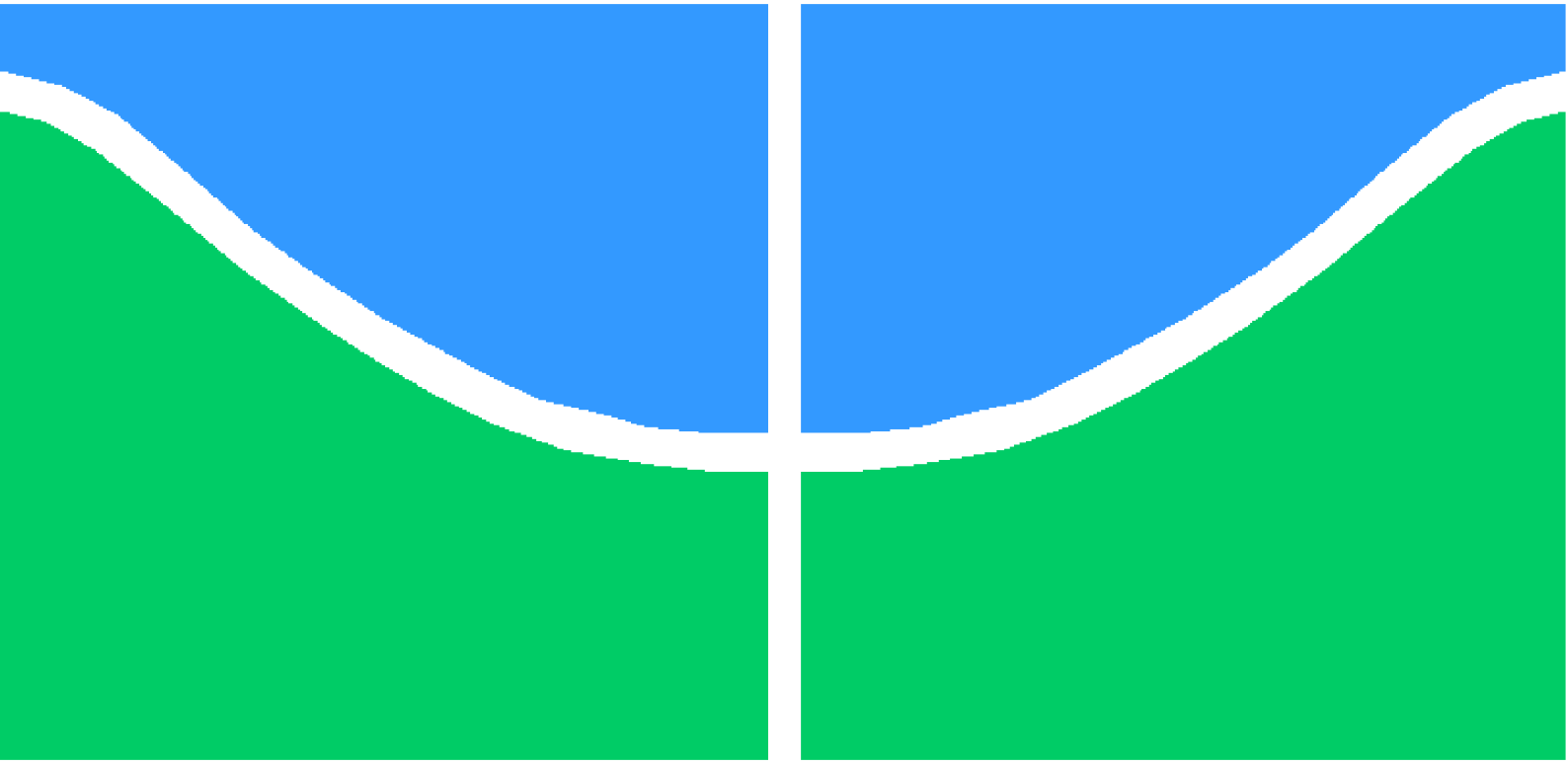


Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Engenharia de Software

Mezuro, Evolução de Software Livre: Da Arquitetura à Experiência do Usuário

Autor: Renan Costa Filgueiras
Vinícius Vieira Meneses
Orientador: Prof. Dr. Paulo Roberto Miranda Meirelles

Brasília, DF
2014

Renan Costa Filgueiras
Vinícius Vieira Meneses

Mezuro, Evolução de Software Livre: Da Arquitetura à Experiência do Usuário

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

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dr. Paulo Roberto Miranda Meirelles

Brasília, DF
2014

Renan Costa Filgueiras
Vinícius Vieira Meneses
Mezuro, Evolução de Software Livre: Da Arquitetura à Experiência do Usuário/ Renan Costa Filgueiras
Vinícius Vieira Meneses. – Brasília, DF, 2014-
128 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Paulo Roberto Miranda Meirelles

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2014.

1. Evolução de Software. 2. Experiência do Usuário. I. Prof. Dr. Paulo Roberto Miranda Meirelles. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Mezuro, Evolução de Software Livre: Da Arquitetura à Experiência do Usuário

CDU 02:141:005.6

Renan Costa Filgueiras
Vinícius Vieira Meneses

Mezuro, Evolução de Software Livre: Da Arquitetura à Experiência do Usuário

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

Trabalho aprovado. Brasília, DF, :

Prof. Dr. Paulo Roberto Miranda Meirelles
Orientador

Prof. Dr. Maurício Serrano
Convidado 1

Msc. Ana Paula Oliveira dos Santos
Convidado 2

Brasília, DF
2014

Agradecimentos

Primeiramente agradeço a Deus por ter permitido seguir trabalhando com saúde e força para superar as dificuldades.

Ao Prof. Dr. Paulo Meirelles pela oportunidade e apoio na elaboração deste trabalho.

Aos integrantes dos laboratórios LAPPIS e ao CCSL-IME-USP por proporcionar um ambiente criativo, amigável e colaborativo.

0.1 Renan Costa Filgueiras

Aos meus pais, Sandra Regina da Costa Filgueiras e Elias Costa Filgueiras, pelo apoio, amor, força e auxílio em me guiar sempre pelo melhor caminho.

Aos meus familiares, especialmente meu irmão Rafael Costa Filgueiras, minha prima Dyelly Costa Filgueiras e meu irmão de coração Newton da Silva Miranda Júnior por me apoiar nos dias difíceis e sempre me motivar a continuar.

Pelos amigos de curso que auxiliaram, motivaram e dividiram momentos de dificuldades e alegrias durante essa caminhada.

0.2 Vinícius Vieira Meneses

Aos meus pais e à minha irmã, Antônia Vieira, Ricardo Mario e Victória Vieira, e demais familiares pela confiança, incentivo e apoio incondicional.

Pelos amigos de curso que auxiliaram, motivaram e dividiram momentos de dificuldades e alegrias durante essa caminhada.

Resumo

Este trabalho apresenta um estudo e as colaborações na evolução de uma plataforma para monitoramento de códigos-fonte chamada Mezuro. Essa plataforma é desenvolvida através de um projeto de software livre. Em sua concepção, foi pensada como um plugin de uma plataforma de redes sociais, o Noosfero. Com sua evolução, ou seja, sucessivas alterações e com o aumento em tamanho e funcionalidades, aumentou-se também a complexidade do Mezuro, assim como a dificuldade em mantê-la. A equipe de desenvolvimento decidiu então por evoluir essa ferramenta para um aplicação independente. Neste trabalho de conclusão de curso, discutimos as principais razões e motivações para a evolução dessa plataforma, assim como os impactos em sua arquitetura e nos seus requisitos de qualidade. Apresentamos também um relato das nossas colaborações nesse projeto de software livre.

Palavras-chaves: software livre. evolução de software. métricas de código-fonte, usabilidade, técnicas de usabilidade, qualidade de software.

Abstract

This work presents a study and our collaboration on a source code monitoring platform called Mezuro. This platform is developed through an free software project. At the first moment, It was designed as Noosfero plugin, a social networking platform. However, with successive changes, increase of size, and features, Mezuro also have increased its complexity. The Mezuro development team have had problems to control its maintainability. In this degree monograph, we are discussing the main reasons and motivations to migrate Mezuro to a standalone platform, according to the Mezuro development analysis, as well as we discuss the impacts of that change on Mezuro architecture and its quality requirements. We also present a report of our collaboration on that free software project.

Key-words: free software. software evolution. source code metrics. usability. usability techniques. software quality.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

AGPL	GNU Affero General Public License
DCU	Design Centrado no Usuário
DRY	Don't Repeat Yourself
DSDM	Dynamic software development method
FGA	Faculdade Gama
GPL	General Public License
HTTP	Hipertext Transfer
IBM	International Business Machines
IDE	Integrated Development Environment
ISO	International Organization for Standardization
JaBUTi	Java Bytecode Understanding and Testing
JSON	JavaScript Object Notation
LGPL	GNU Lesser General Public License
LOC	Lines of Code
MA	Métodos ágeis
MVC	Model View Controller
QualiPSo	Quality Plataform for Open Source
PSSUQ	Psychometric Evaluation Of The Post-Study System Usability Questionnaire
Rails	Ruby on Rails
RITE	Rapid Iterative Testing and Evaluation
RPC	Remote Procedure Call
SOAP	Simple Object Access Protocol
TCC	Trabalho de Conclusão de Curso

UnB	Universidade de Brasília
USP	Universidade de São Paulo
UX	User eXperience
XML	Extensible Markup Language
XP	Extreme programming

Sumário

1 Introdução

Atualmente, as tecnologias da informação exercem cada vez mais influência na sociedade, seja na interação entre pessoas, ou nas relações que empresas possuem com o mercado. Nesse sentido, sistemas de software tem recebido mais atenção, dado que cada vez mais podem determinar o sucesso ou fracasso nessas relações.

Por volta da década de 60, o software era comercializado juntamente com hardware. O código-fonte era disponibilizado junto com o software e muitos usuários compartilhavam código e informações. O software era dito livre até a década de 70, quando a International Business Machines (IBM) decidiu comercializar seus programas separados do hardware. Com isso as restrições de acesso ao código-fonte se tornaram comuns, assim seu compartilhamento se tornava cada vez mais escasso, surgiram assim os sistemas de software proprietários.

Durante a década de 80, porém, o contexto de software livre que permeava o início do software ressurgiu, em suma, por iniciativa Richard Stallman. Em 1983, após uma experiência negativa com softwares proprietários, ele deu origem ao Projeto GNU. Stallman objetivava criar um mecanismo para garantir direitos de cópia, modificação e redistribuição de software. Com isso surgiu a Licença General Public License (GPL).

A definição de software é a mesma, seja ele proprietário ou livre. Sistemas de Software são constituídos por um conjunto de procedimentos, dados, possível documentação, e satisfazem necessidades específicas de determinados usuários. Além da definição, os dois tipos de software mencionados aqui possuem também a necessidade de serem mantidos e evoluídos.

Um exemplo de software que passa por um processo de evolução é a plataforma Mezuro, uma ferramenta para monitorar código-fonte. O Mezuro é desenvolvido como um projeto de software livre, que a princípio era um *plugin* de uma plataforma de desenvolvimento de redes sociais denominada Noosfero. Porém, o Mezuro cresceu em tamanho e complexidade, motivando a reescrita do seu código-fonte, transformando-o em uma plataforma independente do Noosfero, aumentando a consistência em relação ao seu propósito, que é análise e interpretação de código-fonte. Além disso, para tornar o Mezuro mais competitivo em relação às ferramentas semelhantes e ampliar o interesse de utilização por parte de usuários, decidiu-se evoluí-lo.

A necessidade de ampliar o interesse por parte do usuário dentro do projeto Mezuro, advém da crescente preocupação com a usabilidade que o desenvolvimento de software livre vem adquirindo nos últimos anos, ainda que essa preocupação esteja normalmente limitada à projetos de grande visibilidade, geralmente patrocinados por grandes

empresas (NICHOLS; TWIDALE, 2006). Ao lado da própria dificuldade em se mensurar objetivamente a usabilidade de um sistema, é comum em programas de software livre que haja pouco incentivo (ou interesse) nesse aspecto, dado que a prioridade do mesmo é a implementação das funcionalidades. Essa cultura leva o desenvolvedor a iniciar o projeto pelo código, deixando o *design* de interfaces em segundo plano (THOMAS, 2008).

A fim de evitar esse cenário dentro do projeto Mezuro, um ciclo de usabilidade foi integrado dentro do processo de evolução da plataforma. Para poder inserir uma maior preocupação com a usabilidade dentro de um projeto de desenvolvimento livre com práticas ágeis foi necessário um levantamento de técnicas de usabilidade ágeis para ter o menor impacto dentro do ciclo de vida do projeto, gerando assim, colaboradores com foco na usabilidade para evolução das camadas de *front-end* e *back-end*, bem como, dessa forma, disseminar o interesse da melhoria desse requisito não funcional – a usabilidade.

Um dos desafios inerentes ao desenvolvimento de software é a manutenção e aumento da qualidade do sistema desenvolvido. Muitos indícios da qualidade de um software são encontrados no seu código-fonte através de suas métricas. Porém, quanto maior a quantidade de métricas extraídas mais difícil é a leitura e interpretação das mesmas. Visando maximizar o poder de compreensão do usuário em relação as métricas ou dados apresentados, um dos aspectos referentes a interação do usuário com a ferramenta é a visualização de informações, que tem como objetivo exatamente auxiliar o usuário a interpretar os valores apresentados, por meio de representações gráficas e interativas apoiadas por computador (MARTINS, 2012).

1.1 Objetivos

1.1.1 Objetivos Gerais

Neste trabalho de conclusão de curso, há como principal objetivo colaborar com evolução da plataforma de monitoramento de código-fonte Mezuro, de forma que migre do cenário de ser um plugin para torna-se uma aplicação independente.

1.1.2 Objetivos Específicos

Como objetivos específicos, este trabalho visa:

1. Reescrever o código do *background*[1] do Mezuro (modelo e controladores);

2. Reescrever o código do *front-end* do Mezuro (apresentação);

[1] Sequência de passos ou processo que executa em paralelo e "por trás" da camada de visualização sem a intervenção do usuário

3. Verificar resultados da aplicação de uma técnica de usabilidade ágil em uma comunidade de software livre;

4. Comparar a nova arquitetura com a antiga;

5. Colaborar com a evolução do Mezuro, segundo os padrões de contribuição para projetos de Software Livre;

1.2 Metodologia

A pesquisa é utilizada para a melhoria da qualidade do contexto pesquisado através do novo conhecimento adquirido, assim nesse trabalho quanto a natureza da pesquisa foi abordado a pesquisa aplicada ou tecnológica. De acordo com Barros e Lehfeld 2000, a pesquisa aplicada possui como motivação a necessidade de gerar conhecimento para aplicação de seus resultados, com o objetivo de "contribuir para fins práticos, visando à solução mais ou menos imediata do problema encontrado na realidade". Já que um estopim já havia sido dado pelo trabalho de pesquisa fundamental realizado na dissertação de mestrado "Aplicação de práticas de usabilidade ágil em software livre" (SANTOS, 2012) e com esse conhecimento gerado resolveu-se aplicá-lo a fim de se obter uma melhoria na evolução do projeto Mezuro.

Quanto aos objetivos, a pesquisa exploratória vai em conjunto com a necessidade de se verificar um padrão e/ou ideias em relação a introdução de um ciclo de usabilidade dentro de um projeto de desenvolvimento de software livre com práticas ágeis, já que esse é um assunto com poucos estudos a respeito. Associado a pesquisa exploratória foi realizado uma revisão sistemática a fim de se verificar os padrões existentes que poderiam ser aplicados para inserir a perspectiva de usabilidade em um contexto de projetos ágeis e de software livre.

Quanto aos procedimentos foi realizado um estudo de caso, um dos focos principais motivado pela evolução da plataforma Mezuro, onde se poderia aplicar as ideias propostas. A pesquisa foi realizado em associação aos laboratórios da Universidade de São Paulo (USP) e UnB (Universidade de Brasília) Gama. Os resultados desse estudo forneceram dados qualitativos para a análise.

O primeiro passo para realização deste trabalho foi o contato com a equipe que mantém a plataforma Mezuro, que se deu através do professor Dr. Paulo Meirelles, um dos mantenedores. Como muitos projetos de software livre contam com desenvolvedores distantes geograficamente, com o Mezuro não foi diferente. Após o contato com o primeiro mantenedor, iniciou-se os pareamentos remotos, através de vídeo-conferências, para maior familiaridade com o código e funcionalidades fornecidas pela ferramenta.

Após a apresentação geral de todo o código e funcionalidades do Mezuro, iniciou-se o treinamento na ferramenta utilizada no desenvolvimento do Mezuro, o *Ruby on Rails*. Essa fase aconteceu simultaneamente ao desenvolvimento das funcionalidades reservadas a equipe da Universidade de Brasília.

Para o desenvolvimento dessas funcionalidades, foram utilizadas práticas ágeis como *pair programming*[2], divisão das iterações em *sprints*, definição das unidades de trabalho em *backlog*, reuniões de retrospectiva para análise dos resultados produzidos ao fim de uma *sprint*, além das reuniões diárias que satisfazem um dos princípios básicos da metodologia ágil que é a comunicação.

1.3 Organização do Trabalho

Em paralelo à interação e colaborações ao código do Mezuro, houve a escrita deste trabalho, que está organizado em capítulos. O Capítulo 2 apresenta os principais conceitos de software livre, métodos ágeis, padrões de software livre e processo de desenvolvimento de software. O Capítulo 3 trás os principais, dos diversos conceitos, de evolução de projeto de software divido em duas partes, evolução de software e evolução da arquitetura. No Capítulo 4 é introduzido a abordagem da experiência do usuário focada na usabilidade com aspectos de melhoria da interface e na visualização de software sendo essas requisitos não funcionais dentro do projeto. Finalmente, o Capítulo 5 apresenta com mais detalhes a plataforma Mezuro, seu processo de desenvolvimento, fatores que motivaram a reescrita de seu código, assim como o processo e resultados gerados da aplicação de um ciclo de evolução de software focado na usabilidade tanto em *front-end* quanto em *back-end* na plataforma Mezuro. O Capítulo 6 é a conclusão do trabalho, trazendo limitações, trabalhos futuros e como a plataforma mezuro poderá continuar com o legado deixado pelo trabalho realizado.

[2] Dois programadores, piloto e co-piloto, trabalham juntos na mesma estação de trabalho. O piloto codifica, enquanto o co-piloto acompanha e auxilia o piloto na tomada das melhores decisões

2 Métodos Empíricos

2.1 Software Livre

O software, em um sistema computacional, é o componente que contém o conhecimento relacionado aos problemas a que a computação se aplica, contendo diversos aspectos que ultrapassam questões técnicas (MEIRELLES, 2013b), como por exemplo:

- O processo de desenvolvimento de software;
- Os mecanismos econômicos (gerenciais, competitivos, sociais, cognitivos, etc.) que regem esse desenvolvimento e seu uso;
- O relacionamento entre desenvolvedores, fornecedores e usuários de software;
- Os aspectos éticos e legais relacionados ao software;

O que define e diferencia o software livre de software proprietário vai do entendimento desses quatro pontos dentro do que é conhecido como *ecossistema do software livre* (MEIRELLES, 2013a). O princípio básico desse ecossistema refere-se ao direito dos usuários de executar, copiar, distribuir, estudar, alterar e melhorar o software. Estas estão definidas nas quatro liberdades para os usuários do software descritas no portal do GNU:

- A liberdade de executar o programa, para qualquer propósito (liberdade no. 0) [1];
- A liberdade de estudar como o programa funciona, e adaptá-lo para as suas necessidades (liberdade no. 1)[1,2];
- A liberdade de redistribuir cópias de modo que você possa ajudar ao seu próximo (liberdade no. 2)[1];
- A liberdade de aperfeiçoar o programa, e liberar os seus aperfeiçoamentos, de modo que toda a comunidade se beneficie (liberdade no. 3) [1,2].

Um software é considerado livre quando os usuários deste possuem todas essas liberdades. Assim, você deve ser livre para redistribuir cópias, sejam com ou sem modificações, sem custo associado ou cobrando uma taxa pela distribuição, para qualquer um em qualquer lugar. Além disso, a liberdade de redistribuir cópias deve incluir formas binárias ou executáveis do programa, assim como o código-fonte.

[1] Disponível em: <http://www.gnu.org/philosophy/free-sw.pt-br.html>, acessado em: Maio de 2014
[2] Aceso ao código-fonte é um pré-requisito para esta liberdade

Desenvolvedores ou colaboradores devem ser livres para fazer modificações e usá-las privativamente no seu trabalho ou lazer, sem mencionar a fonte do código. A liberdade de executar o programa significa a liberdade para qualquer tipo de pessoa, física ou jurídica, utilizar o software em qualquer tipo de sistema computacional, para qualquer tipo de trabalho ou finalidade. Uma condição necessária para softwares livres é a liberdade de acesso ao código-fonte gnu2013, principalmente no que diz respeito às liberdade número 1 e 3.

O GNU[3] quis dar liberdade aos utilizadores. Contudo precisava usar termos de distribuição que impediriam software livre de ser transformado em software proprietário. O método usado foi chamado de copyleft. Copyleft é um método geral para fazer um software livre e exige que todas as versões modificadas e estendidas do programa sejam também livres. Ele utiliza a lei de direitos autorais, mas veio para servir como oposto de sua finalidade usual: ao invés de um meio de privatizar o software, torna-se um meio de manter o software livre (STALLMAN; GAY, 2009).

2.2 Processo de Desenvolvimento de Software Livre

O aspecto mais importante de um software livre, sob a perspectiva da Engenharia de Software é o seu processo de desenvolvimento. Um projeto de software livre começa quando um desenvolvedor individual ou uma organização decidem tornar um projeto de software acessível ao público. Seu código-fonte é licenciado de forma a permitir seu acesso e alterações subsequentes por qualquer pessoa. Tipicamente, qualquer pessoa pode contribuir com o desenvolvimento, mas mantenedores ou líderes decidem quais contribuições serão incorporadas à *release* oficial. Não é uma regra, mas projetos de software livre, muitas vezes, recebem colaboração de pessoas geograficamente distantes que se organizam ao redor de um ou mais líderes (CORBUCCI, 2011).

Há características presentes no software livre que, a princípio, tornam incompatível a aplicação de métodos ágeis em seu desenvolvimento. Entre essas características estão a distância entre os desenvolvedores e a diversidade entre suas culturas, que dificultam a comunicação, um dos principais valores dos métodos ágeis. Entretanto, o sucesso resultante de alguns projetos de software livre, como é o caso do Kernel do Linux [4], fizeram surgir estudos com foco na união dessas duas vertentes.

Analisando um pouco melhor projetos de software livre, é possível notar que esses compartilham princípios e valores presentes no manifesto ágil [5]. Adaptação a mudanças, trabalhar com *feedback* contínuo, entregar funcionalidades reais, respeitar colaboradores

[3] GNU é um acrônimo para Gnu is Not Unix, além de ser o nome de um mamífero escolhido por Richard Stallman para batizar um sistema operacional completamente livre

[4] <https://www.kernel.org/>

[5] <http://agilemanifesto.org/>

e usuários e enfrentar desafios, são qualidades esperadas em desenvolvedores que utilizam métodos ágeis e são naturalmente encontradas em projetos de software livre.

Num trabalho realizado, Corbucci (2011) analisa semelhanças entre projetos de software livre e métodos ágeis, através de uma relação entre os quatro valores enunciados no manifesto ágil e práticas realizadas em projetos livres.

Conceitualmente, os valores semelhantes são:

- Indivíduos e interações são mais importantes que processos e ferramentas.
- Software em funcionamento é mais importante que documentação abrangente.
- Colaboração com o cliente (usuários) é mais importante que negociação de contratos.
- Responder às mudanças é mais importante que seguir um plano.

Além disso, várias práticas disseminadas pelas metodologias ágeis são usadas no dia-a-dia dos desenvolvedores e equipes das comunidades de software livre (CORBUCCI, 2011):

- Código compartilhado (coletivo);
- Projeto simples;
- Repositório único de código;
- Integração contínua;
- Código e teste;
- Desenvolvimento dirigido por testes, e
- Refatoração.

Observar e entender esses aspectos nos projetos de software livre tornam-se relevantes à medida que muitos projetos de software livre não vão além dos estágios iniciais e muitos acabam sendo abandonados antes de produzir resultados razoáveis. Isso sugere que, mesmo com o sucesso de alguns projetos de software livre, as comunidades, com ou sem a participação de empresas, podem avançar no acompanhamento do desenvolvimento dos projetos de software livre que participam. Olhar o processo de desenvolvimento de software livre do ponto de vista da Engenharia de Software e as possíveis sinergias com os métodos ágeis podem contribuir para um melhor rendimento dessa disposição na criação e colaboração em torno de projetos de software livre (MEIRELLES, 2013b).

Na prática, dentro do processo de desenvolvimento de software livre, após lançar uma versão inicial e divulgar o projeto, os usuários interessados começam a usar o software livre em questão. De acordo com Eric Raymond,"bons programas nascerem de necessidades pessoais", esses usuários podem também ser desenvolvedores, que irão colaborar com o projeto a fim de atenderem às suas próprias necessidades. Destacando a colaboração no código-fonte, essas melhorias são enviadas aos mantenedores do projeto como *patches*, ou seja, arquivos que contém as modificações no código e que serão analisados pelos mantenedores que, caso concordem com a mudança e com a sua implementação em si, irão aplicá-las ao repositório oficial do projeto. Portanto, mesmo que em projetos maiores outros aspectos sejam levados em consideração ou sigam processos mais burocrático de colaboração, a essência da colaboração técnica está no envio e análise de trechos de código-fonte (MEIRELLES, 2013b).

2.3 Padrões de Contribuição a Software Livre

Um software livre é concebido através de um processo de contribuições, o qual possui características especiais que promovem o surgimento de diversas práticas influenciadas por diversas forças. Tais práticas são conhecidas como padrões de software livre. Para simplificar, nesta seção o termo padrão está associado a padrões de software livre. Esses padrões estão organizados dentro de três grupos, que são o resultado de um trabalho de Antônio Terceiro, Rodrigo Rocha e Cristina Chaves (TERCEIRO RODRIGO ROCHA, 2013).

- **Padrões de seleção** auxiliam prováveis colaboradores a selecionar projetos adequados.

  - O primeiro padrão de seleção recomenda colaboradores novatos a "caminhar sobre terreno conhecido", ou seja, se deseja contribuir, começar por algum software que seja familiar, como por exemplo, um *browser*, editor de texto, IDE[6], ou qualquer outro software que já se utiliza.
  - O segundo padrão é similar ao primeiro, porém ao invés da ferramenta ser familiar, esse padrão recomenda que o colaborador tenha conhecimentos na linguagem ou tecnologia utilizada no projeto.
  - Já o terceiro padrão desse grupo motiva colaboradores a procurar por projetos de software livre que ofereçam funcionalidades atrativas, mesmo que o novo colaborador não tenha familiaridade com a ferramenta nem com a tecnologia utilizada em seu desenvolvimento.

[6] Integrated Development Environment, um ambiente integrado para desenvolvimento de software

O terceiro padrão é o que melhor se encaixa ao contexto desse trabalho, já que o Mezuro não era uma ferramenta utilizada no cotidiano e tão pouco familiar. Além disso, as tecnologias utilizadas em seu desenvolvimento não eram as de maior conhecimento. Entretanto, as funcionalidades providas por essa plataforma foi determinante para essa contribuição.

- **Padrões de envolvimento** lidam com os primeiros passos para que o colaborador se familiarize e se envolva com o projeto selecionado.

  - Entrar em contato com mantenedores para aprender sobre o contexto histórico e político no qual aquele projeto está inserido.
  - Realizar instalação e checar se todo o ambiente do projeto está corretamente configurado em um período limitado de tempo (máximo um dia)
  - Durante uma apresentação do sistema, por parte de algum mantenedor, interagir para se familiarizar melhor com funcionalidades e cenários presentes no sistema.
  - Avaliar o estado do sistema através de uma breve, mas intensa revisão de código. Isso ajuda a ter uma primeira impressão sobre a qualidade do código-fonte.
  - Através da leitura, avaliar a relevância da documentação em um período limitado de tempo.
  - Checar a lista de tarefas a serem feitas. Ela pode conter bons pontos de partida para começar uma contribuição.
  - Relacionado ao padrão mencionado acima, está o padrão que recomenda novos colaboradores iniciarem por tarefas mais fáceis. Começar uma tarefa e termina-la é importante para manter colaboradores motivados, e conforme ganharem mais experiencia e familiaridade com o software avançam para tarefas mais complexas.

  No contexto deste trabalho, muitos dos padrões desse grupo foram inseridos ao processo de contribuição. Por exemplo, o orientador deste trabalho é também mantenedor da plataforma Mezuro, assim como outros colaboradores da plataforma, auxiliaram durante o processo de envolvimento, apresentando funcionalidades e principais cenários do sistema, além de fornecer documentação necessária para o entendimento do histórico e contexto no qual o Mezuro está inserido.

- **Padrões de contribuição** documenta as melhores práticas para se contribuir com softwares livres. Os grupos anteriores tratavam como iniciar e se familiarizar com um projeto de software livre. Esse grupo, por sua vez, contém padrões que auxiliam o

fornecimento de insumos para projetos de software livre, seja código-fonte ou outros artefatos presentes no processo de desenvolvimento.

- Uma boa contribuição para projetos de software em geral, é a escrita de documentação. O código-fonte muitas vezes não é o suficiente para que todos os envolvidos entendam o andamento do projeto, pois apesar de promoverem o software não possuem conhecimento técnico suficiente. Além disso, documentação do projeto auxilia na manutenção e evolução do produto.
- Muitos softwares livres não suportam o idioma de diversos colaboradores. Um bom ponto de partida seria a internacionalização do sistema, incluindo a própria linguagem no sistema.
- Reportar *bugs* eficientemente, pois é comum que colaboradores identifiquem bugs mas ao reporta-los não são claros com respeito ao seu contexto, dificultado sua correção.
- Utilizar a versão correta para tarefas. Durante o desenvolvimento de software há diferentes versões, onde há no mínimo uma versão estável e uma versão desenvolvimento. É recomendado utilizar a versão estável para reportar *bugs* e a versão de desenvolvimento para implementar novas funcionalidades e tudo que não está relacionado com correção de defeitos existentes.
- Separar alterações não relacionadas. Se tratando de sistemas de controle de versão[7] há uma ação conhecida como *commit*, onde as alterações realizadas são agrupadas e gravadas. É recomendado que num mesmo *commit* as alterações sejam relacionadas.
- Mensagens de *commit* explicativas para facilitar o entendimento e identificação do que foi desenvolvido ou alterado para o restante dos colaboradores.
- Documentar as próprias modificações. Desenvolvedores, geralmente, alteram o código, corrigem *bugs*, adicionam novas funcionalidades, mas não atualizam a documentação, a qual se torna desatualizada. Por isso é recomendado documentar as alterações antes de submete-as ao repositório.
- Manter-se atualizado com o estado atual do projeto, ajudando a evitar duplicação de esforços e identificar oportunidades de colaboração. Isso é importante pois um projeto de software livre é um esforço coletivo, mas às vezes é difícil coordenar o esforço de pessoas com diferentes horários e prioridades.

Em resumo, esses padrões não são regras, apenas indicam um bom caminho para contribuições. Por exemplo, o software tratado neste trabalho, o autor principal do mesmo, não era familiar no início do processo de contribuição para a colaboração da evolução de um software livre.

[7] SCM - Source Code Management - GIT, SVN, Baazar, Mercurial, entre outros

2.4 Métodos Ágeis

Métodos ágeis (AM) é uma coleção de metodologias baseada na prática para modelagem efetiva de sistemas baseados em software. É uma filosofia onde muitas metodologias se encaixam. As metodologias ágeis aplicam uma coleção de práticas, guiadas por princípios e valores que podem ser aplicados por profissionais de software no dia a dia (ÁGIL, 2001)

2.4.1 Metodologia Ágil

A expressão "Metodologias Ágeis"tornou-se popular em 2001 quando dezessete especialistas em processos de desenvolvimento de software representando diversos métodos como Scrum (SCHWABER; BEEDLE, 2002), Extreme Programming (XP) e outros, estabeleceram princípios comuns compartilhados por todos esses métodos. Foi então criada a Aliança Ágil e o estabelecimento do Manifesto Ágil (ÁGIL, 2001) Os conceitos chave do Manifesto Ágil são: **Indivíduos e interações** ao invés de processos e ferra- mentas. **Software executável** ao invés de documentação. **Colaboração do cliente** ao invés de negociação de contratos. **Respostas rápidas a mudanças** ao invés de seguir planos. O Manifesto Ágil não discrimina processos, ferramentas, documentação, negociação de contratos ou o planejamento, mas simplesmente mostra que eles têm importância secundária quando comparado com os indivíduos, interações, software executável, participação do cliente e *feedback* rápido a mudanças e alterações.

Tory Dyba (2008) em uma revisão sistemática elenca seis dos principais métodos ágeis presente em 2008 e trás uma breve descrição de cada um, o que ajuda a observar características das comunidades ágeis já que os principais especialistas desses métodos ajudaram a escrever o manifesto ágil. Eles são:

1. **Crystal Clear** centra-se na comunicação de pequenas equipes de desenvolvimento de software que não possui ciclo de vida crítico. Seu desenvolvimento tem sete características: entrega frequente, melhoria reflexivo, comunicação osmótica, segurança pessoal, foco, fácil acesso a usuários experientes e os requisitos para o ambiente técnico.

2. **Dynamic software development method (DSDM)** divide projetos em três fases: pré-projeto, o ciclo de vida do projeto e pós projeto. Nove princípios subjacentes ao DSDM: o envolvimento do usuário, capacitando da equipe do projeto, a entrega frequente, atender às necessidades de negócios atuais, desenvolvimento iterativo e incremental, permitir mudanças, o escopo de alto nível a ser fixados antes do início do projeto, testar todo o ciclo de vida e eficiente e eficaz comunicação.

3. **Feature-driven development** combina desenvolvimento *model-driven* e ágil, com ênfase em modelo inicial de objeto, a divisão do trabalho em funções e design iterativo para cada recurso. Afirma ser adequado para o desenvolvimento de sistemas críticos. Uma iteração de um recurso é composto de duas fases: concepção e desenvolvimento

4. **Lean software development** uma adaptação de princípios de produção de carne magra e, em particular, o sistema de produção Toyota para desenvolvimento de software. Consiste em sete princípios: eliminar o desperdício, aumentar a aprendizagem, decidir o mais tarde possível, entregar o mais rápido possível, capacitar a equipe, construir integridade, e ver o todo

5. **Scrum** centra-se na gestão de projetos em situações em que é difícil planejar o futuro, com mecanismos de " controle de processos empíricos ", onde *loops* de *feedback* constituem o elemento central. Software é desenvolvido por uma equipa de auto-organização em incrementos (chamado *sprints*), começando com o planejamento e terminando com uma retrospectiva. Recursos a serem implementadas no sistema estão registrados em um *backlog* . Em seguida, o proprietário do produto decide quais itens do *backlog* deve ser desenvolvido da seguinte *sprint*. Coordenar membros da equipe de seu trabalho em uma reunião diária de *stand-up*. Um membro da equipe, o *scrum master*, é responsável pela resolução de problemas que impedem a equipe de trabalho de forma eficaz.

6. **Extreme programming (XP)** concentra-se nas melhores práticas para o desenvolvimento. Consiste em doze práticas: o jogo de planejamento, pequenos lançamentos, metáfora, design simples, testes, refatoração, programação em pares, propriedade coletiva, integração contínua, 40 h semana, os clientes no local, e os padrões de codificação. A revista XP2 consiste nas seguintes "práticas primárias": sentar-se juntos, toda a equipe, trabalho informativo, o trabalho energizado, programação em pares, histórias, ciclo semanal, ciclo trimestral, slack, construção de 10 minutos, integração contínua, testes primeiro que programação e design incremental. Há também 11 "corollary practices".

Métodos ágeis, extraído da revisão sistemática (DYBÅ; DINGSØYR, 2008)

Alguns desses métodos ainda são amplamente utilizados, discutidos e contribuem para o entendimento do desenvolvimento ágil, pois definem práticas que ajudam a instigar a vivência ágil dentro da equipe. É o método escolhido que caracteriza as práticas a serem seguidas.

2.4.2 Princípios Ágeis

O manifesto ágil define 12 princípios que devem ser seguidos (ÁGIL, 2001)

1. Nossa maior prioridade é satisfazer o cliente, através da entrega adiantada e contínua de software de valor.
2. Aceitar mudanças de requisitos, mesmo no fim do desenvolvimento. Processos ágeis se adéquam a mudanças, para que o cliente possa tirar vantagens competitivas.
3. Entregar software funcionando com frequência, na escala de semanas até meses, com preferência aos períodos mais curtos.
4. Pessoas relacionadas a negócios e desenvolvedores devem trabalhar em conjunto e diariamente, durante todo o curso do projeto.
5. Construir projetos ao redor de indivíduos motivados. Dando a eles o ambiente e suporte necessário, e confiar que farão seu trabalho.
6. O Método mais eficiente e eficaz de transmitir informações para, e por dentro de um time de desenvolvimento, é através de uma conversa cara a cara.
7. Software funcional é a medida primária de progresso.
8. Processos ágeis promovem um ambiente sustentável. Os patrocinadores, desenvolvedores e usuários, devem ser capazes de manter indefinidamente, passos constantes.
9. Contínua atenção à excelência técnica e bom design, aumenta a agilidade.
10. Simplicidade: a arte de maximizar a quantidade de trabalho que não precisou ser feito.
11. As melhores arquiteturas, requisitos e designs emergem de times auto-organizáveis.
12. Em intervalos regulares, o time reflete em como ficar mais efetivo, então, se ajustam e otimizam seu comportamento de acordo.

Os princípios ágeis devem ser vividos pela equipe e é o principal motivador e base para o desenvolvimento ágil. As práticas ágeis adotadas serão de acordo com o método escolhido e ajudarão a obter os objetivos propostos pelo método, sempre levando em consideração os princípios ágeis definidos pelo manifesto ágil.

3 Evolução de Projetos de Software

Atualmente as tecnologias da informação exercem cada vez mais influência na sociedade, seja na interação entre pessoas, ou nas relações que empresas possuem com o mercado. Organizações que possuem parte dos seus lucros associados diretamente, ou não, a sistemas de software, precisam evolui-los, seja para adequá-los à mudanças no ambiente onde estão inseridos, ou para mantê-los competitivos frente aos concorrentes. Além desses fatores, quando os sistemas em questão são desenvolvidos como softwares livres, eles também precisam evoluir para que se mantenham sempre atrativos, motivando a comunidade estabelecida ao seu redor. Sistemas estagnados desmotivam usuários ou colaboradores, o que significa risco de perda de mercado ou enfraquecimento de um projeto de software livre, já que esses são feitos de colaboradores, como foi dito.

Por outro lado, a manutenção desses sistemas é difícil, consome bastante tempo e recursos. Tarefas como adicionar novas funcionalidades, suporte a novos dispositivos de hardware, correção de defeitos, entre outros, se tornam mais difíceis e complexas conforme o sistema cresce e envelhece (GODFREY; TU, 2000), surgindo adversidades. Os problemas de *design* vão além de algoritmos e estruturas de dados, onde a especificação da estrutura geral do sistema surge como um novo obstáculo (GARLAN; SHAW, 1993). Visando amenizar problemas como esse, a arquitetura de software tem recebido grande atenção desde a década passada, já que ela tem auxiliado na obtenção de ótimos resultados quanto ao atendimento de atributos de qualidade (BRAZ, 2009).

Este trabalho visa contribuir diretamente com um software livre, tratando a evolução do mesmo. Dessa forma, neste capítulo, apresentamos o que é evolução de software, assim como esse assunto é apresentado na literatura. Além disso, serão apresentados conceitos da arquitetura do ponto de vista da Engenharia de Software e aspectos relevantes deste processo durante a evolução de um sistema de software.

3.1 Evolução de Software

A evolução de software foi identificada pela primeira vez no final dos anos 60, embora não denominada evolução até 1969, quando Meir M. Lehman realizou um estudo com a IBM, com a ideia de melhorar a efetividade de programação dessa empresa. Apesar de não ter recebido tanta atenção e pouco impactado nas práticas de desenvolvimento dessa companhia , esse estudo fez surgir um novo campo de pesquisa, a evolução de software.

Muitas vezes, na literatura os termos manutenção e evolução de sistemas de software apresentam-se juntas, o que pode causar a falsa impressão que possuem o mesmo significado. Embora se refiram ao mesmo fenômeno, possuem ênfases diferentes. Manutenção é o ato de manter uma entidade num estado de reparo, capacidade ou disponibilidade, prevenindo-a contra falhas, mantendo a satisfação dos envolvidos ao longo do ciclo de vida do software. Já a evolução refere-se a um processo de mudança contínuo de um estado mais baixo, simples ou pior para um estado mais alto, mais complexo e melhor, refletindo a soma de todas as alterações implementadas no sistema.

Durante esses estudos, Lehman formulou as três primeiras, de um total de oito leis, conhecidas atualmente como leis de Lehman. O restante foi formulado em estudos posteriores, conforme a relevância desse campo aumentava. O conjunto dessas oito leis estão listadas abaixo:

Índice (Ano)	Nome	Descrição
1 (1974)	Mudança contínua	Um software deve ser continuamente adaptado, caso contrário se torna progressivamente menos satisfatório.
2 (1974)	Complexidade Crescente	À medida que um software é alterado, sua complexidade cresce, a menos que um trabalho seja feito para mantê-la ou diminuí-la.
3 (1974)	Auto-regulação	O processo de evolução de software é auto-regulado próximo à distribuição normal com relação às medidas dos atributos de produtos e processos.
4 (1978)	Conservação da estabilidade organizacional	A não ser que mecanismos de retro-alimentação tenham sido ajustados de maneira apropriada, a taxa media de atividade global efetiva num software em evolução tende a ser manter constante durante o tempo de vida do produto.
5 (1991)	Conservação da Familiaridade	De maneira geral, a taxa de crescimento incremental e taxa crescimento a longo prazo tende a declinar.
6 (1991)	Crescimento contínuo	O conteúdo funcional de um software deve ser continuamente aumentado durante seu tempo de vida para para manter a satisfação do usuário.
7 (1996)	Qualidade decrescente	A qualidade do software será entendida como declinante a menos que o software seja rigorosamente adaptado às mudanças no ambiente operacional.
8 (1971/96)	Sistema de Retro-alimentação	Processos de evolução de software são sistemas de retro-alimentação em múltiplos níveis, em múltiplos laços (*loops*) e envolvendo múltiplos agentes.

Tabela 1 – Leis de Lehman, extraído de (FERNANDEZ-RAMIL et al., 2008)

Ao contrário das engenharias tradicionais, a engenharia de software tem em mãos um produto abstrato e intangível, o que resulta em alguns desafios inerentes aos processo de desenvolvimento. A evolução de software busca amenizar ou solucionar alguns desses desafios (MENS et al., 2005), entre eles:

- Manter e melhorar a qualidade do software;
- Suportar evolução do modelo de desenvolvimento (não só código-fonte);
- Manter consistência entre artefatos relacionados;
- Integrar mudanças dentro do ciclo de desenvolvimento de software;
- Necessidades de bons sistemas de controle de versão;
- Integração e análise de dados de várias fontes (relatórios de erros, métricas, solicitações de mudança);

Quando inserida ou considerada nos processos de desenvolvimento, ela resulta numa excelente alternativa para evitar os sintomas do envelhecimento e inconsistências entre o próprio software e o ambiente onde está inserido (MENS et al., 2005). Dessa forma, argumentamos que o desenvolvimento de projetos de software livre têm colaborado para a produção de softwares de alta qualidade com grande número de funcionalidades. Um exemplo disso é o sistema operacional Linux, que nas últimas décadas, entre outro pontos, tem experimentado um grande sucesso comercial.

Em geral, sistemas desenvolvidos por meio de projetos de software livre tendem a crescer com o passar do tempo, após sucessivas *releases*. Esse comportamento sugere consistência com a sexta lei de Lehman, que se refere ao crescimento contínuo. Nesse sentido, além de um comportamento necessário para manter a satisfação do usuário, o crescimento contínuo de um software livre é importante para manter a motivação da comunidade estabelecida ao seu redor.

Por exemplo, para avaliar esse comportamento de contínuo crescimento em softwares livres, Godfrey e Tu (2000) realizaram pesquisas, do tipo estudo de caso, baseados no sistema operacional Linux. A Figura 1 mostra o crescimento do sistema operacional Linux desde sua primeira *release*, no ano de 1994. Desde então, ele é mantido por centenas de desenvolvedores que o desenvolvem em dois ramos paralelos: *stable releases* contendo as principais atualizações e correções de defeitos, e *development releases* com funcionalidades experimentais e porções de código não testado.

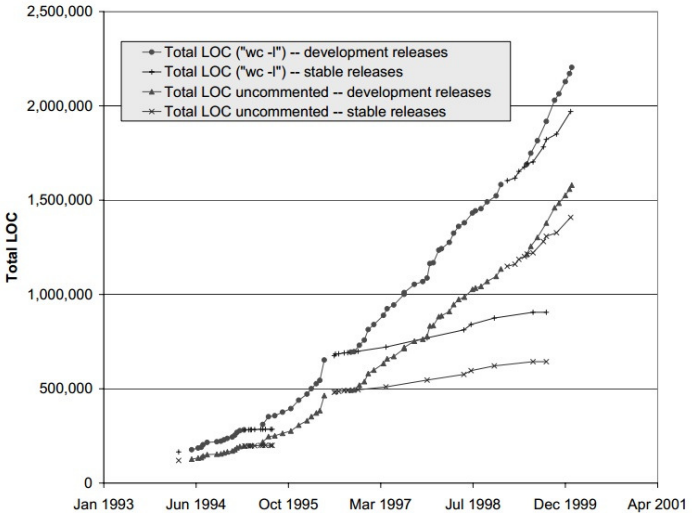


Figura 1 – Evolução do sistema operacional Linux, extraído de (GODFREY; TU, 2000)

Os dados presentes no gráfico, até o início dos anos 2000, vão de encontro à quinta lei de Lehman, citada na Tabela 1. No gráfico, o número de linhas de código (LOC) do *kernel* do sistema, possui taxa de crescimento positiva, enquanto a lei afirma que ao longo do tempo a taxa de crescimento tende a diminuir. Por outro lado, neste trabalho, não estamos tratando a evolução de software apenas do ponto de vista da inserção de novas funcionalidades, o que pode levar ao crescimento de número de linhas de código, como exemplificado acima. Estamos tratando a evolução de um software livre real, do ponto de vista da sua arquitetura, de acordo com as decisões julgadas pelo seus principais desenvolvedores, conforme descrito nas respostas ao questionário apresentado no Apêndice .1, para poderem evoluir o projeto de uma forma mais rápida e objetivando formação de uma comunidade de desenvolvedores.

3.2 Evolução da Arquitetura

3.2.1 Arquitetura de Software

Desde a primeira referência em um relatório técnico intitulado *Software Engineering Tecnhiques* (BUXTON; RANDELL, 1970), na década de 1970, diversos autores buscaram definir o termo arquitetura de software de software. Mary Shaw e David Garlan (SHAW; GARLAN, 1996), Philippe Kruchten, Grady Booch, Kurt Bittner, e Rich Reitman afirmam que: Arquitetura de Software engloba o conjunto de decisões significativas sobre a organização de um sistema de software incluindo: i) seleção de elementos estruturais e suas interfaces pelos quais um sistema é composto; ii) comportamento como especificado em colaboração entre esses elementos; iii) composição dos elementos estruturais e comportamentais dentro de um subsistema maior; iv) um estilo arquitetural que

orienta essa organização. Arquitetura de Software também envolve funcionalidade, usabilidade, flexibilidade, desempenho, reuso, compreensibilidade, restrições econômicas e tecnológicas, vantagens e desvantagens, além de preocupações estéticas.

Com o intuito de estabelecer um padrão sobre o que é e para que serve a arquitetura de software, a International Organization for Standardization (ISO) 1471 estabelece que Arquitetura de Software é a organização fundamental de um sistema incorporada em seus componentes, seus relacionamentos com o ambiente, e os princípios que conduzem seu design e evolução. Em suma, há três conceitos citados por todos os autores quando se trata de arquitetura de software (DIAS; VIEIRA, 2000):

- Elementos estruturais ou de software, também chamados de módulos ou componentes, são as abstrações responsáveis por representar as entidades que implementam funcionalidades especificadas.
- Interfaces ou relacionamentos, também chamados de conectores, são as abstrações responsáveis por representar as entidades que facilitam a comunicação entre os elementos de software.
- Organização ou configuração que consiste na forma como os elementos de software e conectores estão organizados.

Durante a especificação da arquitetura é importante ter a atenção nos relacionamentos entre seus elementos. Essas relações especificam a comunicação, o controle da informação e o comportamento do sistema. Consequentemente, essas relações impactam nos atributos de qualidade, sejam os percebidos pelos usuários, ou apenas pelos desenvolvedores (GERMOGLIO, 2010). Os atributos de qualidade são uma das principais preocupações da arquitetura. Eles representam a maneira que o sistema executará suas funcionalidades e são impostos pelos diversos envolvidos no sistema. Podem ser de três tipos:

- Atributos de produto ditam como o sistema irá se comportar. Exemplos clássicos são: desempenho, disponibilidade, manutenibilidade, escalabilidade, disponibilidade e portabilidade;
- Atributos organizacionais são padrões ou regras impostas por organizações envolvidas para satisfazer determinados requisitos.
- Atributos externos são leis impostas sobre softwares ou requisitos de interoperabilidade entre sistemas.

Para satisfazer esses atributos a arquitetura não pode ter suas estruturas definidas aleatoriamente. É necessário que o engenheiro de software opte por alternativas,

divida o sistema em elementos e defina seus relacionamentos para alcançar os atributos de qualidade desejados. Esse conjunto de decisões é conhecido por decisões arquiteturais. Qualquer software possui arquitetura, independente dela ser documentada ou projetada. Entretanto, uma arquitetura apenas implementada, ou seja, arquitetura sem projeto, não fornece benefícios ou vantagens que uma arquitetura projetada e bem documentada pode oferecer. Entre os benefícios da documentação da arquitetura estão: i) arquitetura como ferramenta de comunicação entre os participantes do projeto; ii) um método ou modelo para a análise antecipada do sistema a ser desenvolvido; iii) ferramenta de rastreabilidade entre os requisitos e os elementos que compõem o sistema, o que é de grande relevância dada a volatilidade dos requisitos durante o processo de desenvolvimento. Além da participação no desenvolvimento e o estudo da nova arquitetura do Mezuro, neste trabalho, também colaboraremos com documentação de sua arquitetura, e de modo que possa ser mantida de acordo com as práticas das comunidades de software livre, ou seja, também mantida e distribuída junto ao código no seu repositório.

3.2.2 O arcabouço Ruby on Rails

O Ruby on Rails é um *arcabouço*, disponível como software livre, criado em 2003 por David Heinemeier Hansson. Sua primeira versão foi lançada em 2004, e desde então seu desenvolvimento e utilização são cada vez maiores. Diversos programadores e empresas em todo mundo utilizam esse arcabouço para construírem suas aplicações. Entre as mais conhecidas estão o Twitter (nas primeiras versões), GitHub e Groupon. O Rails utiliza a linguagem de programação Ruby, criada no Japão em 1995 por Yukihiro "Matz"Matsumoto. A linguagem Ruby é interpretada, multiparadigma, com tipagem dinâmica e gerenciamento de memória automático. O Rails é basicamente uma biblioteca Ruby ou *gem* e é construído utilizando o padrão arquitetural **MVC**.

Um de seus fundamentos é facilitar o desenvolvimento. O Rails utiliza diversos princípios para orientá-lo do "modo certo"("Rails way"), possibilitando concentrar esforços no problema do cliente, poupando a equipe do esforço de organizar a estrutura da aplicação a qual é feita pelo arcabouço. Alguns princípios são:

- **DRY** - “Don’t Repeat Yourself” - Propõe que um mesmo trecho ou porção de conhecimento em um código deve possuir representação única no sistema, livre de redundância e repetições. Quando aplicado, esse princípio possibilita que uma alteração seja feita em um único local no código, evitando “bad smells” como código duplicado e facilitando a manutenibilidade do sistema.

- **Convenção ao invés de Configuração** (Convention over configuration)- Considerado um paradigma de design que visa diminuir o número de decisões que os desenvolvedores precisam tomar, ganhando simplicidade, sem perder simplicidade.

- **REST** - É um estilo arquitetural para aplicações web. O termo REST é um acrônimo para **RE**presentaional **S**tate **T**ransfer. Propõe princípios (não exclusivos ao REST) que definem como Web Standards como HTTP e URIs devem ser usados. Os princípios são:

  1. Todos os recursos devem ter um identificador. Um conceito comum para identificador na web é a URI;

  2. Utilize links (hipermídia) para referenciar recursos;

  3. Utilize os métodos padrão - São eles GET, POST, PUT, DELETE. Os métodos GET e PUT e DELETE são idempotentes, ou seja, há garantia de que podemos enviar a requisição novamente. Quando emitimos um GET, por exemplo, e não recebermos resposta, não saberemos se a requisição foi perdida ou a resposta que se perdeu. Mas nesse caso podemos simplesmente enviar a solicitação novamente;

  4. Múltipla representação de recursos - diversos formatos dos recursos para diferentes necessidades, como formatos XML, HTML. Isso faz com que seus recursos sejam consumidos não apenas pelo seu aplicativo, mas também por qualquer navegador web;

  5. Comunicação sem estado - Um servidor não deveria guardar o estado da comunicação de qualquer um dos clientes que se comunique com ele além de uma única requisição. A razão para isso é escalabilidade - o número de clientes que podem interagir com o servidor seria consideravelmente impactado se fosse preciso manter o estado do cliente;

Evolução do Ruby on Rails

O arcabouço Rails, desde seu lançamento, sofre frequentes alterações, e com isso novas versões são lançadas constantemente, conforme apresentado na Tabela 2. Muitos desenvolvedores enxergam essas constantes mudanças como um ponto negativo, ao passo que há incompatibilidade de algumas "gems"de uma versão para outra. Por outro lado, outros aprovam essa característica pois a cada atualização há melhora no produto, com adição de novos recursos, além de ser um incentivo para a implementação de testes, já que eles auxiliam a manter a integridade da aplicação a cada atualização.

Versão	Data
1.0	13/12/2005
1.2	19/01/2007
2.0	7/12/2007
2.1	01/06/2008
2.2	21/11/2008
2.3	16/03/2009
3.0	29/08/2010
3.1	31/08/2011
3.2	20/01/2012
4.0	25/06/2013

Tabela 2 – Versões do Rails

Entre os novos recursos oferecidos pela versão 4 do Rails, que é a versão usada no novo Mezuro, estão:

- Páginas mais rápidas através da utilização de Turbolinks. Ao invés de deixar o navegador recompilar o JavaScript e CSS entre cada mudança de página, a instância da página atual é mantida, substituindo apenas o conteúdo e o título.
- Suporte para a expiração de cache baseado em chave, que automatiza a invalidação do cache e deixa mais fácil a implementação de estruturas de cache sofisticadas.
- *Streaming* de vídeo ao vivo em conexões persistentes.
- Melhorias no ActiveRecord para aprimorar a consistência do escopo e da estrutura das queries.
- Padrões de segurança *locked-down*.
- *Threads* seguras por padrão e a eliminação da necessidade de configurar servidores com *thread*.

É importante discutir sobre a evolução deste arcabouço pois esse foi um dos motivos que motivaram a evolução da plataforma Mezuro. Isso para aproveitar os recursos mais novos disponíveis nas versões mais recentes, além de desfrutar de maior suporte da comunidade deste arcabouço como da linguagem utilizada por ele, a linguagem Ruby. Um questionário com questões relacionadas a evolução do Mezuro se encontra no Apêndice.1 deste documento e respostas de alguns dos colaboradores se encontram no Anexo.2.

Padrão Arquitetural MVC

O padrão Model View Controller (MVC) começou como um arcabouço desenvolvido por Trygve Reenskaug para a plataforma SmallTalk, no final dos anos 70. Desde então, ele exerce grande influência sobre diversos arcabouços que promovem interação com usuário, como é o caso do Ruby on Rails. O MVC visa separar a representação da informação da interação com o usuário. Para atingir esse objetivo são utilizados três "papéis", conforme apresentado do na Figura 2: (i) O modelo (*model*) que representa informações do domínio, como dados da aplicação, regras de negócio, lógica e funções; A visão (*view*) que são saídas de representação dos dados do modelo ao usuário. Um exemplo comum de visão é uma página HTML contendo dados presentes no modelo. O último papel, o controlador (*controller*) é responsável por receber requisições da visão, manipula-las, utilizando dados do modelo, e atualizar a visão para satisfazer as requisições do usuário.

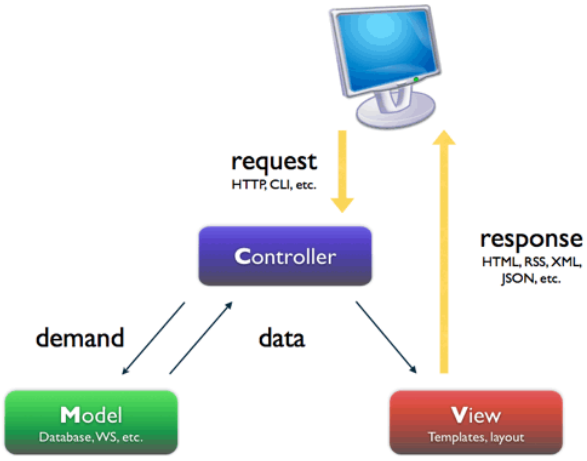


Figura 2 – Representação do padrão MVC

O exemplo abaixo ilustra um cenário de execução de uma funcionalidade com o padrão MVC.

Exemplo

Um usuário navega em um site de vendas de veículos. Ao pressionar o botão para visualizar determinado veículo, o navegador carrega a página de detalhes e a devolve ao usuário. Esse processo começa quando na VIEW o usuário pressiona o botão de visualização. A URL gerada é processada por um método da CONTROLLER de veículos. Esse método solicita ao MODEL o veículo correspondente a URL, então a VIEW é renderizada com os dados do veículo correspondente, obtidos pela CONTROLLER.

Arquitetura do Rails

Entre os elementos da arquitetura de software, há elementos estáticos e dinâmicos. Elementos estáticos definem as partes de um sistema e sua organização. Entre elementos estáticos estão: i) elementos de software; ii) elementos de dados; iii) elementos de hardware. As características do Rails estão distribuídas entre elementos, conforme ilustrado na

Figura 3. Os relacionamentos entre os elementos também estão inclusos, e também compõem o aspecto estático da arquitetura do sistema (GERMOGLIO, 2010). Já os elementos dinâmicos definem o comportamento do sistema e representa o sistema em execução. Nele estão incluídos processos, protocolos, módulos e classes que realizam comportamento.

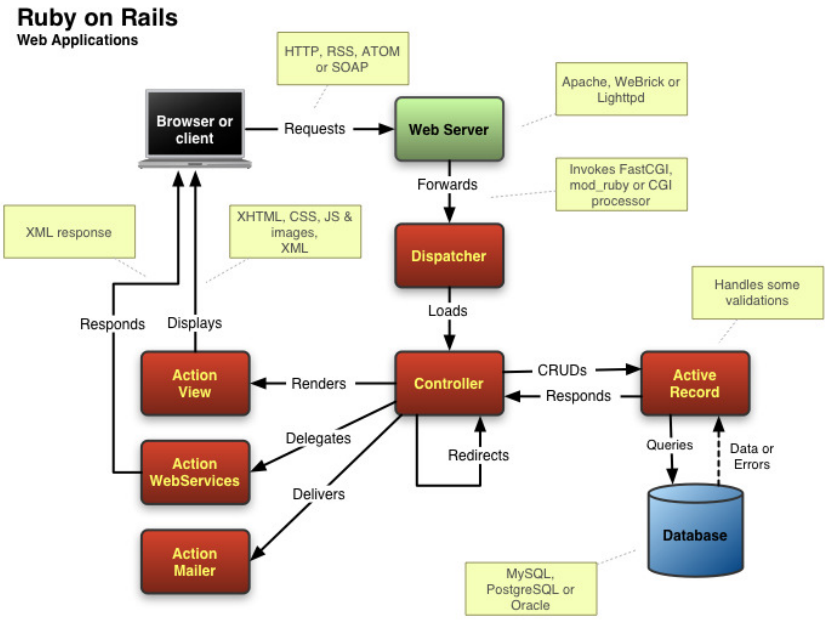


Figura 3 – Interação entre os componentes do Rails. Extraído de (MEJIA, 2011)

1. **Action Mailer** fornece a capacidade de criação de serviços de e-mail. Podendo enviar e-mails baseados em *templates* adaptáveis ou receber e processar um e-mail;

2. **Action Pack** é composto por:

   Action Controller é o componente que gerencia os *controllers* em uma aplicação Rails. Ele processa as requisições que chegam de uma aplicação Rails, recebe seus parâmetros e os envia para as ações pretendidas;

   Action View gerencia as *views* da aplicação. Isto é, as saídas HTML e XML por padrão. Gerencia a renderização de *templates* aninhados ou parciais , e inclui suporte embutido para AJAX;

3. **Active Record** é a base dos *models*. Ele fornece a independência de banco de dados e CRUD básico. Ele é utilizado para criar a representação, em orientação a objetos, dos dados presentes no banco de dados;

4. **Active Resource** gerencia conexões entre objetos de negócio e serviços web RESTful. Ele mapeia recursos baseados em web para objetos locais com lógica CRUD;

5. **Active Support** uma coleção de classes de utilidade e extensões da biblioteca padrão do Ruby que são utilizadas pelo Rails, tanto em seu núcleo quanto para suas aplicações;

6. **Railties** é o núcleo do código Rails e é ele quem constrói novas aplicações Rails e junta os diversos componentes;

4 Experiência do Usuário

Este trabalho visa contribuir diretamente com um software livre, tratando a evolução do mesmo. Dessa forma, neste capítulo, apresentamos os principais conceitos relacionados com esse tipo de software, passando pelas definições básicas, processos de desenvolvimento e os padrões para se contribuir com um projeto de software livre. Complementarmente, discutimos o que é evolução de software, tratando as Leis de Lehman e como está apresentada na literatura os estudos sobre a evolução de projeto de software livre.

4.1 Requisitos Não-Funcionais

Um requisito não funcional em engenharia software é aquele que descreve não o que o sistema fara, mas como ele fará. A avaliação dos requisitos não funcionais é feita, em parte, por meio de testes, enquanto que outra parte e avaliada de maneira subjetiva (FILHO, 2010). Tanto requisitos funcionais quanto não funcionais possuem importância no desenvolvimento de um sistema de software. Entretanto, os requisitos não funcionais, também denominados de características de qualidade, tem um papel relevante durante o desenvolvimento de um sistema, atuando como critérios na seleção ou composição de uma arquitetura de software, dentre as varias alternativas de projeto (FILHO, 2010).

4.2 Usabilidade

4.2.1 Terminologias

O termo usabilidade de modo geral pode ser escrito como a facilidade com a qual um equipamento ou programa pode ser usado. Esse termo dentro da computação foi diversas vezes refinado como nas ISO 9126, 12119, 9241, 14598 e, por fim, na ISO 25010, que define como uma medida pela qual um produto pode ser usado por usuários específicos para alcançar metas específicas com eficácia, eficiência e satisfação em um contexto específico de uso (SYSTEMS. . . , 2010).

A usabilidade não é uma qualidade intrínseca de um sistema, é dependente de um acordo entre as características de sua interface e as características de seus usuários na busca de determinados objetivos e situação de uso (CYBIS; BETIOL; FAUST, 2010).

Por esse motivo uma interface que pode ser considerada satisfatória para determinado grupo de usuários pode ser inviabilizada por outros, como usuários experientes *versus* novatos, além de uma percepção diferente dependendo do ambiente onde esse sis-

tema se encontra, um computador lento *versus* computador rápido. Dessa forma, podemos definir que a usabilidade é um acordo entre interface, usuário, tarefa e ambiente. Baseado nesta definição é que pautaremos nossas discussões e estudos de caso neste trabalho.

A necessidade de se garantir que sistemas e dispositivos estejam adaptados à maneira como o usuário pensa, comporta-se e trabalha, entra o conceito de ergonomia. Tal conceito surgiu logo após a II Guerra Mundial, como consequência do trabalho interdisciplinar realizado por diversos profissionais, tais como engenheiros, fisiologistas e psicólogos, durante a guerra (LIDA, 2005).

Há algumas definições formais para o termo "ergonomia", de acordo com a *Ergonomics Society*, a Associação Brasileira de Ergonomia e a *International Ergonomics Association*. Para este trabalho, adotamos a definição da Associação Brasileira de Ergonomia, que a conceitua como o estudo das interações das pessoas com a tecnologia, a organização e o ambiente, objetivando intervenções e projetos que visem melhorar, de forma integrada e não-dissociada, a segurança, o conforto, o bem-estar e a eficácia das atividades humanas (ERGONOMIA, 2013).

Neste contexto, a questão que norteia este trabalho é como pode-se avaliar, entender, verificar, observar a interface de uma aplicação em determinado contexto ou sistema? Para respondermos essa questão, mapeamos as definições de alguns especialistas em usabilidade e ergonomia, que estabeleceram critérios, regras e princípios para nortear essa necessidade.

- Jakob Nielsen, em seu livro *Usability Engineering* , propõe um conjunto de dez heurísticas de usabilidade (NIELSEN, 1994):
  - Viabilidade do estado do sistema;
  - Mapeamento entre o sistema e o mundo realizada;
  - Liberdade e controle ao usuário;
  - Consistência e padrões;
  - Prevenção de erros;
  - Reconhecer em vez de relembrar;
  - Flexibilidade e eficiência de uso;
  - Design estético e minimalista;
  - Suporte para o usuário reconhecer, diagnosticar e recuperar erros;
  - Ajuda e documentação.
- Ben Shneiderman, em seu livro *Designing The User Interface*, propõe, o que ele denominou de "oito regras de ouro" (SHNEIDERMAN; BEN, 2003)

  - Perseguir a consistência;
  - Fornecer atalhos;
  - Fornecer feedback informativos;
  - Marcar o final dos diálogos;
  - Fornecer prevenção e manipulação simples de erros;
  - Permitir o cancelamento das ações;
  - Fornecer controle e iniciativa ao usuário;
  - Reduzir a carga de memória de trabalho.

- Christian Bastien e Dominique Scapin definiram 8 critérios ergonômicos (BASTIEN; SCAPIN et al., 1993)

  - Condução;
  - Carga de trabalho;
  - Controle;
  - Adaptabilidade;
  - Gestão de erros;
  - Coerência;
  - Significado dos códigos;
  - Denominações e Compatibilidade.

Portanto, baseado nas heurísticas e critérios de ergonomia listados acima, Walter Cybis, no livro Ergonomia e Usabilidade, propôs uma tabela que relaciona todas essas definições, conforme apresentado na Tabela 3 (CYBIS; BETIOL; FAUST, 2010).

Condução	Qualidade da ajuda e da documentação Adequação ao aprendizado Apresentação do estado do sistema Convite Agrupamento e distinção por localização Agrupamento e distinção por formato Feedback imediato
Carga de trabalho	Legibilidade Brevidade das entradas individuais Concisão das apresentações individuais Ações mínimas Densidade informacional Design minimalista e estético
Controle	Ações explícitas Controle do usuário
Adaptabilidade	Flexibilidade Personalização Consideração da experiência do usuário
Gestão de erros	Proteção de erros Tolerância aos erros Qualidade das mensagens de erro Correção de erros
Coerência	Homogeneidade interna a uma aplicação Homogeneidade externa a plataforma
Significado dos códigos e denominações	Interface clara
Compatibilidade	Compatibilidade com o usuário Compatibilidade com a tarefa dos usuários Compatibilidade com a cultura dos usuários

Tabela 3 – Conjunto integrador de critérios, princípios, regras e heurísticas de ergonomia

4.2.2 Técnicas de Usabilidade Ágeis

Técnicas de usabilidade e desenvolvimento ágil têm muito em comum, principalmente o fato de que, muitas vezes, estão envolvidos no desenvolvimento do mesmo software. Apesar disso, tem havido pouca investigação ou discussão sobre a forma como os dois processos trabalham em conjunto e os resultados dessa parceria (FERREIRA; NOBLE; BIDDLE, 2007a).

Para tanto realizou-se uma revisão sistemática. A revisão sistemática é uma forma

de síntese das informações disponíveis em dado momento, sobre um problema específico, de forma objetiva e reproduzível, por meio de método científico. Ela tem como princípios gerais a exaustão na busca dos estudos analisados, a seleção justificada dos estudos por critérios de inclusão e exclusão explícitos e a avaliação da qualidade metodológica, bem como a quantificação do efeito dos tratamentos por meio de técnicas estatísticas (LIMAA; SOARESB; BACALTCHUKC, 2000).

O Objetivo da revisão sistemática era levantar as técnicas de usabilidade em projetos ágeis aplicadas no desenvolvimento de software livre. A busca foi realizada nas bibliotecas digitais Google Academic e Springer Link que possuem máquinas de busca com bom funcionamento e abrangência, além das bases de teses e dissertações da UnB e USP. Como resultado do trabalho levantou-se uma lista de técnicas e foram selecionadas 4 das que mais poderiam ser facilmente aplicadas em projetos de software livre e evidenciadas abaixo.

1. Santos e Kon (2009) propõe a aplicação do próprio processo de Design Centrado no Usuário (DCU) para levantamento dos usuários e do contexto de uso da metodologia. Os usuários do processo são membros de equipes de software livre, que desejam inserir práticas de usabilidade no processo de desenvolvimento. O contexto de uso são sistemas distribuídos geograficamente, com pessoas trabalhando colaborativamente. Os métodos podem ser aplicados tanto para o início de novos projetos, que tenham como usuários, pessoas não acostumadas a sistemas livres e métodos de usabilidade ou mesmo para a melhoria de sistemas existentes, com respeito à usabilidade.

2. Já Lee, Judge e McCrickard (2011) defendem a aplicação de definição de histórias de usuário de usabilidade dirigidas pela decisão do responsável por design realizando a prototipação dessas histórias e depois sendo validadas através de testes de usabilidade conforme Figura 31.

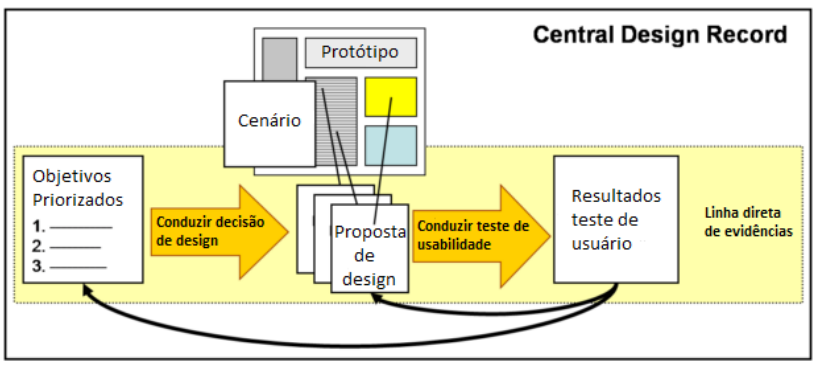


Figura 4 – Ciclo de atividades de usabilidade (LEE; JUDGE; MCCRICKARD, 2011)

As atividades de usabilidade seguem o modelo de ciclo de vida das outras atividades dentro do desenvolvimento ágil com a diferença que o ciclo de usabilidade tem sua iteração primeiro que as outras atividades de desenvolvimento conforme Figura 32:

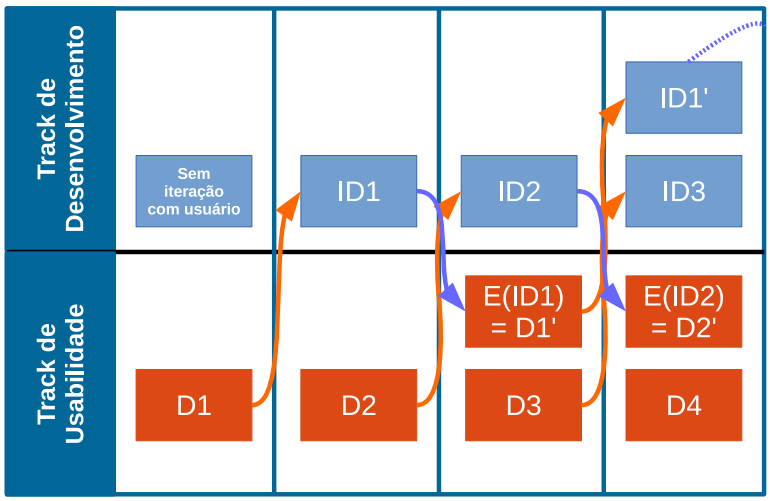


Figura 5 – Ciclo das tarefas de usabilidade x desenvolvimento (LEE; JUDGE; MCCRICKARD, 2011)

3. Santos (2012) em uma nova pesquisa aplicada propôs a aplicação das técnicas de usabilidade de acordo com as fases do DCU, conforme a descrição das técnicas de usabilidade das comunidades de métodos ágeis e de software livre. Para a fase do DCU Criar soluções de design, não foram identificadas propostas de adaptações, sendo portanto, descritas propostas para as fases: Identificar necessidades para design centrado em humano, Especificar contexto de uso, Especificar requisitos e Avaliar Designs conforme tabela 4.

Práticas de usabilidade para Software Livre	
Fases DCU	**Práticas de Usabilidade**
Identificar necessidades para design centrado em humano	Equipe-Núcleo como Donos do Produto Caminhos Completos Especialista-generalista
Especificar contexto de uso	Pouco design antecipado e distribuído
Especificar Requisitos	Definir metas de usabilidade automáticas
Avaliar designs	RITE (Rapid Iterative Testing and Evaluation) para desenvolvedores de software livre

Tabela 4 – Práticas de usabilidade no contexto de Software Livre

4. Moreno e Yagüe (2012) trás em seu trabalho a inserção da usabilidade por meio da criação de *User Stories* de usabilidade que denominam de *Usability Stories*. Tendo

que especificar os recursos funcionais de usabilidade que serão utilizados no projeto, o artigo trás um tabela que é necessária para definir as características dessa funcionalidade.

Status do Sistema	Para informar os usuários sobre a situação interna do sistema
Aviso	Para informar os usuários de qualquer ação com consequências importantes
Longa ação Feedback	Para informar aos usuários que o sistema está processando uma ação que vai demorar algum tempo para completar
Desfazer global	Para desfazer as ações do sistema em vários níveis
Abortar Operação	Para cancelar a execução de uma ação ou toda a aplicação
Abortar comando	Para cancelar a execução de uma tarefa em andamento
Voltar	Para voltar a um estado particular em uma sequência de execução de comando
Entrada de texto estruturada	Para ajudar a prevenir o usuário de cometer erros de entrada de dados
Execução Passo-a-Passo	Para ajudar os usuários a realizar tarefas que requerem diferentes passos com a entrada do usuário e tal entrada correta
Preferências	Para gravar as opções de cada usuário para usar as funções do sistema
Favoritos	Para gravar determinados locais de interesse para o usuário
Ajuda Multinível	Para fornecer diferentes níveis de ajuda para usuários diferentes

Tabela 5 – Mecanismos de usabilidade

Associado a essas características são definidas três maneiras pelas quais a incorporação de usabilidade influencia as histórias de usuário.

a) A adição de novas histórias para representar requisitos diretamente derivados usabilidade.

b) Adição ou modificação de tarefas em histórias de usuários existentes. Isto significa que algumas ações decorrentes de limitações de usabilidade devem ser realizadas por um usuário existente na história. Esta tarefa pode ser tão simples ou detalhados, conforme necessário.

c) Adição ou modificação de critérios de aceitação. Estes critérios de aceitação aparecem porque a funcionalidade da história de usuário precisa incluir algumas ações específicas para modificar o ambiente operacional.

Com isso os autores realizaram um mapeamento entre os mecanismos de usabilidade e as ações a serem tomadas finalizando a proposta de como a usabilidade deve ser abordada por equipes ágeis.

	Nova Tarefa	Modificar Tarefa	Novo Critério de Aceitação	Modificar Critério de Aceitação	Nova História de Usabilidade	Nova História de Usuário
Status do Sistema		X	X	X	X	
Aviso	X		X	X	X	
Longa ação		X	X			X
Abortar Operação		X	X			
Abortar comando	X	X	X			
Voltar	X	X	X			
Entrada de texto estruturada		X	X			
Execução Passo-a-Passo	X		X	X		
Preferências						X
Favoritos		X	X		X	
Ajuda Multinível		X	X		X	

Tabela 6 – Mapeamento entre os mecanismos de usabilidade e ações

O levantamento de técnicas de usabilidade da comunidade de métodos ágeis, técnicas de usabilidade ágil, e das técnicas de usabilidade da comunidade de software livre, foi realizado com o objetivo de refletir como técnicas de usabilidade poderiam ser aplicadas em comunidades de software livre. E essa lista possibilitou uma escolha da técnica que melhor se adéqua ao estudo de casa de acordo com as características de cada uma encontrada na revisão sistemática. O trabalho completo da revisão sistemática pode ser encontrado na seção .5 no apêndice do TCC.

4.2.3 Usabilidade em Software Livre

Para aplicação em ambientes distribuídos, abertos e colaborativos, como em comunidades de software livre, implica-se uma adaptação em métodos de usabilidade. Isso ocorre porque em comunidades de desenvolvimento de software livre, não se pode garantir a existência de um indivíduo especialista em usabilidade ou de uma equipe dedicada a essas atividades. A distância física de membros que contribuem com o projeto também dificulta a utilização de métodos de usabilidade que dependem de comunicação face a face. Mesmo assim, a usabilidade deve ainda ser considerada, afinal ela é muito importante para a criação de um sistema de qualidade, em qualquer ambiente.

A usabilidade deve fazer parte do desenvolvimento, não apenas em busca de produtos usáveis, mas também de práticas usáveis e de modo a envolver todos os membros de

uma equipe considerando o contexto em que as práticas serão aplicadas. Dessa forma, não se tem uma equipe de acordo com os valores de métodos ágeis e de métodos de usabilidade se apenas alguns se preocupam em atender as necessidades dos usuários típicos e clientes, pois todos os membros precisam entender a importância de atendê-las (SANTOS, 2012).

Ana Paula Oliveira dos Santos , em sua dissertação de mestrado (SANTOS, 2012), propôs práticas de usabilidade para a comunidade de software livre através da associação das fases do DCU(Design Centrado no Usuário) com as técnicas de usabilidade da comunidade ágil conforme a tabela 7.

Práticas de usabilidade para Software Livre	
Fases DCU	**Práticas de Usabilidade**
Identificar necessidades para design centrado em humano	Equipe-Núcleo como Donos do Produto Caminhos Completos Especialista-generalista
Especificar contexto de uso	Pouco design antecipado e distribuído
Especificar Requisitos	Definir metas de usabilidade automáticas
Avaliar designs	RITE (Rapid Iterative Testing and Evaluation) para desenvolvedores de software livre

Tabela 7 – Práticas de usabilidade no contexto de Software Livre

As práticas são descritas apresentando um contexto, um problema e uma solução visando a adequação a comunidades de software livre. As fases do DCU são:

Identificar necessidades para design centrado em humano Trazer o DCU para dentro da equipe-núcleo do projeto, tendo um responsável para assumir o papel de Proprietário do Produto,além da equipe-núcleo realizar o levantamento dos usuários, necessidades e o contexto do sistema e assumir o ciclo completo de DCU sem paralelismo com uma equipe externa de usabilidade;

Especificar contexto de uso Levar a equipe-núcleo a responsabilidade de especificar as práticas de usabilidade a serem utilizadas no contexto de uso do sistema;

Especificar requisitos Realizar a escrita de testes de aceitação automáticos baseados em BDD (*Behavior Driven Development*);

Avaliar designs Aplicação do método RITE (Rapid Iterative Testing and Evaluation) pela equipe-núcleo do projeto, sem necessidade de laboratórios de usabilidade podendo ser substituído por um acompanhamento da utilização dos usuários de um pequeno conjunto de funcionalidades.

A descrição completa dessas práticas pode ser encontrada na seção .4 no apêndice do TCC. Esta foi retirada da tese de mestrado da Ana Paula Oliveira dos Santos (SANTOS, 2012) e removido os exemplos aplicados em projetos realizados em sua tese.

Assumindo as práticas descritas ao estudo de caso Mezuro, o ciclo de vida e aplicação destas na comunidade de software livre serão seguidos dentro do projeto junto com a equipe de desenvolvimento, visando suprir as deficiências e problemas citados no capítulo 1.

4.2.4 Métodos de Avaliação

Uma vez que conceituamos as práticas de usabilidade ágeis e o que encontramos sobre usabilidade em projetos de software livre, nesta seção detalharemos os métodos escolhidos para a avaliação da ergonomia das interfaces. Os métodos de avaliação trás um diagnóstico através de verificações e inspeções de aspectos ergonômicos das interfaces que possam ser um problema ao usuário durante sua interação com o sistema. Através desse diagnóstico é possível priorizar e classificar os problemas encontrados de acordo com o método escolhido.

4.2.4.1 Avaliação heurísticas

Uma avaliação heurística representa um julgamento de valor sobre as qualidades ergonômicas das Interfaces Humano-Computador. Essa avaliação é realizada por especialistas em ergonomia, com base em sua experiência e competência no assunto (CYBIS; BETIOL; FAUST, 2010).

Para utilização de uma avaliação heurística serão definidos os graus de severidade. A severidade do problema de usabilidade é uma combinação de três fatores:

- A frequência com que ocorre o problema: é comum ou raro?
- O impacto do problema caso ocorra: Será que vai ser fácil ou difícil para os usuários a superar ?
- A persistência do problema: É um problema que com o tempo os usuários possam superar, uma vez que sabe sobre ele ou os usuários repetidamente serão incomodados pelo problema ?

A escala de classificação seguinte de 0 a 4 pode ser usada para avaliar a severidade dos problemas de usabilidade: (NIELSEN, 1995):

- 0 = Não há consenso quanto a ser um problema de usabilidade
- 1 = Problema cosmético
- 2 = Problema menor
- 3 = Problema importante de usabilidade

- 4 = Catástrofe de usabilidade

A presentação dos resultados seguirá um modelo simples similar ao que é utilizado em desenvolvimento ágil para documentação de defeitos, elencando o problema, a possível solução e o grau de severidade.

4.2.4.2 Inspeções por listas de verificação

As inspeções de ergonomia por meio de listas de verificação permitem que profissionais, não necessariamente especialistas em ergonomia, identifiquem problemas menores e repetitivos das interfaces. Nesse tipo de técnica, ao contrário das avaliações heurísticas, são mais as qualidades explicativas da ferramenta e menos os conhecimentos implícitos dos avaliadores que determinam as possibilidades para a avaliação (CYBIS; BETIOL; FAUST, 2010).

Através das inspeções de ergonomia será possível suprir um deficit ocasionado pela falta de experiência do avaliador dentro de determinados contextos do sistema que este não esteja familiarizado. A ISO 9241 fornece listas de verificação de ergonomia bem definidas, porém será utilizado as listas do laboratório LabIUtil do projeto ErgoList [1], que fornece um serviço na Internet para aplicar uma avaliação simplificada e objetiva (*check-list*) e obtermos os resultados imediatamente. Com a aplicação da lista pode obter vantagens como obter conhecimentos ergonômicos, reduzir a subjetividade normalmente associada a processos de avaliação e sistematizar as avaliações se tratando de abrangência de componentes a inspecionar.

Com a aplicação dos métodos de avaliação serão obtidos os relatórios com os diagnósticos da verificação e inspeção dos aspectos ergonômicos. Estes possibilitarão a classificação (de acordo com cada método) e a priorização dos problemas encontrados, fator fundamental para sustentar as práticas de usabilidade adotadas para a implementação no estudo de caso Mezuro descritas na seção 4.2.3. Ter o controle dos problemas levantados e dos seus impactos na aplicação, irá auxiliar na tomada de decisão da equipe e deixará todos cientes do andamento da usabilidade no projeto e sua importância dentro do desenvolvimento.

4.3 Visualização de Software

O progresso alcançado com sistemas de hardware possibilita que grandes quantidades de informações sejam armazenadas por sistemas computacionais. Aproximadamente um exabyte, quantidade equivalente a um milhão de terabytes, de dados é gerado a cada ano, das quais a maior parte está disponível na forma digital (KEIM, 2002). Um dos

[1] <http://www.labiutil.inf.ufsc.br/ergolist/check.htm>

fatores que contribuem para isso são o avanço das tecnologias da informação e das telecomunicações, além da diminuição dos custos dos dispositivos de armazenamento. Dados armazenados podem auxiliar no apoio aos mais diversos tipos de atividades como definição de políticas públicas, investigações científicas, estratégias de negócios, melhoria da qualidade de sistemas de software. Mas para que os dados possam ser aproveitados ao máximo, é necessário primeiro compreendê-los (HEER; SHNEIDERMAN, 2012).

A visualização fornece meios poderosos para a compreensão de grandes conjuntos de dados. Por meio de representações gráficas e interativas apoiadas por computador é possível mapear atributos ou características, relativos ao domínio observado, em propriedades visuais como posição, tamanho, forma e cor potencializando as habilidades sensoriais do ser humano para o entendimento, tomadas de decisão e interpretação de padrões, agrupamentos, tendências e discrepâncias. A partir de uma representação inicial, o usuário extrai observações e conclusões sobre os dados e interage diretamente com a visualização, moldando-a para atingir os objetivos de sua tarefa (MARTINS, 2012).

O estudo da visualização de informações pode ser dividida em duas grandes vertentes. A primeira é a visualização científica que trata de dados físicos ou geométricos, como o planeta Terra, o corpo humano e fenômenos da natureza. A segunda vertente está relacionada a informações não-físicas, como dados financeiros, coleções de documentos, código-fonte de sistemas de software, os quais podem se beneficiar de representações visuais (MARTINS, 2012). Porém, para esse tipo de informação é necessário aplicar técnicas de visualização, pois não há formas diretas para representá-las.

A visualização de informação é composta por dados de entrada, que consistem em grandes conjuntos de registros. Cada registro contem certo número de atributos ou dimensões. A quantidade de dimensões define a dimensionalidade do conjunto de dados, que pode ser bidimensional ou multidimensional. Conjuntos de dados multidimensionais exigem técnicas mais sofisticadas para sua representação, já que não podem ser mapeados diretamente nos espaços 2D ou 3D, ao contrário dos bidimensionais.

Plataformas de avaliação de qualidade baseadas em conjuntos de ferramentas, como é o caso do Mezuro, expõem o avaliador de qualidade a dezenas de valores numéricos relacionados as métricas calculadas. Sem uma apresentação especial, esses dados são de pouco valor, pois são numerosos e difíceis de avaliar. Muitas vezes as métricas não são correlacionadas e a formatação da apresentação consiste apenas na aplicação de valores de referência.

Uma possibilidade para a solução desse problema é a exploração da visualização de software, uma sub-área da visualização de informações cujos objetivos são auxiliar a compreensão de software e melhorar a produtividade do seu processo de desenvolvimento utilizando visualização, já que essa sub-área gera representações visuais de diversos aspectos do software e de seu processo de desenvolvimento (MARTINS, 2012), como processos

de análise e especificação de requisitos, design e arquitetura, implementação, manutenção e evolução de software.

A visualização de software é uma das ramificações da visualização de informação que mais cresce atualmente. A visualização de software é dividida em três categorias principais: estrutura, comportamento e evolução.

- **Estrutura** é a categoria que representa as partes estáticas do sistema, ou seja, aquelas que podem ser computadas sem executá-lo como por exemplo o código-fonte, estrutura de dados do programa, o grafo de chamadas estático, e a organização do programa em módulos.

  A maioria das ferramentas, técnicas, propostas nessa categoria compartilham um mesmo modelo conceitual: a geração de um grafo, onde vértices representam entidades do programa, como arquivos ou classes, e arestas representam dependências como usos, chamadas, ou heranças.

- **Comportamento** se refere a compreensão do que ocorre com o sistema durante seu tempo de execução, quais instruções são executadas e como seus estados mudam dado um conjunto de possíveis entradas. Entre as aplicações dessa categoria de visualização estão a análise de traces, que são registros de valores de variáveis em certos instantes, animação de algoritmos, depuração visual, e apoio visual a atividade de teste.

- **Evolução** representa as características modificadas ao longo do tempo em um sistema. Técnicas de visualização dessa categoria podem auxiliar analistas a verificarem quais as relações entre artefatos modificados e quais tendências dessas modificações. Analisar a evolução de um sistema pode ser tão ou mais importante que a análise de sua estrutura, já que manutenções, sejam elas corretivas, preventivas ou adaptativas representam grande parte do custo envolvido em um projeto, podendo chegar a até 80% do total.

Como o Mezuro é uma plataforma de análise de código-fonte, ou seja, análise estática, a categoria de visualização de software comportamento, responsável por fornecer informações da execução do sistema, não será relevante para este trabalho, ao contrário das categorias estrutura e evolução.

Selecionar uma técnica de visualização mais relevante para um objetivo ou aplicação particular não é trivial, já que nenhuma técnica específica funciona bem para todos os casos ou problemas (MARTINS, 2012).

Para alcançar as representações gráficas e todas as vantagens proporcionadas pela visualização de software, é necessário primeiro extrair as métricas de código-fonte e repositório.

4.3.1 Métricas de Qualidade

O processo de extração de métricas, muitas vezes, gera um grande volume de métricas, com valores predominantemente numéricos, que ainda são difíceis de analisar sem ferramentas apropriadas. O objetivo da visualização de informação aplicada a softwares, não apenas livres, é possibilitar que grandes quantidades de informações, as métricas extraídas, sejam analisadas objetivamente para compreensão de sua qualidade. Isso permite estabelecer um processo para apoiar as tarefas de avaliação, monitoramento e melhoria da qualidade.

Muitas vezes as avaliações de projetos de software livre são feitas informalmente, com a leitura de documentação e análise de opiniões de usuários anteriores, que nem sempre geram resultados confiáveis. Nesses projetos, muitos dados, tanto de processo como de produto, estão disponíveis publicamente. Exemplos desses dados são: código-fonte, históricos de evolução do repositório, conjunto de testes, relatório de erros, entre outros. Todas essas informações, se corretamente processadas, podem ser utilizadas para avaliar a qualidade de projetos.

Métricas podem ser usadas para medir características e atributos definidos por modelos de qualidade de software voltados para a análise tanto do processo quanto do produto. Alguns modelos propostos especificamente para software livre se baseiam em modelos tradicionais como o CMMI[2](PAULK et al., 1994) e a norma ISO 9126(ISO, 2003) mas incluem extensões que consideram especificamente aspectos importantes do desenvolvimento aberto. No entanto, a multiplicidade de indicadores de qualidade que podem ser extraídos de um projeto de software livre, considerando todas as suas diversas perspectivas (principalmente código, testes e repositório), cada um com suas próprias interpretações e valores de referência, torna complicada a realização de uma avaliação objetiva e direta. Abaixo se encontram as métricas disponibilizadas pelas ferramentas base utilizadas na plataforma Mezuro.

[2] Capability Maturity Model Integration

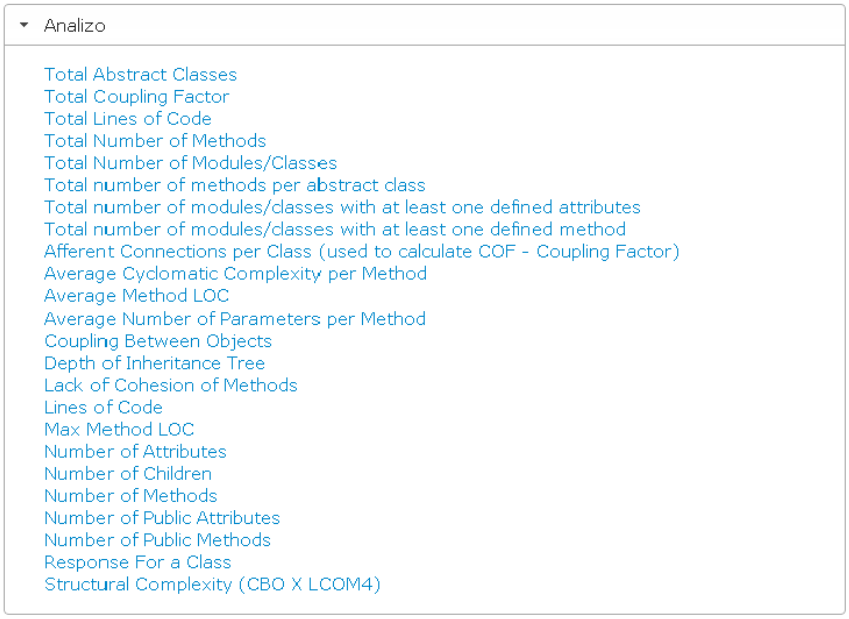


Figura 6 – Métricas fornecidas pela ferramenta base Analizo

O Analizo possui suporte nativo às linguagens C, C++ e Java, extraindo métricas de código-fonte (entre elas há mais de quinze métricas de módulos e sete métricas de projetos). Além dessas características, o Analizo também apoia a evolução de software já que permite a análise de diferentes versões do software em relação ao tempo .

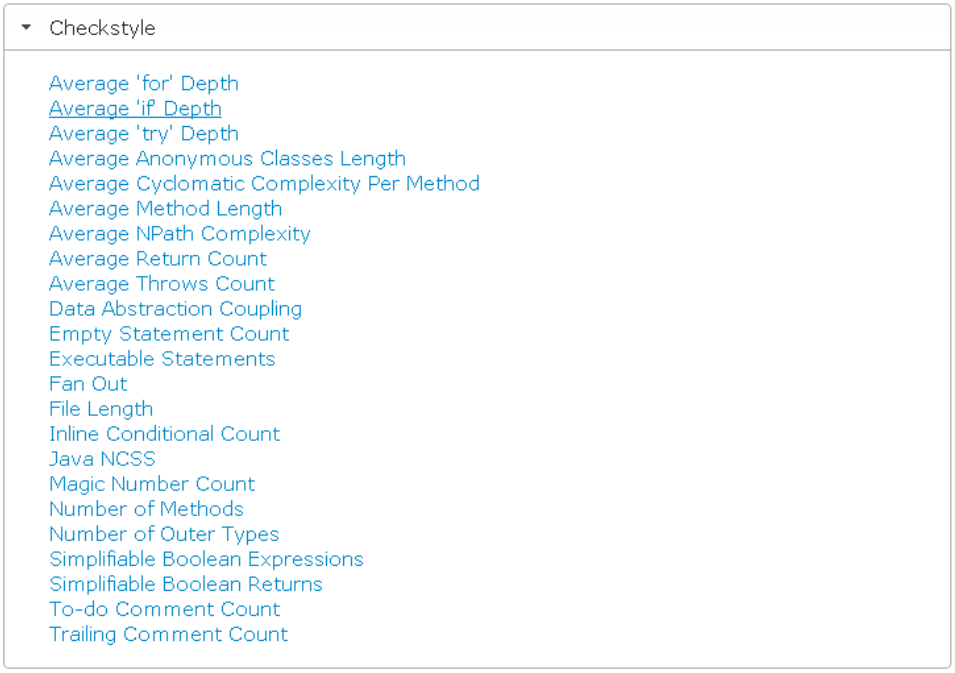


Figura 7 – Métricas fornecidas pela ferramenta base Checkstyle

O Checkstyle é uma ferramente de desenvolvimento que tem como principal objetivo auxiliar os desenvolvedores a seguirem um padrão de codificação. Assim como o

Analizo, é através da extração de métricas de código-fonte Java que essa ferramenta visa alcançar o padrão de codificação pretendido.

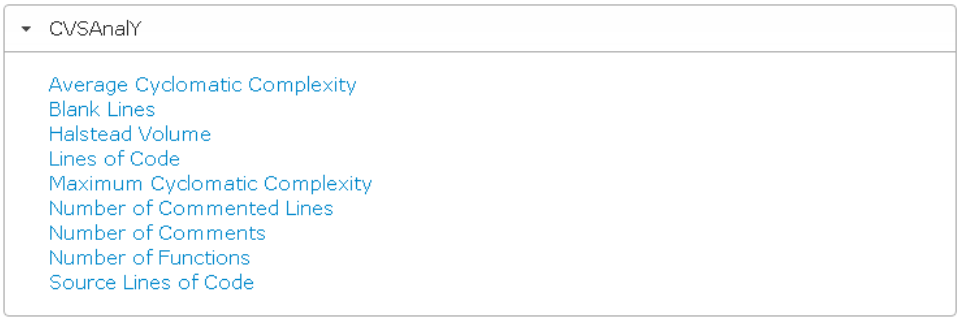


Figura 8 – Métricas fornecidas pela ferramenta base CVS Analy

Dentre os coletores de métricas utilizados no Mezuro, o último a ser adicionado foi o CVSAnaly, que apesar de não ampliar o suporte do Mezuro a novas linguagens, ele aumenta a variedade de métricas disponíveis já que muitas das métricas extraídas por essa ferramenta são complementares às métricas do Analizo e do Checkstyle.

5 Estudo de Caso: Mezuro, uma plataforma de Monitoramento de Código Fonte

A prática da engenharia de software exige compreensão do sistema desenvolvido como um todo, onde o código-fonte é uma das partes mais importantes. O engenheiro de software precisa analisar um código-fonte diversas vezes, seja para desenvolver novas funcionalidades ou melhorar as existentes (MEIRELLES et al., 2010). Como parte fundamental do projeto de software, o código-fonte é um dos principais artefatos para avaliar sua qualidade (MEIRELLES et al., 2009). Essas avaliações não são meramente subjetivas, sendo necessário extrair informações que possam ser replicadas e entendidas da mesma forma, independente de quem analisa o código. As métricas de código-fonte permitem esse tipo de avaliação pois possibilitam analisar, de forma objetiva, as principais características para aceitação de um software.

Há várias características que fazem do software um sistema de qualidade ou não. Entre elas há algumas que são obtidas exclusivamente através do código-fonte. Quando compilamos um software, por exemplo, características podem ser analisadas, mas outras como organização e legibilidade não. Isso não refletiria tamanho, modularidade, manutenibilidade, complexidade, flexibilidade, que são características encontradas na análise de códigos-fonte (MEIRELLES, 2013b). O esforço necessário para extrair métricas de código-fonte manualmente pode ser considerado imensurável. Quanto maior e mais complexo é o código, esse tipo de atividade se torna ainda mais distante das equipes de desenvolvimento, dada a quantidade de propriedades e linhas de códigos a se analisar, além de não ser viável pelo tempo despendido. Existem ferramentas que auxiliam nessas atividades, por meio da extração e monitoramento automático de métricas de código-fonte, auxiliando a equipe durante o processo de desenvolvimento. Porém, existem existem poucas ferramentas disponíveis, e muitas delas nem sempre são adequadas para análise do projetos de software livre (MEIRELLES et al., 2010), que é ponto central deste trabalho.

5.1 Concepção do Projeto Mezuro

A partir da necessidade de extrair métricas de código-fonte e interpretar seus valores, foi desenvolvida uma plataforma chamada Mezuro [1]. Ela possibilita o monitoramento de características específicas do software. O Mezuro foi concebido através de um longo processo de amadurecimento de diversas ferramentas, que teve seu início com o projeto

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

Qualipso[2]. O projeto Qualipso foi um consórcio formado por indústria, academia, e governo. Seu principal objetivo é potencializar as práticas de desenvolvimento de software livre, tornando-as confiáveis, reconhecidas e estabelecidas na indústria, através da implementação de tecnologias, procedimentos e leis (QUALIPSO..., 2009).

No contexto do trabalho desenvolvido com o Mezuro, o principal projeto com características e propósitos similares ao Mezuro é o Code Climate[3]. Essa plataforma auxilia a inserção de qualidade em softwares desenvolvidos com o arcabouço *Ruby on Rails* e com a linguagem Javascript, por meio da analise de quatro indícios (smells):

- **Duplicação** - Estruturas sintáticas repetidas ou similares no projeto.
- **Método complexo** - Alta complexidade para a definição de um método.
- **Classe complexa** - Quando há classes muito grandes no projeto, o que pode ser um sinal de baixa coesão[4].
- **Alta complexidade total em classes** - Mesmo que os métodos da classe sejam simples, se a classe for muito grande é sinal que ela tem muitas responsabilidades.

Em particular, o projeto Mezuro surgiu antes do CodeClimate, que provê serviços similares aos pretendidos pelo Mezuro, mas monitorando apenas três métricas para códigos Ruby. Diferentemente do Mezuro, que hoje, suporta dezenas de métricas (apresentadas na seção 4.3) providas pelos coletores de métricas:

1. Analizo [5]
2. CheckStyle [6]
3. CVSAnalY [7]

O Mezuro utiliza o Kalibro Metrics [8] para fornecer a funcionalidade de análise e avaliação de métricas de código-fonte. O Mezuro e o Kalibro se comunicam através da interface do Kalibro em forma de *web service* conhecida como Kalibro Service. Os criadores do projeto Mezuro argumentam que serviços web são uma boa solução de interoperabilidade, com popularidade crescente na última década. O Mezuro se comunica com este

[2] Quality Platform for Open Source: <http://qualipso.icmc.usp.br/>
[3] <http://codeclimate.com>
[4] Representa o grau de especialização de uma classe para desempenhar papeis em um contexto. Quanto menos responsabilidades tiver uma classe, mais coesa ela será.
[5] <http://anzalizo.org>
[6] <http://checkstyle.sourceforge.net/>
[7] <http://tools.libresoft.es/cvsanaly>
[8] <http://kalibro.org>

web service através de um protocolo conhecido como SOAP[9], baseado em XML[10] para o formato de mensagens. Por meio de requisições SOAP, o Mezuro acessa os *end-points*[11] do Kalibro Service. Em resumo, isso permite ao Mezuro prover aos seus usuários as seguintes funcionalidades:

1. Baixar códigos-fonte de repositórios dos tipos GIT, Subversion, Baazar e CVS
2. Criação de configurações, que conjuntos pré-definidos de métricas relacionadas para serem utilizadas na avaliação de projetos de software.
3. Criação de intervalos relacionados com a métricas e avaliações qualitativas.
4. Criação de novas métricas compostas, de acordo com aquelas fornecidas pelos coletores do Kalibro.
5. Cálculo de resultados estatísticos para módulos com alta granularidade.
6. Possibilidade de exportar arquivos com os resultados gerados.
7. Interpretação dos resultados com interface mais amigável aos usuários com a utilização de cores nos intervalos das métricas.

5.2 Mezuro como Plugin do Noosfero

O Mezuro foi concebido como instância de uma plataforma web conhecida como Noosfero, com o plugin Mezuro ativado. O Noosfero é um software livre para criação de redes sociais, que está disponível sob licença AGPL[12] V3, com o intuito de permitir que os usuários criem sua própria rede social livre e personalizada de acordo com suas necessidades.

A linguagem de programação Ruby e o arcabouço *MVC Ruby on Rails* foram utilizados para desenvolver o Noosfero. Essas tecnologias foram escolhidas pois a linguagem Ruby possui uma sintaxe simples, que facilita a manutenibilidade do sistema, característica importante em projetos de software livre que tendem a atrair colaboradores externos a equipe. Já o arcabouço *Ruby on Rails* influencia em maior produtividade graças a conceitos como *convention over configuration* e DRY . Por esse motivo o Noosfero "herda" sua arquitetura, a qual é baseada no padrão arquitetural MVC, assim como os plugins que estendem suas funcionalidades.

A arquitetura do Noosfero permite a adição de novas funcionalidades através de plugins. Essa característica é interessante, pois colaboradores podem incorporar novas

[9] Simple Object Access Protocol
[10] Linguagem de Marcação Extensível
[11] Métodos disponibilizado pelo *web service*, onde cada método define uma funcionalidade
[12] Licença de software GNU Affero General Public License

funcionalidades ao Noosfero, já que os plugins possuem o código isolado, mantendo o baixo acoplamento e alta coesão dos módulos do sistema. Embora plugins sejam totalmente independentes do sistema alvo, no Noosfero os plugins são mantidos com o código principal para auxiliar no controle de qualidade do ambiente. Pensando nisso os plugins devem ter testes automatizados. Quando houver a necessidade de alterar o código do Noosfero, os testes dos plugins são executados para verificar se as mudanças não afetaram seu funcionamento [13]. O funcionamento dos plugins é inspirado no paradigma de orientação a eventos. O núcleo do Noosfero dispara um evento durante sua execução e os plugins interessados nesse evento saberão como tratá-lo. Os eventos que são disparados pelo Noosfero são chamados de "hotspots".

5.3 Mezuro como aplicação independente

As colaborações com a plataforma Mezuro, relacionadas a este trabalho, se iniciaram somente após a decisão de reescrita de seu código, para transformá-la em uma aplicação independente. Por isso, decidimos elaborar um questionário destinado à equipe de desenvolvimento do Mezuro. Esse questionário teve como objetivo extrair informações que embasassem a evolução da plataforma, do ponto de vista do código-fonte e sua arquitetura. O questionário encontra-se disponível no Apêndice.1 deste documento e as respostas de alguns dos desenvolvedores se encontram no Anexo.2. De acordo com as informações obtidas com o questionário, é percebido que não foi um fator isolado que motivou a evolução da plataforma Mezuro, e sim um conjunto deles.

Como já foi mencionado, o Mezuro foi concebido como um plugin do Noosfero. Porém, até a decisão de evoluir o Mezuro, ainda não havia previsões de atualização do Noosfero, o qual se encontrava nas versões 1.8 do Ruby e 2 do *Ruby on Rails*. Atualmente o Noosfero passa por um processo de atualização para as versões 1.9.2 e 3.2 do Ruby e *Ruby on Rails*, respectivamente, ou seja, mesmo com a atualização recente, o Noosfero ainda não fornecerá os recursos mais novos do arcabouço *Rails*.

Como apresentado na seção 3.2.2, o Rails encontra-se na quarta versão, com mais recursos e melhorias em relação às versões anteriores. Além disso, a versão 1.8 do Ruby já não recebe suporte dos desenvolvedores desde o início de 2012, em favor das versões 1.9 e superiores. Isso era um fator limitante, pois os desenvolvedores do Mezuro ficavam restritos aos recursos disponíveis nessas versões usadas pelo Noosfero.

Acompanhando a evolução do Rails, visando os novos recursos, além do suporte da comunidade[14] desse arcabouço, a equipe de desenvolvimento da plataforma Mezuro decidiu atualizá-la para as novas versões 2 do Ruby e 4 do Rails. A comunidade do Rails

[13] <http://noosfero.org/Development/Plugins>
[14] <http://rubyonrails.org/>

favorece a utilização das últimas versões, e as versões mais antigas vão perdendo força, passando a receber cada vez menos suporte dos desenvolvedores.

Conforme observado na Figura 14, o Mezuro é um "cliente" do Kalibro. Ele como um plugin para o Noosfero, faz com que possua muitos recursos, que primeiro foram vistos com vantagem, mas depois foi avaliado pela equipe do Mezuro como desnecessários para uma ferramenta de monitoramento de código-fonte. Entre esses recursos estão: blog, fórum, upload de arquivos, CMS[15], chat e relacionamento entre "amigos". Evolução de software também significa retirar funcionalidades que não se encaixam mais ao ambiente que o software está inserido. Foi exatamente isso que a equipe de desenvolvimento levou em consideração ao decidir por esse passo na evolução do Mezuro.

A manutenibilidade do Mezuro como plugin se tornava cada vez menor conforme ele evoluia, pois ao invés de fornecer um número baixo de funcionalidades, ou funcionalidades muito específicas, que é o princípio de um plugin, o Mezuro acabou se transformando em uma aplicação, a qual depende de outra aplicação, no caso o Noosfero. Neste ponto, podemos observar a segunda lei de Lehman disponível na Tabela 1, à medida que um software é alterado sua complexidade tende a crescer, a não ser que um trabalho seja feito para mantê-la ou diminuí-la. E assim fez a equipe de desenvolvimento do Mezuro. Em suma, pensando no desempenho e na evolução do desenvolvimento do Mezuro, com o objetivo de formatar um comunidade de software livre para atrair desenvolvedores, resolveu-se retirar o Mezuro como um plugin do Noosfero, para não mais o limitar ao andamento do desenvolvimento do Noosfero.

Do ponto de vista do desenvolvimento, a Figura 9 representa o projeto de alto-nível da plataforma Mezuro como uma aplicação independente. Como mencionado anteriormente, ele funciona como a camada de visualização do Kalibro Metrics, que por sua vez utiliza coletores de métricas (Analizo, CheckStyle e CVSAnalY) para executar suas funcionalidades.

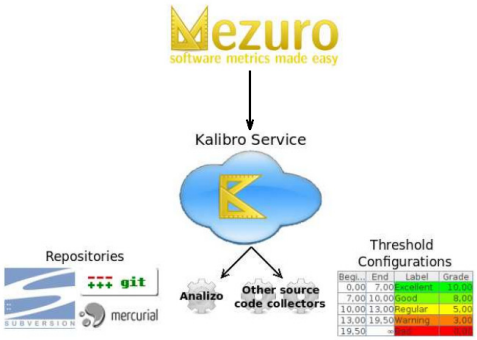


Figura 9 – Design de alto-nível do Mezuro. Editado de (MEIRELLES et al., 2010)

[15] Sistemas de Gerenciamento de Conteúdo, do inglês Content Management System. Os mais conhecidos são o Wordpress e o Joomla!

Sem as restrições impostas pelo desenvolvimento do Noosfero, uma importante decisão arquitetural tomada pela equipe do Mezuro foi o desenvolvimento de uma *gem* (equivalente a uma *Library* do Java) do Kalibro Metrics. Anteriormente, quando o Mezuro estava incorporado ao Noosfero, toda a interface de utilização do serviço do Kalibro estava implementada no próprio Mezuro. Com o desenvolvimento dessa *gem*, esse serviço se torna mais reutilizável, resultando numa contribuição a toda a comunidade do Rails, já que projetos desenvolvidos com esse arcabouço que necessitarem utilizar serviços do Kalibro, bastam instalar e utilizar a *gem*, sem a necessidade de reimplementar a interface do serviço.

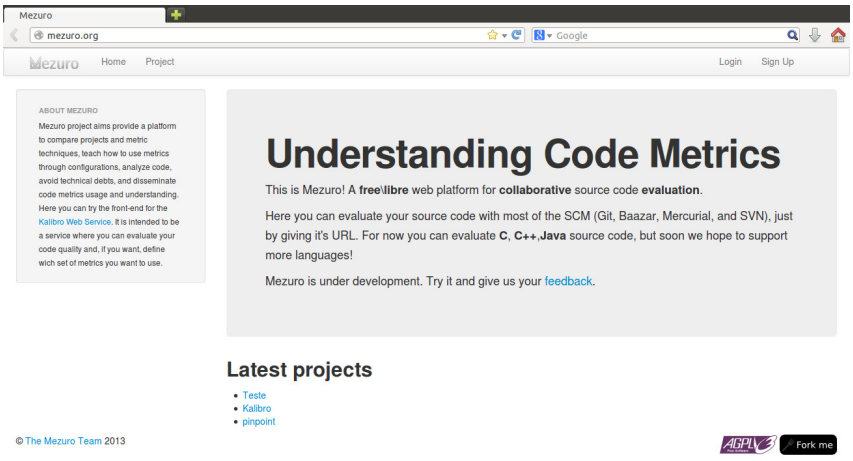


Figura 10 – Tela principal do Mezuro. Disponível em <http://mezuro.org/>

Dentro da evolução do Mezuro, numa primeira fase deste trabalho, visando conhecer a tecnologia, se integrar com a equipe *core*, e se familiarizar com o código-fonte, foi desenvolvida a funcionalidade de inserção de repositórios nos projetos cadastrados. Um projeto pode conter conter vários repositórios, porém apenas um é processado a cada instante. Ao término de um processamento, métricas que representam o estado atual do repositório são fornecidas ao usuário. O diagrama da Figura 14 representa bem a relação entre as entidades citadas acima.

Porém, para processar um repositório, é necessário informar ao Mezuro quais métricas deseja-se obter ao final. Para isso criou-se uma entidade para representar quais informações o processamento deve retornar, essa entidade é a Configuração. A implementação da configuração iniciou-se após a conclusão das funcionalidades relacionadas a entidade Repositório, que incluem as *actions* básicas como criação, atualização, remoção e visualização. Os objetivos desse primeiro ciclo incluem não apenas a conclusão de uma funcionalidade, mas também a familiarização com o código-fonte, interação com a equipe, que se encontra distante geograficamente, além da aplicação de padrões de contribuição a softwares livre, principalmente padrões de envolvimento, como visto na seção 2.3.

Essas funcionalidades, relacionadas as entidade Repositório e Configuração, desenvolvidas durante este trabalho, se encontram disponíveis no ambiente de produção da

plataforma Mezuro como aplicação independente[16], ilustrada na Figura 11.

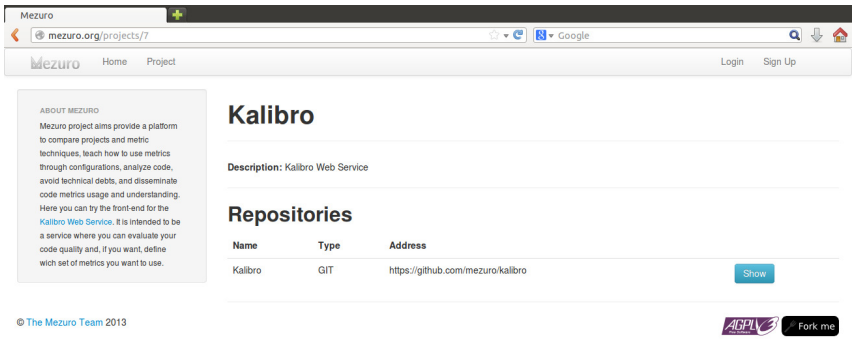


Figura 11 – Tela de visualização de um projeto

Ao clicar em algum projeto disponível no Mezuro, sua tela de visualização é carregada. Nela são exibidas todos os repositórios relacionados a esse projeto. Na Figura 11, por exemplo, é exibida a tela do monitoramento do código do Kalibro cadastrado no Mezuro, assim como um único repositório associado a ele.

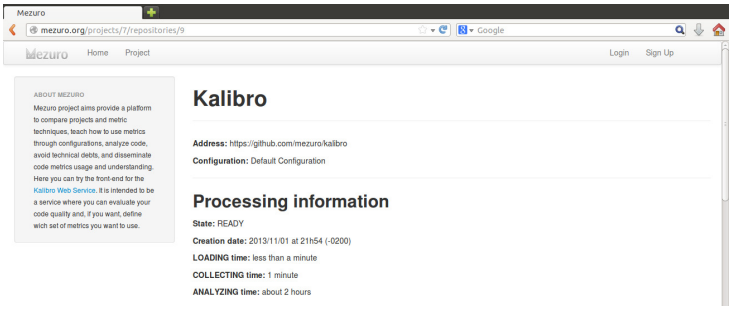
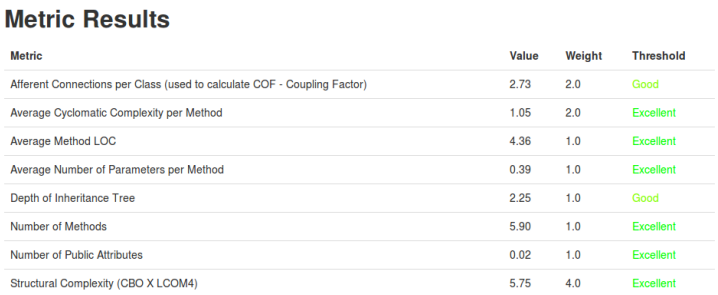


Figura 12 – Tela de informações do repositório



Metric Results

Metric	Value	Weight	Threshold
Afferent Connections per Class (used to calculate COF - Coupling Factor)	2.73	2.0	Good
Average Cyclomatic Complexity per Method	1.05	2.0	Excellent
Average Method LOC	4.36	1.0	Excellent
Average Number of Parameters per Method	0.39	1.0	Excellent
Depth of Inheritance Tree	2.25	1.0	Good
Number of Methods	5.90	1.0	Excellent
Number of Public Attributes	0.02	1.0	Excellent
Structural Complexity (CBO X LCOM4)	5.75	4.0	Excellent

Figura 13 – Métricas do repositório após processamento

Ao clicar em visualizar um repositório, é feito um processamento desse, e a tela de detalhes do mesmo é carregada. A Figura 12 e 13 são os resultados do processamento do

16 <http://mezuro.org/>

único repositório associado ao projeto Kalibro monitorado pelo Mezuro. A Figura 12 exibe informações gerais do processamento do repositório, como a configuração, estado, data de criação e tempo de processamento. A Figura 13 mostra o conjunto de métricas relacionas à configuração do repositório processado (uma configuração *default*, por exemplo), assim como os intervalos e interpretação de cada uma dessas métricas.

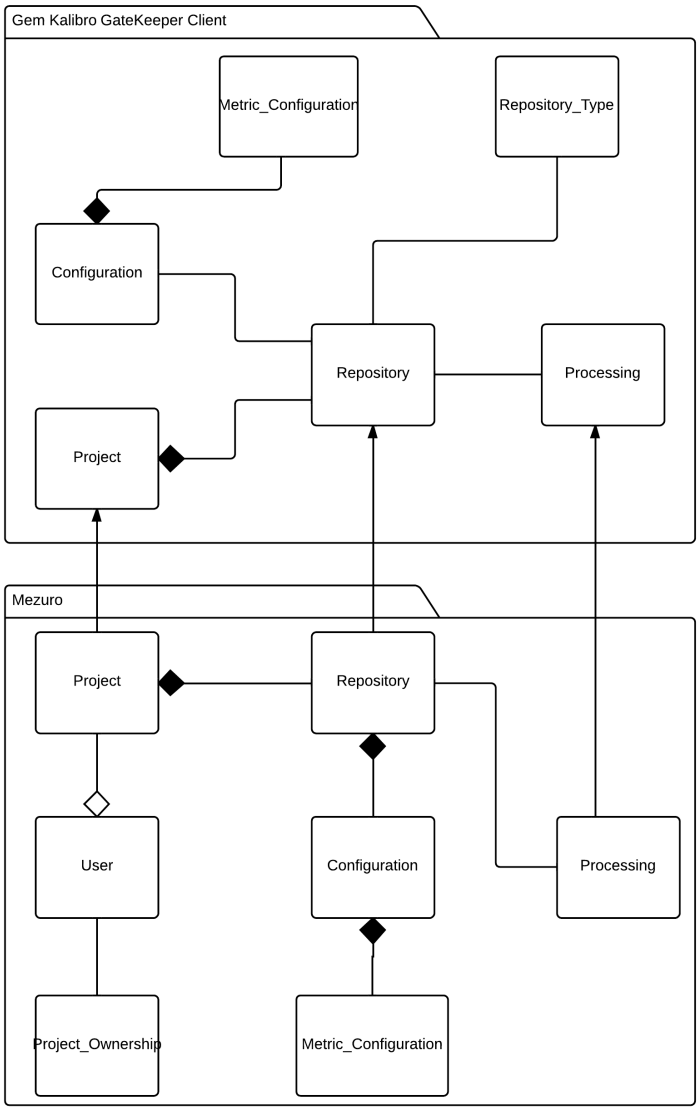


Figura 14 – Diagrama de classes simples do Mezuro

Por fim, a Figura 14 apresenta um diagrama de classes que representa as principais entidades já implementadas no Mezuro como plataforma independente. A classe *Project* representa o projeto a ser analisado. Um projeto pode conter vários repositórios, de onde

as métricas são extraídas. Um repositório pode ser de vários tipos (Git[17], Subversion[18], Bazaar[19], Mercurial[20]), assim como possui uma configuração relacionada a ele. Uma configuração representa um conjunto de métricas, um intervalo para cada uma delas, além da interpretação de cada métrica após o processamento de um repositório. No Mezuro as métricas extraídas de um repositório são representadas pela classe *Metric Configuration*. Já a classe *Processing* representa o processamento de um repositório para obter os resultados das métricas extraídas de acordo com a configuração selecionada.

5.3.1 Evolução com foco na usabilidade

Após uma primeira interação com a equipe e estando mais a par do projeto, iniciou-se um novo ciclo com foco na usabilidade a fim de inseri-la no ciclo de desenvolvimento. Seguindo as técnicas de usabilidade adotas, descrita na seção 4.2.3, as tarefas de usabilidade a serem realizadas no projeto Mezuro serão:

- Coletar requisitos de tarefas e necessidades dos usuários;
- Criar protótipos de tela;
- Testar protótipos com avaliações heurísticas;
- Testar a usabilidade de protótipos com usuários locais e remotos (manipulação de variáveis independentes);
- Coletar métricas (variáveis dependentes);
- Apresentar resultados das análises dos testes (*tracking*);
- Definição de histórias de usabilidade baseadas nos requisitos fornecidos pelos clientes e no resultado dos testes de usuário;
- Implementação das histórias definidas.

A forma mais usual para avaliar usabilidade de uma tela propondo sua evolução em um ciclo ágil é através da avaliação heurística descrito na seção 4.2.4.1, porém essa avaliação é realizada por especialistas em ergonomia, com base em sua experiência e competência no assunto (CYBIS; BETIOL; FAUST, 2010). A fim de melhorar e adquirir essa competência para as futuras avaliações, essa primeira interação de usabilidade foi realizada utilizando inspeções por listas de verificação conforme na seção 4.2.4.2. Como essas listas trazem os principais problemas que podem ser encontrados e uma padronização

[17] <http://git-scm.com/>
[18] <http://subversion.apache.org/>
[19] <http://bazaar.canonical.com/en/>
[20] <http://mercurial.selenic.com/>

da avaliação, isso ajuda a conhecer os aspectos que devem ser observados com maior frequência em uma avaliação heurística.

Os artefatos gerados nessa interação levam em consideração os seguinte aspectos de usabilidade que são encontrados nas listas do laboratório LabIUtil do projeto ErgoList [21]:

Presteza verifica se o sistema informa e conduz o usuário durante a interação.

Agrupamento por localização Verifica se a distribuição espacial dos itens traduz as relações entre as informações.

Agrupamento por formato Verifica os formatos dos itens como meio de transmitir associações e diferenças.

Feedback Avalia a qualidade do *feedback* imediato às ações do usuário.

Legibilidade Verifica a legibilidade das informações apresentadas nas telas do sistema.

Concisão Verifica o tamanho dos códigos e termos apresentados e introduzidos no sistema.

Ações Mínimas Verifica a extensão dos diálogos estabelecidos para a realização dos objetivos do usuário.

Densidade Informacional Avalia a densidade informacional das telas apresentadas pelo sistema.

Ações Explícitas Verifica se é o usuário quem comanda explicitamente as ações do sistema.

Controle do Usuário Avalia as possibilidades do usuário controlar o encadeamento e a realização das ações.

Flexibilidade Verifica se o sistema permite personalizar as apresentações e os diálogos.

Experiência do Usuário Avalia se usuários com diferentes níveis de experiência têm iguais possibilidades de obter sucesso em seus objetivos.

Proteção contra erros Verifica se o sistema oferece as oportunidades para o usuário prevenir eventuais erros.

Mensagens de erro Avalia a qualidade das mensagens de erro enviadas aos usuários em dificuldades.

Correção de erros Verifica as facilidades oferecidas para que o usuário possa corrigir os erros cometidos.

[21] <http:labiutil.inf.ufsc.br/ergolist/check.htm>

Consistência Avalia se é mantida uma coerência no projeto de códigos, telas e diálogos com o usuário.

Significados Avalia se os códigos e denominações são claros e significativos para os usuários do sistema.

Compatibilidade Verifica a compatibilidade do sistema com as expectativas e necessidades do usuário em sua tarefa.

Aspectos de usabilidade retirado da ErgoList (INFORMáTICA, 2013)

A avaliação foi realizada em um conjunto de telas que tinha como principal foco a informação e compreensão das mensagens pelo usuário que são as telas com entrada de dados através de formulários. As telas escolhidas foram: *SignUp*, *New Project*, *Edit Account*, *New Configuration* e *New Reading Group*. A avaliação das telas utilizando o *checklist* da ErgoList gerou o seguinte resultado

Laudo Final				
Aspectos-Questões	Conformes	Não conformes	Não aplicáveis	Total
Presteza	4	7	6	17
Agrupamento por localização	6	1	4	11
Agrupamento por formato	9	2	6	17
Feedback	9	2	1	12
Legibilidade	14	1	12	27
Concisão	6	3	5	14
Ações Mínimas	3	2	0	5
Densidade Informacional	5	1	3	9
Ações Explícitas	4	0	0	4
Controle do Usuário	2	0	2	4
Flexibilidade	0	2	1	3
Experiência do Usuário	4	1	1	6
Proteção contra erros	4	2	1	7
Mensagens de erro	7	2	0	9
Correção de erros	1	0	4	5
Consistência	10	1	0	11
Significados	7	2	3	12

Tabela 8 – Resultado da avaliação de telas do Mezuro

Como o *checklist* da ErgoList é utilizado para avaliar todos os âmbitos de um projeto e o trabalho é realizado em um conjunto específico de telas, então ao analisar os resultados foram encontradas muitas questões que não foram aplicáveis devido a esse domínio. Dentro desse mesmo domínio algumas questões não estão conforme, pois não é o objetivo do projeto, um exemplo é a utilização do campo piscante para destaque de uma informação que não convém com o padrão de aparência utilizado. Desconsiderando esses

fatores é possível observar uma debilidade na presteza, agrupamento e concisão das telas o que foi pontuado nos protótipos de tela.

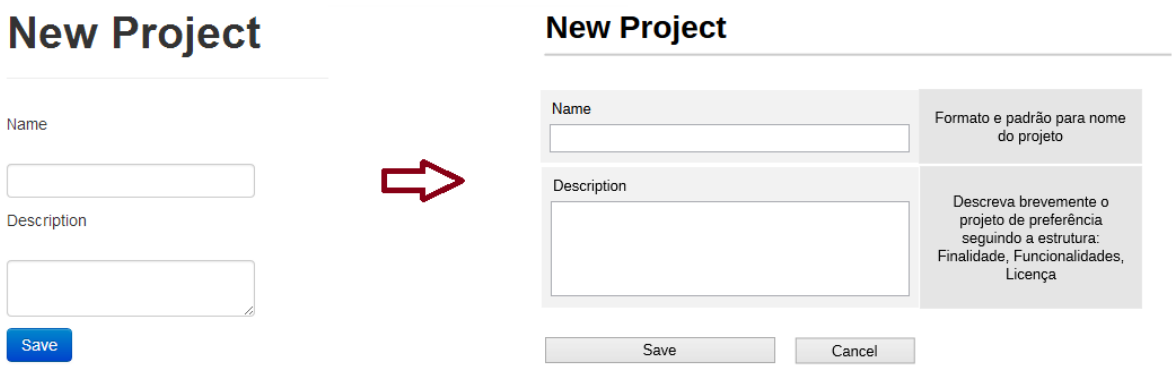


Figura 15 – Versão Avaliada x Protótipo de tela - New Project

Os protótipos de tela foram apresentados a equipe de desenvolvimento juntamente com uma descrição do comportamento para melhor entendimento do que seria realizado. Após refinamento da proposta foi realizado a implementação dentro da camada de front-end do projeto, ao replicar a funcionalidade as outras telas, percebeu-se um padrão o que possibilitou gerar um CSS encontrado na seção do apêndice .6.4 denominado formwithto-oltip.css que possui um conjunto de padrões com foco em suprir a debilidade dessas telas em relação a presteza, agrupamento e concisão, mas precisamente nos campos de descrição da tela. Esse CSS serviu como legado a equipe para aplicação em outros contexto, sendo já utilizado por uma equipe externa da disciplina LabXP da USP.

Tela *SignUp* reformulada levando em consideração as pontuações de usabilidade levantadas, uma das principais telas em relação a instrução da entrada correta de informações do usuário.

Tela *New Project* para apresentar outro contexto da aplicação, já que no contexto do projeto Mezuro foram reformuladas um total de 5 telas utilizando esse padrão. Com a implementação da melhoria proposta foi feita uma nova avaliação utilizando o mesmo *checklist* da ErgoList que apresentou os seguintes resultados

Ao analisar os novos dados foi constatado uma melhoria na presteza, agrupamento e concisão conforme desejado, porém a melhoria também impactou em outros fatores como consistência e significados.

Com um domínio de abrangência aos aspectos de usabilidade um pouco menor houve uma interação responsável pela tela de *SignIn*. Como essa tela difere das outras implementadas na primeira interação, então uma nova avaliação e protótipo foram realizados gerando os resultados a seguir.

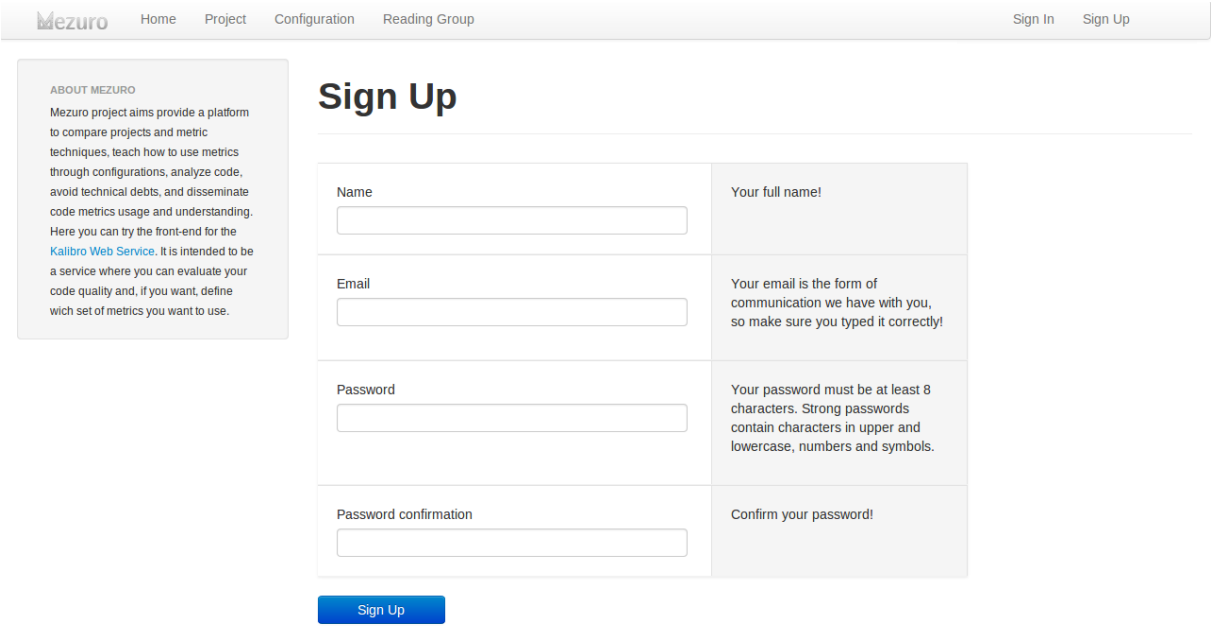


Figura 16 – Resultado implementação tela SignUp

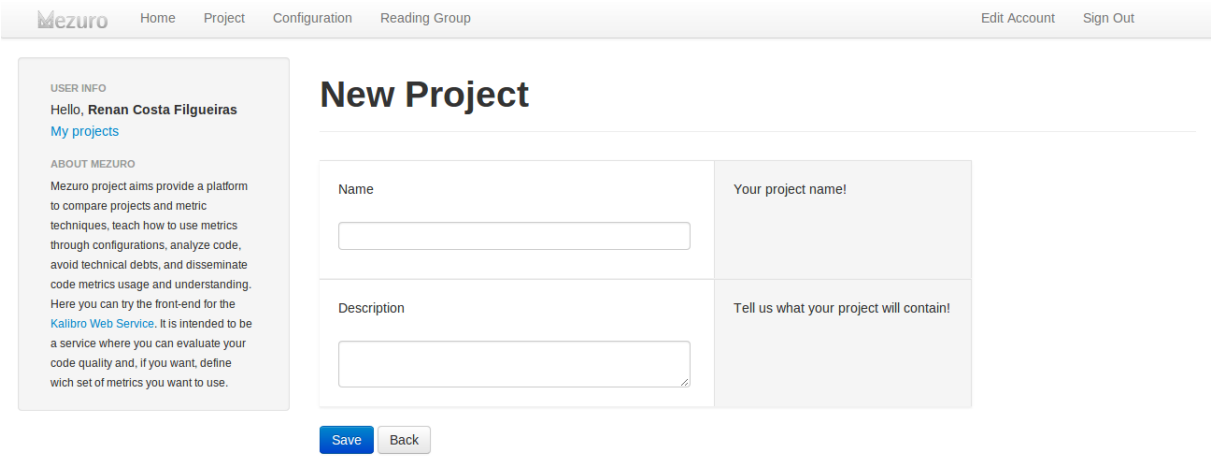


Figura 17 – Resultado implementação tela New Project

Com a analise dos dados e levando em consideração os mesmos porém levantados na interação anterior referente aos itens não aplicáveis e não conforme, gerou-se um protótipo de tela para avaliação pela equipe.

Com o protótipo de tela a equipe pode refinar e opinar sobre como seria o resultado final e contribuir com a evolução das interfaces. A implementação dessas melhorias resultou na seguinte tela

Laudo Final				
Aspectos-Questões	Conformes	Não conformes	Não aplicáveis	Total
Presteza	8	3	6	17
Agrupamento por localização	7	0	4	11
Agrupamento por formato	10	1	6	17
Feedback	9	2	1	12
Legibilidade	14	1	12	27
Concisão	7	2	5	14
Ações Mínimas	4	1	0	5
Densidade Informacional	6	0	3	9
Ações Explícitas	4	0	0	4
Controle do Usuário	2	0	2	4
Flexibilidade	0	2	1	3
Experiência do Usuário	4	1	1	6
Proteção contra erros	5	1	1	7
Mensagens de erro	7	2	0	9
Correção de erros	1	0	4	5
Consistência	11	0	0	11
Significados	9	0	3	12

Tabela 9 – Resultado avaliação telas reformuladas

Laudo Final				
Aspectos-Questões	Conformes	Não conformes	Não aplicáveis	Total
Legibilidade	11	4	12	27
Agrupamento por formato	8	2	7	17
Presteza	5	3	9	17
Agrupamento por localização	3	3	5	11

Tabela 10 – Resultado avaliação tela SignIn anterior

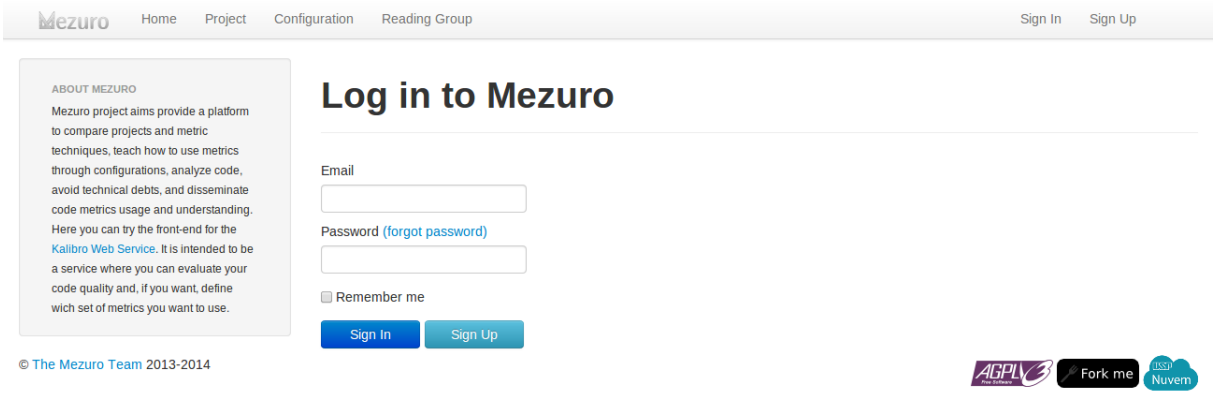


Figura 19 – Implementação tela Sign In

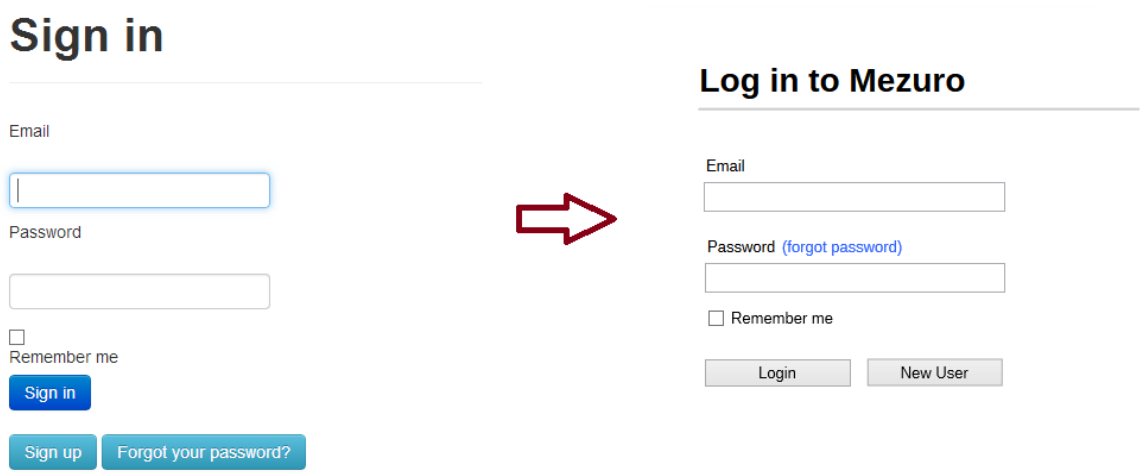


Figura 18 – Versão Avaliada x Protótipo de tela - Sign In

A avaliação utilizando o *checklist* da ErgoList foi refeita chegando aos seguintes resultados

Laudo Final				
Aspectos-Questões	Conformes	Não conformes	Não aplicáveis	Total
Legibilidade	14	1	12	27
Agrupamento por formato	9	1	7	17
Presteza	5	3	9	17
Agrupamento por localização	7	0	4	11

Tabela 11 – Resultado avaliação implementação tela SignIn

Com a implementação realizada na tela *Sign In* foi possível observar uma melhoria nos aspectos de usabilidade mais significativamente no agrupamento por localização devido ao um não agrupamento de informações que a versão anterior da tela de *Sign In* não possuía.

Com a primeira interação de usabilidade dentro do ciclo de desenvolvimento da equipe, foi possível adquirir experiência em relação a avaliação de usabilidade, o processo de trabalho da equipe, as tecnologias necessárias para a camada de *front-end* e a melhor forma de conduzir o entendimento da equipe quanto as melhorias propostas. Através desses conhecimentos obtidos uma próxima interação de usabilidade foi proposta seguindo o método de avaliação heurística para avaliar as telas e sugerir se necessário uma melhoria. Para essa etapa uma reunião foi montada junto a equipe que sugeriu o conjunto de telas da funcionalidade de configuração de métricas que apresentava um caminho muito exaustivo para o usuário.

Analisando a funcionalidade *Configuration* foi percebido um fluxo massante que o usuário tinha que percorrer para executar a funcionalidade. Os prontos mais críticos

levantados foram:

1. A escolha das métricas na tela *Choose Metric*: Não possibilita escolha múltipla de métricas forçando o usuário acessar diversas vezes a tela;
2. Informações iniciais das métricas na tela *New Metric*: Preenchimento repetitivo dos dados e falta de sugestão para o campo *Code*, onde poderia definir um padrão;
3. Excesso de passos: Quantidade excessiva de passos para termino da configuração de métricas do projeto.
4. Preenchimento do formulário: Devido ao processamento realizado em cima de cada métrica um meio dinâmico de edição do formulário torná-o mais eficiente ao longo do cadastro.

Baseado nesses pontos críticos e pretendendo resolvê-los foi construído os protótipos de tela e apresentado a equipe para discussão já que o comportamento da funcionalidade *Configuration* foi alterado. Todos os pontos críticos foram classificados dentro da severidade 3 - problema importante de usabilidade conforme definido na seção 4.2.4.1. A tela de configuração onde são listados os métodos passou a contar com mais campos de informação relevantes as métricas escolhidas.

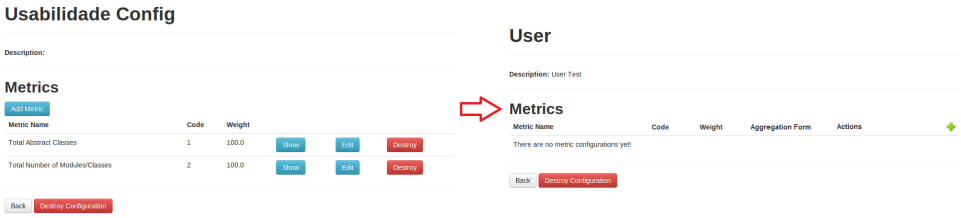


Figura 20 – Versão Avaliada x Protótipo de tela - Configuration

A tela *Choose metric* passou a contar com a múltipla escolha de métricas, sendo essa funcionalidade exercida por *checkboxs* que permitem a seleção das métricas desejadas e que serão adicionadas ao projeto.

A tela de *New Metric* foi unificada a tela inicial de configuração, com isso não há mais a necessidade de outra tela, o preenchimento dos dados inicias passaram a ser dinâmico realizado na página que lista as métricas e o campo *Code* passou a ser pré-definido com preenchimento automático.

Com os protótipos avaliados, iniciou-se a implementação do novo comportamento da funcionalidade *Configuration*, devido as funcionalidades dinâmicas na tela foi necessário o uso a mais das tecnologias JavaScript, Ajax e JQuery a fim de deixar a funcionalidade o mais transparente possível. A implementação do *Configuration* resultou nas seguintes telas.

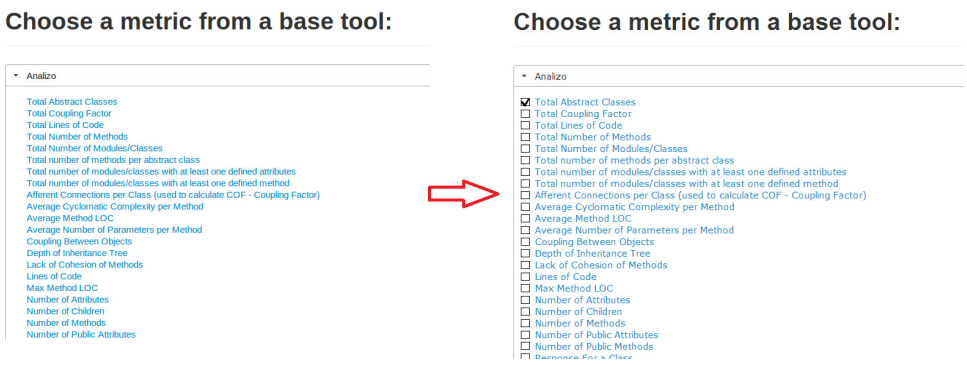


Figura 21 – Versão Avaliada x Protótipo de tela - Choose Metric

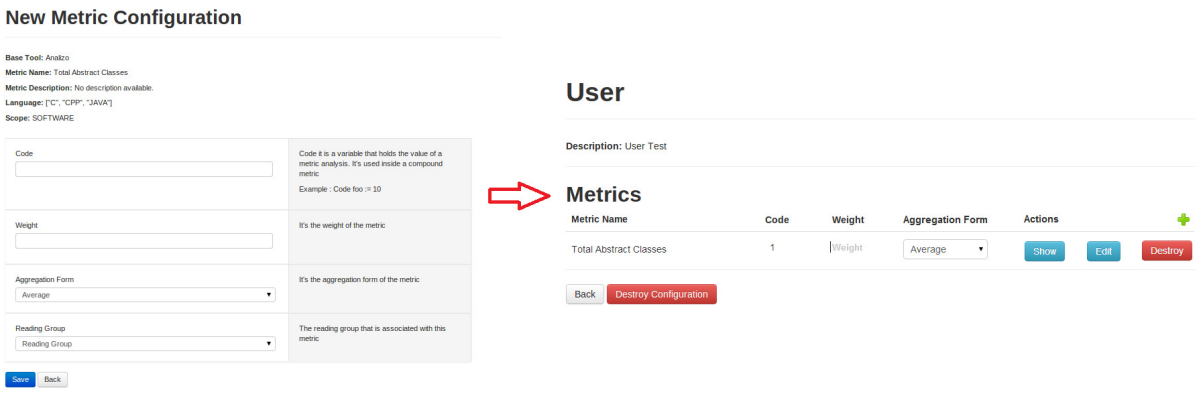


Figura 22 – Versão Avaliada x Protótipo de tela - New Metric

A tela implementada apresenta uma lista dinâmica para a alteração das informações das métricas, sem a necessidade de passar por um novo fluxo. O code das métricas são gerados automaticamente como uma proposta que pode ser seguida ou alterada pelo usuário diminuindo o tempo de configuração das métricas.

A implementação da tela *Choose metric* possibilitou a múltipla escolha de métricas evitando um *loop* no fluxo para cadastro de mais de uma métrica que gerava uma exaustão ao usuário já que normalmente um projeto possui mais de 4 a 5 tipos de métricas.

O resultado quanto ao nível da melhoria de usabilidade foi avaliado pela equipe core do Mezuro, já que a ferramenta ainda está em um ciclo de desenvolvimento e não se encontra totalmente em produção, levando em consideração todas as melhorias realizadas pelo trabalho na plataforma.

Para essa avaliação foi aplicado o questionário Psychometric Evaluation Of The Post-Study System Usability Questionnaire (PSSUQ). O PSSUQ foi publicado em 1992 por James R. Lewis, composto inicialmente por 18 questões e atualizado para 19 questões em 1995, esse questionário avalia a satisfação do usuário após a participação em estudos de usabilidade baseadas em cenários (LEWIS, 1992). As características avaliada são facilidade de uso e de aprendizado, simplicidade, eficácia, informação e a interface com o

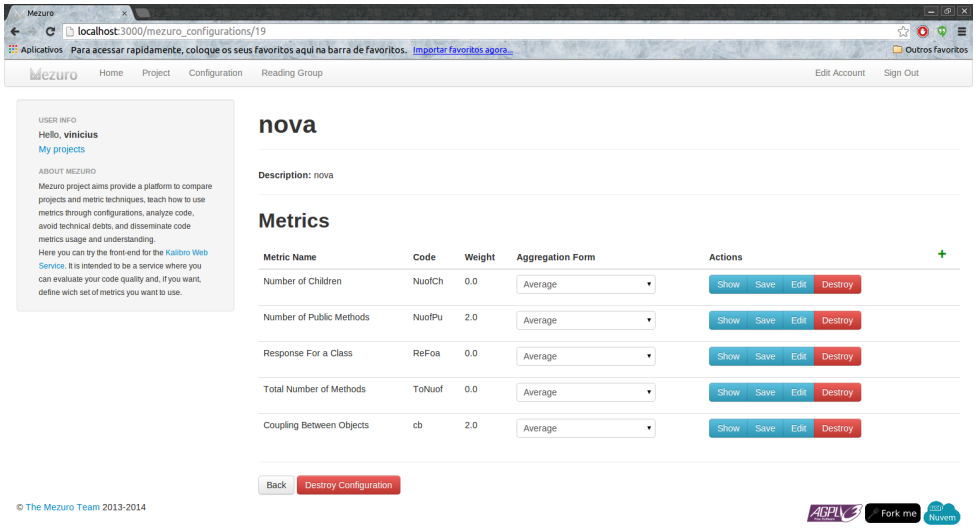


Figura 23 – Implementação tela

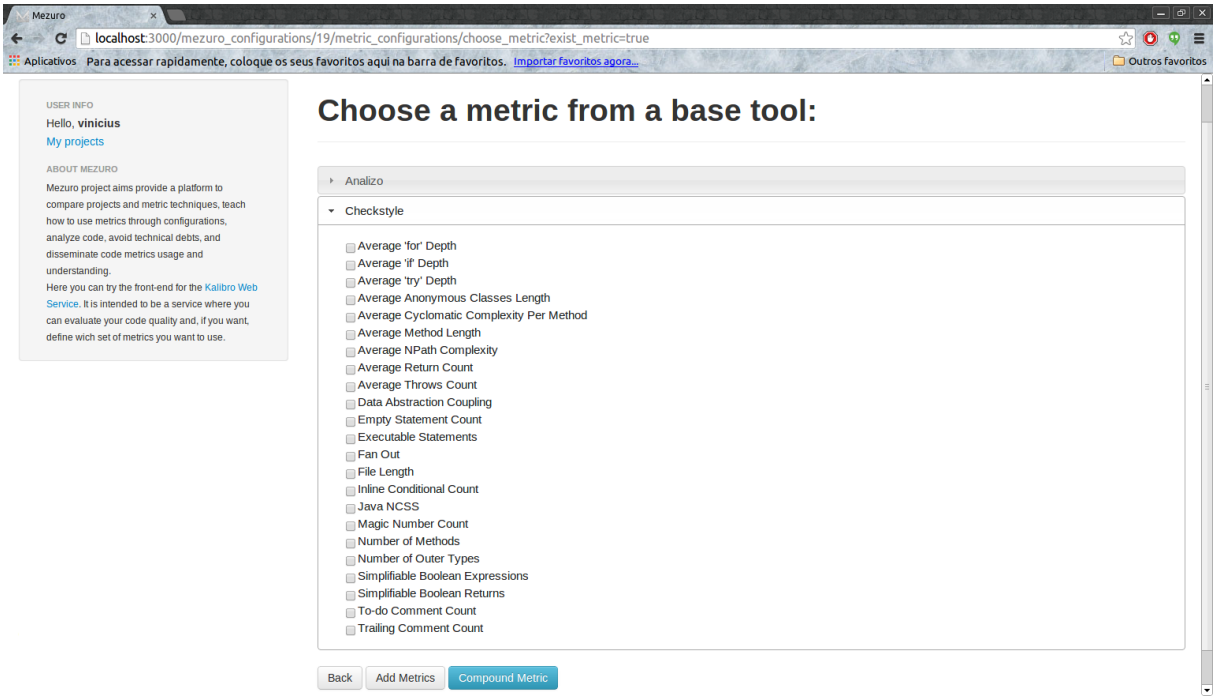


Figura 24 – Implementação tela

usuário. O PSSUQ possui questões intuitivas, de fácil compreensão e um tempo em média de 10 minutos para completar todo o questionário, juntamente com a escala de 1 a 7 de satisfação. As respostas são classificadas quanto ao:

- Pontuação da satisfação geral (OVERALL): Avalia os itens de 1 a 19;
- Utilidade do sistema (SYSUSE): Avalia os itens de 1 a 8;
- Qualidade da informação (INFOQUAL): Avalia os itens de 9 a 15;

- Qualidade da interface (INTERQUAL): Avalia os itens 16 até 18.

A tabela a baixo representa todos os dados coletados na pesquisa realizada com a equipe core do Mezuro.

	Questões																		
Usuários	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
A	3	3	2	4	3	3	4	3	3	3	4	4	4	4	4	2	2	2	2
B	2	2	1	3	2	2	3	2	2	1	3	3	2	4	2	1	1	1	2
C	1	1	1	2	2	1	2	1	2	2	2	1	3	2	2	3	3	2	1
D	3	3	3	3	2	2	1	3	2	2	2	2	3	5	3	2	2	5	2
E	6	2	4	2	2	1	2	4	3	2	3	4	2	4	2	2	1	1	1
F	3	3	3	3	3	3	3	2	3	2	3	3	3	3	4	2	3	4	3

Tabela 12 – Dados coletados de 6 participantes

Com a coleta dos dados foram levantados a média, desvio padrão e variância de acordo com a classificação das respostas.

	Classificação			
Participantes	OVERALL	SYSUSE	INFOQUAL	INTERQUAL
A	3.11	3.13	3.71	2
B	2.05	2.13	2.43	1
C	1.79	1.38	2	2.67
D	2.63	2.5	2.71	3
E	2.32	2.38	2.86	1.34
F	2.95	2.88	3	3
Média	2.47	2.40	2.79	2.17
Variância	0.514	0.61	0.58	0.86
Desvio Padrão	0.26	0.38	0.33	0.74

Tabela 13 – Média, Variância e Desvio Padrão dos dados

Para melhor evidenciar os resultados estes foram apresentados em modelos gráficos.

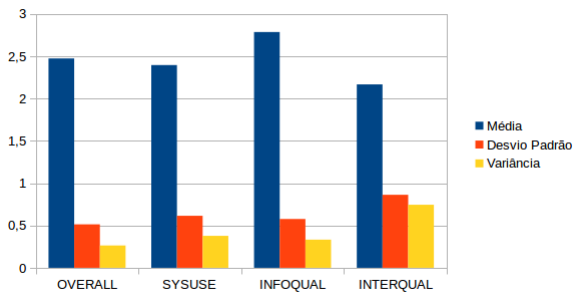


Figura 25 – Gráfico resultados dentro da classificação das respostas.

Com a análise dos dados foi observado que as 4 classificações mantém uma média similar com desvio padrão e variância baixos. Com isso, foi percebido uma tendencia aproximada ao resultado 2.45, onde levando em consideração que o projeto avaliado teve apenas 2 interações com foco em usabilidade, foi obtido bons resultados já que a escala vai de 1 a 7 de concordo fortemente a discordo fortemente. Levando em consideração os resultados também do *checklist* realizado anteriormente foi observado uma melhoria progressiva com as interações de usabilidade sem impactar fortemente na rotina da equipe do projeto. As contribuições foram tanto em nível de *front-end* quanto de *back-end* e os membros com o foco de usabilidade participaram ativamente do desenvolvimento do código-fonte do projeto.

O questionário encontra-se no apêndice na seção .3.

5.3.1.1 Visualização de Software

Como retratado na seção 4.3, a apresentação de informações relacionadas a softwares pode se tornar ineficiente conforme o tamanho e complexidade inerente ao código-fonte aumentam. Por esse motivo, para facilitar o entendimento e compreensão das informações apresentadas, é comum utilizar técnicas de visualização, empregadas cada vez mais em softwares, com o intuito de apoiar o processo de desenvolvimento.

A plataforma Mezuro como plugin do Noosfero, contava com uma técnica de visualização, as coordenadas paralelas, presente na figura 26, um tipo de projeção geométrica que representa bem dados multidimensionais ou com muitos atributos, como é o caso das informações geradas por um processamento do Mezuro.

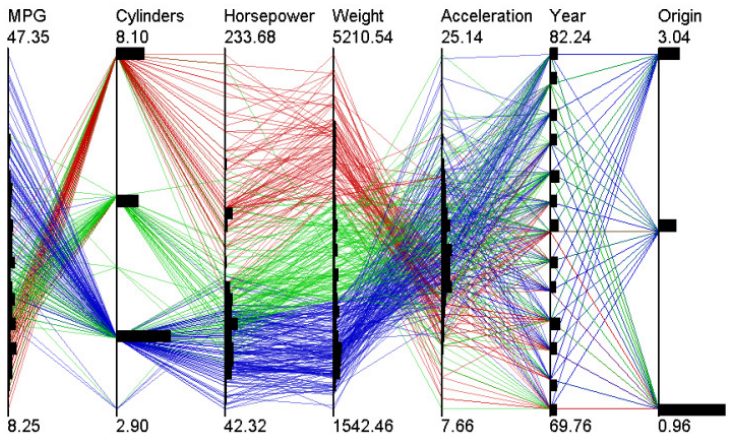


Figura 26 – Coordenadas Paralelas (MCDONNELL; MUELLER, 2008)

Como foi decidido reescrever o código do Mezuro, transformando-o numa aplicação independente, durante boa parte do período de desenvolvimento não havia nenhuma técnica para visualização definida. Como um primeiro passo sobre a visualização de informações no Mezuro, uma possibilidade seria aplicar a técnica das coordenadas paralelas

novamente. Porém, assim como no desenvolvimento de software, em visualização e análise de dados, não existe solução para todos os problemas. É necessário analisar o contexto ou problema e encontrar e julgar a solução que melhor se encaixa.

Pensando nisso, e já que que a técnica das coordenadas paralelas já é conhecida por parte da equipe, neste trabalho buscou-se uma técnica que, assim como as coordenadas paralelas, representasse bem um grande número de atributos. Optou-se então pela técnica do radar, que exibe os dados na forma de um gráfico bidimensional de três ou mais atributos, representados por eixos que iniciam do mesmo ponto como visto no gráfico radar da figura 27.

Embora um sistema que tem como foco a interpretação de dados, visando maximizar a compreensão do usuário a respeito dos valores apresentados, tratar visualização de informação como uma funcionalidade, neste primeiro passo, rumo a aplicação de técnicas de visualização no Mezuro como aplicação independente, a visualização de software será tratada como um requisito não-funcional.

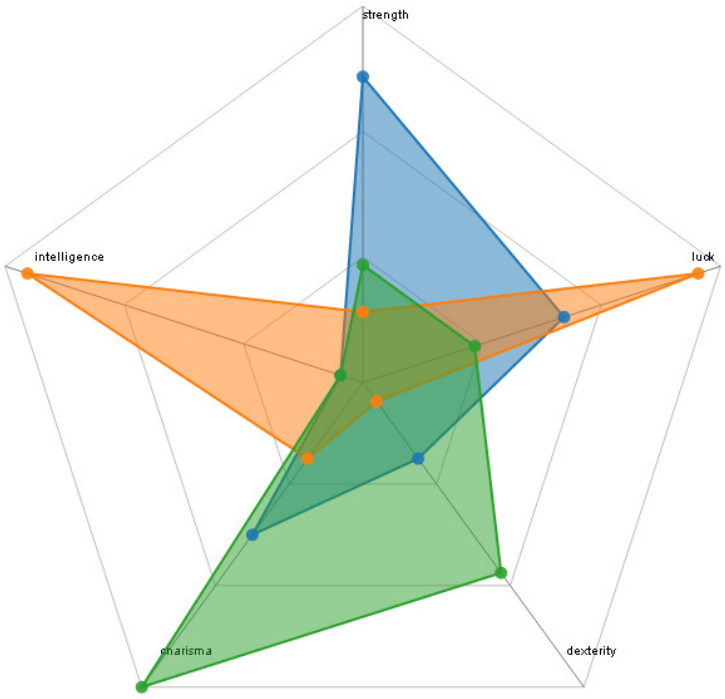


Figura 27 – Gráfico Radar (GRAVES, 2014)

Para aplicar essa técnica ao Mezuro, foi utilizada a biblioteca *D3.js Data-Driven Documents*. D3.js é uma biblioteca Javascript para manipulação de documentos baseados em dados. Ela auxilia na representação dos dados através de tecnologias bem difundidas

como o HTML[22], SVG[23] e CSS[24]. Além disso, a preocupação dessa biblioteca com *web standards* possibilita que o usuário usufrua de todos os recursos dos navegadores web mais modernos, sem amarra-lo a nenhum *framework* proprietário (ABOUT..., 2014).

D3.js, assim como JQuery e o Prototype, é baseado na especificação DOM[25], onde é possível modificar o objeto dinamicamente, ou seja, os dados a serem representados são relacionados a um objeto e consecutivas transformações podem ser aplicadas a esse objeto para satisfazer as necessidades do usuário.

D3.js otimiza a manipulação de documentos em comparação com a especificação DOM, já que essa última utiliza uma abordagem imperativa, a qual requer iterações manuais entre os elementos e armazenamento do estado temporário de cada transformação. D3.js utiliza uma abordagem declarativa, que reduz o número de linhas de código e aumenta a legibilidade.

Além da enorme quantidade de exemplos da utilização dessa biblioteca aplicadas a técnicas de visualização[26] a D3.js está licenciada sob *BSD 3-Clause License*[27] que a torna compatível com a licensa na qual o Mezuro é distribuído, a AGPL V3[28], pois em caso de incompatibilidade não seria possível utilizar os componentes por conta de restrições de distribuição.

Entre os possíveis gráficos gerados após sucessivas transformações dos dados de entrada, está o Radar. A figura 28 representa os dados de um processamento do Mezuro, os quais são lidos de um arquivo de extensão .tsv[29], com valores presentes na tabela 14.

axis	values
Number of Public Methods	32
Number of Methods	45
Number of Modules/Classes	12

Tabela 14 – Dados de entrada

Optou-se por ler os respectivos dados de entrada a partir de um arquivo pois o Mezuro, no momento, passa por uma fase de melhoria de desempenho e não está proces-

22 Linguagem de marcação de hipertexto, do inglês HyperText Markup Language. É uma linguagem de marcação para construir páginas na web

23 Gráficos vetoriais escaláveis, do inglês Scalable Vector Graphics. Linguagem que descreve vetorialmente gráficos bidimensionais

24 Cascading Style Sheets. Linguagem utilizada para definir a apresentação de documentos escritos em linguagens de marcação

25 Document Object Model. Especificação para uma interface, independente de linguagem ou plataforma, onde é possível alterar um documento ou objeto.

26 <https://github.com/mbostock/d3/wiki/Gallery>

27 <http://opensource.org/licenses/BSD-3-Clause>

28 <http://www.gnu.org/licenses/agpl-3.0.html>

29 Tab-separeted values, ou seja, as colunas são separadas por um tab, e os registros por quebras de linha

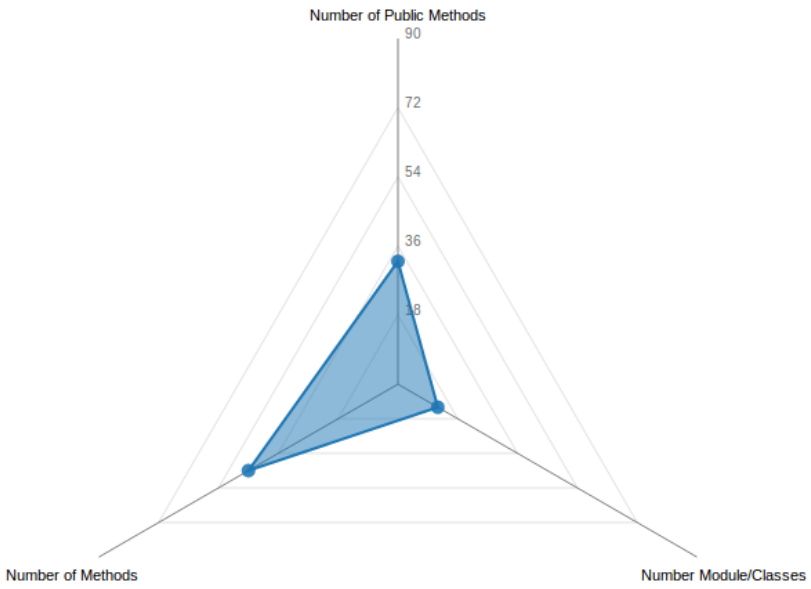


Figura 28 – Gráfico Radar aplicado no Mezuro

sando nem mesmo um repositório por vez. Esses dados representam métricas de mesma natureza, o tamanho do software.

As implementações básicas utilizando a D3.js inserem os dados de entrada no próprio *script* (*hard coding*), com um *array* por exemplo, que apesar da separação de responsabilidades relacionadas a leitura de dados e geração do gráfico em si, não oferece flexibilidade para alteração desses valores. Já utilizando um arquivo *.tsv* permite alterar os valores com facilidade, aumentando a coesão do código (disponível no Anexo .6).

O arquivo *.tsv* representado pela tabela 14, especificamente, representa os valores processados de um repositório, a partir de uma configuração, que conta com cinco métricas de configuração: Total Abstract Classes, Total Lines of Code, Total Number of Methods, Total Number Module/Classes, Average Method LOC, todas com escopo de software.

5.4 Desenvolvimento

O objetivo desta seção é descrever o desenvolvimento das funcionalidades, ou contribuições relacionadas a este trabalho. Serão relatadas as dificuldades encontradas, os métodos utilizados, assim como os processos envolvidos em cada funcionalidade.

5.4.1 Funcionalidade 1: Manter Repositórios

Descrição: Funcionalidade de baixa complexidade, foi a primeira contribuição da equipe à plataforma Mezuro. Essas características fizeram com que a equipe core a selecionasse por ser o primeiro contato dos integrantes da equipe UnB com a tecnolo-

gia envolvida, além da baixa familiridade com o código-fonte já desenvolvido (apesar de poucas entidades implementadas, a gem do Kalibro já era utilizada)

Objetivos: Entre os objetivos dessa funcionalidade estão o registro de um novo repositório (sempre associado a um projeto), edição do repositório criado, visualização e exclusão do mesmo. Além desses objetivos funcionais estão os objetivos de interação com o projeto, como aumento da familiaridade com o código-fonte, interação com a equipe core, a qual é remota (cituada no laboratório CCSL IME/USP), e introdução ao treinamento nas tecnologias envolvidas (arcabouço Ruby on Rails, política de commits no sistema de gestão de configuração GIT).

Relevância para o Mezuro: Apesar de sua simples implementação, os repositórios são importantes para o funcionamento do Mezuro, pois os processamentos que extram as métricas são feitas a partir de um repositório. A relevância dessa funcionalidade não está limitada apenas na sua implementação, mas também porque essa funcionalidade representa um marco para o projeto Mezuro, já que foi a primeira funcionalidade a ser desenvolvida por co-desenvolvedores, ou seja, houve uma descentralização do desenvolvimento da equipe core.

Principais dificuldades: As dificuldades relacionadas a essa funcionalidade estão relacionadas aos problemas comuns que todo desenvolvedor encontra ao interagir com um novo sistema já em desenvolvimento e com uma nova tecnologia. Porém os problemas relacionados ao conhecimento do sistema em sí foram rapidamente suprimidos em função do padrão de codificação já adotado pela equipe core do Mezuro, isso acelerou o entendimento e familiaridade com sistema.

Entre as metodologias utilizadas para essa funcionalidade estão:

5.4.1.1 Divisão das Iterações em Sprints, Sprint Plannig Meeting, Sprint Review Meeting

Descrição: Um ciclo de trabalho de trabalho é organizado em sprint, assim como propõe so Scrum. O tempo de um ciclo pode variar entre uma semana ou duas, dependendo da complexidade e tamanho das atividades a serem desenvolviddas. No início de cada sprint há uma reunião de planejamento (Sprint Planning Meeting) onde são selecionadas estórias de usuários, as quais representam unidades de trabalho a serem desenvolvidas. Se a complexidade da estória for muito alta, ela pode ser quebrada em tarefas, com granularidade mais baixa. No caso da funcionalidade de Manter Reposiórios a sprint foi de pouco mais de duas semanas, pois foi levado em consideração o treinamento na tecnologias utilizadas. Ao final da sprint houve nova reunião (sprint review meeting), para revisar o que foi produzido e avaliar se os objetivos propostos foram alcançados.

Relevância: O planejamento da sprint foi importante para selecionar a melhor funcionalidade a ser desenvolvida pela equipe naquele momento, levando em consideração

a familiaridade com o projeto e conhencimento da tecnologia. O tempo para o desenvolvimento das atividades propostas mais flexível foi importante para conciliar o treinamento com implementação. Já a reunião ao final foi relevante para avaliar o progresso durante o ciclo de trabalho e auxiliar o planejaemento do próximo ciclo.

5.4.1.2 Daily Meeting

Descrição: No caso do projeto Mezuro, formado pela sua equipe core e equipe de co-desenvolvedores, é inviável a realização de daily meetings com os integrantes de todas as equipes todos os dias, asssim como propõe o scrum. Por isso as reuniões eram semanais, sempre no mesmo horário, com objetivo de superação de algum obstáculo encontrado durante a semana.

Relevância: Importante para superação de impasses que surgiram durante o desenvolvimento, além de ampliar a interação com a equipe.

Limitação: O intervalo de uma semana, muitas vezes pareceia muito grande. Com isso os integrantes recorriam a lista de emails para tirar dúvidas, não tão eficientes quanto reuniões.

5.4.2 Funcionalidade 2: Manter Configurações

Descrição: Assim como a funcionalidade de Manter Repositórios, esta funcionalidade também possui baixa complexidade. Seu fluxo de execução e propósitos são similares a funcionalidade descrita anteriormente.

Objetivos: Entre os objetivos dessa funcionalidade estão o registro de uma nova configuração (associação de um para um com a entidade projeto, diferente dos dos repositórios, onde a relação é de muitos para um), edição da configuração criada, visualização e exclusão da mesma.

Relevância para o Mezuro: Como os repositórios, as configurações são importantes para o funcionamento do Mezuro, pois os resultados dos processamentos são baseados na configuração associada ao repositório. Essa entidade é um *container* de outra entidade, as métricas de configuração, as quais representam uma métrica de código-fonte.

Principais dificuldades: As dificuldades encontradas durante o desenvolvimento desta funcionalidade foram menores que a funcionalidade 5.4.1. Isso pois o grau de complexidada das duas são similares e a familiaridade e conhecimento técnico das tecnologias eram maiores nesse ponto do desenvolvimento, comparando com o início das contribuições.

As metodologias utilizadas nesta contribuição foram basicamente as mesmas da funcionalidade 5.4.1.

5.4.3 Funcionalidade 3: Refatoração do Fluxo de Adição de Métricas a uma Configuração

Descrição: Esta funcionalidade teve complexidade de implementação maior em relação às duas funcionalidades citadas nas seções 5.4.1 e 5.4.2. Isso pois surgiram novas tecnologias, como o AJAX[30] e JQuery[31] além das já utilizadas no projeto Mezuro.

Objetivos: Essas novas tecnologias foram importantes pois era necessário melhorar a forma como métricas eram adicionadas a uma configuração. Anteriormente só era possível adicionar uma métrica a cada instante, prencheendo um formulário para cada métrica. Ou seja, se numa dada configuração fossem necessárias oito métricas, por exemplo, o usuário iria levar entre cinco e dez minutos aproximadante para adequar essa configuração. Atualmente, essas novas tecnologias permitem adicionar múltiplas métricas ao mesmo tempo, tornando o fluxo de adição mais eficiente.

Relevância para o Mezuro: Essa refatoração, apesar de não ser crucial para o funcionamento do Mezuro, pode ser considerada uma otimização importante já que melhora a experiência do usuário ao adicionar novas métricas. O estado anterior poderia desestimular o uso do sistema pelos usuários finais.

Principais dificuldades: As dificuldades encontradas durante a refatoração estavam restritas à tecnologia utilizada, pois o fluxo pretendido estava muito bem definido graças aos protótipos de tela desenvolvidos como resultado do ciclos de usabilidade que aconteciam em paralelo às demais contribuições.

5.4.3.1 *Pair Programming*

Para lidar com as dificuldades, durante o desenvolvimento dessa funcionalidade, foi utilizada com mais rigor a prática do *pair programming*, uma prática ágil onde dois programadores trabalham juntos num mesmo computador.

Descrição: Enquanto um programador, piloto, desenvolve o código, o outro, chamado de co-piloto, observa as linhas escritas e palpita sobre melhores soluções. A dupla revesa o posto de piloto a cada intervalo de tempo definido.

Relevância: Além de diminuir a chance de erros, tanto de sintaxe quanto erros lógicos, as chances de obter uma melhor solução são maiores, além do conhecimento das novas tecnologias ser repassado do piloto para o co-piloto e vice-versa durante o pareamento.

[30] Conjunto metodológico de tecnologias utilizadas para fazer requisições assíncronas, tornando as páginas mais interativas

[31] Biblioteca *Javascript*. <*http://jquery.com/*>

5.4.4 Funcionalidade 4: Inserir uma Técnica de Visualização de Software

Descrição: Técnicas de visualização de software são meios ou recursos utilizados para facilitar o entendimento de grande quantiade informação. Geralmente são empregados gráficos, cores, posições e tamanhos diversos para aumentar o poder cognitvo do usuário.

Objetivos: Afim de facilitar o entendimento das informações, as métricas extraídas, e a análise do código são aplicadas técnicas de visualização de informação. Na figura 28 se encontra a técnica aplicada ao Mezuro, neste primeiro momento.

Relevância para o Mezuro: A aplicação de visualização de informação por meio de técnicas de visualização de software é crucial para o Mezuro. Isso pois a plataforma Mezuro lida com um grande número de informações que são difíceis de interpretar sem uma técnica aplicada.

Principais dificuldades: As dificuldades encontradas nesta funcionalidade são relativas aos conceitos relacionados a visualização de software, e ,num primeiro momento, a biblioteca para manipulação de documentos baseados em dados, a D3.js[32] . A definição da melhor técnica está entre essas dificuldades, já que uma técnica não se encaixa bem em todos os casos, necessitando a análise do contexto para a seleção da melhor técnica.

Além das metodologias já utilizadas nas funcionalidades anteriores

5.4.5 Processos

Os processos utilizados durante os ciclos de desenvolvimento foram os mesmos em todas as funcionalidades. Entre eles estão:

- Engenharia de Requisitos de Software: De maneira macro, esse processo é responsável por traduzir as necessidades dos envolvidos, em especificações bem definidas do sistema, os requisitos. Para levantar os requisitos podem ser utilizadas várias técnicas[33], entre elas foram utilizadas o *brainstorming*[34] e a prototipagem de telas.

- Implementação: Transformação da especificação em código, a principal saída deste processo.

- Gerencia de configuração: Processo de apoio ao desenvolvimento de software, fornece controle de versões, controle de mudanças, e auditoria das configurações, ou produtos de trabalho (PRESSMAN, 2011).

[32] <http://d3js.org/>
[33] Entrevistas e questionários, *brainstorming*, prototipagem, cenários
[34] Dinâmica de grupo onde ideias hipotéticas são sugeridas

- Manutenção e evolução de software: Como definido no capítulo 3 a manutenção e evolução de software tem os objetivos de manter a satisfação dos usuários, a disponibilidade, além de levar o sistema a um estado superior, tanto em relação a requisitos funcionais como não-funcionais.

6 Conclusão

A evolução do Mezuro não foi motivada por um motivo isolado. Um conjunto de fatores influenciaram e convenceram a equipe que o melhor para o futuro do projeto Mezuro seria um conjunto de modificações em sua estrutura. Os desenvolvedores estavam restritos aos ultrapassados recursos do Rails 2 e do Ruby 1.8, a qual não recebia mais suporte de seus desenvolvedores.

A equipe almejava por liberdade na tomada de decisões, como por exemplo atualizar o Mezuro para as novas versões do Rails e do Ruby, aproveitando suas melhorias e novos recursos. Porém, estavam subordinados as decisões e andamento do projeto Noosfero. A equipe se deu conta que os recursos relacionados a redes sociais fornecidos não eram necessários para uma ferramenta de monitoramento de código-fonte. Além disso, a manutenibilidade e inserção de novos desenvolvedores ao projeto eram tarefas difíceis, já que o Mezuro crescia bastante e se tornava cada vez mais uma aplicação dentro de outra aplicação, ao invés de um plugin com funcionalidades bem específicas.

Levando em conta todos esses fatores, a equipe do Mezuro decidiu retirá-lo do Noosfero, transformando-o em uma aplicação independente, além de formatar uma comunidade de software livre para atrair novos desenvolvedores. Para consolidar esses fatores, que levaram a evolução do Mezuro, foi elaborado um questionário (encontrado no Apêndice .1) direcionado e equipe que o mantém. Dos fatores que motivaram a evolução, os que mais foram citados pela equipe foram aqueles que limitam sua liberdade ou poder de decisão, que é o fato do Mezuro estar contido dentro do Noosfero, tendo seu desenvolvimento limitado.

É importante destacar que, embora o Mezuro esteja evoluindo para uma aplicação distinta do Noosfero, haverá uma integração, ainda a ser discutida pela comunidade de desenvolvedores, entre essas duas ferramentas futuramente.

6.1 Contribuições

A contribuição deste trabalho inclui os seguintes tópicos relacionados a implementação do Mezuro:

1. “Manter Repositórios”;
2. “Manter Configurações”;
3. Melhorias na composição de Configurações, ou seja, o fluxo de adição de métricas de configuração, assim como a edição das mesmas;

4. Adequação da identidade visual, de acordo com as melhores práticas relacionadas à usabilidade;

5. Aplicação de uma técnica de visualização de software;

Os aspectos relevantes deste trabalho não estão restritos apenas aos pontos práticos ou de implementação citados. A colaboração com um software livre, de acordo com os padrões de contribuição destacados na seção 2.3, passando por vários dos processos[1] que compõem a engenharia de software, além da aplicação de princípios ágeis[2], interação e envolvimento com uma equipe remota também são pontos notáveis deste trabalho.

A técnica aplicada para a inserção do ciclo de usabilidade foi implementada seguindo as práticas de usabilidade para software livre proposto na dissertação de mestrado de Ana Paula Oliveira dos Santos. O grupo de desenvolvimento era composto por 15 membros, divididos em 8 membros da equipe Core, 5 co-desenvolvedores e 2 usuários ativos. Dentro do trabalho realizado foram abordadas todas as fases propostas de forma completa ou parcial das práticas de usabilidade para software livre.

- Identificar necessidades para design centrado em humano: Membro da equipe de desenvolvimento com foco em usabilidade a fim de coletar requisitos de tarefas e necessidades dos usuários;

- Especificar contexto de uso: Realizado através de reuniões diárias e no planejamento da interação. As práticas definidas foram: Criar protótipos de tela, Coletar métricas (variáveis dependentes), Apresentar resultados das análises dos testes e Definição de histórias de usabilidade baseadas nos requisitos fornecidos pelos clientes e no resultado dos testes de usuário;

- Especificar requisitos: Testar protótipos com avaliações heurísticas;

- Avaliar designs: Testar a usabilidade de protótipos com usuários locais e remotos (manipulação de variáveis independentes)

Os resultados obtidos através dos métodos de avaliação heurística, questionário PSSUQ e Checklist foram descritos na subseção 5.3.1. Levando em consideração todos esses resultados realizado anteriormente foi possível observar uma melhoria progressiva com as interações de usabilidade sem impactar fortemente na rotina da equipe do projeto. As contribuições foram tanto em nível de *front-end* quanto de *back-end* e os membros com o foco de usabilidade participaram ativamente do desenvolvimento do código-fonte

[1] Implementação, gerencia de configuração, requisitos e manutenção e evolução de software

[2] *Pair programming*, divisão de iterações em *sprints*, definição das unidades de trabalho dentro de *backlog*

do projeto sem gerar nenhum tipo de deadlock. Isso possibilitou uma melhoria da usabilidade dentro da evolução de uma ferramenta da comunidade de software livre sem uma desestruturação de uma equipe tradicional das comunidades livres e sem a necessidade de gastos adicionais com uma equipe externa.

6.1.1 Limitações

O código presente no apêndice .6.3, referente a técnica de visualização aplicada, o gráfico do radar, se encaixa bem na solução atual que o Mezuro fornece para o processamento de repositórios, que é o processamento de um repositório a cada instante. Porém é desejável que seja apresentado ao usuário o resultado de um ou mais processamentos, para possíveis comparações. Para a visualização dos resultados de mais de um processamento a solução atual não é satisfatória dado que os dados são lidos de um arquivo de extensão *.tsv* onde não é possível separar os dados em grupos que representariam os resultados de cada processamento.

A avaliação de satisfação de usabilidade não pôde ser aplicada aos usuários devido ao fato da plataforma Mezuro estar em um ciclo de desenvolvimento (melhoria) e apesar de estar acessível[3], não é possível completar um ciclo de utilização devido ao módulo de análise do código estar em estado de refatoração em uma mudança de linguagem java para Ruby on Rails.

A quantidade de tecnologias envolvidas para geração de uma nova funcionalidade ou melhoria de um módulo gera uma deadline de aprendizado numa primeira contribuição ao projeto. Porem essa dificuldade é minimizada com a colaboração da equipe para a disseminação de conhecimento.

6.1.2 Continuação

As funcionalidade e melhorias realizadas no projeto possuem um intuito de continuidade e propagação dos seus benefícios ao longo de outras interfaces pela comunidade da plataforma Mezuro. O padrão gerado na primeira interação foi utilizado por desenvolvedores externos da equipe core a fim de aplicar o comportamento em outras telas que não sofreram a interação conforme citado na subseção 5.3.1. O comportamento dinâmico e com a diminuição de fluxo da funcionalidade Configuration deverá ser aplicado nas funcionalidades *New Project*, *New Configuration* e *New Repository*, por necessitarem de um comportamento similar. Enquanto a visualização, essa deverá ser aplicada para as métricas do projeto ao termino da fase de melhoria de desempenho da análise de código conforme na subseção 5.3.1.1. Com o termino dessas aplicações, a contribuição dessas

[3] Plataforma Mezuro: http://mezuro.org/

melhorias deverá afetar diretamente ou indiretamente aproximadamente 80% da interface da plataforma.

6.1.3 Trabalhos Futuros

Pensando em deixar uma perspectiva de trabalhos com foco em usabilidade para uma possível nova interação da comunidade Mezuro uma nova avaliação heurística foi realizada com foco em levantar pontos de melhoria. Os pontos de melhorias levantados foram

- Feedback ao usuário fixo

  Os *feedbacks* implementados na plataforma conforme Figura 29, modificam a estrutura da tela dificultando ao usuário o que diz respeito a facilidade de memorização, algumas das mensagens necessitam ser encerradas e com o acumulo dessas mensagens o agrupamento e a legibilidade da tela vai se perdendo.

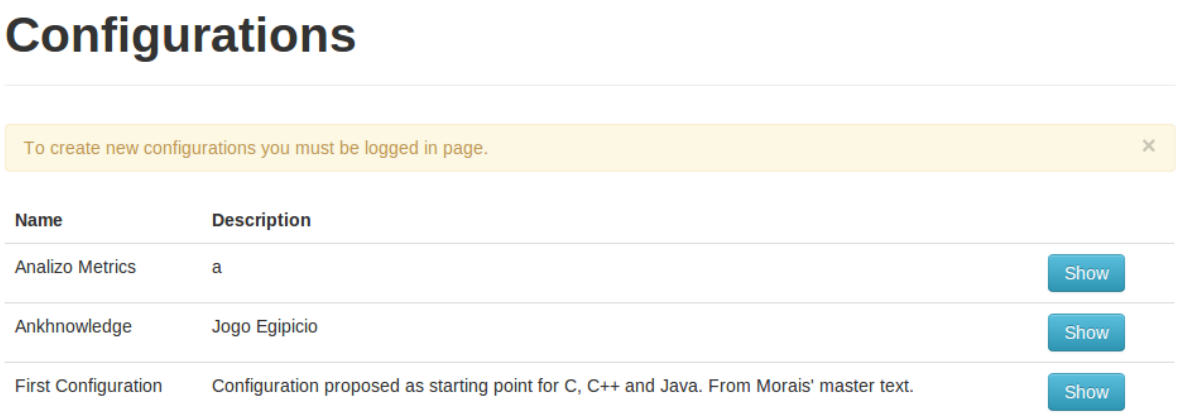


Figura 29 – Feedback ao usuário

  O esperado seria mensagens que não alterasse o agrupamento e legibilidade da tela, sem necessitar uma interação do usuário para encerramento dessas mensagens, tendo uma exibição limitada pelo tempo;

- Feedback com tempo mínimo

  Os *feedbacks* que não são fixos na tela apresentam tempo inapropriado, pois estão baseados no tempo de processamento das funcionalidades, em um máquina com maior processamentos as mensagens praticamente não são percebidas e a leitura delas não são possíveis. Esse tempo deve ser prefixado para um tempo médio da leitura da mensagem;

- Internacionalização da plataforma

A plataforma Mezuro é apresentada totalmente na língua inglesa, por ser uma língua amplamente conhecida, porém há uma necessidade de se possibilitar a criação de internacionalização para que a comunidade possa desenvolver dicionários das mensagens em outras línguas e assim aumentar a abrangência do software.

As contribuições realizadas neste trabalho possuem relativa relevância dada a crescente força adquirida por projetos de software livre, principalmente o Mezuro que visa auxiliar a melhoria da qualidade de códigos-fonte. A metodologia aplicada a essas contribuições também merecem destaque já que foram aplicadas práticas ágeis que são cada vez mais utilizadas no desenvolvimento de software, e de vários processos que são objeto de estudo da engenharia de software.

A distância entre as equipes dos laboratório Lappis - FGA/UnB e CCSL - IME/USP poderia se tornar um grande empecilho ao desenvolvimento e colaboração da equipe da UnB ao Mezuro. Porém a comunicação, uma característica que é priorizada pelos métodos ágeis, supriu essa desvantagem através das reuniões semanais e grupo de emails ativo.

A comunicação entre os membros das equipes é importante em vários aspectos. Entre eles está a aplicação de padrões durante o desenvolvimento, os quais impactam diretamente na manutenibilidade do sistema. Segundo PIGOSKI o esforço para compreensão de programas ou documentos (código-fonte), compreende cerca de 47% a 62% do esforço total de desenvolvimento de um software. Essa afirmação ficou clara durante as contribuições com o Mezuro, principalmente no início, onde a tecnologia era pouco conhecida pelos membros da equipe da UnB. Porém, conforme a tecnologia se tornava mais familiar aos desenvolvedores e a interação e comunicação entre as equipes se tornava maior, o esforço para compreensão diminuía, o que era auxiliado também pela característica da linguagem Ruby, que favorece a manutenibilidade, dada sua legibilidade.

.1 Questionário

Evolução da Plataforma Mezuro

Esta é uma pesquisa relacionada à evolução do Mezuro. Direcionada aos colaboradores dessa plataforma, ela tem como objetivo extrair informações, que serão utilizadas no trabalho de conclusão de curso do aluno Vinícius Vieira, sob orientação do professor Paulo Meirelles.

1. Quais os principais problemas, do ponto de vista do código e da arquitetura, do antigo Mezuro? Por que os mantenedores decidiram escrevê-lo do zero? *

2. Há aspectos do código ou da arquitetura anteriores melhores que do novo código, ou vice-versa? Quais são eles? *

3. Em relação ao código antigo, o novo código fornece (ou está previsto): *

   a) As mesmas funcionalidades

   b) Menos funcionalidades

   c) Mais funcionalidades

4. Em relação a questão anterior. Em caso de mais funcionalidades ou menos, quais são elas?

.2 Respostas

Respostas do Questionário

Respostas 1

1. Quais os principais problemas, do ponto de vista do código e da arquitetura, do antigo Mezuro? Por que os mantenedores decidiram escrevê-lo do zero? *

> A arquitetura de plugins do Noosfero impõe limitações para a estrutura da aplicação, como rotas e também herança de controllers por exemplo. Além disso o problema com tecnologias obsoletas era sério. Ruby 1.8, além de já não receber suporte dos desenvolvedores há algum tempo, tem sérios problemas de performance corrigidos nas versões posteriores. Da mesma forma, a versão 2 do Rails é incompatível com boa parte das gemas atuais.

2. Há aspectos do código ou da arquitetura anteriores melhores que do novo código, ou vice-versa? Quais são eles? *

> Em suma, o novo código é melhor por que resolvemos todos os problemas da resposta anterior.

3. Em relação ao código antigo, o novo código fornece (ou está previsto): *

   a) As mesmas funcionalidades

   b) Menos funcionalidades

   c) Mais funcionalidades

> As mesmas funcionalidades, menos funcionalidades, mais funcionalidades

4. Em relação a questão anterior. Em caso de mais funcionalidades ou menos, quais são elas?

> As novas funcionalidades que temos previstas além das que já existiam estão todas descritas nas issues do github.A menos, não pretendemos fornercer uma rede social com páginas pessoais, muito menos comunidades nem suporte a temas.

Respostas 2

1. Quais os principais problemas, do ponto de vista do código e da arquitetura, do antigo Mezuro? Por que os mantenedores decidiram escrevê-lo do zero? *

> Por antes o Mezuro ser um plugin de um software maior, nossa arquitetura era limitada ao que este software permitia fazer. Sobre o código, éramos obrigados a usar versões antigas de bibliotecas, o que fazia com que nossas soluções ficarem atrasadas com relação com o que está sendo desenvolvido no mundo do Ruby on Rails. Por esses motivos, resolvemos escrever o código do zero, pois agora temos liberdade para mudar a arquitetura sempre que necessário e podemos usar as tecnologias mais novas.

2. Há aspectos do código ou da arquitetura anteriores melhores que do novo código, ou vice-versa? Quais são eles? *

> Antes o Mezuro era um plugin de um sistema maior, portanto a arquitetura era de um plugin e não de uma aplicação rails completa. No novo código, podemos desfrutar de todas as vantagens que o rails fornece. Por outro lado, antes tínhamos muita coisa já implementada que agora temos que refazer.

3. Em relação ao código antigo, o novo código fornece (ou está previsto): *

   a) As mesmas funcionalidades

   b) Menos funcionalidades

   c) Mais funcionalidades

> As mesmas funcionalidades, mais funcionalidades

4. Em relação a questão anterior. Em caso de mais funcionalidades ou menos, quais são elas?

> Ampliar o escopo para analisar código Ruby, melhorar a visualização dos gráficos e dos resultados, notificação de alerta quando uma métrica atingir certo valor considerado ruim, entre outras.

.3 Questionário PSSUQ - Satisfação de Usabilidade Plataforma Mezuro

	1 2 3 4 5 6 7		
1. No geral, estou satisfeito com o quão fácil é usar o sistema.	Concordo Fortemente	○ ○ ○ ○ ○ ○ ○	Discordo Fortemente
Comentários:			
2. Foi fácil utilizar este sistema.	Concordo Fortemente	○ ○ ○ ○ ○ ○ ○	Discordo Fortemente
Comentários:			
3. Eu pude completar as tarefas e cenários de forma **efetiva**, usando este sistema.	Concordo Fortemente	○ ○ ○ ○ ○ ○ ○	Discordo Fortemente
Comentários:			
4. Eu fui capaz de completar as tarefas e cenários de forma **rápida**, usando este sistema.	Concordo Fortemente	○ ○ ○ ○ ○ ○ ○	Discordo Fortemente
Comentários:			
5. Eu fui capaz de completar as tarefas e cenários de forma **eficiente**, usando este sistema.	Concordo Fortemente	○ ○ ○ ○ ○ ○ ○	Discordo Fortemente
Comentários:			
6. Eu me senti confortável usando este sistema.	Concordo Fortemente	○ ○ ○ ○ ○ ○ ○	Discordo Fortemente
Comentários:			
7. Foi fácil aprender a usar este sistema.	Concordo Fortemente	○ ○ ○ ○ ○ ○ ○	Discordo Fortemente
Comentários:			

8. Eu acredito que eu poderia me tornar produtivo rapidamente usando este sistema.	Concordo Fortemente	○ ○ ○ ○ ○ ○ ○	Discordo Fortemente
Comentários:			
9. As mensagens de erros do sistema foram claras o suficiente para me ajudar na correção erros.	Concordo Fortemente	○ ○ ○ ○ ○ ○ ○	Discordo Fortemente
Comentários:			
10. Sempre que eu cometi algum erro, eu pude recuperar de forma fácil e rápida.	Concordo Fortemente	○ ○ ○ ○ ○ ○ ○	Discordo Fortemente
Comentários:			
11. As informações (como ajuda on-line, na tela de mensagens e outros documentos) fornecidas com este sistema foram claras.	Concordo Fortemente	○ ○ ○ ○ ○ ○ ○	Discordo Fortemente
Comentários:			
12. Foi fácil encontrar a informação que eu precisava.	Concordo Fortemente	○ ○ ○ ○ ○ ○ ○	Discordo Fortemente
Comentários:			
13. A informação fornecida pelo sistema é fácil de entender.	Concordo Fortemente	○ ○ ○ ○ ○ ○ ○	Discordo Fortemente
Comentários:			
14. A informação foi eficaz em me ajudar a completar as tarefas e cenários.	Concordo Fortemente	○ ○ ○ ○ ○ ○ ○	Discordo Fortemente
Comentários:			
15. A organização das informações nas telas do sistema é clara.	Concordo Fortemente	○ ○ ○ ○ ○ ○ ○	Discordo Fortemente
Comentários:			
16. A interface deste sistema é agradável.	Concordo Fortemente	○ ○ ○ ○ ○ ○ ○	Discordo Fortemente

Comentários:			
17. Eu gostei de usar a interface deste sistema.	Concordo Fortemente	○ ○ ○ ○ ○ ○ ○	Discordo Fortemente
Comentários:			
18. Este sistema tem todas as funções e capacidades que eu esperava que ele tivesse.	Concordo Fortemente	○ ○ ○ ○ ○ ○ ○	Discordo Fortemente
Comentários:			
19. No geral, estou satisfeito com este sistema.	Concordo Fortemente	○ ○ ○ ○ ○ ○ ○	Discordo Fortemente
Comentários:			

.4 Proposta de práticas de usabilidade ágil para a comunidade de software livre

[4]

Identificar necessidades para design centrado em humano

Equipe-Núcleo como Donos do Produto

Contexto: Equipe de desenvolvimento de software livre composta por desenvolvedores e que não possui especialistas em usabilidade ou UX como membros. Contudo, a equipe-núcleo do projeto percebe a necessidade de compreender melhor os requisitos de negócios e de usabilidade, levando em consideração a visão de clientes e usuários típicos. Problema: Integrar requisitos de negócio com requisitos de usabilidade em um ambiente que não possui especialistas em usabilidade. Principais forças envolvidas:

- Força 1: Necessidade de levantamento de requisitos de negócios com clientes e requisitos de usabilidade com usuários típicos, de modo a integrá-los para o desenvolvimento do sistema.

- Força 2: Não existe garantia de que especialistas em usabilidade ou UX participarão voluntariamente do projeto e/ou não é possível contratá-los. Também não é possível garantir que desenvolvedores voluntários queiram participar dessas atividades.

Solução: Uma adaptação da prática Especialistas em UX como Donos do Produto, da comunidade de métodos ágeis, na qual a equipe-núcleo de um projeto de software livre assumiria o papel de Proprietários do Produto, que levam em consideração a usabilidade do sistema. Dessa forma, podem controlar as contribuições para o projeto, com a visão das necessidades de usuários típicos e clientes.

Caminhos Completos

Contexto: Equipe de desenvolvimento de software livre composta por desenvolvedores e que não possui especialistas em usabilidade ou UX como membros. Contudo, a equipe-núcleo do projeto precisa empregar práticas de usabilidade durante o desenvolvimento do sistema. Problema: Realizar práticas de usabilidade em projetos de software livre que não possuem especialistas em usabilidade ou UX. Principais forças envolvidas:

- Força 1: Necessidade de realizar práticas de usabilidade para pesquisa de usuários, levanta- mento de requisitos e metas de usabilidade, definição de design e avaliações com usuários e clientes.

[4] Seção retirada da dissertação de mestrado Ana Paula Oliveira dos Santos

- Força 2: Não existe garantia de que especialistas em usabilidade ou UX participarão voluntariamente do projeto e/ou não é possível contratá-los.
- Força 3: Desenvolvimento distribuído e participação esporádica de membros.

Solução: Em vez de caminhos paralelos entre equipe de desenvolvimento e de UX, como ocorre na prática Caminhos paralelos da comunidade de métodos ágeis, a equipe de desenvolvimento executa o ciclo completo de DCU para um conjunto específico de funcionalidades, utilizando-se de Pouco design antecipado ou Pouco design antecipado e distribuído para coletar informações. A prática pode ser executada apenas pela equipe-núcleo do projeto ou mesmo com a participação dos demais contribuidores que desejem participar.

Especialista-generalista

Contexto: Equipes de desenvolvimento de software livre, que não possuem especialistas em usabilidade ou UX como membros do time, compostas por desenvolvedores que desejam desenvolver sistemas com melhor usabilidade para usuários típicos. Problema: Ausência de especialistas em usabilidade ou UX na equipe de desenvolvimento do projeto. Principais forças envolvidas:

- Força 1: Não existe garantia de que especialistas em usabilidade ou UX participarão voluntariamente do projeto e/ou não é possível contratá-los.
- Força 2: Desenvolvimento distribuído e participação esporádica de membros.

Solução: Os desenvolvedores da equipe-núcleo do projeto aplicam práticas de usabilidade para entender quem são os usuários típicos do sistema, quais são as suas necessidades e em que contexto o sistema seria utilizado, de modo a incluir essas considerações nos requisitos da aplicação. As pesquisas têm baixa granularidade, ou seja, realiza-se apenas o necessário para o entendimento das funcionalidades da próxima iteração. Os requisitos podem ser definidos por meio da escrita de cartões de histórias de usuários, que são validados com o cliente conforme ocorre em comunidades de métodos ágeis. A documentação detalhada dos requisitos pode ser encontrada nos testes de aceitação, que podem ser acessados por qualquer desenvolvedor do sistema, conforme a prática Testes de aceitação de comunidades de métodos ágeis, o que mantém um relatório atualizado das funcionalidades do sistema que atendem ao comportamento esperado. As metas de usabilidade do sistema também podem ser descritas por meio da proposta de prática Definir metas de usabilidade automáticas.

Especificar contexto de uso

Pouco design antecipado e distribuído

Contexto: Equipe de desenvolvimento de software livre composta por desenvolvedores e que não possui especialistas em usabilidade ou UX como membros. Membros da

equipe-núcleo do projeto e contribuidores encontram-se distribuídos em diversas localidades. Contudo, existe a necessidade de realização de pesquisas presenciais com usuários típicos para melhor compreensão do contexto de uso do sistema. Problema: Utilizar práticas de usabilidade, em ambiente de desenvolvimento de software livre, para especificar contexto de uso de um sistema, onde membros da equipe estão dispersos em vários locais diferentes. Principais forças envolvidas:

- Força 1: Distância física entre membros de uma comunidade de desenvolvimento de software livre.
- Força 2: Necessidade de realização de pesquisas de usabilidade para definição de perfil de usuários típicos e o contexto de uso do sistema.
- Força 3: Possibilitar a participação de voluntários de diversas culturas.

Solução: Equipe-núcleo do projeto é responsável por definir quais são as práticas de usabilidade a serem utilizadas para especificação do contexto de uso de um sistema e também por realizar as práticas presenciais na sua cidade. Membros da equipe, que se encontram dispersos em locais distintos, poderiam aplicar a mesma prática em sua localidade, de modo a obter feedback de usuários com culturas diferentes; por exemplo, replicando testes, sessões de grupos focais ou entrevistas presenciais, em sua região ou país. Desse modo, possibilita-se a obtenção da percepção cultural de vários locais distintos, de modo a explorar o contexto de projetos abertos, no qual podem existir desenvolvedores, usuários, membros da equipe-núcleo e contribuidores em diversas localidades.

Especificar requisitos

Definir metas de usabilidade automáticas

Contexto: Desenvolvimento aberto, distribuído e colaborativo, onde desenvolvedores podem entrar e sair do projeto durante o processo de desenvolvimento. Também não existe uma equipe de usabilidade trabalhando em conjunto com a equipe de desenvolvimento. Problema: Definir metas de usabilidade de modo que todos os desenvolvedores que contribuam com um projeto aberto possam conhecer as metas definidas. Principais forças envolvidas:

- Força 1: Necessidade de definição de metas de usabilidade que atendam às necessidades de usuários típicos.
- Força 2: Possibilitar que todos os desenvolvedores tenham contato diário com as metas de usabilidade definidas
- Força 3: Desenvolvimento distribuído e participação esporádica de membros.

- Força 4: Manter documentação atualizada das metas de usabilidade tratadas pelo sistema.

Solução: Escrita de testes de aceitação automáticos baseados em Behaviour Driven Development (BDD) para definição de metas de usabilidade. Para o contexto de desenvolvimento livre, seria mais eficiente escrever as metas de usabilidade diretamente no ambiente de desenvolvimento do que em documentos separados, que correm o risco de não serem lidos. Sendo assim, conforme grupos de funcionalidades são selecionados para desenvolvimento, descreve-se as metas de usabilidade que precisam ser cumpridas para essas funcionalidades. Membros da equipe-núcleo do projeto podem escrever testes de aceitação automáticos, envolvendo usuários típicos e/ou clientes, o que possibilita documentar o comportamento esperado para a funcionalidade, e também gerar um relatório do funcionamento do sistema, exibindo quais funcionalidades e quais cenários são implementados de acordo com as necessidades dos usuários reais.

Avaliar designs

RITE (Rapid Iterative Testing and Evaluation) para desenvolvedores de software livre

Contexto: Equipes de desenvolvimento de software livre que não possuem especialistas em usabilidade ou UX como membros do time, mas que tem a necessidade de realizar testes de usabilidade com usuários típicos do sistema de modo a desenvolver sistemas com melhor usabilidade. Problema: Possibilitar a identificação e correção de problemas de usabilidade no menor tempo possível durante o desenvolvimento de software livre. Principais forças envolvidas:

- Força 1: Diminuir a distância entre a identificação e a correção de problemas de usabilidade encontrados em testes com usuários.

- Força 2: Não existe garantia de que especialistas em usabilidade ou UX participarão voluntariamente do projeto e/ou não é possível contratá-los.

Solução: O método RITE pode ser aplicado por membros da equipe-núcleo do projeto, não sendo necessário utilizar laboratórios de usabilidade com a aplicação de testes formais. Os desenvolvedores da equipe-núcleo podem observar os usuários utilizando um pequeno conjunto de funcionalidades do sistema e solicitar que falem em voz alta o que estão pensando, enquanto o utilizam (Protocolo Pensando em voz alta). Não seria necessária a criação de relatórios e análises de vídeo dos testes, pois os desenvolvedores que estarão envolvidos na correção dos problemas encontrados podem participar do teste como moderadores ou observadores, de modo que possam obter o conhecimento das melhorias necessárias que precisam ser implementadas. Para documentar problemas referentes a um comportamento

esperado do sistema, os testes de aceitação automáticos, utilizados pela comunidade de métodos ágeis, podem servir como forma de documentação, como também, para verificar se o sistema está realizando a tarefa do modo que se espera. Nesse caso, o relatório de teste de usabilidade seria substituído por testes de aceitação automáticos. A criação dos testes de aceitação, nesse caso, seria feita pelos próprios desenvolvedores que participaram do teste e conhecem o problema a ser resolvido. Um breve brainstorming após a sessão de teste serviria para consolidar as impressões dos membros da equipe envolvidos, possibilitando definir como os problemas serão corrigidos. A correção dos problemas encontrados seria realizada na sequência da realização do teste. Dessa forma, os testes de aceitação serviriam para registrar como corrigir um problema de usabilidade, detectado no teste com usuários típicos, para um determinado cenário de uso do sistema.

.5 Revisão Sistemática: Técnicas de usabilidade em projetos ágeis aplicadas no desenvolvimento de software livre

[5]

O objetivo desta revisão sistemática é analisar relatos de experiência da abordagem de usabilidade nas comunidades ágeis que possam ser aplicadas na comunidade de software livre, com o propósito de identificar e analisar técnicas de usabilidade, com relação à forma que a usabilidade é abordada em projetos ágeis, do ponto de vista de organizações que implementam técnicas de usabilidade no processo de desenvolvimento envolvidas nas iniciativas e no contexto de projetos e estudos de caso reais.

Questão de pesquisa definida para alcançar o objetivo descrito:

1. Quais técnicas de usabilidade são abordadas nas comunidades ágeis e software livre?

Em relação ao escopo da pesquisa, os critérios adotados para selecionar as fontes de busca foram: bases com renome e difundidas na área de TI; que grande quantidade de material publicado e engenhos de busca intuitivos para filtrar os resultados; e possuir relação com o tema a ser pesquisado. Outro ponto levantado foi selecionar bases que pudessem evidenciar o cenário das universidades brasileiras em relação ao tema.

Foram selecionadas as bases de busca Google Academic e Springer Link que possuem máquinas de busca com bom funcionamento e abrangência, além das bibliotecas digitais da UnB e USP.

Foram utilizados os termos em inglês e uma tradução em português. Uma primeira string de busca foi formulada, porém por mais que abordasse aspectos da hipótese continha termos muito genéricos como palavras chave (Usability, Free Software e Agile Method). Obteve assim resultados demasiados, chegando a encontrar mais de mil artigos em apenas uma única base, dificultando a filtragem e busca de artigos adequados e relevantes ao tema.

A fim de obter um resultado mais expressivo em relação a questão levantada, em um domínio mais específico criou-se uma nova String: ((”technical usability” or ”usability practices” or ”interaction design” and usability) and (“free software” or ”open source”) and “agile software development”) com uma variação de sintaxe para se adequar a todas as bases selecionadas: ((”technical usability” OR ”usability practices” OR ”interaction design” AND usability) AND (“free software” OR ”open source”) AND “agile software development”). Os termos da expressão relacionados a usabilidade estão voltados para implementação, aplicação e parte prática. Os outros termos restringem os resultados para as abordagens das comunidades e projetos ágeis e de software livre.

[5] Seção retirada da revisão sistemática realizada no processo de desenvolvimento do trabalho

A seleção dos artigos foi realizada em 5 etapas: 1. Seleção e catalogação preliminar dos artigos coletadas nas fontes a partir da expressão de busca; 2. Filtro de seleção dos artigos relevantes - 1, verificar data de publicação aplicando o critério de seleção ”CS1 – ter sido publicado entre 2007 até 2013”; 3. Filtro de seleção dos artigos relevantes - 2, por meio da ferramenta de busca filtrar resultados por relevância (critérios definidos pela base de busca) e selecionar os 20 primeiros da lista; 4. Filtro de seleção dos artigos relevantes - 3, por meio de análise do resumo (abstract) e aplicando o critério de seleção ”CS2 - possuir informações sobre técnicas de usabilidade na comunidade ágil ou software livre”; 5. Filtro de seleção dos artigos relevantes - 4, por meio da leitura completa dos artigos e aplicando os critérios de seleção ”CS3- possuir evidência de técnicas que foram praticadas, utilizadas, discutidas dentro de um projeto ágil ou de software livre”.

Para os artigos considerados relevantes, os seguintes dados foram extraídos: dados do artigo (título, autor(es), data da publicação, fonte de publicação), resumo do artigo, listagem de como as técnicas de usabilidade foram adotadas e quais técnicas de usabilidade foram abordadas em projetos ágeis ou de software livre.

Os dados coletados nos artigos selecionados foram analisados quantitativa e qualitativamente. A análise qualitativa se deu através de um mapeamento das técnicas encontradas e como elas interagiram. A análise quantitativa resultou em uma lista das técnicas que podem ser aplicadas pelas comunidades de software livre.

Resultado da Revisão sistemática

Com o estabelecimento do protocolo da revisão sistemática, a pesquisa foi executada em Outubro de 2013. Na primeira etapa de seleção dos artigos, a expressão de busca foi executada nas máquinas de buscas Google Academic e Springer Link e verificada nos arquivos das bibliotecas digitais das universidades USP e UnB. No Google Academic, 165 artigos foram obtidas e no Springer Link foram retornadas 55, sendo 19 comuns com o Google Academic. Nas universidades nacionais, a USP retornou 6 resultados e na UnB nenhum artigo que atendesse à expressão de busca foi encontrada.

Na primeira etapa de seleção dos artigos, o critério de seleção CS1, foi aplicado filtrando os artigos de entre o período de 2007 até 2013. No Google Academic, 118 resultados foram obtidos, porém 12 destes eram livros ficando 106 artigos e no Springer Link foram retornadas 52, sendo 16 comuns com o Google Academic e na USP 3 dos resultados eram apenas links sem referência a artigos, restando assim 3 artigos.

Na segunda etapa de seleção dos artigos, os artigos foram filtradas de acordo com a relevância e foram selecionados os 20 primeiros resultando em um total de 43 artigos selecionados.

Na terceira etapa de seleção dos artigos, o resumo (abstract) de cada artigo foi lido. Seguindo o critério de seleção CS2, os artigos que não apresentavam o resumo (abstract)

via engenho de busca foram descartadas automaticamente, assim selecionando dez artigos conforme apresentado na Figura 30.

Springer	Tailoring Usability into Agile Software Development Projects
	A Green Paper on Usability Maturation
	Up-Front Interaction Design in Agile Development
	Second XP Workshop about Dealing with Usability in an Agile Domain
	Values and Assumptions Shaping Agile Development and User Experience Design in Practice
Google Academic	Agile User Stories Enriched with Usability
	Current state of agile user-centered design: A survey
	Evaluating eXtreme scenario-based design in a distributed agile team
	Adaptaçao de metodologias de usabilidade para o contexto de desenvolvimento de software livre
USP	Aplicação de práticas de usabilidade ágil em software livre

Figura 30 – Terceira etapa de seleção dos artigos

Durante a obtenção dos artigos 3 foram eliminados devido a falta de acesso e um por não conter seu desenvolvimento apenas breve descrição e resumo. Na quarta etapa de seleção dos artigos com os 7 artigos remanescentes, foi realizada a leitura completa dos artigos aplicando o critério de seleção CS3 para verificar a adequação junto a questão levantada.

Os artigos selecionados levaram a seleção de 8 técnicas de usabilidade utilizadas na comunidade ágil que podem ser aplicadas na comunidade de software livre. Essas técnicas foram propostas visando os princípios e práticas ágeis e algumas validadas em projetos ágeis, elas podem ser agrupas em 3 tipos, histórias de usabilidade, design centrado no usuário e outros.

Design Centrado no Usuário Santos e Kon (2009) propõe a aplicação do próprio processo de Design Centrado no Usuário (DCU) para levantamento dos usuários e do contexto de uso da metodologia. Os usuários do processo são membros de equipes de software livre, que desejam inserir práticas de usabilidade no processo de desenvolvimento. O contexto de uso são sistemas distribuídos geograficamente, com pessoas trabalhando colaborativamente. Os métodos podem ser aplicados tanto para o início de novos projetos, que tenham como usuários, pessoas não acostumadas a sistemas livres e métodos de usabilidade ou mesmo para a melhoria de sistemas existentes, com respeito à usabilidade.

Santos (2012) em uma nova pesquisa aplicada propôs a aplicação das técnicas de usabilidade de acordo com as fases do DCU, conforme a descrição das técnicas de usabilidade das comunidades de métodos ágeis e de software livre. Para a fase do DCU Criar soluções de design, não foram identificadas propostas de adaptações, sendo portanto, descritas propostas para as fases: Identificar necessidades para design centrado em humano, Especificar contexto de uso, Especificar requisitos e Avaliar Designs conforme tabela 15.

Práticas de usabilidade para Software Livre	
Fases DCU	**Práticas de Usabilidade**
Identificar necessidades para design centrado em humano	Equipe-Núcleo como Donos do Produto Caminhos Completos Especialista-generalista
Especificar contexto de uso	Pouco design antecipado e distribuído
Especificar Requisitos	Definir metas de usabilidade automáticas
Avaliar designs	RITE (Rapid Iterative Testing and Evaluation) para desenvolvedores de software livre

Tabela 15 – Práticas de usabilidade no contexto de Software Livre

Histórias de Usabilidade Já Lee, Judge e McCrickard (2011) defendem a aplicação de definição de histórias de usuário de usabilidade dirigidas pela decisão do responsável por design realizando a prototipação dessas histórias e depois sendo validadas através de testes de usabilidade conforme Figura 31.

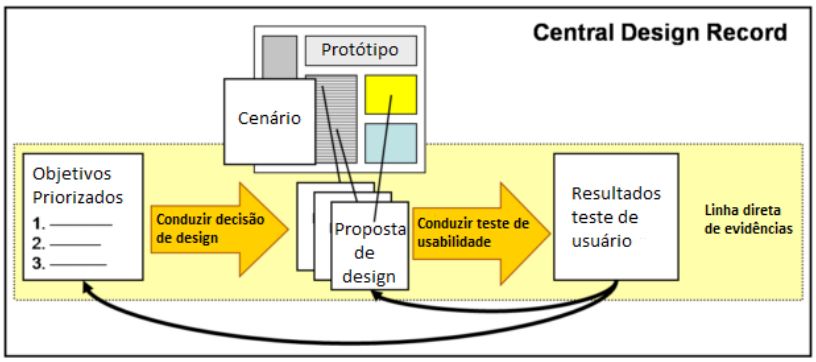


Figura 31 – Ciclo de atividades de usabilidade (LEE; JUDGE; MCCRICKARD, 2011)

As atividades de usabilidade seguem o modelo de ciclo de vida das outras atividades dentro do desenvolvimento ágil com a diferença que o ciclo de usabilidade tem sua iteração primeiro que as outras atividades de desenvolvimento conforme Figura 32:

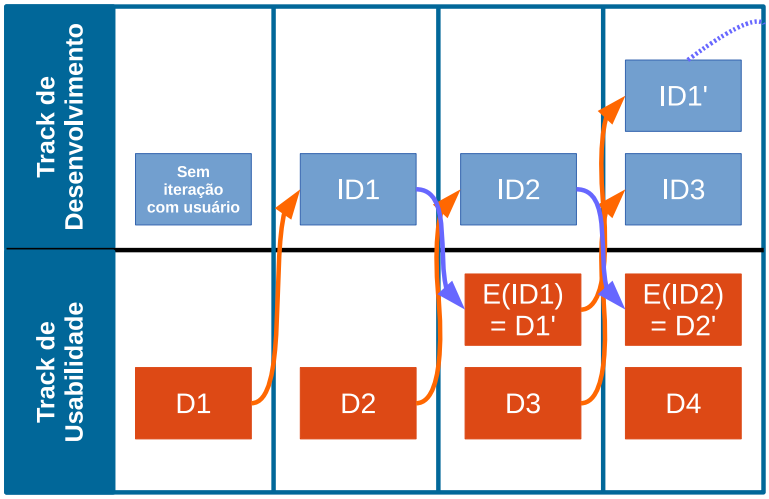


Figura 32 – Ciclo das tarefas de usabilidade x desenvolvimento (LEE; JUDGE; MCCRICKARD, 2011)

Moreno e Yagüe (2012) trás em seu trabalho a inserção da usabilidade por meio da criação de *User Stories* de usabilidade que denominam de *Usability Stories*. Tendo que especificar os recursos funcionais de usabilidade que serão utilizados no projeto, o artigo trás um tabela que é necessária para definir as características dessa funcionalidade.

Status do Sistema	Para informar os usuários sobre a situação interna do sistema
Aviso	Para informar os usuários de qualquer ação com consequências importantes
Longa ação Feedback	Para informar aos usuários que o sistema está processando uma ação que vai demorar algum tempo para completar
Desfazer global	Para desfazer as ações do sistema em vários níveis
Abortar Operação	Para cancelar a execução de uma ação ou toda a aplicação
Abortar comando	Para cancelar a execução de uma tarefa em andamento
Voltar	Para voltar a um estado particular em uma sequência de execução de comando
Entrada de texto estruturada	Para ajudar a prevenir o usuário de cometer erros de entrada de dados
Execução Passo-a-Passo	Para ajudar os usuários a realizar tarefas que requerem diferentes passos com a entrada do usuário e tal entrada correta
Preferências	Para gravar as opções de cada usuário para usar as funções do sistema
Favoritos	Para gravar determinados locais de interesse para o usuário
Ajuda Multinível	Para fornecer diferentes níveis de ajuda para usuários diferentes

Tabela 16 – Mecanismos de usabilidade

Associado a essas características são definidas três maneiras pelas quais a incorporação de usabilidade influencia as histórias de usuário.

1. A adição de novas histórias para representar requisitos diretamente derivados usabilidade.
2. Adição ou modificação de tarefas em histórias de usuários existentes. Isto significa que algumas ações decorrentes de limitações de usabilidade devem ser realizadas por um usuário existente na história. Esta tarefa pode ser tão simples ou detalhados, conforme necessário.
3. Adição ou modificação de critérios de aceitação. Estes critérios de aceitação aparecem porque a funcionalidade da história de usuário precisa incluir algumas ações específicas para modificar o ambiente operacional.

Com isso os autores realizaram um mapeamento entre os mecanismos de usabilidade e as ações a serem tomadas finalizando a proposta de como a usabilidade deve ser abordada por equipes ágeis.

	Nova Tarefa	Modificar Tarefa	Novo Critério de Aceitação	Modificar Critério de Aceitação	Nova História de Usabilidade	Nova História de Usuário
Status do Sistema		X	X	X	X	
Aviso	X		X	X	X	
Longa ação		X	X			X
Abortar Operação		X	X			
Abortar comando	X	X	X			
Voltar	X	X	X			
Entrada de texto estruturada		X	X			
Execução Passo-a-Passo	X		X	X		
Preferências						X
Favoritos		X	X		X	
Ajuda Multinível		X	X		X	

Tabela 17 – Mapeamento entre os mecanismos de usabilidade e ações

Outros Hussain, Slany e Holzinger (2009) não evidenciam proposta de técnicas de usabilidade, eles realizam uma busca sobre o que está sendo utilizado nas comunidades ágeis e trazem como resultados:

- Técnicas de IHC (Iteração Humano Computador).As técnicas mais utilizadas são de baixa fidelidade de protótipos, seguido de projetos conceituais, observa-se estudos adicionais dos usuários, avaliações de especialistas de usabilidade, estudos de campo, testes rápidos iterativo e testes de usabilidade em laboratório, respectivamente;
- Técnicas de IHC utilizadas com menor frequência. Contém investigação contextual, os testes não-formais de usabilidade (em pessoa), design participativo, especificações de interface do usuário completas, de alta fidelidade de protótipos e inquérito model-driven.

Como conclusão observam que o uso de protótipos de baixa fidelidade, avaliações de especialistas de usabilidade, e teste rápido iterativo é facilmente alocado dentro das iterações rápidas de métodos ágeis (HUSSAIN; SLANY; HOLZINGER, 2009).

Ferreira e Biddle (2007b) defendem o foco em design de iteração ágil antes do início do desenvolvimento de software dentro da iteração. Para eles a inserção de técnicas de design no início do ciclo de desenvolvimento podem trazer diversos benefícios sem

atrapalhar ou prejudicar a equipe. Para validar essa teoria utilizaram de entrevistas com designers de interação e desenvolvimento de software em várias equipes ágeis e acompanharam a aplicação desta dentro do projeto, em um contexto de 3 projetos de desenvolvimento ágil (FERREIRA; NOBLE; BIDDLE, 2007b).

.6 Código-Fonte

.6.1 Feature Repositórios

.6.1.1 *Controller*

```
1  include OwnershipAuthentication
2
3  class RepositoriesController < ApplicationController
4    before_action :authenticate_user!, except: [:show, :state]
5    before_action :project_owner?, only: [:new, :create]
6    before_action :repository_owner?, only: [:edit, :update, :destroy, :
       process_repository]
7    before_action :set_repository, only: [:show, :edit, :update, :destroy,
        :state, :process_repository]
8
9    # GET /projects/1/repositories/1
10   # GET /projects/1/repositories/1.json
11   # GET /projects/1/repositories/1/modules/1
12   # GET /projects/1/repositories/1/modules/1.json
13   def show
14     set_configuration
15   end
16
17   # GET projects/1/repositories/new
18   def new
19     @project_id = params[:project_id]
20     @repository = Repository.new
21     @repository_types = Repository.repository_types
22   end
23
24   # GET /repositories/1/edit
25   def edit
26     @project_id = params[:project_id]
27     @repository_types = Repository.repository_types
28   end
29
30   # POST /projects/1/repositories
31   # POST /projects/1/repositories.json
32   def create
33     @repository = Repository.new(repository_params)
34     @repository.project_id = params[:project_id]
35     respond_to do |format|
36       create_and_redir(format)
37     end
38   end
39
40   # PUT /projects/1/repositories/1
```

```
41  # PUT /projects/1/repositories/1.json
42  def update
43    respond_to do |format|
44      if @repository.update(repository_params)
45        format.html { redirect_to(project_repository_path(params[:
              project_id], @repository.id), notice: 'Repository was
              successfully updated.') }
46        format.json { head :no_content }
47      else
48        failed_action(format, 'edit')
49      end
50    end
51  end
52
53  # DELETE /projects/1/repositories/1
54  # DELETE /projects/1/repositories/1.json
55  def destroy
56    @repository.destroy
57    respond_to do |format|
58      format.html { redirect_to project_path(params[:project_id]) }
59      format.json { head :no_content }
60    end
61  end
62
63  # POST /projects/1/repositories/1/state
64  def state
65    if params[:last_state] != 'READY'
66      if params[:day].nil?
67        @processing = @repository.last_processing
68      else
69        year, month, day = params[:year], params[:month], params[:day]
70        @processing = Processing.processing_with_date_of(@repository.id,
              "#{year}-#{month}-#{day}")
71      end
72
73      respond_to do |format|
74        if @processing.nil?
75          format.js { render action: 'unprocessed' }
76        elsif @processing.state == 'READY'
77          format.js { render action: 'load_ready_processing' }
78        else
79          format.js { render action: 'reload_processing' }
80        end
81      end
82    else
83      head :ok, :content_type => 'text/html' # Just don't do anything
84    end
```

```
85    end
86
87    # GET /projects/1/repositories/1/process
88    def process_repository
89      @repository.process
90      set_configuration
91      respond_to do |format|
92        format.html { redirect_to project_repository_path(@repository.
              project_id, @repository.id) }
93      end
94    end
95
96  private
97    # Duplicated code on create and update actions extracted here
98    def failed_action(format, destiny_action)
99      @project_id = params[:project_id]
100     @repository_types = Repository.repository_types
101
102     format.html { render action: destiny_action }
103     format.json { render json: @repository.errors, status: :
            unprocessable_entity }
104   end
105
106   # Use callbacks to share common setup or constraints between actions.
107   def set_repository
108     @repository = Repository.find(params[:id].to_i)
109   end
110
111   def set_configuration
112     @configuration = MezuroConfiguration.find(@repository.
            configuration_id)
113   end
114
115   # Never trust parameters from the scary internet, only allow the white
          list through.
116   def repository_params
117     params[:repository]
118   end
119
120   # Code extracted from create action
121   def create_and_redir(format)
122     if @repository.save
123       format.html { redirect_to project_repository_process_path(
              @repository.project_id, @repository.id), notice: 'Repository
              was successfully created.' }
124     else
125       failed_action(format, 'new')
```

```
126     end
127   end
128
129 end
```

.6.2 Feature Configuration

.6.2.1 *Controller*

```
1  class MetricConfigurationsController <
       BaseMetricConfigurationsController
2    def choose_metric
3      @mezuro_configuration_id = params[:mezuro_configuration_id].to_i
4      @metric_configuration_id = params[:metric_configuration_id].to_i
5      @base_tools = KalibroGatekeeperClient::Entities::BaseTool.all
6      @exist_metric = params[:exist_metric]
7    end
8
9    def new
10     super
11     @metric_configuration = MetricConfiguration.new
12     @metric_configuration.configuration_id = params[:
           mezuro_configuration_id].to_i
13     @metric_configuration.base_tool_name = params[:base_tool_name]
14   end
15
16   def create
17     @configuration_metrics = params[:configuration_metrics_names]
18
19     @configuration_metrics.each do |configuration_metric|
20       code = automatic_code configuration_metric
21       @metric_configuration = MetricConfiguration.new
22       @metric_configuration.configuration_id = params[:
             mezuro_configuration_id].to_i
23       @metric_configuration.base_tool_name = "Analizo"
24       @metric_configuration.metric = KalibroGatekeeperClient::Entities::
             BaseTool.find_by_name("Analizo").metric(configuration_metric.
             to_s)
25       @metric_configuration.code = code.to_s
26       @metric_configuration.weight = "0"
27       @metric_configuration.aggregation_form = "AVERAGE"
28       @metric_configuration.reading_group_id = 1
29       @metric_configuration.save
30     end
31     respond_to do |format|
32       create_and_redir(format)
33     end
34   end
```

```
35
36  def automatic_code(metric_name)
37    array = metric_name.split(" ", 3);
38    automatic_code = ""
39    array.each do |metric_word|
40      automatic_code = automatic_code << metric_word[0...2]
41    end
42    automatic_code
43  end
44
45  def edit
46    #FIXME: set the configuration id just once!
47    @mezuro_configuration_id = params[:mezuro_configuration_id]
48    @metric_configuration.configuration_id = @mezuro_configuration_id
49  end
50
51  def update
52    puts '=' * 100
53    puts 'I reached the action controller!'
54    respond_to do |format|
55      @metric_configuration.configuration_id = params[:
         mezuro_configuration_id]
56      if @metric_configuration.update(metric_configuration_params)
57        format.html { redirect_to(mezuro_configuration_path(
             @metric_configuration.configuration_id), notice: 'Metric
             Configuration was successfully updated.') }
58        format.json { head :no_content }
59      else
60        failed_action(format, 'edit')
61      end
62    end
63  end
64
65  def destroy
66    @metric_configuration.destroy
67    respond_to do |format|
68      format.html { redirect_to mezuro_configuration_path(params[:
         mezuro_configuration_id]) }
69      format.json { head :no_content }
70    end
71  end
72
73  protected
74
75  def metric_configuration
76    @metric_configuration
77  end
```

```
78
79    def update_metric_configuration (new_metric_configuration)
80      @metric_configuration = new_metric_configuration
81    end
82
83    private
84
85    # Duplicated code on create and update actions extracted here
86    def failed_action(format, destiny_action)
87      @mezuro_configuration_id = params[:mezuro_configuration_id]
88
89      format.html { render action: destiny_action }
90      format.json { render json: @metric_configuration.errors, status: :
            unprocessable_entity }
91    end
92
93    #Code extracted from create action
94    def create_and_redir(format)
95      if @metric_configuration.save
96        format.html { redirect_to mezuro_configuration_path(
              @metric_configuration.configuration_id), notice: 'Metric
              Configuration was successfully created.' }
97      else
98        failed_action(format, 'new')
99      end
100   end
101 end
```

```
1  include OwnershipAuthentication
2
3  class MezuroConfigurationsController < ApplicationController
4    before_action :authenticate_user!, except: [:index, :show]
5    before_action :mezuro_configuration_owner?, only: [:edit, :update, :
         destroy]
6
7    # GET /mezuro_configurations/new
8    def new
9      @mezuro_configuration = MezuroConfiguration.new
10   end
11
12   # GET /mezuro_configurations
13   # GET /mezuro_configurations.json
14   def index
15     @mezuro_configurations = MezuroConfiguration.all
16   end
17
18   # POST /mezuro_configurations
19   # POST /mezuro_configurations.json
```

```
20  def create
21    @mezuro_configuration = MezuroConfiguration.new(
          mezuro_configuration_params)
22    respond_to do |format|
23      create_and_redir(format)
24    end
25  end
26
27  # GET /mezuro_configurations/1
28  # GET /mezuro_configurations/1.json
29  def show
30    set_mezuro_configuration
31    @mezuro_configuration_metric_configurations = @mezuro_configuration.
          metric_configurations
32  end
33
34  # GET /mezuro_configurations/1/edit
35  # GET /mezuro_configurations/1/edit.json
36  def edit
37    set_mezuro_configuration
38  end
39
40
41  def update
42    set_mezuro_configuration
43    if @mezuro_configuration.update(mezuro_configuration_params)
44      redirect_to(mezuro_configuration_path(@mezuro_configuration.id))
45    else
46      render "edit"
47    end
48  end
49
50  # DELETE /mezuro_configurations/1
51  # DELETE /mezuro_configurations/1.json
52  def destroy
53    set_mezuro_configuration
54    current_user.mezuro_configuration_ownerships.
          find_by_mezuro_configuration_id(@mezuro_configuration.id).destroy
55    @mezuro_configuration.destroy
56    respond_to do |format|
57      format.html { redirect_to mezuro_configurations_url }
58      format.json { head :no_content }
59    end
60  end
61
62  private
63  # Use callbacks to share common setup or constraints between actions.
```

```
64    def set_mezuro_configuration
65      @mezuro_configuration = MezuroConfiguration.find(params[:id])
66    end
67
68    # Never trust parameters from the scary internet, only allow the white
          list through.
69    def mezuro_configuration_params
70      params[:mezuro_configuration]
71    end
72
73    # Extracted code from create action
74    def create_and_redir(format)
75      if @mezuro_configuration.save
76        current_user.mezuro_configuration_ownerships.create
              mezuro_configuration_id: @mezuro_configuration.id
77
78        format.html { redirect_to mezuro_configuration_path(
              @mezuro_configuration.id), notice: 'mezuro configuration was
              successfully created.' }
79        format.json { render action: 'show', status: :created, location:
              @mezuro_configuration }
80      else
81        format.html { render action: 'new' }
82        format.json { render json: @mezuro_configuration.errors, status: :
              unprocessable_entity }
83      end
84    end
85  end
```

.6.3 Gráfico Radar

Nesta subseção se encontra o código-fonte referente a aplicação da técnica de visualização do Radar. O script abaixo encontra-se na *view show.html* da entidade *Repository*, a qual é responsável pela abertura do *modal*, que é a estrutura visual gerada, por cima da *view* principal, para apresentar o gráfico.

```
1
2   <script type="text/javascript">
3
4   $('#my-modal').on('show', function () {
5
6       $(this).find('.modal-body').css({
7         width:'auto',
8         height:'auto',
9         'max-height':'100%'});
10  });
11
```

```
var w = 500,
h = 500;

var colorscale = d3.scale.category10();

//Data
var myData = new Array([]);

d3.tsv("/data.tsv", type, function(error, data) {

  for (var i = 0; i < data.length; i++) {
    var map = {};
    map["axis"] = data[i].axis;
    map["value"] = data[i].value;
    myData[0][i] = map;
  }
  console.log(myData[0]);

  //Call function to draw the Radar chart
  RadarChart.draw("#chart", myData, mycfg);
});

function type(d) {
  d.value = +d.value; // coerce to number
  return d;
}

//Options for the Radar chart, other than default
var mycfg = {
  w: w,
  h: h,
  maxValue: 150,
  levels: 5,
  ExtraWidthX: 200
}

</script>
```

O código abaixo é referente a estrutura que contem o gráfico, o chamado *modal*.

```
<div class="modal hide fade" id="my-modal" title="Radar Chart">
  <div class="modal-header">
    <button aria-hidden="true" class="close" data-dismiss="modal" type="
       button">x</button>
    <h3 id="myModalLabel">Visualization</h3>
  </div>
  <div class="modal-body" id="chart">
    <h4>Radar Chart</h4>
```

```
  </div>
  <div class="modal-footer">
    <button aria-hidden="true" class="btn" data-dismiss="modal">Close</
       button>
  </div>
</div>
```

```
<%= link_to "Visualize", "#my-modal", :class => "btn", "data-toggle" =>
   "modal" %>
```

.6.4 CSS formwithtooltip

```
.form-table {
  margin: 0 !important;
  background: #fff;
  -webkit-box-shadow: 0 1px 2px rgba(0, 0, 0, 0.075);
  box-shadow: 0 1px 2px rgba(0, 0, 0, 0.075);
}

.form-row {
  float: left;
  width: 100%;
  border-bottom: 1px solid #f2f2f2;
  border-top: 1px solid #e3e3e3;
  display: inline-flex;
}

.field-container {
  position: relative;
  float: left;
  width: 653px;
  padding: 20px;
}

.help-container {
  float: left;
  width: 400px;
  padding: 20px;
  background: #f5f5f5;
  border-left: 1px solid #e3e3e3;
  border-right: 1px solid #e3e3e3;
}

.text-field {
  width: 95% !important;
  margin: 0;
  background: #f9f9f9;
```

```
36 }
37
38 .text-area {
39   width: 95% !important;
40   margin: 0;
41 }
42
43 .field-container select {
44   width: 99% !important;
45   margin: 0;
46 }
```

Referências

ABOUT D3.js. 2014. <http://d3js.org/>. Accessed: 10/06/2014. Citado na página 82.

BARROS, A. J. d. S.; LEHFELD, N. A. d. S. *Fundamentos de metodologia: um guia para a iniciação científica. ampl.* [S.l.]: São Paulo: Makron Books, 2000. Citado na página 21.

BASTIEN, J. C.; SCAPIN, D. L. et al. Ergonomic criteria for the evaluation of human-computer interfaces. 1993. Citado na página 47.

BRAZ, F. A. *Instrumentação da Análise e Projeto de Software Seguro Baseada em Ameaças e Padrões*. Tese (Doutorado) — Universidade de Brasília, Faculdade de Tecnologia, Abril 2009. Citado na página 33.

BUXTON, J. N.; RANDELL, B. *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee*. [S.l.]: NATO Science Committee; available from Scientific Affairs Division, NATO, 1970. Citado na página 36.

CORBUCCI, H. *Métodos ágeis e software livre: um estudo da relação entre estas duas comunidades*. 2011. Citado 2 vezes nas páginas 24 e 25.

CYBIS, W.; BETIOL, A. H.; FAUST, R. *Ergonomia e usabilidade*. [S.l.: s.n.], 2010. Citado 5 vezes nas páginas 45, 47, 54, 55 e 69.

DIAS, M. S.; VIEIRA, M. E. Software architecture analysis based on statechart semantics. In: IEEE. *Software Specification and Design, 2000. Tenth International Workshop on*. [S.l.], 2000. p. 133–137. Citado na página 37.

DYBÅ, T.; DINGSØYR, T. Empirical studies of agile software development: A systematic review. *Information and software technology*, Elsevier, v. 50, n. 9, p. 833–859, 2008. Citado 2 vezes nas páginas 29 e 30.

ERGONOMIA, A. B. de. *O que é ergonomia @ONLINE*. 2013. Disponível em: <http://www.abergo.org.br/>. Citado na página 46.

FERNANDEZ-RAMIL, J. et al. Empirical studies of open source evolution. In: *Software evolution*. [S.l.]: Springer, 2008. p. 263–288. Citado 2 vezes nas páginas 13 e 34.

FERREIRA, J.; NOBLE, J.; BIDDLE, R. Up-front interaction design in agile development. In: *Agile Processes in Software Engineering and Extreme Programming*. [S.l.]: Springer, 2007. p. 9–16. Citado na página 48.

FERREIRA, J.; NOBLE, J.; BIDDLE, R. Up-front interaction design in agile development. In: *Agile processes in software engineering and extreme programming*. [S.l.]: Springer, 2007. p. 9–16. Citado 2 vezes nas páginas 111 e 112.

FILHO, A. M. d. S. Requisitos não funcionais: Critérios para análise arquitetural. *Engenharia de Software Magazine*, 2010. Citado na página 45.

GARLAN, D.; SHAW, M. An introduction to software architecture. *Advances in software engineering and knowledge engineering*, Singapore, v. 1, p. 1–40, 1993. Citado na página 33.

GERMOGLIO, G. *Fundamentos da arquitetura de software*. 2010. Citado 2 vezes nas páginas 37 e 42.

GODFREY, M. W.; TU, Q. Evolution in open source software: A case study. In: IEEE. *Software Maintenance, 2000. Proceedings. International Conference on*. [S.l.], 2000. p. 131–142. Citado 4 vezes nas páginas 11, 33, 35 e 36.

GRAVES, A. *Example Radar Chart*. 2014. <http://graves.cl/radar-chart-d3/>. Accessed: 12/06/2014. Citado 2 vezes nas páginas 11 e 81.

HEER, J.; SHNEIDERMAN, B. Interactive dynamics for visual analysis. *Queue*, ACM, v. 10, n. 2, p. 30, 2012. Citado na página 56.

HUSSAIN, Z.; SLANY, W.; HOLZINGER, A. *Current state of agile user-centered design: A survey*. [S.l.]: Springer, 2009. Citado na página 111.

INFORMáTICA, L. de Utilizabilidade da. *ErgoList - Checklist @ONLINE*. 2013. Disponível em: <http://www.labiutil.inf.ufsc.br/ergolist/check.htm>. Citado na página 71.

ISO, N. Iec 9126-1. *Engenharia de software-Qualidade de produto. Parte*, v. 1, 2003. Citado na página 58.

KEIM, D. A. Information visualization and visual data mining. *Visualization and Computer Graphics, IEEE Transactions on*, IEEE, v. 8, n. 1, p. 1–8, 2002. Citado na página 55.

LEE, J. C.; JUDGE, T. K.; MCCRICKARD, D. S. Evaluating extreme scenario-based design in a distributed agile team. In: ACM. *PART 1———-Proceedings of the 2011 annual conference extended abstracts on Human factors in computing systems*. [S.l.], 2011. p. 863–877. Citado 6 vezes nas páginas 11, 12, 49, 50, 108 e 109.

LEWIS, J. R. Psychometric evaluation of the post-study system usability questionnaire: The pssuq. In: SAGE PUBLICATIONS. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*. [S.l.], 1992. v. 36, n. 16, p. 1259–1260. Citado na página 77.

LIDA, I. *Ergonomia: projeto e produção. 2ª edição*. [S.l.: s.n.], 2005. Citado na página 46.

LIMAA, M. S. de; SOARESB, B. G.; BACALTCHUKC, J. Psiquiatria baseada em evidências. *Rev Bras Psiquiatr*, SciELO Brasil, v. 22, n. 3, p. 142–6, 2000. Citado na página 49.

MARTINS, R. M. *Técnicas de Visualização para Avaliação e Melhoria de Qualidade de Software Livre e Aberto*. Tese (Doutorado) — Instituto de Ciências Matemáticas e de Computação - ICMC-USP, 2012. Citado 3 vezes nas páginas 20, 56 e 57.

MCDONNELL, K. T.; MUELLER, K. Illustrative parallel coordinates. In: WILEY ONLINE LIBRARY. *Computer Graphics Forum*. [S.l.], 2008. v. 27, n. 3, p. 1031–1038. Citado 2 vezes nas páginas 11 e 80.

MEIRELLES, P. et al. *Mezuro: A Source Code Tracking Platform*. Tese (Doutorado) — FLOSS Competence Center – University of São Paulo, Instituto de Matemática e Estatística, Maio 2010. Citado 3 vezes nas páginas 11, 61 e 65.

MEIRELLES, P. R. et al. Crab: Uma ferramenta de configuração e interpretação de métricas de software para avaliação de qualidade de código. *XXIII SBES-Simpósio Brasileiro de Engenharia de Software (XVI Sessão de Ferramentas). Citado na pág*, v. 33, 2009. Citado na página 61.

MEIRELLES, P. R. M. *Monitoramento de métricas de código-fonte em projetos de software livre*. Tese (Doutorado) — Universidade de São Paulo, 2013. Citado na página 23.

MEIRELLES, P. R. M. *Monitoramento de métricas de código-fonte em projetos de software livre*. Tese (Doutorado) — Instituto de Matemática e Estátistica – Universidade de São Paulo (IME/USP), 2013. Citado 4 vezes nas páginas 23, 25, 26 e 61.

MEJIA, A. *Ruby on Rails Architectural Design*. 2011. Disponível em: <http://adrianmejia.com/blog/2011/08/11/ruby-on-rails-architectural-design/>. Citado 2 vezes nas páginas 11 e 42.

MENS, T. et al. Challenges in software evolution. In: IEEE. *Principles of Software Evolution, Eighth International Workshop on*. [S.l.], 2005. p. 13–22. Citado na página 35.

MORENO, A. M.; YAGÜE, A. Agile user stories enriched with usability. In: *Agile Processes in Software Engineering and Extreme Programming*. [S.l.]: Springer, 2012. p. 168–176. Citado 2 vezes nas páginas 50 e 109.

NICHOLS, D. M.; TWIDALE, M. B. Usability processes in open source projects. *Software Process: Improvement and Practice*, Wiley Online Library, v. 11, n. 2, p. 149–162, 2006. Citado na página 20.

NIELSEN, J. *Usability engineering*. [S.l.]: Academic Press Limited, 1994. Citado na página 46.

NIELSEN, J. Severity ratings for usability problems. *Papers and Essays*, 1995. Citado na página 54.

PAULK, M. C. et al. *The capability maturity model: Guidelines for improving the software process*. [S.l.]: Addison-wesley Reading, MA, 1994. Citado na página 58.

PIGOSKI, T. M. *Practical software maintenance: best practices for managing your software investment*. [S.l.]: John Wiley & Sons, Inc., 1996. Citado na página 93.

PRESSMAN, R. S. *Engenharia de software*. [S.l.]: McGraw Hill Brasil, 2011. Citado na página 87.

QUALIPSO Project - O Projeto Qualipso. 2009. Disponível em: <http://qualipso.icmc.usp.br/>. Citado na página 62.

SANTOS, A. P. *Aplicação de práticas de usabilidade ágil em software livre*. 2012. Citado 4 vezes nas páginas 21, 50, 53 e 107.

SANTOS, A. P.; KON, F. Adaptação de metodologias de usabilidade para o contexto de desenvolvimento de software livre. 2009. Citado 2 vezes nas páginas 49 e 107.

SCHWABER, K.; BEEDLE, M. *Agile software development with Scrum*. [S.l.]: Prentice Hall Upper Saddle River, 2002. Citado na página 29.

SHAW, M.; GARLAN, D. Software architecture: perspectives on an emerging discipline. Prentice Hall, 1996. Citado na página 36.

SHNEIDERMAN, B.; BEN, S. *Designing The User Interface: Strategies for Effective Human-Computer Interaction, 4/e (New Edition)*. [S.l.]: Pearson Education India, 2003. Citado na página 46.

STALLMAN, R. M.; GAY, J. *Free software, free society: Selected essays of Richard M. Stallman*. [S.l.]: CreateSpace, 2009. Citado na página 24.

ISO 25010, SYSTEMS and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. 2010. Citado na página 45.

TERCEIRO RODRIGO ROCHA, C. C. A. *Padrões para Contribuição em Projetos de Software Livre*. 2013. <https://gitorious.org/flosspapers/free-software-patterns/>. [Online; accessed 04-June-2014]. Citado na página 26.

THOMAS, M. P. Why free software has poor usability, and how to improve it. *Computing & Internet, Usability*, 2008. Citado na página 20.

ÁGIL, M. *Manifesto para o desenvolvimento ágil de software @ONLINE*. 2001. Disponível em: <http://manifestoagil.com.br/>. Citado 2 vezes nas páginas 29 e 31.