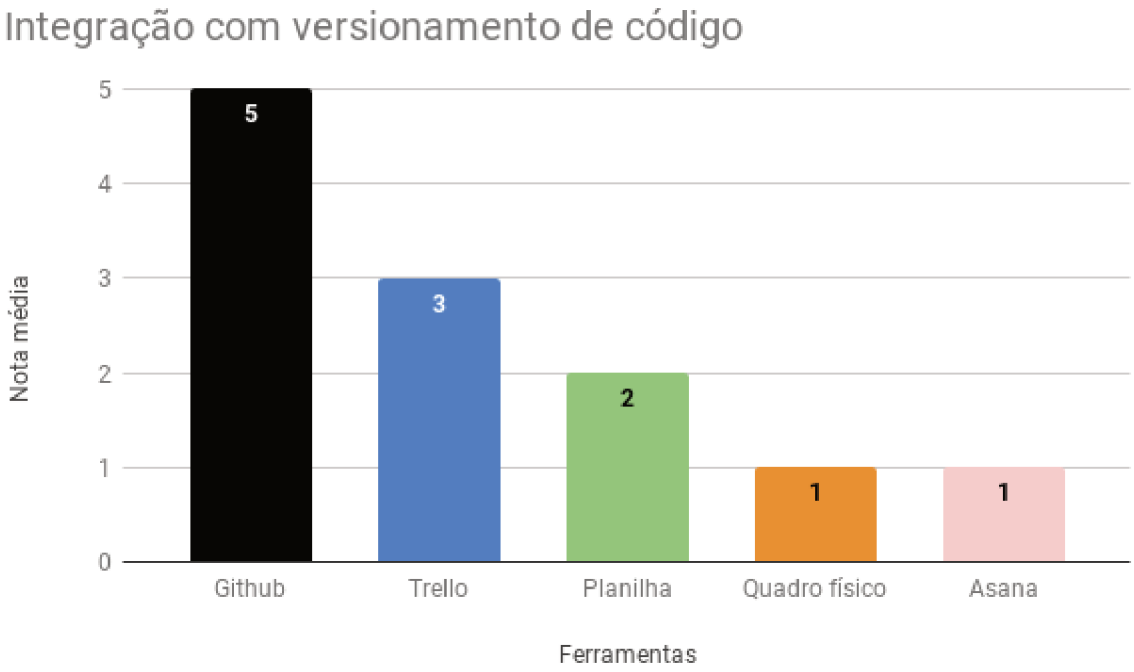
Quanto a rastreabilidade, os entrevistados de Github, Trello e Asana estão contentes com a forma que a ferramenta provê isso à eles. Algo notado, particularmente na ferramenta Github, é a forma como pesquisas podem ser mais complicadas para usuário não tão avançados. Isso se dá, porque neste sistema de reporte de *bugs* há uma sintaxe avançada e especial para fazer pesquisas dentro dos filtros, por exemplo "is:issue"ou "is:open", quando se quer relatar que o que se quer procurar é uma tarefa e ainda está com *status* de aberta. Em contrapartida, uma outra equipe relatou que não necessariamente isso é algo ruim, uma vez que os usuários dessa ferramenta, dentro do ambiente de desenvolvimento ágil do Laboratório Bridge, não são leigos, eles facilmente podem utilizar esse artefato poderosíssimo para fazer pesquisas mais refinadas.

Os usuários de planilha e quadro físico, apontaram que a rastreabilidade fica um pouco complicada uma vez que as ferramentas que eles utilizam são de propósito geral, e não especificamente para reporte de *bugs*. Na planilha, por exemplo, é necessário utilizar a ferramenta de localização de caracteres provida pelo navegador (*control-f*) e ainda, assim, ela só achará o que se pesquisa exatamente com as palavras digitadas. Não possui nenhum tipo de inteligência artificial ou método de pesquisa avançado para trazer possíveis relações com o que o usuário procura. Outro problema apontado, é que se a planilha possui mais de uma única aba, a pesquisa deve ser feita manualmente em cada uma das abas. Quanto ao

quadro físico, como a colocação das *issues* são adicionadas no quadro em forma de *post-its*, é necessário procurar um a um, caso se deseje encontrar o que precisa. Além disso, não há nenhum tipo de informação adicionada nos papéis sobre quais fases do desenvolvimento ele passou (se por codificação, teste ou se a tarefa já foi fechada). A única informação colocada no topo do *post-it* são as datas de entrada e saída do *board*.

5.6.3 Integração com versionamento de código

Figura 4 – Integração com versionamento de código



Fonte: o autor, 2019.

Não coincidentemente, a única ferramenta com nota máxima neste tópico é a do Github. Isso se dá, porque o versionamento de código dos projetos do laboratório é feito com ele. E a parte de manejo de *issues* desse sistema é amplamente voltada para trabalhar em conjunto com o código hospedado na plataforma. Inclusive esse foi um dos pontos positivos mais elencados por todas as equipes: a centralização em uma única ferramenta de código, tarefas e o poder que os usuários têm de referenciar e integrar código com partes processuais e atividades de desenvolvimento.

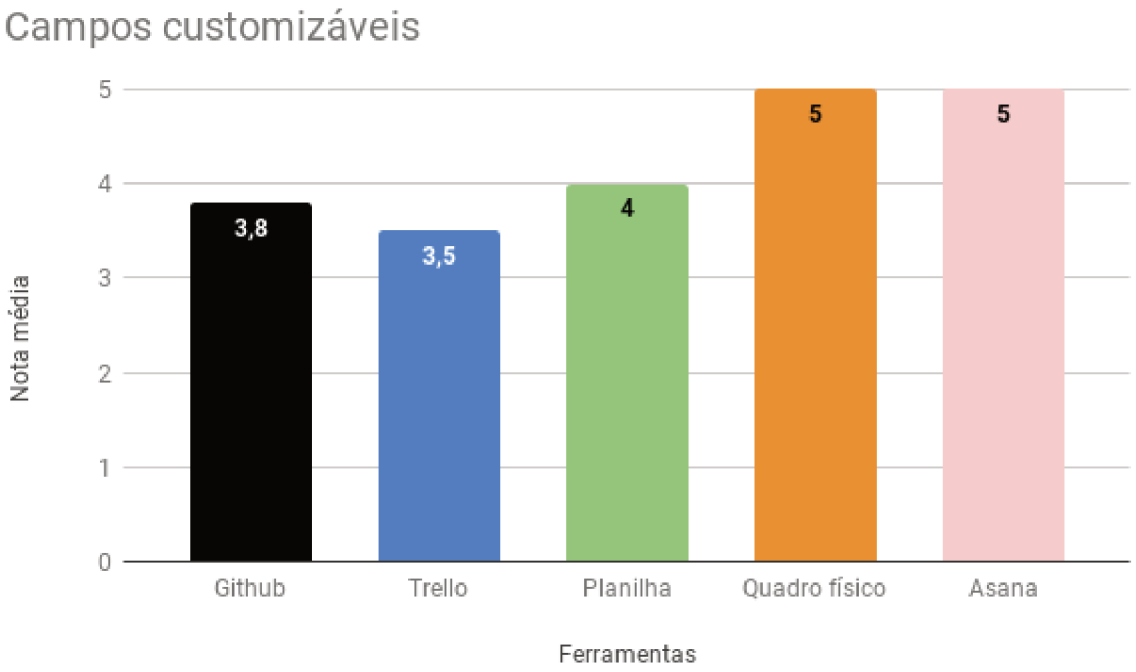
O Trello recebeu média 3, pois possui os chamados *power ups*. Os *power ups*, permitem que usuários tragam funcionalidades adicionais aos seus quadros para integrar aplicações com o Trello. Um *power up* possível de ser adicionado é o de integração com o Github, portanto, versionamento de código. Com ele consegue-se anexar *branches*, *commits*, *issues* e *pull requests*.

A planilha recebeu nota 2, porque mesmo que não haja nenhuma integração com versionamento de código, uma das suas principais qualidades é sua ampla customização. Podendo, então, a equipe criar uma coluna adicional e incluir lá um *link* para a referenciada página no Github do código.

Quadro físico e Asana, como apontam as equipes, não possuem esse tipo de integração, portanto receberam nota mínima.

5.6.4 Campos customizáveis

Figura 5 – Campos customizáveis



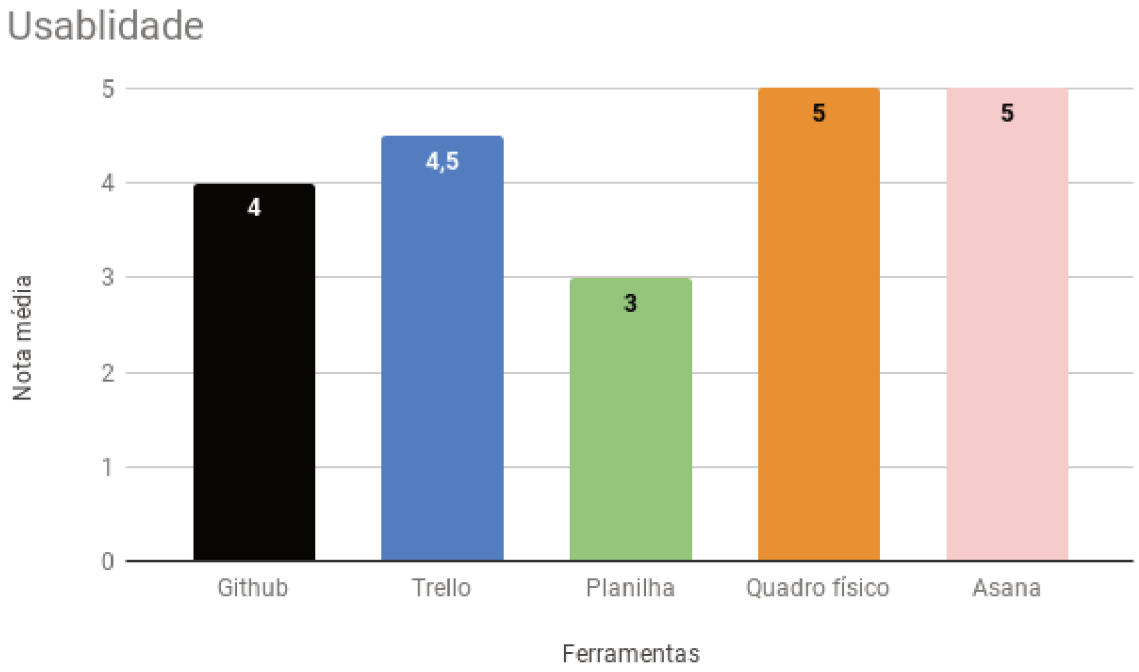
Fonte: o autor, 2019.

Como apontado pelos entrevistados, neste quesito, a ferramenta do Github é um pouco mais limitada quando se comparada a outros sistemas. Tudo se resume ao uso das *labels*. Elas ajudam a organizar e priorizar o trabalho e podem ser aplicadas tanto a *issues* quanto *pull requests*. Nelas é possível adicionar uma cor e uma descrição (normalmente com uma informação útil), por exemplo um rótulo na cor vermelha escrito "prioridade alta". Outro problema apontado para esse tópico não atingir nota máxima, é que caso haja a necessidade de customizar algo na tela atual (criando uma tarefa, por exemplo), é necessário sair dessa página, ir até a customização, fazê-la e depois voltar ao fluxo de trabalho normal. As respostas para o Trello também não diferiram muito, ao passo que as customizações são feitas nos nomes dos quadros e nas criações de *labels* similares as do Github.

Os usuários de Planilha, quadro físico e Asana apontaram que conseguem customizar tudo o necessário para o seu fluxo de trabalho, especialmente quando se trata do quadro. Uma vez que é possível mexer em tamanho de fontes, *post its*, linhas, nomes das colunas e cores.

5.6.5 Usabilidade

Figura 6 – Usabilidade



Fonte: o autor, 2019.

Aqui, as notas médias, novamente, foram notadamente altas, com exceção da planilha. O principal motivo apontado pelos usuários, foi que nem sempre os objetivos eram alcançados com eficácia e eficiência, por se tratar de uma ferramenta de propósito geral. Um exemplo relatado foi o de inserção de filtros em colunas, para ordenar e pesquisar por tarefas lá reportadas. Era necessário alguns cliques e, nem sempre, os botões indicavam que seria possível fazer um filtro.

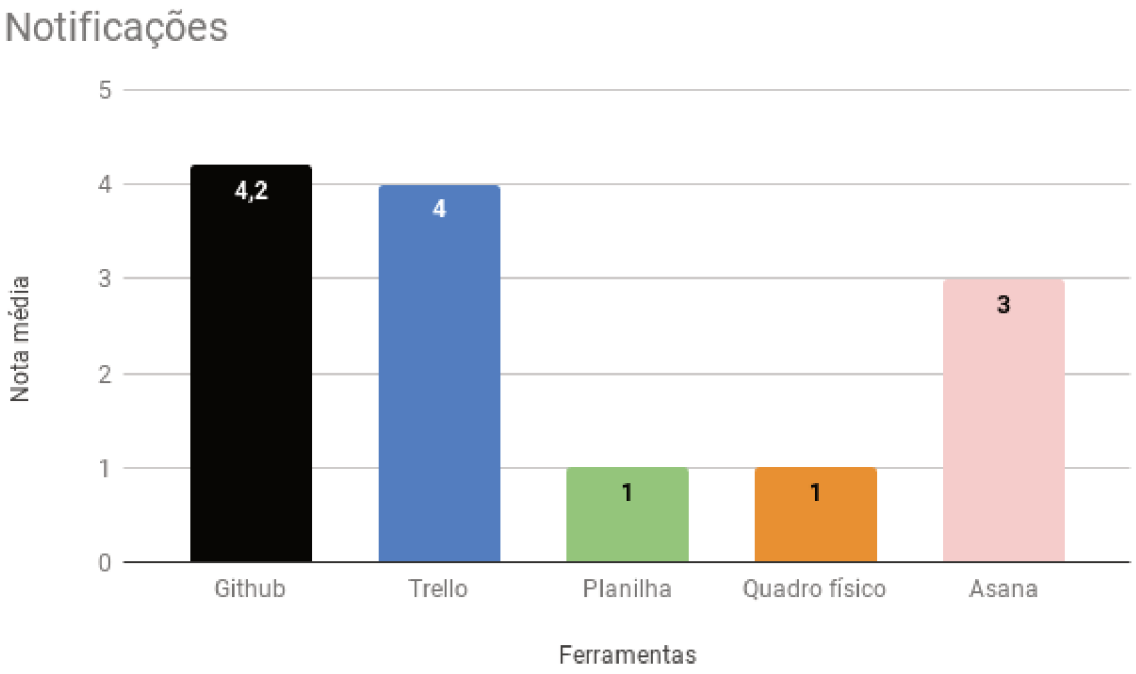
Quanto a Github e Trello foi levantado a questão sobre o *Markdown* - linguagem de marcação simples - que é presente no primeiro, mas que causa certa confusão e dificuldade na hora de criar uma lista, anotações ou *check lists* por exemplo. No Trello, ao criar um *card*, é possível clicar em componentes que são adicionados automaticamente na tela, através de atalhos, o que torna o processo muito mais rápido. Para obter o mesmo resultado na ferramenta do Github, é necessário primeiro criar uma *issue* e, depois disso, no corpo

da tarefa (descrição) fazer o uso da linguagem *Markdown* para estilizar o conteúdo. Pela falta destas pequenas facilidades, o Github teve uma nota pouco menor que a do Trello.

Já os usuários do quadro físico e do Asana deram nota máxima para suas ferramentas, apontando que não possuem problemas de usabilidade com elas.

5.6.6 Notificações

Figura 7 – Notificações



Fonte: o autor, 2019.

Notificações, como já descrito no capítulo anterior, são importantes para que pessoas do mesmo time saibam o que seus colegas estão fazendo, se prioridades são modificadas, novos problemas encontrados ou pessoas são designadas para determinadas correções. Os usuários de quadro físico, embora tenham dado nota mínima para este quesito, porque, de fato, o quadro não é capaz de emitir nenhum *e-mail* ou até mesmo um *pop-up* de aviso, alegaram que isso não necessariamente é um problema na equipe deles. O quadro, só é mexido (adicionado uma funcionalidade, retirado uma *issue* ou trocado *status* de desenvolvimento), durante o *daily meeting*, quando todos os integrantes da equipe estão participando. Eles fazem um bom uso de um dos princípios ágeis, onde pessoas valem mais do que ferramentas e exploram da boa comunicação intra-equipe para funcionar com êxito.

Do mesmo modo, os usuários de Planilha dizem que não há nenhuma forma de notificação presente nesta ferramenta. Isso foi um dos motivos pelo qual a equipe

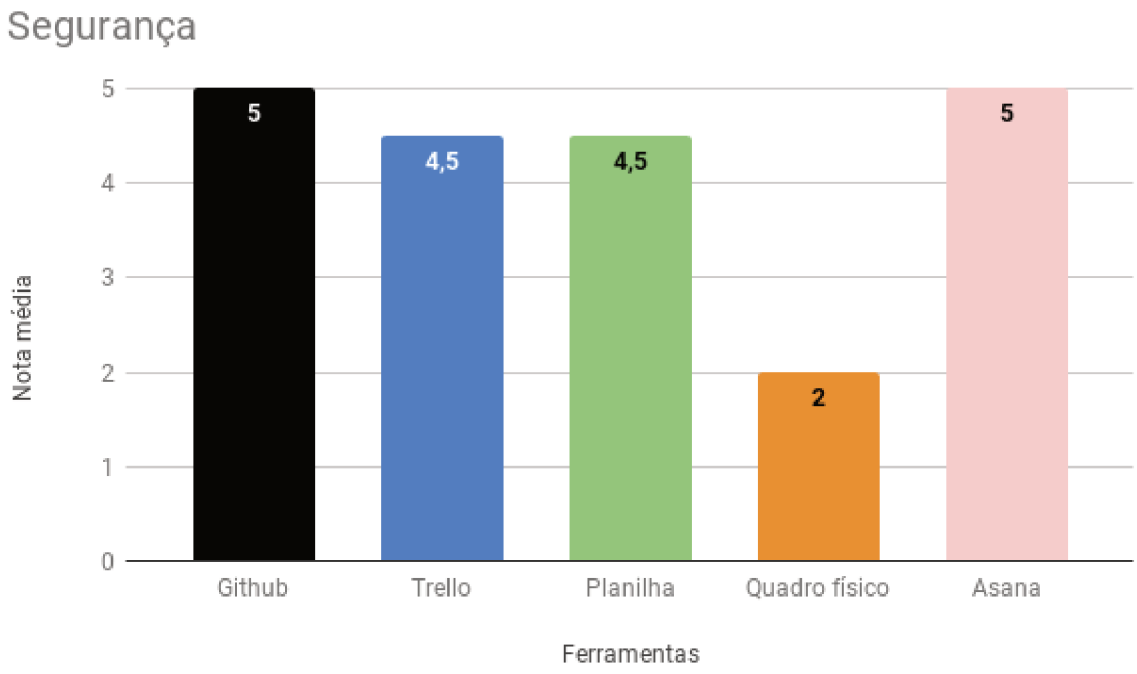
decidiu migrar para o Github. Eles também precisavam fazer um bom e constante uso de comunicação para discutir sobre as informações presentes na planilha.

Os usuários de Asana, Trello e Github apontam que suas ferramentas fazem bom uso de notificações, principalmente na forma de *e-mails* sobre alterações, mas acreditam que ainda faltam algumas informações diretamente na tela sobre mudanças. Também foi discutido que se uma determinada tarefa, por exemplo, com prioridade definida como alta ficar por muito tempo parada, sem alterações ou atualizações no *status* de desenvolvimento, seria interessante surgir um aviso por parte da ferramenta sobre esse problema. Nenhuma das ferramentas estudadas possuem tal funcionalidade no momento.

Outra situação levantada pelas equipes, relacionada a campos customizáveis, é o fato de que nas ferramentas que possuem sistema de notificações, é possível configurar e determinar quais tipos de notificações são disparadas por *e-mail*. Por exemplo, receber *e-mails* apenas quando uma tarefa for atribuída para si ou for fechada. O que é considerado excelente, pois receber muitas notificações ou *e-mails* pode prejudicar o desenvolvimento (com perda de foco) e poluição visual.

5.6.7 Segurança

Figura 8 – Segurança



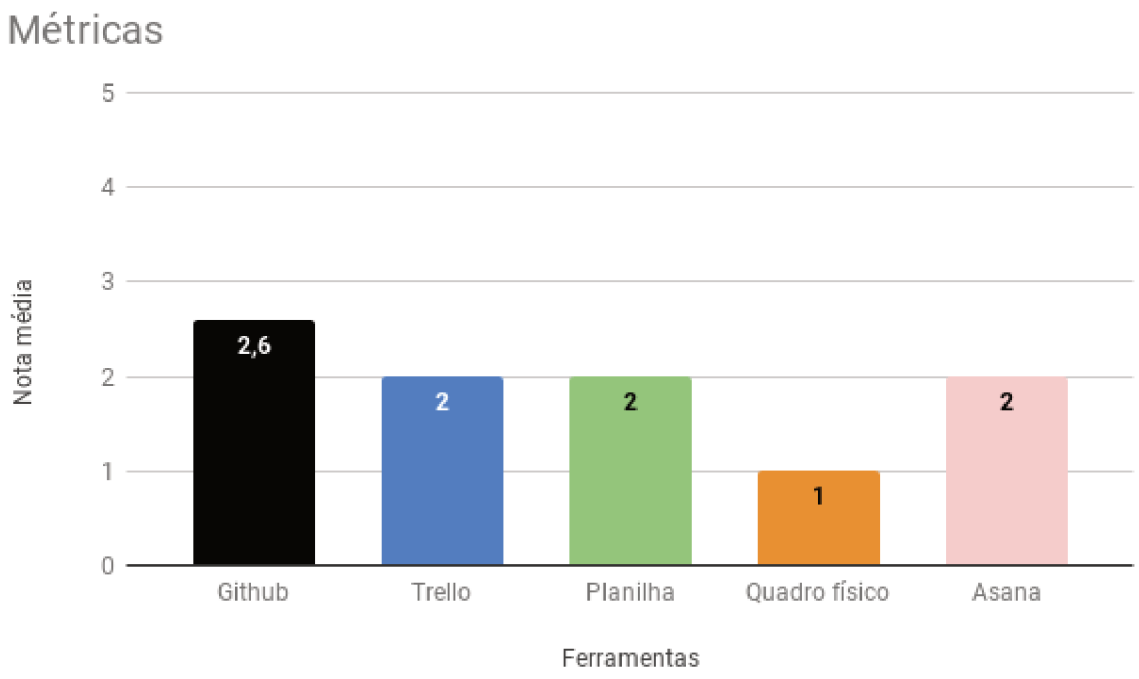
Fonte: o autor, 2019.

No tópico sobre segurança as notas foram muito altas com exceção do quadro físico. Todos os usuários das outras ferramentas apontam que se sentem muito seguros em usá-las. Por se tratarem de sistemas *web*, todas possuem redundância e são fortes contra perda de informações. Ainda sobre elas, todas possuem sistemas de *login*, onde é necessário estar autenticado para acessar determinados recursos; e, dentro de suas configurações é possível dizer quais pessoas ou grupo de pessoas podem acessar, alterar e excluir certas atividades.

Já no quadro físico, brinca o entrevistado: "a única segurança que temos é esta câmera de monitoramento que consegue visualizar o quadro". Qualquer pessoa que tenha acesso à sala e, por consequência ao quadro, pode modificá-lo, rasgar os *post its*, riscar ou escrever nele, visualizar todas as informações e andamentos das tarefas. Além disso, num caso de sinistro maior, todas as informações seriam perdidas. Razões estas que tornam sua nota tão baixa.

5.6.8 Métricas

Figura 9 – Métricas



Fonte: o autor, 2019.

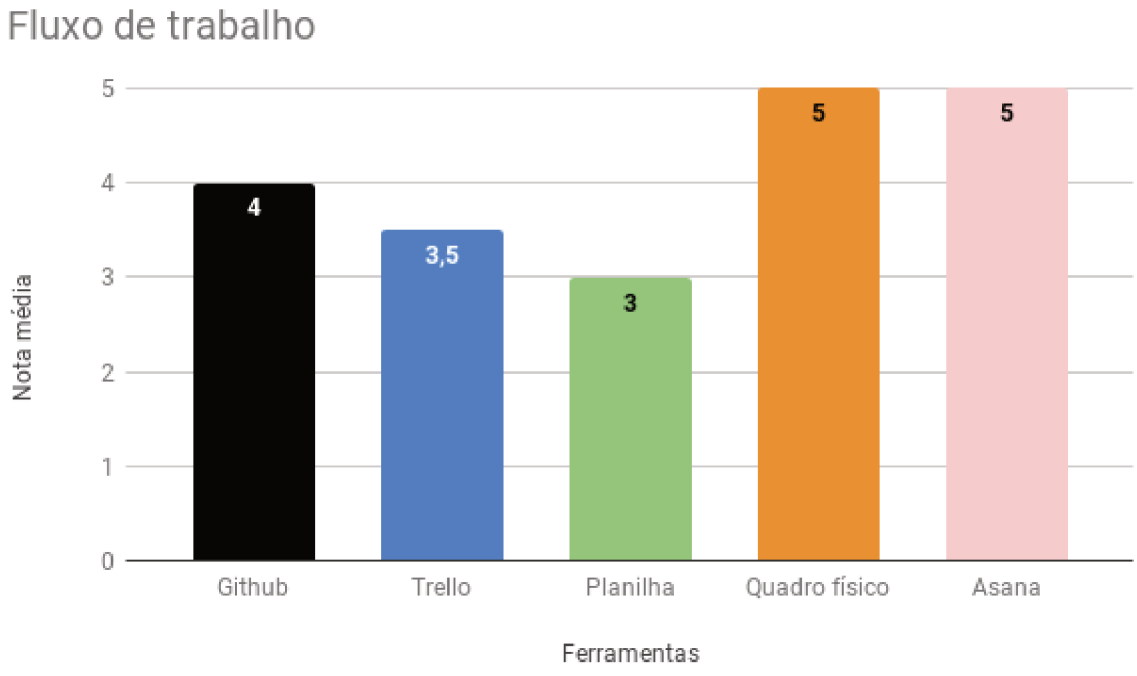
Todas as ferramentas alvo de estudo deste trabalho, sem nenhuma exclusão, receberam notas muito baixas neste tópico. Isso ocorreu, porque os usuários destas ferramentas alegaram que elas não possuem geração de relatórios e ou alguma funcionalidade que permita contagens (sejam elas de problemas encontrados, resolvidos, ainda abertos, para uma determinada pessoa...) de forma automatizada afim de relatar informações úteis para

o processo de desenvolvimento de *software*. Caso os usuários queiram ou necessitem, e assim muitas equipes o fazem, eles devem fazer manualmente uma pesquisa com base nos filtros pré-disponíveis e fazer a contagem para retirarem seus próprios diagnósticos e ter estimativas de acordo com o que necessitam.

As equipes que fazem uso de Github deram notas um pouco mais altas, pelo fato da ferramenta também ter implementado o conceito de *milestones*. Elas são "marcos"do projeto e dão noção de eventos como data, título e descrição - geralmente utilizados para definir prazos de entrega. Assim, é possível atribuir tarefas para um determinado marco (normalmente uma *sprint*) e conforme essas *issues* são finalizadas, uma barra no topo do projeto cresce, indicando que a *milestone* está próxima de ser completada. Mas, ainda assim, a ferramenta carece de relatórios que apontem dados consistentes e mais úteis, tais como *bugs* mais acessados e recorrentes, quantidade n de tarefas atrasadas ou criadas em um determinado marco.

5.6.9 Fluxo de trabalho

Figura 10 – Fluxo de trabalho



Fonte: o autor, 2019.

De acordo com este tópico, uma ferramenta de *BTS* deve estar presente no ambiente de desenvolvimento de um projeto para otimizar o trabalho e os processos na produção de *software*. Ele deve auxiliar a resolver os problemas de forma clara e concisa; e de forma alguma atrapalhar ou confundir o fluxo de trabalho de uma equipe.

Os usuários de quadro físico e Asana indicaram que essas ferramentas se adequaram perfeitamente no fluxo deles, uma vez que a comunicação é o carro-chefe da equipe, e o efeito visual que eles proporcionam, principalmente na hora do *daily meeting* mantém toda a equipe unida, bem informada, organizada e, sobretudo, alinhada, de forma que todos saibam o que os outros estão fazendo.

Os usuários de planilha disseram que por ela ser um pouco mais burocrática, às vezes o fluxo de trabalho é um pouco atrapalhado, principalmente pela falta de comunicação e notificação sobre novas linhas adicionadas às tabelas (tais como novas *issues* ou *status* de tarefas atualizados).

Para as equipes que usam Github e Trello as notas estão aceitáveis. Especialmente para o Github, a nota não foi máxima por questões levantadas quanto a assuntos sobre interequipes, ou seja, quando mais de um time precisa fazer uso de um mesmo repositório, acaba ocorrendo uma quebra de fluxo e a ferramenta não provê suporte para isso. Por exemplo, se uma pessoa externa a equipe (ainda que dentro do mesmo projeto e organização) necessita de informações sobre uma tarefa específica, ela necessariamente precisa ser adicionada dentro do repositório de um time, dando noção, assim, de que ela faz parte daquele time, além dela ter acesso a todo o fluxo de trabalho e poder fazer alterações.

5.7 Comparando funcionalidades das ferramentas

Essa seção contém algumas funcionalidades presentes nas ferramentas. A partir disso, é montado uma tabela para cada funcionalidade e atribuído um identificador (F#) na tabela 5.

Tabela 5 – Exemplos de funcionalidades existentes

#	Descrição
F1	O uso da ferramenta é gratuito
F2	Possui uma versão paga com novas funcionalidades inclusas
F3	Possui integração direta com o código da aplicação
F4	Há a possibilidade de criação de *boards* e *cards*
F5	Atribuição de pessoas a determinados recursos
F6	Possibilidade de adicionar comentários
F7	Possibilidade de incluir *checklists*
F8	Ferramenta de uso *web* (necessita de acesso a *internet*)
F9	Presença da linguagem Markdown para estilizar
F10	Necessidade de autenticação para visualizar/editar artefatos
F11	Possui automatização
F12	Possibilidade de *upload* de arquivos como *pdfs* e imagens
F13	Presença de filtros customizáveis
F14	Presença de campo de pesquisa
F15	Criação e customização de *labels*

A lista geral de funcionalidades citadas nessa tabela foram os exemplos de funcionalidades existentes nas ferramentas foco de estudo deste trabalho e apontadas pelos entrevistados durante as conversas. São vistos tópicos tais como gratuidade, presença de campos em específico, possibilidades de customizações, *upload* de arquivos e integração com outras ferramentas. Já na tabela 6 é marcado com um x se tal funcionalidade está presente ou não na respectiva ferramenta.

Tabela 6 – Comparação entre ferramentas com suas funcionalidades

Ferramenta	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15
Github	x	x	x	x	x	x	x	x	x	x	x	x		x	x
Asana	x	x		x	x	x	x	x		x		x		x	x
Trello	x	x		x	x	x	x	x	x	x		x		x	x
Quadro	x			x		x	x								
Planilha	x					x		x		x		x	x		

Realizando a análise da tabela 6, pode-se perceber que a ferramenta Github é a que mais possui funcionalidades em quantidade (quatorze ocorrências de x); inclusive, uma delas muito comentada pelas equipes é a *F11* (que diz sobre possibilidade de automatização). Portanto, é a mais completa em termos de funções segundo o que os entrevistados elencaram ser importantes nas conversas. Aqui também vale ressaltar que existem outras funcionalidades não presentes na tabela 5 e que dentro da cultura do projeto e-SUS AB do Laboratório Bridge não foram elencadas como importantes, por isso não apareceram. Mas, possivelmente, poderiam aparecer em outro contexto, em uma outra empresa ou projeto.

Em contrapartida, o quadro físico é a que menos possui (apenas quatro ocorrências) e a única que não é acessada via *web*. O alerta aqui, é que essa tabela comparativa tem apenas o intuito de demonstrar as funcionalidades que cada ferramenta possui ou não. Pode acontecer de que para determinadas equipes e projetos alguma dessas funcionalidades seja irrelevante, portanto essa ferramenta é uma candidata ao uso da equipe. Este é o caso da equipe que faz uso do quadro físico. Mesmo ela possuindo poucas funcionalidades em concorrência com as demais, ainda assim se enquadra bem no fluxo da equipe.

Uma outra análise que podemos ter é que as ferramentas Trello e Asana se comportaram de forma idêntica em todas as funcionalidades, exceto na *F9* (presença de linguagem Markdown para estilizar). Essas ferramentas realmente tem um propósito muito parecido, inclusive a forma de uso e a interface visual também são similares.

E uma funcionalidade de destaque da ferramenta Planilhas Google, é a capacidade de criar filtros próprios e customizáveis. Uma vez que há uma coluna na planilha, é possível selecioná-la e montar uma filtragem com parâmetros escolhidos pelo usuário para mostrar os dados de forma ordenada.

6 Conclusões

Com o término das entrevistas e compilação dos resultados obtidos para apresentação neste trabalho, foi possível perceber que diferentes equipes possuem diferentes realidades, além de suas diferentes demandas do dia a dia. Mesmo algumas delas utilizando as mesmas ferramentas, apontaram limitações e pontos de melhoria distintos, com base nos seus anseios e necessidades diárias.

Foi constatado, durante a execução do trabalho, que a maior parte das equipes buscou migrar para o Github, fazendo uso dos seus módulos de *issues* e *projects*. Isso ocorreu, principalmente, pela sua ótima integração com o código, conforto em centralizar tudo numa única ferramenta e possibilidade de automatizações. Além disso, na parte processual, a metodologia *Kanban* foi tomando espaço, se tornando um catalizador do *Scrum*, que já era utilizado por anos no laboratório.

Mesmo com essa migração para o Github, foi possível perceber que ele ainda não é excelente, possuindo limitações e melhorias citadas pelas equipes. Das melhorias apontadas, a mais importante e que traria muitos benefícios ao fluxo de trabalho dentro do projeto é a noção de intertimes, ou seja, vários times trabalhando em conjunto dentro de um mesmo repositório. Como foi visto, atualmente, os usuários são tratados como independentes, e colocar mais do que um time de usuários dentro de um mesmo repositório pode ficar insustentável e desorganizado.

Um ponto que chamou atenção, não só do Github, mas como de todas as ferramentas analisadas, foi a nota recebida quando perguntado sobre se a ferramenta fornece métricas para a equipe; e, se fornece, se a faz de forma adequada. Todas as ferramentas receberam notas baixíssimas nesse quesito, pois nenhuma realmente gera um relatório apontando ocorrências, sejam elas quantidade de tarefas criadas, fechadas, reabertas, versões com muitas tarefas de alta prioridade, entre outras. Inclusive, sobre este caso, uma colega do Laboratório está fazendo de trabalho de conclusão de curso, um *plugin* para o Github que permite disponibilizar tais métricas. Hoje, o que as equipes fazem para ter insumos e gerenciar suas produções, é fazer uso dos filtros das ferramentas e com base neles, contar as ocorrências e colocá-las numa planilha.

Outro fator percebido foi que as equipes respeitam muito e até comentaram um dos princípios ou conceitos chave mais importantes do Manifesto ágil, que trata de indivíduos e interações serem mais importantes do que processos e ferramentas. Não adianta ter uma ótima ferramenta, considerada a melhor do mercado, se a equipe não tem boa comunicação e baixa produtividade. Esse fato foi muito percebido quando tratamos da questão de adaptabilidade das ferramentas, e as ferramentas Planilhas Google e quadro físico receberam

notas máximas. Dentro dessas equipes a comunicação é tida como componente essencial e a aproximação profissional intratime é muito boa, fazendo com que até ferramentas de propósito geral sejam adaptáveis a realidade deles.

Além das migrações dos *BTS* utilizados pelas equipes e implantação do *Kanban* dentro delas, algo que ocorria quando se iniciou esse trabalho de conclusão de curso e, que de certa forma incomodava algumas pessoas dentro do projeto, eram as duplicações de tarefas. Optou-se então, colocar uma pergunta no questionário sobre o tema, para saber a opinião das equipes sobre o assunto e verificar se ele ainda ocorre.

Para surpresa, as equipes apontaram que não veem mais ocorrências de tarefas duplicadas com frequência, tampouco se sentem incomodados quando isso acontece. Contudo, levantaram uma questão muito importante que deve estar inclusa em um bom sistema de reporte de *bugs*: a capacidade de procurar e filtrar com qualidade as tarefas da base de dados. Inclusive, um dos trabalhos correlatos, que trata sobre esse tema, elenca dentre as 5 maiores razões para que haja duplicação de tarefas o filtro de pesquisa da ferramenta ser fraco ou ineficiente. Foi visto ainda, que as equipes se sentem importunadas quando precisam utilizar o filtro da ferramenta Redmine antes de criar uma nova tarefa. Isso se sucedeu, porque, diversas vezes, pesquisar se um problema já foi reportado é mais custoso do que criar a tarefa em si.

Por fim, este trabalho teve um bom propósito de entender como as equipes ágeis do projeto e-SUS AB atuam no seu dia a dia interagindo com as ferramentas de *bug* e *tracking report*. Problemas foram elencados e melhorias foram discutidas e sugeridas. A centralização, automatização e facilidade no uso das ferramentas são o ponto número um para seu uso, seguidos de bons filtros e funcionalidades que ajudam nas caracterizações das tarefas reportadas pelos usuários. A partir da análise dos resultados, não se pode dizer que existe uma melhor ferramenta ou não. Diferentes equipes, possuem diferentes realidades, e têm diferentes demandas. Ainda que fazendo uso de uma mesma ferramenta, elas apontaram limitações e melhorias distintas. Portanto, o que há são várias ferramentas e, antes de fazer uso delas, deve-se analisar as funcionalidades presentes e verificar se elas se encaixam nos requisitos necessários para o projeto em questão.

7 Trabalhos futuros

Para possíveis trabalhos futuros seria interessante realizar os seguintes tópicos:

- Realizar a mesma pesquisa, porém em outro projeto corrente dentro do Laboratório Bridge. Ainda que dentro da mesma cultura da organização, diferentes projetos podem demandar diferentes situações e preferências no uso de ferramentas;
- Ainda sobre o espaço de experiência da pesquisa, pode-se fazê-la em outra empresa ou organização. Diferentemente do item de cima, além de ser outro projeto, será outra cultura. Seria relevante fazê-la para acompanhar mais de uma realidade no uso dessas ferramentas, e até adicionar novos sistemas além dos usados no Laboratório Bridge;
- Propor as melhorias elencadas para as equipes diretamente para as ferramentas utilizadas, juntamente das dificuldades e limitações que elas possuem. Assim, talvez haja a possibilidade das empresas implementarem essas funcionalidades;
- Criar uma ferramenta nova e implementá-la com base em tudo apontado pelos entrevistados. Essa ferramenta terá todas as funcionalidades embasadas nas respostas dos usuários, suprindo suas necessidades atuais;
- Aproveitando o benefício da ferramenta Redmine ser de código livre, pode ser interessante, porém custoso, mexer no seu fonte e adequar, principalmente, sua parte de filtros e pré-definições, que foram tratadas no escopo desse trabalho como ruins ou ineficientes pelos participantes.

Referências

ASANA. <https://asana.com/pt>. Acesso em: 02/07/2019. Citado na página 26.

BARTIÉ, A. *Garantia da qualidade de software*. [S.l.]: Gulf Professional Publishing, 2002. Citado na página 32.

BETTENBURG, N. et al. What makes a good bug report? In: ACM. *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. [S.l.], 2008. p. 308–318. Citado na página 44.

BETTENBURG, N. et al. Duplicate bug reports considered harmful... really? In: IEEE. *Software maintenance, 2008. ICSM 2008. IEEE international conference on*. [S.l.], 2008. p. 337–345. Citado na página 49.

BLAIR, S. A guide to evaluating a bug tracking system. *Retrieved from Scribd. com: http://www. scribd. com/doc/7046090/A-Guide-to-Evaluating-a-Bug-Tracking-System*, 2004. Citado 4 vezes nas páginas 26, 45, 53 e 61.

BOEG, J. Kanban em 10 passos. *Tradução de Leonardo Campos, Marcelo Costa, Lúcio Camilo, Rafael Buzon, Paulo Rebelo, Eric Fer, Ivo La Puma, Leonardo Galvão, Thiago Vespa, Manoel Pimentel e Daniel Wildt. C4Media*, 2010. Citado na página 42.

DELAMARO, M.; JINO, M.; MALDONADO, J. *Introdução ao teste de software*. [S.l.]: Elsevier Brasil, 2017. Citado 2 vezes nas páginas 35 e 36.

FILHO, R. M. T.; RIOS, E. *Projeto & engenharia de software: teste de software*. [S.l.]: Rio de Janeiro: Alta Books, 2003. Citado na página 33.

GITHUB issues. <https://github.com>. Acesso em: 02/07/2019. Citado na página 25.

KITCHENHAM, B. Procedures for performing systematic reviews. *Keele, UK, Keele University*, v. 33, n. 2004, p. 1–26, 2004. Citado na página 29.

KOSCIANSKI, A.; SOARES, M. dos S. *Qualidade de Software-2ª Edição: Aprenda as metodologias e técnicas mais modernas para o desenvolvimento de software*. [S.l.]: Novatec Editora, 2007. Citado 6 vezes nas páginas 31, 33, 34, 38, 39 e 40.

PLANILHAS Google. <https://www.google.com/intl/pt-BR/sheets/about/>. Acesso em: 02/07/2019. Citado na página 25.

PRODANOV CLEBER CRISTIANO; FREITAS, E. C. d. *Metodologia do Trabalho Científico: Métodos e Técnicas da Pesquisa e do Trabalho Acadêmico - 2ª Edição*. [S.l.]: Editora Freevale, 2013. Citado na página 30.

REDMINE. <https://www.redmine.org/>. Acesso em: 02/07/2019. Citado na página 25.

RIOS, E.; MOREIRA, T. *Teste de software*. [S.l.]: Alta Books Editora, 2006. Citado na página 34.

TRELLO. <https://trello.com>. Acesso em: 02/07/2019. Citado na página 26.

UNITO. <https://unito.io/>. Acesso em: 02/07/2019. Citado na página 59.

WAZLAWICK, R. *Engenharia de software: conceitos e práticas*. [S.l.]: Elsevier Brasil, 2013. v. 1. Citado 9 vezes nas páginas 23, 24, 31, 33, 35, 38, 39, 40 e 41.

ZIMMERMANN, T. et al. Improving bug tracking systems. In: IEEE. *Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*. [S.l.], 2009. p. 247–250. Citado na página 47.

Apêndices

APÊNDICE A – Entrevistas

Como as entrevistas se deram como uma conversa direcionada pelas perguntas, algumas das perguntas estão aqui descritas em mais de um parágrafo ou o mesmo parágrafo respondendo-as.

A.1 Equipe A

1) Github issues. Usamos só essa.

2) Antes usavamos o Trello. Tinhamos mais colunas do que tem hoje no Projects, mas ele ficava desatualizado mais fácil. Por exemplo, estávamos trabalhando em uma tarefa e aconteceram algumas vezes dela ser integrada e acabar ficando o card esquecido lá na ferramenta. Hoje não, com a automatização não precisa mais disso. Ele (o Github) manda sozinho de doing para done. Só que o Trello era mais fácil de usar e tinha mais funcionalidades, então acabamos abrindo mão disso. A gente tinha só um projeto, então tinham vários cards espalhados, com o Github a gente cria um projeto novo, por exemplo o projeto do Neuro. Todos os cards ficam lá e só vão pro board da equipe quando vão ser efetivamente desenvolvidos. Então isso ajuda muito na organização e não deixar as coisas espalhados.

3) O Github é mais adequado, porque conseguem fazer automatização. E porque fica tudo concentrado em uma ferramenta só. Não precisa de mais de uma ferramenta.

4) Automatização em primeiro lugar. Ferramenta unificada e centralizada para tudo.

5) O fato de trabalhar com repositórios diferentes e não ter essa integração intertimes complica um pouco a parte da automatização, se desse para fazer isso da mesma forma que é feito intra repositório seria muito bom. A usabilidade tem muito a melhorar, principalmente a fluidez. Precisamos apertar f5 algumas vezes pra funcionar, por exemplo.

6) Interação mais clara entre projetos pode ser melhorada, pode ser deixado mais claro funcionalidades que ele tem, mas que ficam escondidas. Automatização poderia ser mais fáceis, ter mais itens. A maioria é "se o PR for fechado, mexe", mas as vezes queremos que algo aconteça no meio do processo e não só no fim. Fazer a filtragem por cards também é complicado, porque tem uma linguagem de pesquisa própria, com expressão própria e fica meio difícil de entender. Até para nós que somos da área é difícil de usar. Seria bacana o Github gerar indicadores do Kanban, melhorar mais automatizações. Ofertar mais métricas e relatórios aos usuários.

7) Todos usam.

8) Ele é bom para coisas que envolvam mais de uma equipe. por exemplo, desenvolvemos um módulo e vamos passar para a equipe de testes (qualidade). Questão inter equipes é legal o Redmine. Se tivesse uma integração entre o Github e o Redmine seria bom. Ele é de certa forma o mal necessário. Cuidamos muito com burocracia e certas coisas tem que ser de formas bem específicas e ficarem gravadas. O fluxo de uma equipe dentro dela, é muito diferente de fora dela. Até mesmo dentro de uma equipe pode haver fluxos diferentes, com diferentes demandas. Por exemplo, hoje a equipe de qualidade está testando algo que já foi desenvolvido a duas semanas e nós estamos com uma demanda nova. Se a qualidade encontrar alguma coisa, ela reporta formalmente no Redmine e nós vamos receber esse chamado, tratar ele e devolver novamente para teste. Então isso é o legal dele, é uma ferramenta de uso coletivo para muitas pessoas. Seria complicado ter a equipe de qualidade inserida dentro do nosso espaço de trabalho, usando as nossas ferramentas do dia a dia, porque os fluxos dos trabalhos não são os mesmo; nós já estamos trabalhando coisas lá da frente. Os pontos negativo pro Redmine são os filtros, são muito ruins. Esses dias eu estava tentando salvar uma pré configuração de filtro e desisti por causa da demora para conseguir.

9) Não acontecem muito hoje em dia, mas também acho que não deveriam acontecer. Pode acontecer de eu pegar uma tarefa para desenvolver, por exemplo, ela passa pelo teste, passa por tudo, mas ainda tem uma outra tarefa idêntica aberta. Aí nós vamos fazer um levantamento do que foi feito ou não e ficamos naquela: mas nós desenvolvemos isso aqui sim, porque tem essa tarefa aberta? Aí vamos ver e tem 2 tarefas iguais. O que complica é o filtro do Redmine. Ele é muito ruim. E temos muitas tarefas. Já coloca ai na conta: tempo para achar o bug, tempo para reportar o bug da melhor forma possível para outras pessoas entenderem, e ainda ter que gastar tempo procurando na base de dados se aquele bug já foi encontrado antes e reportado, isso é muito ruim. As vezes o tempo de fazer isso tudo é menor do que o tempo de ficar procurando por uma tarefa, pra ver se ela já existe.

A.2 Equipe B

1) Usamos atualmente github issues, mas antigamente usavamos planilhas google.

2) Antes nós usávamos a planilha do Google. Dava muito certo principalmente porque a nossa comunicação dentro da equipe era muito boa. Qualquer coisinha já virávamos as cadeiras e discutíamos. Sempre eu sabia o que os outros membros da minha equipe estavam fazendo e eles sabiam também o que eu estava fazendo. A planilha dava pra organizar da forma como queríamos, porque ela é de propósito geral, aí lá nós tínhamos abas separando as coisas e colocávamos os números das tarefas numa coluna, descrições, problemas e status em outras. As vezes colocavamos o link pro Github ou pro Redmine.

Tínhamos um pouco de problema para fazer pesquisa, até porque não existe filtros, tínhamos que usar o ctrl f do navegador e isso exigia que escrevêssemos exatamente o que queríamos na pesquisa, não tinha uma forma de busca melhor. Nós mudamos justamente por causa da integração com o código, centralização em uma ferramenta. basicamente conseguimos fazer tudo hoje no Github que fazíamos na planilha, só que de forma muito mais melhorada e otimizada. Agora conseguimos filtrar e organizar melhor as issues, outro ponto foi a automatização e conseguir referenciar tudo dentro de tudo.

3) Manter tudo numa ferramenta só e a facilidade, pois ela é fácil de aprender e de utilizar. O principal mesmo é integração e unificação. Uma ferramenta só para controlar tudo, conseguir referenciar commits. Ele também tem o esquema dos projects. Nós fazemos os cards como se fosse o Kanban.

4) Facilidade de uso, manter tudo em um único lugar.

5, 6) Não vimos nenhuma limitação que impeça de alguma coisa, mas o maior problema é o histórico. Organização de evolução de uma issue, principalmente pq quando troca label, ele coloca comentário e começa a ficar muito gigante para baixo, isso torna difícil precisar achar alguma coisa de tal issue, porque ela fica muito grande. Independente da ferramenta, tem que ver primeiro a equipe. Por exemplo antes nós tínhamos a planilha e tudo funcionava bem porque a nossa comunicação era excelente. Os filtros não funcionavam muito bem, até porque não tinha né. Precisávamos dar ctrl f para achar algo na página e escrever exatamente o que estávamos procurando. O contrário também pode valer: pegar a melhor ferramenta do mundo, mas um desastre de equipe, que vai continuar não funcionando. Talvez fosse legal ter algo para filtrar só comentários nas issues, melhor visualização de histórico, uma forma melhor de customizar o que quer ver nos históricos das issues. Por exemplo ocultar todos os processos daquela issue e deixar só os comentários, ou mostrar um diff, ordenar como a timeline (mais recente - pq hoje temos que descer até o final) se a issue for grande tem que descer muito.

7) Todos usam.

8) Talvez aqui no laboratório haja uma repulsa, porque ela é sinônimo de burocracia. Mas é uma ferramenta super importante, porque ela nos provê muito contato com o cliente, uma vez que eles comentam tarefas, abrem novas demandas por lá e pedem que as coisas sejam reportadas e fiquem salvas lá para auditoria. Assim é mais organizado, ninguém faz nada "a torto e a direito", tem agentes externos que usam e por ser uma ferramenta centralizada e de uso coletivo total a gente tem mais cuidado. Uma ferramenta ótima que se flexibiliza da forma como quiser, pois ela é bem configurável e de código livre. Ela é boa para organização-cliente. Mas se fosse para as equipes ágeis usarem ia ser mais complicado, por ela ser mais formal. Ele é o mal necessário. Ele é burocrático por um lado, mas ele ajuda a ter uma linguagem única dentro do laboratório. Não importa como as equipes se ajeitam internamente, mas como a tarefa chega sempre será entendível por todos. Por

exemplo é dito os pré requisitos do teste, o que aconteceu, como aconteceu. Isso num laboratório muito grande, sem essa organização e padronização, não ia funcionar.

9) Duplicação de tarefas, pensando em organização, quanto menor o escopo, mais controlado, mais tu consegue tomar ações. Quanto mais tarefas têm, mais difícil é controlar. Nosso redmine hoje tem 12000 tarefas, isso é ruim. Talvez se tivessem filtros melhores, não aconteceriam tanto duplicações de tarefas.

A.3 Equipe C

1) Usamos o Github, mas já usamos outras antes, inclusive mais de uma ao mesmo tempo. Migramos porque buscamos centralizar tudo em um único local, por isso hoje usamos uma só.

2) Usavamos o Trello e Github. Usavamos a parte de power ups do Trello para vincular certas partes dele com o Github. Mas não conseguiamos fechar um card quando um PR fosse "mergeado". Então além da centralização buscavamos automatização, por isso migramos para o Github issues.

3, 4) Quando decidimos adotar o Github, a gente usava mais de uma ferramenta e aí viamos que sempre em algum lugar as informações estavam desatualizadas, tínhamos que usar de retrabalho e tinha desencontro de informações. Daí com a centralização nós nos demos melhor. Atualiza em um lugar só. E às vezes até em um terceiro local, que eram tarefas desenvolvidas na equipe e ainda tinha que mexer no Redmine.

5) Uma das coisas que a gente tem dificuldade é com a organização das colunas dentro do módulo do Project para ficar mais parecido com o Kanban, por exemplo, nem sempre cabe tudo que a gente precisa por ali dentro, dentro daquele layout. Tem que ficar fazendo scroll horizontal. Então nós nos preocupamos bastante para organizar o número de colunas que coubesse em uma página. Não é bem uma limitação, mas é mais de usabilidade que de certa forma incomoda a gente. Outra questão foi que nós precisamos criar um novo Project para colocar principalmente as tarefas discutidas direto com o cliente e criando outro project todas as vezes que você precisa ver a informação de um, tem que sair e ir até ele. Não tem um link fácil para visualização dessas coisas. Isso surte impacto para nós mais no inicio da sprint que é quando precisamos olhar essas informações para o desenvolvimento. De resto não tem impacto tão negativo.

5 e 6) Quando decidimos adotar o issues, nós desejamos muito a parte de automatização, por exemplo quando um PR for fechado fecha uma issue também. Porque a gente tinha um problema que era lembrar de atualizar as tarefas que tinham que ser fechadas. Internamente, melhorou bastante nosso fluxo pós atualização. Mas nós temos uma limitação da ferramenta de fechar issues através de pull requests e quando o PR

precisa fechar uma issue de outro repositório complica. Hoje o PR só fecha a issue que seja do mesmo repositório que ele está vinculado. Existe o comando para quando você abrir um PR para outro repositório fechar uma issue de dentro do seu repositório, mas ele tem uma limitação muito grande que é quando a pessoa que for revisar e fechar o PR no repositório externo, ela também tem que ser admin dentro do nosso repositório. Já participei de outras equipes que por causa dessa limitação, eles não adotaram essa ferramenta. Hoje ainda é uma limitação que ainda não foi corrigida, imagino que por motivos de segurança, mas não vejo que seja motivo crítico. Eu acabo tendo que dar permissões dentro do nosso repositório para que quando eles fechem PR lá no repositório master, feche a issue dentro da nossa equipe. Então essa parte toda de automatização do Github é muito ferida quando se trata de repositórios distintos.

6) Uma funcionalidade que possa ser adicionada é que assim, quando a gente vai descrever uma issue, nós precisamos usar o markdown, essa é uma das coisas que tanto no nosso time, quanto em outro time, pesa na hora de migrar do Trello pro Github. É uma das coisas que o Trello oferece, mas o Github não oferece. Que é quando você cria um card no Trello, ele já oferece alguns componentes que tu consegue ir colocando na tela, com coisas mais rápidas, por exemplo check list, nós usamos muito checklist (dentro de issues, cards, anotações). No trello era bem fácil, no Github só da pra fazer por meio de markdown. Nós precisamos criar uma issue e no corpo da issue, na descrição, precisamos usar o markdown para criar uma lista, um check list, estilizar nosso conteúdo. No Github não tem uma ferramenta que nos permita adicionar essas coisas rapidamente. Outra coisa que foi melhorada, mas ainda assim não é tão eficiente é a inserção de labels na propria issue dentro do board. No Trello consegue clicar no próprio card e já ir adicionando informações. No Github você clica, abre a modal e daí sim através da modal que você consegue fazer as edições. Geralmente os problemas que nós temos, seria implementar esses atalhos para inserção desses componentes e facilidades de inserção dos próprios componentes já oferecidos.

7) Todos da equipe fazem uso da ferramenta.

8) Antigamente era complicado, eu até dei ideias para reduzirmos o uso de ferramentas. Por exemplo, usavamos Trello, Github e Redmine. Quando partimos só para o Github ficaram só 2 ferramentas. Ainda assim é uma ferramenta a mais que precisamos utilizar. Mas o Redmine nós sabemos que ele tem funcionalidade diferente, por ele ser mais para integração externa, temos visão que ele é importante, até mesmo por causa da equipe de qualidade, então é necessário ter descrição de cenários e problemas encontrados de qualquer módulo dos problemas; ele prove histórico das tarefas em cima de cada módulo e além disso hoje, no nosso fluxo, se vem uma tarefa para nós no Redmine, a gente pega e replica essa tarefa dentro do nosso repositório. A gente cria uma issue vinculada aquela tarefa. Colocamos no corpo da issue o link na tarefa, mas precisamos fazer tudo na mão.

Não tem nenhum plugin que faça isso por nós hoje; mas seria mto interessante que existisse. Considero o Redmine como mal necessário, principalmente por causa do rastreamento dos problema encontrados, interface com o pessoal do suporte, equipe de qualidade e cliente. Ele é bom por causa da centralização e parte burocrática. Poderíamos ter um só repositório e fazer tudo através das issues, mas de certa forma precisaria ter integração inter times, que hoje não existe.

9) Não possuimos problemas com duplicação de tarefas, dificilmente acontece dentro da equipe. Mas independente da ferramenta isso pode acontecer. Acho que isso depende muito do filtro né, porque hoje a gente precisa fazer pesquisar antes de abrir uma tarefa, então com filtros melhores é mais fácil achar ou não se aquilo que você vai escrever já existe.

A.4 Equipe D

1) Nós usamos as issues do Github. Mas usámos Trello um tempo atrás, se a demanda era puramente do time. Migramos recentemente pro Github, porque ele é completo e mais simples de usar. No momento usamos só o Github issues e criamos um Kanban dentro do módulo de Projects.

2) Não usamos mais de uma ferramenta. Mas achamos interessante o quadro físico pela parte de aproximação do time, coisas físicas são mais palpáveis e ajuda na comunicação do time. Não fazemos uso do quadro por preferir a ideia do Github de centralizar tudo numa única ferramenta, mas talvez o físico se sobressaia ao Github por causa da interação dentro do time.

3 e 4) O Github é muito usado para controle de versão de código. O issues possui uma forma de reportar e escrever as coisas muito parecida com o Github já faz com PR e revisão de código. É a mesma interface, mesmo fluxo, mesmo ID de pré-issue. Pode-se referenciar recursos direto da issue, PR direto na issue, automatizar o gerenciamento de PR com a issue, algumas automatizações que o próprio Github já fornece ou através de plugins que a API fornece. Ele é bem rico, completo e centralizado. Quanto menos ferramentas diferentes usar melhor, porque é menos burocracia e mais direto. Para quem controla e gerencia as paradas é mto bom. A parte mais importante é a parte de integração.

5 e 6) A ferramenta expõe uma limitação de processo. Dentro de um repositório pode colocar vários boards que são representações dos times ágeis. Mas cada time trabalha de um jeito, tem um processo diferente, mas como o board faz parte do repositório, tudo que tem de configuração acaba sendo compartilhada, como o uso de label podem afetar uns aos outros, porque elas são artefatos do repositório e aí acaba tendo esse conflito de ideias de acordo com que cada time usa. Sinto falta de uma maneira mais eficiente e isolada para que times diferentes dentro de um mesmo repositório possam usar com

mais comodidade o issues. Hoje em dia é contornado com uma gambiarra fazendo cada time ter seu próprio; mas a ideia é não ter mais isso e ver como isso vai ser feito. Segunda maior limitação: a eficiência do uso da ferramenta para muitos usuários. 1 time ou 2 é "ok" de gerenciar. Colocar mais times é muita gente mexendo, referenciando, colocando coisas. É complicado gerenciar muitas issues. Todas elas vão aparecer na mesma listagem para todas as equipes. Eu tenho medo de ficar insustentável, até porque nunca tivemos essa experiência. Eu peso pela desorganização e caos que isso pode causar. O Redmine, por exemplo, tem muita tarefa perdida, ninguém sabe que tarefa existe ou o que fazer, mas elas tão lá. Não gosto muito dele não. Um ponto a ser melhorado é a organização de labels, que podia ser feito por boards ou por time direto dentro, sem precisar criar uma organização, para ficar mais claro. Uma forma de visualizar melhor e separar as issues por time. Implementar a ideia de times e não de usuários 100

7) O designer nem sempre usa a ferramenta, mas se precisar mexer e fazer alguma alteração em algo ele tem permissão e faz.

8) Não gosto do Redmine. Acho ele mais ou menos. Ele é o mal necessário no nosso projeto. É uma boa ferramenta, mas a configuração dele é difícil, os filtros são difíceis, a ferramenta tem alguns bugs que incomodam como salvar uma pré config de filtro para reusar depois. Isso é algo que não funciona e vive dando problemas. Ela já teve seus tempos áureos, mas a integração que o Github tem com o versionamento de código é tão grande e útil que não consigo mais voltar e olhar pra trás. É meio que uma dependência ter uma ferramenta que controle issues e problemas e tarefas no mesmo lugar que eu guardo meu código ou que ao menos tenha uma ótima integração com meu código, porque isso facilita demais para todo mundo mundo (teste, analista, programador). Sinto falta disso no Redmine. No Redmine era colado um link do código ou uma página da doc, fazer upload do pdf da doc. Porque não dá pra mostrar um html lá então é colocado um link para anexar na tarefa. Não precisava, poderia colocar o link direto. No issues tu bota o link e ele já mostra direto o conteúdo da página, mostra direto o recurso. No redmine não tem isso. É uma demanda mais recente para times. Talvez para projetos muito simples e pequenos ele seja mais eficiente. Mas para projetos como o nosso com 11 mil tarefas é mta coisa. Dessas 11 a maior parte nem importa mais e muitas coisas já foram perdidas, não tem uma maneira rápida de filtrar. No Github dá pra ver todas as issues para o time tal e que nunca foram modificadas. Para quem gerencia é muito mais rápido e eficiente. O Redmine não foi extinto ainda, porque funciona.

9) Duplicação de tarefas não é um problema hoje muito grave com relação do que já foi, pois tínhamos um controle bem pior das tarefas. Hoje já é muito mais claro e tem alguém responsável por olhar o macro e o pessoal tem um escopo bem definido. Sabemos o que cada time está fazendo e uma duplicação de tarefa é detecta muito rapidamente. Mas isso pode acontecer em qualquer ferramenta, o que precisa melhorar é o processo.

Outra coisa que pode mitigar isso, é o poder que a ferramenta tem na sua fase de busca (filtros) para que as pessoas procurem se a tarefa já exista, menor é a chance de duplicar a tarefa. O usuário nunca vai escrever perfeitamente o que ele quer. A ferramenta que for mais eficiente nesse sentido é a que menor terá duplicação de tarefas.

A.5 Equipe E

1) Usamos hoje Asana e o quadro físico. Antigamente usavamos apenas o Asana para fazer tudo, mas hoje fazemos o planejamento semanal. E dau no asana, como tinha muita separação por módulo, ficava difícil de acompanhar, então para deixar o daily meeting mais saudável e visível ficou melhor usar o quadro físico. Fizemos o experimento, gostamos e adotamos. Com o quadro físico ficou melhor fazer o planejamento de sprints (de duas semanas). A Asana era complicado de usar, por se tratar de uma ferramenta web virtual, nosso daily meeting não era olhando diretamente para ele, já no quadro físico a gente mexe nas tarefas durante a reunião e todos sabem tudo que os outros estão fazendo e vão fazer. As tarefas não são replicadas no quadro físico e no Asana. No asana ficam as tarefas macros e definições com o cliente e o supervisor de desenvolvimento e no quadro físico as tarefas menores e o andamento que elas têm. Por exemplo, um quadro no Asana diz meio que formalmente que uma funcionalidade terá quatro características, isso então é colocado no quadro físico como 4 tarefas diferentes nos post-its.

2) Não usamos outras ferramentas antes, sempre foi o Asana e posteriormente aderimos ao quadro também.

3 e 4) O Asana é legal, porque dá de fazer subtarefas de subtarefas, nós podemos fazer vários sub níveis dentro de uma única tarefa. Por exemplo uma funcionalidade tem o nível de análise, de testes, casos de testes, testes automatizados e aí tu vai adentrando nos níveis. Na época o Trello não tinha, o Github tinha um visual desagradável. E o quadro por ser físico, mexer tatualmente são as principais características de porque usamos eles. Esses são os pontos positivos. Conseguimos fazer itens, checkbox, checklists, clicar para completar coisas. Certamente, não usamos todo o potencial da ferramenta, mas também há a questão histórica. Funcionamos muito bem assim e precisamos de um motivo muito bom para mudar. Já tivemos várias discussões para migrar, mas ainda não vimos vantagem.

5) O Asana não tem integração com o Github diretamente, isso talvez pudesse nos ajudar, e acho que esse é o principal motivo que as pessoas estão migrando, porque o Github automatiza muito e o quadro não tem ligação com nada né, se ele pegar fogo se perde tudo, a única segurança que temos é essa câmera de monitoramento que consegue visualizar o quadro. Pode ser que o tamanho dele um dia seja pequeno, para acomodar tantos post-its, mas acredito que pra nossa realidade nunca vai exceder o que usamos hoje. Além do daily meeting, né, o Asana não permitia a gente usar ele no daily meeting, só se

fosse com um projetor ou algo do tipo.

6) No Asana integração com Github. O quadro já é muito maleável, depende de quem mexe nele. Os post-its poderiam vir mais preparados para colocarmos as informações nele. Hoje é um papel em branco que a gente pode escrever o que quiser dentro. Inclusive, parece meio "zoeira", mas fazemos uma data em cima dele. Data de entrada e saída. Post-its preparados para receber anotações diferentes seria bom. Uma das dificuldades que nós temos é ter que pegar os post-its e colocar numa planilha de lead time do Kanban depois.

7) Todos usam.

8) Redmine é extremamente útil. É o pilar da organização. Se não fosse por ele, a gente não teria acesso aos boards das outras equipes e até mexer neles seria ruim. Então precisa de uma ferramenta que centraliza tudo e que seja mais burocrática. Dentro das equipes precisamos apoiar mais a comunicação, mas fora delas precisa ser mais formal. E também porque é nossa comunicação com o cliente, se foi definido algo em reunião é registrado em Redmine, se a gente não registrou e entramos em impasse com eles, eles perguntam "porque não colocaram no Redmine?".

9) Acontecia, mas hoje não tem mais tanto. Dentro da nossa equipe faz muito tempo que não vemos isso. Como te falei antes, a gente preza por uma comunicação perfeita, então sempre sabemos o que os outros estão fazendo. Aí quando uma tarefa é criada, a gente sabe. Então é difícil criar outra igual. Agora no caso do Redmine, quando precisa procurar alguma tarefa lá pra ver se já tem é mais complicado. Os filtros dele são meio ruins, tem que gastar muito tempo em pesquisa, às vezes, nem vale a pena ficar pesquisando. Aí numa dessas você pesquisa, mas tem a infelicidade de não por uma palavra-chave exata e não acha a tarefa igual a que você tá criando. Pronto, tarefa duplicada.

A.6 Equipe F

1) Usamos Github.

2) Nunca utilizamos outra ferramenta além dele, desde a criação da equipe foi essa.

3 e 4) Integração com versionamento de código, centralização, automatização. Na parte de desenvolvimento dá para integrar com ferramentas de teste e de build contínuo, tipo marcar lá no PR se aquilo passou ou não. Daí o código é travado para não ser integrado.

5) Para fazer o gerenciamento de tarefas, como era feito no Redmine, é mais limitado, porque tem menos opções de filtros, ele não tem muita categoria, muita coisa pra classificar tuas issues, tem que se virar com as tags. Comparando com o Redmine que tem muito combo, dá pra classificar bem a tarefa. O github não tem muito isso além do uso de tags. Fora que ali você pode colocar se quiser, no Redmine eram campos obrigatórios.

Algumas opções de automatizações extras além da que tem hoje. Hoje ele só faz o processo final, mas a gente, internamente, ainda tem um desenvolvimento intermediário, pq ainda tem que passar pelo QA, e não é interessante que ela vá direto pro done. É porque assim, o básico dela te joga pra esse lado. No board por exemplo, não dá pra ver que a issue tem um PR atribuído, você precisa abrir ela e entrar pra ver. É meio complicado usar o Github Projects, usando o método Kanban dessa forma. É difícil contar as tarefas de QA por exemplo, por causa dessa falta de automatização.

6) Automatização já existe mas é limitada, deveria existir mais opções de automatização. Eu não quero sempre levar para done, eu posso querer levar para uma outra etapa do desenvolvimento, outra coluna por exemplo. Outra coisa na nossa configuração de automatização, é o nosso fluxo. Ele diz que pelo menos 2 pessoas tem que aprovar a fase de code review. Só que se uma pessoa aprovar, o Github já muda automaticamente o card da coluna. Aí quando a segunda pessoa for olhar o código para aprovar, ele não vai mais estar na coluna de pedindo review. Então não tem essa configuração, por exemplo, de escolher quantos você quer. Se tiver um ele já vai e isso é ruim para nós. Ele só aceita 3 etapas de processo: todo, doing e done e aí divide entre eles. E como nossas outras categorias de dev, qa fazem parte do doing, ele entende que nossos processos são 1 só. Se eles conseguissem refinar melhor isso seria útil.

7) Todos usam.

8) O Redmine ainda é usado pela nossa analista, porque ela precisa fazer triagem das tarefas que estão lá que precisam de melhoria no novo PEC. Então é criada uma issue relacionada no Github e colocado o link do Redmine. Mas a equipe em si, não adiciona nenhuma issue nova no Redmine. Mas acredito que precisa existir um intermediário entre as equipes e o meio externo que é o cliente e a equipe de qualidade. Uma coisa que eu não gosto no Redmine são os filtros dele, são muito ruins de usar. Principalmente a parte de salvar um filtro pra ter reuso depois.

9) Tivemos uma ou outra recorrência de tarefas duplicadas aqui. Como não tem esquema de módulos e como categorizar bem uma issue, nós tivemos esse problema. E também porque no início criávamos uma única issue com um checklist enorme de problemas dentro. E vimos que isso era ruim para o fluxo, porque aquela issue demorava muito a ser fechada até dar check em todos os problemas de dentro dela. E aí começamos a desmembrar elas em issues menores, separadas e diferentes. Nesse processo acabou ficando uma ou outra duplicada. E aí pra contornar isso, nós tivemos que inventar uma forma nossa, padronizada, porque a ferramenta não provê isso. Então nós colocamos entre colchetes o nome do módulo no início do título, seguido daí pelo título da tarefa. Então assim dava pra filtrar mais fácil onde o problema tava acontecendo. mas foi uma decisão nossa, a ferramenta não ajudou. Quer dizer, o filtro e a busca dela são boas, porque daí permite a gente de filtrar pelo nome do módulo no título direto desse jeito.

APÊNDICE B – Artigo

O impacto que as ferramentas de Bug Tracking Report têm sobre o ambiente de desenvolvimento e teste ágil

Christian Silva de Pieri[1]

[1]Departamento de Informática e Estatística – INE
Universidade Federal de Santa Catarina (UFSC)
Florianópolis – SC – Brasil
christiandepieri12@gmail.com

Abstract. *The main focus in this work is to do a case study with agile teams in order to know about the best Bug Tracking Report tools and its feedback, whether positive or negative, that they can give to an agile development and testing environment.There will be two major research areas: the systematic bibliography and exploratory field research in the laboratory. The e-SUS AB project has 6 different multidisciplinary teams, including testers, developers and software analysts. Each team is also self-manageable, that is, they have freedom to decide how to work, repecting the agile development philosofy. Several tools are used by teams. The study will be based on the bibliography, methodologies and agile tests with the use of these tools, in search of a greater understanding if there is a better tool of bug tracking report that the team adapts with more excellence. The most searched parameters was understant what is the bests existing functionalities in these tools and which of them are essential from the point of view of development and agile test.*

Resumo. *O foco principal deste trabalho de conclusão de curso foi fazer um estudo de caso com equipes ágeis, a fim de entender melhor as melhores ferramentas de Bug Tracking Report e o feedback, seja positivo ou negativo, que elas podem dar a um ambiente de desenvolvimento e de teste ágil. Foram realizados dois tipos de pesquisa: a bibliográfica sistemática e a pesquisa exploratória em campo no laboratório. O projeto e-SUS AB possui seis diferentes equipes ágeis multidisciplinares, que compreendem testadores, desenvolvedores, analistas de sistemas e designers. Cada equipe é também auto-gerenciável, ou seja, possui liberdade para trabalhar da melhor forma que decidir dentro do desenvolvimento ágil. Diversas ferramentas são utilizadas pelas equipes. O estudo se deu em cima da bibliografia, metodologias e testes ágeis com o uso destas ferramentas, em busca de uma maior compreensão se existe ou não uma melhor ferramenta de bug tracking report que a equipe se adapte com mais excelência. Buscou-se entender quais as melhores funcionalidades existentes nessas ferramentas e quais delas são essenciais do ponto de vista do desenvolvimento e do teste ágil.*

1. Introdução

Atualmente, segundo [Wazlawick 2013], o desenvolvimento ágil de software – que é uma metodologia de software que providencia uma estrutura conceitual para reger projetos de engenharia de software, é utilizado por diversas empresas e equipes. Interligado ao desenvolvimento de software tem-se o teste de software, elemento importante no ciclo de

vida de um sistema e usado para garantir a qualidade final de um determinado produto. No desenvolvimento do software ágil, ao passo que o desenvolvedor programa um determinado algoritmo, módulo, classe, etc., o analista de qualidade faz o teste do mesmo. Ao final do processo, se erros (bugs) forem encontrados, eles são reportados em alguma ferramenta de armazenamento, chamadas de Bug Tracking System, e o programador, assim, consegue saber qual problema aconteceu e em qual parte do sistema, para que ele possa corrigir.

Essas ferramentas estão sujeitas a avaliação e ao se fazer isso, de forma individual, deve-se ter em mente alguns tópicos para serem levados em consideração:

a) como se dá um bom *report* de *bug*? Como efetivamente escrever um bom passo de teste de um erro que ocorreu para ser corrigido depois? Seja este pela própria pessoa que o reportou ou algum outro desenvolvedor.

b) entendidas as necessidades que um *report* precisa atender, como deve-se avaliar e elencar os campos, funcionalidades e componentes que um *Bug Track System* deve possuir?

c) estas ferramentas podem e devem ser aperfeiçoadas para se adequarem a organização a qual estão atendendo, mas quais melhorias são necessárias e como fazê-las?

d) duas tarefas referenciando um mesmo problema são extremamente comuns em projetos de grande porte e com um vasto *backlog* de tarefas a serem corrigidas e desenvolvidas. Como, então, identificar duplicação de tarefas e ter o rastreio de problemas recorrentes?

No Laboratório Bridge, que é um laboratório ligado à Universidade Federal de Santa Catarina e que atua na pesquisa e desenvolvimento de soluções tecnológicas, conectando governo e cidadãos, foi feita toda a pesquisa proposta neste trabalho abrangendo as equipes ágeis e o ferramental que seus colaboradores utilizam no dia-a-dia.

O laboratório possui quinze equipes ágeis distribuídas nos seus projetos de atuação, sendo seis delas do projeto e-SUS AB, ao qual o autor já esteve inserido em uma destas equipes. Cada equipe é auto-organizável, autogerenciável e composta por colaboradores de forma interdisciplinar. Normalmente esta distribuição se dá aos pares, ou seja, dois programadores, dois testadores e dois analistas de requisitos. Esta formação não é uma regra geral, algumas equipes, por exemplo, contam com 3 programadores e 1 analista de requisitos. Existe também um núcleo especializado em *design*, composto apenas por profissionais da área. E outro núcleo de qualidade, com profissionais que desempenham o papel do usuário final do sistema e reportam erros, defeitos de usabilidade e inconsistências entre sistema e documentação. Esta é a última barreira para testes e descobrimento de *bugs* antes do sistema ir para produção.

O laboratório não segue a risca um determinado modelo ágil para gestão, planejamento e desenvolvimento de *software*, mas é fortemente inclinado para o Scrum, com o catalizador Kanban.

Cada equipe pode escolher e fazer uso da ferramenta que mais se adéqua à sua realidade; então foram identificadas 5 ferramentas de uso pelas equipes, são elas: Github,

Asana, Trello, Planilhas Google e Quadro físico. Há também uma sexta ferramenta, chamada Redmine. Ela é de uso mútuo de todos os colaboradores do projeto e serve como um centralizador de tarefas, bem como interface de comunicação com o cliente.

O objetivo do estudo então, foi de avaliar se existe ou não uma melhor ferramenta que se adéqua a cada particularidade da equipe e se esta equipe tem sua produtividade aumentada com o uso da ferramenta. Além disso, após a aplicação da pesquisa com base nos tópicos descritos por [Blair 2004], teve-se uma maior noção das limitações de cada ferramenta. Buscou-se, também, por propostas de melhorias nas ferramentas para aplicá-las e melhorar os fluxos de trabalho das equipes. Outro fator notório no estudo, é as equipes levam em consideração princípios ágeis tais como comunicação e os consideram mais importantes do que o uso de ferramentas e processos.

1.1. Organização

Este artigo está organizado da seguinte forma:

- Esta introdução: parte introdutória para contextualização do problema;
- Ponto de vista da pesquisa: explicação dos dois tipos de pesquisa utilizados no trabalho;
- Entrevistas, coleta e processamento de dados: como se deram as entrevistas com as equipes ágeis;
- Resultados compilados obtidos: gráficos explicativos sobre tópicos relevantes das ferramentas;
- Comparação de ferramentas: uma tabela comparativa com as funcionalidades mais citadas nas entrevistas;
- Conclusões: conclusões retiradas após o término da pesquisa;
- Trabalhos futuros: possíveis trabalhos futuros com base no que foi concluído.

2. Ponto de vista da pesquisa

A pesquisa foi dividida em dois tipos: a bibliográfica sistemática e a exploratória de campo.

Conforme [Kitchenham 2004], fazer uma revisão sistemática de uma literatura é identificar, avaliar e interpretar todas as pesquisas relevantes a um determinado tópico. Existem muitas razões para realizar este tipo de exame, como verificar novos parâmetros para a pesquisa em foco; identificar falhas na pesquisa atual e sugerir ideias; resumir evidências e elencar benefícios.

Ao passo em que a pesquisa bibliográfica sistemática elenca a relevância de trabalhos correlatos, de um determinado tópico presentes na literatura, a pesquisa exploratória faz uma investigação sobre o assunto. ”Pequisa exploratória é quando a pesquisa se encontra na fase preliminar, tem como finalidade proporcionar mais informações sobre o assunto que vamos investigar, possibilitando sua definição e seu delineamento, isto é, facilitar a delimitação do tema da pesquisa; orientar a fixação dos objetivos e a formulação das hipóteses ou descobrir um novo tipo de enfoque para o assunto. Assume, em geral, as formas de pesquisas bibliográficas e estudos de caso.” [Prodanov 2013]. São feitas entrevistas com pessoas que interagem com o objeto de estudo da pesquisa, ou seja, possuem experiência prática com as ferramentas descritas no capítulo anterior e possibilitam análise de exemplos que incitam a compreensão.

3. Entrevistas, coleta e processamento de dados

A pesquisa de campo, foi aplicada no Laboratório Bridge, que é um laboratório da Universidade Federal de Santa Catarina (UFSC), onde o autor deste trabalho ocupa a função de estagiário em qualidade de *software*.

O ambiente conta com aproximadamente 120 colaboradores multidisciplinares alocados em um dos três projetos em andamento dentro do laboratório. E, mesmo retirando os profissionais que não fazem parte de times ágeis, como os do administrativo, limpeza e gestão, ainda seriam muitas pessoas para aplicar a pesquisa e coletar informações. Portanto, o escopo das entrevistas foi reduzido para apenas o líder de cada uma das equipes ágeis.

Não há problema em ouvir apenas um colaborador por equipe, no caso os líderes, pois dentro do laboratório, os líderes refletem os anseios dos demais, ou seja, a resposta do líder de usar uma determinada ferramenta não se dá por decisão única, mas sim, porque em conversa com a equipe todos optaram por ela e chegaram a um consenso que ela era a melhor de se utilizar.

A pesquisa de campo se deu a partir de uma conversa com cada entrevistado e foi direcionada pelo autor deste trabalho com base em dez perguntas, são elas:

- 1) Qual(is) ferramenta(s) de BTS a equipe utiliza no dia a dia? Se usam mais de uma, porque o fazem? Há limitação em alguma das ferramentas sendo necessário o uso de outra?
- 2) Utilizavam alguma outra ferramenta antes desta? Qual? O que motivou a mudança? O que melhorou?
- 3) Por que utilizam tal(is) ferramenta(s)?
- 4) Quais os pontos positivos da ferramenta atual?
- 5) A ferramenta atual possui alguma limitação? Tanto do ponto de vista da equipe (para o seu uso corrente no dia a dia) ou que os usuários acham que ela possa conter no geral.
- 6) Sobre melhorias na ferramenta atual:
  - a) Quais pontos já existentes da ferramenta poderiam ser melhorados?
  - b) E quais funcionalidades poderiam ser adicionadas para melhorá-la?
- 7) Todos da sua equipe utilizam essa ferramenta? Se não, por quê?
- 8) O que você pensa sobre o *Redmine*?
- 9) O que você acha sobre duplicação de tarefas? É um problema?
- 10) Dê uma nota de 1 a 5 quanto a cada um dos tópicos a seguir: adaptabilidade, rastreabilidade, integração com versionamento de código, campos customizáveis, usabilidade, notificações, segurança, métricas e fluxo de trabalho.

- Notas sobre as perguntas

- Na questão 8 foi perguntado sobre a ferramenta *Redmine*, pois ela é a ferramenta central, utilizada por todos os projetos e colaboradores do laboratório. Também é por ela que o cliente pode acompanhar a evolução das tarefas e, se desejar, abrir tarefas entregáveis, sugerir mudanças, dispor opiniões e comentários para desenvolvimento.
- Os tópicos da questão 10 foram pensados e retirados do guia disposto por [Blair 2004].

4. Resultados compilados obtidos

Aqui nesta seção estão descritos alguns dos resultados obtidos durante as entrevistas. Esses resultados contemplam desde a elencação de pontos positivos, limitações das ferramentas e melhorias discutidas; até pontos curiosos e relevantes segundo o guia de [Blair 2004].

4.1. Principais motivos para uso das ferramentas e seus pontos positivos

Os usuários da ferramenta Github alegaram que ela possui um módulo chamado de *Issues*. Neste módulo é possível reportar e descrever tarefas de forma muito parecida como já era feito anteriormente ao se abrir ou revisar um *Pull Request*. Ele tem a mesma interface, fluxo e identificadores que os *PRs*. Isso torna possível referenciar recursos diretamente entre eles, por exemplo fechar uma determinada tarefa ao integrar um código que fazia referência a ela.

O fato de tudo estar presente dentro de um único sistema de forma centralizada e unificada foi relatado por todos os entrevistados, em conjunto a parte de automatização que o coloca à frente das demais ferramentas. Além disso, ele possui menos burocracia, mais facilidade de uso e de aprendizado. Outro ponto indicado por duas equipes foi a parte de integração que ele tem com outros programas ou *plugins*. Na parte de desenvolvimento, por exemplo, existem várias ferramentas de teste e de *build* contínuo, que indicam dentro do *PR* se há alguma anomalia e tudo isso pode ser configurado para o código não ser integrado até que tudo seja corrigido.

Já a equipe que faz uso de Asana e quadro físico, diz que a principal característica da segunda ferramenta é a aproximação do time e o forte impulsionamento que ele provê para a comunicação interna dar certo. Ele é altamente customizável e os membros podem mexer manualmente nas tarefas. Ele tem ajudado muito no *daily meeting* e aponta ainda que pelo fato das pessoas retirarem um tempo ficando paradas em frente ao quadro, há uma maior absorção dos afazeres durante a semana. Já o Asana possui várias características que a equipe utiliza diariamente e que não encontraram em outra ferramenta, tais como a criação de sub-níveis dentro de uma mesma tarefa. Por exemplo, uma mesma tarefa pode ter o nível de *teste* e dentro do *teste* ter sub-níveis de *teste automatizado*, *testes manuais* e *casos de teste*. Outro ponto forte dito foi a possibilidade de criar *checkboxes*, *checklists*, rápidos cliques para marcar itens como completos.

4.2. Limitações das ferramentas

Para a ferramenta Asana, a limitação mais discutida foi a falta de integração direta com o Github para fazer o versionamento de código e também a falta de automatização que ela tem, comparando com a outra ferramenta. Essa seria uma funcionalidade que ajudaria muito a equipe e, ela disse ainda que acha que esse é o principal motivo dos outros times estarem migrando de plataforma. Além disso, o Asana provia uma forte limitação para a equipe, conforme explicado, para realização dos *daily meetings*. Motivo esse pelo qual adotaram o quadro físico. Durante a reunião diária, eles precisariam fazer uso de um projetor, por exemplo, para verificar as informações presentes no Asana.

Já quando questionados sobre o quadro físico, apontaram que a maior limitação é a de segurança, uma vez que ele não tem ligação com nada. Qualquer pessoa com acesso a sala onde o quadro está pode alterar ou omitir as informações presentes nele. Ele também

pode possuir limitações físicas, em questões de tamanho. Mas a equipe alegou que isso não chega a ser um problema, por ele ser grande nunca precisaram utilizar todo o espaço.

A ferramenta Github recebeu bastantes e variadas críticas sobre seu funcionamento. As mais comentadas foram sobre não dar suporte para múltiplos times ou quando muitos usuários necessitam utilizar a ferramenta ao mesmo tempo. A equipe D disse, por exemplo, que acha muito bom o funcionamento da ferramenta para 1 único time. Quando se coloca um segundo time dentro do mesmo repositório já fica mais complicado; e, acima de dois times, pesa pela desorganização e caos que isso pode causar. Disse que: "É complicado gerenciar muitas *issues*. Todas elas vão aparecer na mesma listagem para todas as equipes. Tenho medo de ficar insustentável, até porque nunca tivemos essa experiência". E completou comparando com o Redmine, pois tem medo que aconteça de ficar muitas tarefas perdidas, que ninguém sabe que elas ainda existem ou não sabem o que fazer com elas.

Outro ponto levantado pelas equipes A e C, ainda relacionado com a interação entre os times, foi de que quando eles desejam fechar uma tarefa dentro do próprio repositório, fazendo uso da automatização provida por ela, mas abrindo um *Pull Request* para outro repositório externo, é necessário que a pessoa responsável por aceitar o PR no segundo repositório tenha privilégio de administrador do primeiro. Disse que é complicado ter que dar esse tipo de privilégio para pessoas de fora do time apenas para utilizarem dessa funcionalidade. Acredita ainda, que o Github tem essa política por fatores de segurança, mas não vê isso com tanta criticidade para ter que funcionar dessa maneira. O líder da equipe alegou que: "Já participei de outras equipes que por causa dessa limitação, eles não adotaram essa ferramenta. Essa parte toda de automatização do Github é muito ferida quando se trata de repositórios distintos".

A equipe F, que utiliza o Github *issues* desde que foi criado, também apontou que a automatização é limitada e eles gostariam que existissem mais opções. Eles fazem uso do módulo *projects* do Github. Essa funcionalidade é bem parecida com os *cards* do Trello e do Asana, porém possui automatização e pode ser relacionado com *issues* e *Pull Requests*. Quando alguma alteração for feita neles, um *card* do *board* é movido para a aba de *doing* (tarefas em desenvolvimento) para *done* (tarefas completadas), por exemplo. Mas nem sempre o fluxo da equipe funciona assim. Eles possuem outra coluna dentro da etapa de desenvolvimento, e gostariam que o *card* fosse movido por dentro dela também, o que hoje não é possível. Outro ponto dito pela equipe foi que quando eles abrem um PR, é necessário que pelo menos duas pessoas revisem e aprovem-no para que ele seja integrado. O que acontece com a automatização, atualmente, é que se uma única pessoa aprovar, ele já troca o *status* de em desenvolvimento para feito. Eles gostariam, então, que houvesse mais configurações de automatização, para customizar essa parte e indicar que o *card* só trocará de coluna de forma automática se *n* pessoas aprovarem as alterações no código.

A equipe B disse que não vê nenhuma limitação na sua ferramenta atual que impeça seu uso. Apontaram que pelo time ter uma ótima comunicação, não importa a ferramenta que usem, o processo ainda vai funcionar. Relembraram que quando usavam as Planilhas do Google, possuíam a limitação de filtrar tarefas de forma eficiente; uma vez que era necessário utilizar a ferramenta de pesquisa do próprio navegador (comando *ctrl* + *f*) para encontrar algo na página. E isso era ruim, pois eles precisavam digitar exatamente

o que desejavam, uma vez que não existe um algorítimo de busca como das ferramentas mais complexas. Porém, como estavam sempre conversando, o fluxo de trabalho era bom e produtivo.

Um ponto levantado pela equipe que eles não tinham problema anteriormente e que têm hoje com o uso do Github, é que o histórico, principalmente na parte de evolução das *issues* pode acabar ficando muito grande e atrapalhar quando necessitarem saber algo dentro dela. Citaram o exemplo de quando adicionam ou retiram *labels* da tarefa, o histórico é mantido. Gostariam de poder filtrar para ler apenas comentários ou trocar a forma de apresentação, para que o mais recente fique em cima. Atualmente a linha do tempo de uma tarefa no Github é da mais antiga primeiro para a mais recente no final da página.

4.3. Melhorias discutidas

Muitas melhorias foram sugeridas pelas equipes, de acordo com suas necessidades de uso diário e também de acordo com as limitações apontadas por elas anteriormente.

A equipe A, exemplificando, disse que gostaria que ficasse mais claro alguns funcionalidades que o Github possui, mas que ficam escondidas. Muitas vezes eles precisam recorrer a documentação da ferramenta para saber como proceder. Outra situação de melhoria discutido foi a questão dos filtros, que, do ponto de vista deles, possui sintaxe própria e não amigável para usuários não tão avançados. Por exemplo, hoje, a ferramenta de comunicação das equipes ágeis com o cliente e equipes de qualidade e suporte do projeto é o Redmine; e, caso todo o laboratório migrasse para o Github, tanto o cliente, quanto alguns membros da equipe de suporte (que não são da área de tecnologia) teriam que aprender a usar, o novo filtro.

Uma outra melhoria apontada por essa equipe, juntamente da equipe F, é sobre adequações no processo de automatização de tarefas e *Pull Requests*: como explicado na seção anterior, hoje é de certa forma limitado e essas equipes gostariam de ter mais customizações e liberdade para automatizar melhor seus processos internos.

Já a equipe C, alertou sobre algumas funcionalidades que estão presentes na ferramenta Trello, mas não estão presentes no Github, que é a possibilidade de adicionar pequenos compontens nos *cards* e nas tarefas com apenas um clique. Esses componentes podem ser *checklists*, listas e anotações. Hoje, eles precisam fazer uso de *Markdown* para criar essas customizações nas suas *issues*. Outra questão levantada que existe no Trello e não no Github é a possibilidade de atribuir *labels* diretamente dentro de um *card*. Atualmente, é necessário sair da página, criar a *label* e depois adicioná-la manualmente no *card*.

A equipe D disse que, no momento, é essencial para a ferramenta e também para o laboratório, que tenha alguma forma de organizar, visualizar e separar as *labels*, os *cards* e *projects* do Github por times. Realmente, implantar a noção de times nesse sistema, que hoje não existe. No presente momento, ela conta apenas com usuários independentes e é necessário contornar isso, para a realidade atual do projeto, com a criação de vários repositórios (diferentes para cada equipe ágil) que abrem PR para um principal.

Dando continuidade nas melhorias para a ferramenta Github, a equipe B contribuiu dizendo que gostaria que a parte de visualização de uma tarefa fosse mais organi-

zada e customizada de acordo com o que eles necessitavam ver no momento. Atualmente, quando se abre uma *issue*, é fornecido o título dela e uma descrição opcional. Conforme vão se adicionando artefatos nela, ela vai aumentando seu tamanho de forma progressiva para baixo. De modo que o mais antigo aparece no topo e o mais recente no final da página. Ao adicionar *labels*, descrições, comentários e referências na tarefa, o seu corpo só tende a crescer. Muitas dessas informações podem ser irrelevantes, na hora de verificar algo dentro da *issue*, caso ela tenha tomado proporções gigantes. A equipe então relatou que seria interessante a ferramenta prover um filtro, para ocultar ou mostrar determinados tópicos. Por exemplo apresentar apenas os comentários, ocultando as outras modificações. Outra melhoria sugerida foi a de poder ordenar a *timeline*, ao invés de ser de forma cronológica (da mais antiga para a mais recente), ser da forma contrária (da mais recente para a mais antiga); mostrando o mais atual primeiro (em cima).

E por fim, a equipe E, que faz uso das ferramentas Asana e Quadro físico apontou melhorias pontuais nos dois casos. Para o quadro, eles disseram que seria interessante se os *post-its* já viessem pré-preparados para receber as tarefas de acordo com a realidade deles. Hoje, eles usam um papel totalmente em branco e, nele, desenham pequenas caixas para colocar data de entrada e saída do quadro e outras customizações padronizadas pela equipe. Para a ferramenta virtual, eles disseram que gostariam que ela tivesse interação com o Github, porque hoje não possui.

Em pesquisa, redigindo esse trabalho, o autor verificou que de fato não existe interação nativa da ferramenta Asana com o Github, mas encontrou uma ferramenta terceira chamada [fer]. Com ela é possível não só sincronizar as duas ferramentas em questão como também outras ferramentas existentes no mercado: Trello, Jira, Gitlab, Bitbucket, Basecamp e Wrike. Dessas descritas comportadas pelo Unito, apenas o Trello está presente no escopo deste trabalho, por ser relevante para as equipes do laboratório.

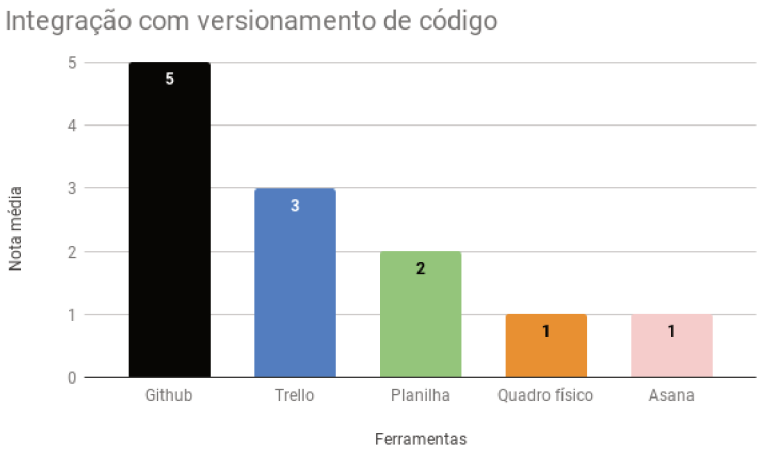
O Unito consegue conectar e sincronizar as ferramentas e, a partir disso, é possível aplicar ações que serão direcionadas para as demais ferramentas tais como fechar tarefas, atualizar títulos e descrições, atribuir um colaborador à *issue*, adicionar *tags* e *labels*, fazer *upload* de arquivos e comentários. Ele tem como desvantagem ser uma ferramenta paga, mas possui uma versão de *free trial* para teste. Constatado tudo isso, foi então apresentado a equipe essa ferramenta, que, por sua vez, comprometeu-se a avaliar e possivelmente implementar seu uso no dia a dia.

4.4. Guia de Blair: integração com versionamento de código

[Blair 2004] afirma que um ponto que vale a pena ressaltar é a capacidade dessas ferramentas fazerem um *link* entre o *bug* e o código fonte do programa; ou seja, ter uma forma do documento que descreve o erro estar ligado ou relacionado com o código, podendo assim ter um maior rastreio e saber onde efetivamente mexer para correção do problema

Não coincidentemente, a única ferramenta com nota máxima neste tópico é a do Github. Isso se dá, porque o versionamento de código dos projetos do laboratório é feito com ele. E a parte de manejo de *issues* desse sistema é amplamente voltada para trabalhar em conjunto com o código hospedado na plataforma. Inclusive esse foi um dos pontos positivos mais elencados por todas as equipes: a centralização em uma única ferramenta de código, tarefas e o poder que os usuários têm de referenciar e integrar código com partes processuais e atividades de desenvolvimento.

Figure 1. Integração com versionamento de código



Fonte: o autor, 2019.

O Trello recebeu média 3, pois possui os chamados *power ups*. Os *power ups*, permitem que usuários tragam funcionalidades adicionais aos seus quadros para integrar aplicações com o Trello. Um *power up* possível de ser adicionado é o de integração com o Github, portanto, versionamento de código. Com ele consegue-se anexar *branches*, *commits*, *issues* e *pull requests*.

A planilha recebeu nota 2, porque mesmo que não haja nenhuma integração com versionamento de código, uma das suas principais qualidades é sua ampla customização. Podendo, então, a equipe criar uma coluna adicional e incluir lá um *link* para a referenciada página no Github do código.

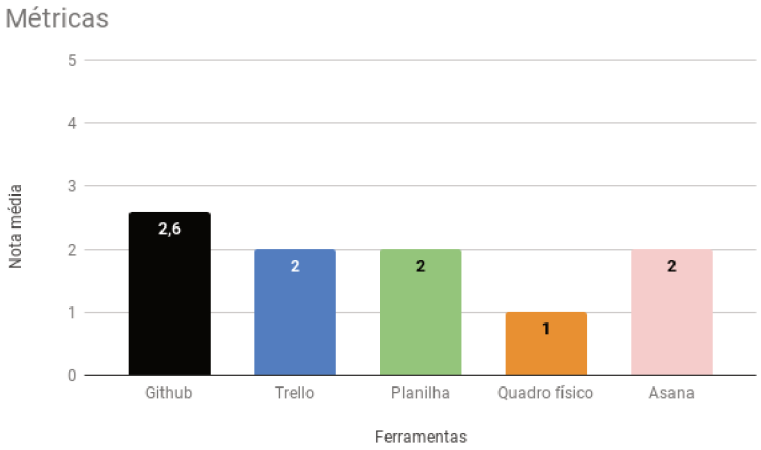
Quadro físico e Asana, como apontam as equipes, não possuem esse tipo de integração, portanto receberam nota mínima.

4.5. Guia de Blair: métricas

Conforme aponta [Blair 2004] é útil que um *BTS* provenha, de forma rápida e organizada, informações e relatórios, com métricas detalhadas para tomadas de decisão. Por exemplo: número de *issues* criadas ou corrigidas, organizadas por data e prioridade; sinalizar *bugs* que ainda não foram corrigidos e sua funcionalidade é requerida na próxima entrega; ter uma biblioteca de reportes predefinidos; listar os *trending bugs*, os mais acessados ou recorrentes.

Todas as ferramentas alvo de estudo deste trabalho, sem nenhuma exclusão, receberam notas muito baixas neste tópico. Isso ocorreu, porque os usuários destas ferramentas alegaram que elas não possuem geração de relatórios e ou alguma funcionalidade que permita contagens (sejam elas de problemas encontrados, resolvidos, ainda abertos, para uma determinada pessoa...) de forma automatizada afim de relatar informações úteis para o processo de desenvolvimento de *software*. Caso os usuários queiram ou necessitem, e assim muitas equipes o fazem, eles devem fazer manualmente uma pesquisa com base nos filtros pré-disponíveis e fazer a contagem para retirarem seus próprios diagnósticos e ter estimativas de acordo com o que necessitam.

Figure 2. Métricas



Fonte: o autor, 2019.

As equipes que fazem uso de Github deram notas um pouco mais altas, pelo fato da ferramenta também ter implementado o conceito de *milestones*. Elas são ”marcos” do projeto e dão noção de eventos como data, título e descrição - geralmente utilizados para definir prazos de entrega. Assim, é possível atribuir tarefas para um determinado marco (normalmente uma *sprint*) e conforme essas *issues* são finalizadas, uma barra no topo do projeto cresce, indicando que a *milestone* está próxima de ser completada. Mas, ainda assim, a ferramenta carece de relatórios que apontem dados consistentes e mais úteis, tais como *bugs* mais acessados e recorrentes, quantidade *n* de tarefas atrasadas ou criadas em um determinado marco.

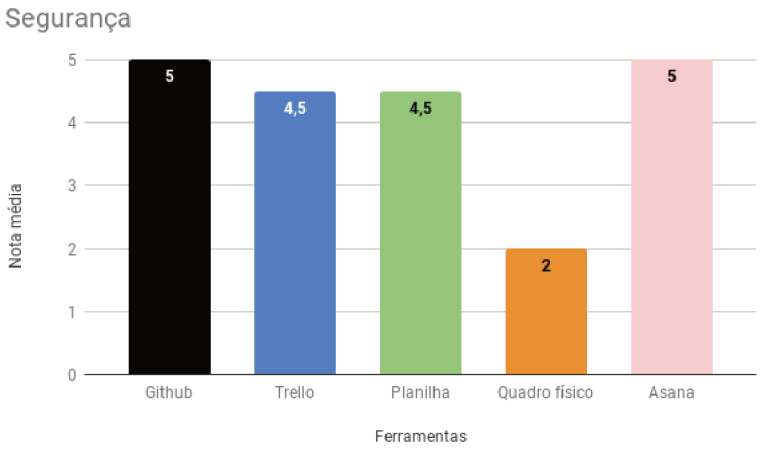
4.6. Guia de Blair: segurança

Segundo [Blair 2004] dependendo do cenário, é preciso que alguns grupos de usuários tenham permissões diferentes de outros. Nem todos são aptos a fazerem modificações como inclusão, edição, visualização ou customizações. Algumas funcionalidades devem ser apenas superficiais. Um repositório comunitário, por exemplo, não deveria permitir que um usuário comum tenha privilégio de exclusão de *issues*; visto que todo o trabalho produzido por outros poderia ser descartado por um erro ou até mesmo de forma maliciosa.

No tópico sobre segurança as notas foram muito altas com exceção do quadro físico. Todos os usuários das outras ferramentas apontam que se sentem muito seguros em usá-las. Por se tratarem de sistemas *web*, todas possuem redundância e são fortes contra perda de informações. Ainda sobre elas, todas possuem sistemas de *login*, onde é necessário estar autenticado para acessar determinados recursos; e, dentro de suas configurações é possível dizer quais pessoas ou grupo de pessoas podem acessar, alterar e excluir certas atividades.

Já no quadro físico, brinca o entrevistado: ”a única segurança que temos é esta câmera de monitoramento que consegue visualizar o quadro”. Qualquer pessoa que tenha acesso à sala e, por consequência ao quadro, pode modificá-lo, rasgar os *post its*, riscar ou escrever nele, visualizar todas as informações e andamentos das tarefas. Além disso, num

Figure 3. Segurança



Fonte: o autor, 2019.

caso de sinistro maior, todas as informações seriam perdidas. Razões estas que tornam sua nota tão baixa.

5. Comparação de ferramentas

Essa seção contém algumas funcionalidades presentes nas ferramentas. A partir disso, é montado uma tabela para cada funcionalidade e atribuído um identificador (F#) na tabela 1. Já na tabela 2 é marcado com um *x* se essa funcionalidade está presente ou não na respectiva ferramenta.

Table 1. Exemplos de funcionalidades existentes

#	Descrição
F1	O uso da ferramenta é gratuito
F2	Possui uma versão paga com novas funcionalidades inclusas
F3	Possui integração direta com o código da aplicação
F4	Atribuição de pessoas a determinados recursos
F5	Possibilidade de incluir *checklists*
F6	Ferramenta de uso *web* (necessita de acesso a *internet*)
F7	Presença da linguagem Markdown para estilizar
F8	Necessidade de autenticação para visualizar/editar artefatos
F9	Possui automatização
F10	Possibilidade de *upload* de arquivos como *pdfs* e imagens
F11	Presença de filtros customizáveis
F12	Presença de campo de pesquisa
F13	Criação e customização de *labels*

A lista geral de funcionalidades citadas na tabela 1 foram os exemplos de funcionalidades existentes nas ferramentas foco de estudo deste trabalho e apontadas pelos entrevistados durante as conversas.

Table 2. Comparação entre ferramentas com suas funcionalidades

Ferramenta	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13
Github	x	x	x	x	x	x	x	x	x	x		x	x
Asana	x	x		x	x	x		x		x		x	x
Trello	x	x		x	x	x	x	x		x		x	x
Quadro	x				x								
Planilha	x						x		x		x	x	

Realizando a análise da tabela 2, pode-se perceber que a ferramenta Github é a que mais possui funcionalidades em quantidade (doze ocorrências de *x*). Em contrapartida, o quadro físico é a que menos possui (apenas duas ocorrências). O alerta aqui, é que essa tabela comparativa tem apenas o intuito de demonstrar as funcionalidades que cada ferramenta possui ou não. Pode acontecer de que para determinadas equipes e projetos alguma dessas funcionalidades seja irrelevante, portanto essa ferramenta é uma candidata ao uso da equipe.

6. Conclusões

Com o término das entrevistas e compilação dos resultados obtidos para apresentação neste trabalho, foi possível perceber que diferentes equipes possuem diferentes realidades, além de suas diferentes demandas do dia a dia. Mesmo algumas delas utilizando as mesmas ferramentas, apontaram limitações e pontos de melhoria distintos, com base nos seus anseios e necessidades diárias.

Um ponto que chamou atenção, em todas as ferramentas analisadas, foi a nota recebida quando perguntado sobre se a ferramenta fornece métricas para a equipe; e, se fornece, se a faz de forma adequada. Todas as ferramentas receberam notas baixíssimas nesse quesito, pois nenhuma realmente gera um relatório apontando ocorrências, sejam elas quantidade de tarefas criadas, fechadas, reabertas, versões com muitas tarefas de alta prioridade, entre outras. Inclusive, sobre este caso, uma colega do Laboratório está fazendo de trabalho de conclusão de curso, um *plugin* para o Github que permite disponibilizar tais métricas. Hoje, o que as equipes fazem para ter insumos e gerenciar suas produções, é fazer uso dos filtros das ferramentas e com base neles, contar as ocorrências e colocá-las numa planilha.

Por fim, este trabalho teve um bom propósito de entender como as equipes ágeis do projeto e-SUS AB atuam no seu dia a dia interagindo com as ferramentas de *bug* e *tracking report*. Problemas foram elencados e melhorias foram discutidas e sugeridas. A centralização, automatização e facilidade no uso das ferramentas são o ponto número um para seu uso, seguidos de bons filtros e funcionalidades que ajudam nas caracterizações das tarefas reportadas pelos usuários. A partir da análise dos resultados, não se pode dizer que existe uma melhor ferramenta ou não. Diferentes equipes, possuem diferentes realidades, e têm diferentes demandas. Ainda que fazendo uso de uma mesma ferramenta, elas apontaram limitações e melhorias distintas. Portanto, o que há são várias ferramentas e, antes de fazer uso delas, deve-se analisar as funcionalidades presentes e verificar se elas se encaixam nos requisitos necessários para o projeto em questão.

7. Trabalhos futuros

- Realizar esta mesma pesquisa em outra empresa ou organização. Assim, se poderá ter uma ideia de impacto tendo em vista outro projeto e outra cultura. Seria relevante fazê-la para acompanhar mais de uma realidade no uso dessas ferramentas, e até adicionar novos sistemas além dos usados no Laboratório Bridge;
- Propor as melhorias elencadas para as equipes diretamente para as ferramentas utilizadas, juntamente das dificuldades e limitações que elas possuem. Assim, talvez haja a possibilidade das empresas implementarem essas funcionalidades;
- Criar uma ferramenta nova e implementá-la com base em tudo apontado pelos entrevistados. Essa ferramenta terá todas as funcionalidades embasadas nas respostas dos usuários, suprindo suas necessidades atuais.

References

Unito. `https://unito.io/`.

Blair, S. (2004). A guide to evaluating a bug tracking system. *Retrieved from Scribd. com: http://www. scribd. com/doc/7046090/A-Guide-to-Evaluating-a-Bug-Tracking-System*.

Kitchenham, B. (2004). Procedures for performing systematic reviews. *Keele, UK, Keele University*, 33(2004):1–26.

Prodanov, Cleber Cristiano; Freitas, E. C. d. (2013). *Metodologia do Trabalho Científico: Métodos e Técnicas da Pesquisa e do Trabalho Acadêmico - 2ª Edição*. Editora Feevale.

Wazlawick, R. (2013). *Engenharia de software: conceitos e práticas*, volume 1. Elsevier Brasil.