



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

# **Evolução de Software: De Plugin para Aplicação Independente**

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

Brasília, DF  
2013





Vinícius Vieira Meneses

## **Evolução de Software: De Plugin para Aplicação Independente**

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

2013

---

Vinícius Vieira Meneses

Evolução de Software:

De Plugin para Aplicação Independente / Vinícius Vieira Meneses. – Brasília, DF, 2013-

56 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 , 2013.

1. Evolução de Software. 2. Arquitetura de Software. I. Prof. Dr. Paulo Roberto Miranda Meirelles. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Evolução de Software:  
De Plugin para Aplicação Independente

CDU 02:141:005.6

---

Vinícius Vieira Meneses

## **Evolução de Software: De Plugin para Aplicação Independente**

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. Luiz Augusto Laranjeiras**  
Convidado 1

---

**Prof. Dr. Fabrício Braz**  
Convidado 2

Brasília, DF  
2013



# 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, ela 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.





# Abstract

This work presents a study and our collaboration on a source code monitoring platform called Mezuro. This platform is developed through an open source 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. This undergraduate work, 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.



# Lista de ilustrações

Figura 1 – Evolução do sistema operacional Linux, extraído de (GODFREY; TU, 2000)	30
Figura 2 – Representação do padrão MVC . . . . .	36
Figura 3 – Interação entre os componentes do Rails. Extraído de (MEJIA, 2011) .	37
Figura 4 – Design de alto-nível do Mezuro. Editado de (MEIRELLES et al., 2010)	44
Figura 5 – Tela principal do Mezuro. Disponível em < <a href="http://mezuro.org/">http://mezuro.org/</a> > . . . .	44
Figura 6 – Tela de visualização de um projeto . . . . .	45
Figura 7 – Tela de informações do repositório . . . . .	45
Figura 8 – Métricas do repositório após processamento . . . . .	46
Figura 9 – Diagrama de classes simples do Mezuro . . . . .	46



# Lista de tabelas

Tabela 1 – Leis de Lehman, extraído de (FERNANDEZ-RAMIL et al., 2008) . . .	28
Tabela 2 – Versões do Rails . . . . .	35
Tabela 3 – Cronograma para atividades do TCC2 . . . . .	50



# Lista de abreviaturas e siglas

MVC	Model-View-Controller
Rails	Ruby on Rails
TCC	Trabalho de Conclusão de Curso
UnB	Universidade de Brasília
LOC	Lines of Code
SOAP	Simple Object Access Protocol
XML	Extensible Markup Language
RPC	Remote Procedure Call
HTTP	Hipertext Transfer
JaBUTi	Java Bytecode Understanding and Testing
USP	Universidade de São Paulo
DRY	Don't Repeat Yourself
AGPL	GNU Affero General Public License
ISO	International Organization for Standardization





# Sumário

<b>1</b>	<b>Introdução</b>	<b>17</b>
1.1	Objetivos	18
1.1.1	Objetivos Gerais	18
1.1.2	Objetivos Específicos	18
1.2	Organização do Trabalho	18
<b>2</b>	<b>Evolução de Projetos de Software</b>	<b>21</b>
2.1	Software Livre	21
2.2	Processo de Desenvolvimento de Software Livre	22
2.3	Padrões de Software Livre	23
2.4	Evolução de Software	26
<b>3</b>	<b>Arquitetura de Software</b>	<b>31</b>
3.1	Arquitetura na Engenharia de Software	31
3.2	O arcabouço Ruby on Rails	33
3.2.1	Evolução do Ruby on Rails	34
3.2.2	Padrao Arquitetural MVC	35
3.2.3	Arquitetura do Rails	36
<b>4</b>	<b>Mezuro: Uma plataforma de Monitoramento de Código Fonte</b>	<b>39</b>
4.1	Concepção do Projeto Mezuro	39
4.2	Mezuro como Plugin do Noosfero	42
4.3	Mezuro como aplicação independente	42
<b>5</b>	<b>Considerações Finais</b>	<b>49</b>
5.1	Cronograma	49
.1	Questionário	51
.2	Respostas	52
	<b>Referências</b>	<b>55</b>



# 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 IBM<sup>1</sup> 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 softwares 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<sup>2</sup>. Stallman objetivava criar um mecanismo para garantir direitos de cópia, modificação e redistribuição de software. Com isso surgiu a Licença GPL<sup>3</sup>.

A definição de software é a mesma, seja ele proprietário ou livre. Softwares 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.

A evolução de software começou a receber mais atenção a partir dos estudos realizados por Meir M. Lehman no final da década de 60, assim se tornou uma área de conhecimento da engenharia de software. A partir de seus estudos Lehman elaborou um conjunto de oito leis que são conhecidas "Leis de Lehman", as quais definem um padrão para evolução de softwares.

A evolução de software é importante para manter o software consistente com seu ambiente, competitivo em relação a softwares concorrentes e manter os usuários interessados. Além disso, quando tratamos de software livre a evolução é essencial para manter a comunidade estabelecida ao seu redor, contribuindo para manter esse projeto ativo.

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

---

<sup>1</sup> Internacional Business Machines é uma empresa americana da área de tecnologia da informação. Comercializa hardware e software além de serviços de hospedagem e infra-estrutura.

<sup>2</sup> Projeto que tem como objetivo a criação de um sistema operacional livre

<sup>3</sup> General Public License ou Licença Pública Geral

são encontrados no seu código-fonte através de suas métricas. Porém, extrair métricas de código-fonte manualmente não é uma tarefa considerada pelas equipes de desenvolvimento dado o esforço necessário para tal. Para isso existem ferramentas que auxiliam as equipes de desenvolvimento. Entre essas ferramentas está a plataforma Mezuro, utilizada para o monitoramento de códigos-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. Por isso, a equipe de desenvolvimento do Mezuro decidiu pela reescrita de seu código, transformando-o em uma plataforma independente fora do Noosfero, visando mantê-la consistente com seu ambiente e com seus propósitos, competitivo em relação às ferramentas semelhantes e ampliar o interesse de utilização por parte de usuários.

## 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*<sup>4</sup> do Mezuro (modelo e controladores);
2. Analisar a nova arquitetura e compará-la com a antiga;
3. Fornecer documentação para o Mezuro, facilitando sua manutenção e evolução;
4. Colaborar com a comunidade de software livre do Mezuro, seguindo os padrões para colaboração.

## 1.2 Organização do Trabalho

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

---

<sup>4</sup> Sequência de passos ou processo que executa em paralelo e “por trás” da camada de visualização sem a intervenção do usuário

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 framework Ruby on Rails. Essa fase aconteceu em paralelo com os pareamentos, onde comecei a implementar a primeira funcionalidade como colaborador da plataforma Mezuro.

Em paralelo, houve a escrita desta trabalho que está organizado em capítulos como a seguir. O Capítulo 2 apresenta os principais conceitos de softwares livre, e padrões de contribuições, com uma fundamentação teórica sobre evolução de software. O Capítulo 3 apresenta os principais, dos diversos conceitos, sobre arquitetura de software, assim como uma visão da arquitetura do Rails. Finalmente, o Capítulo 4 apresenta com mais detalhes a plataforma Mezuro, seu processo de desenvolvimento, fatores que motivaram a reescrita de seu código, assim como uma visão geral do estado atual do desenvolvimento.



## 2 Evolução de Projetos de Software

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.

### 2.1 Software Livre

O software é um componente de um sistema computacional de interesse, uma vez que vários aspectos relacionados a ele ultrapassam questões técnicas (MEIRELLES, 2013), 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 entendimento desses quatro pontos é o que diferencia softwares ditos restritos dos livres, bem como, é o que define o que é conhecido como “ecossistema do software livre”. O princípio básico desse ecossistema é promover a liberdade do usuário, sem discriminar quem tem permissão para usar um software e seus limites de uso, baseado na colaboração e num processo de desenvolvimento aberto (MEIRELLES, 2013).

Ao contrário do software restrito, o software livre tem a característica do compartilhamento do seu código-fonte. Essa característica oferece vantagens em relação ao software proprietário. O compartilhamento permite a simplificação de aplicações personalizadas, já que não necessitam serem codificadas do zero, podendo se basear em soluções existentes. Outra vantagem é a melhoria da qualidade (RAYMOND, 1999), por conta da grande quantidade de colaboradores, que com diferentes perspectivas e necessidades, propõem melhorias para o sistema, além de identificar e corrigir bugs com mais rapidez.

Em resumo, software livre representa uma classe de sistemas de software, os quais são distribuídos sob licenças cujos termos permitem aos seus usuários utilizar, estudar, modificar e redistribuir o software (TERCEIRO, 2012).

## 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, por exemplo. 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<sup>1</sup>, 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<sup>2</sup>. Adaptação a mudanças, trabalhar com *feedback* contínuo, entregar funcionalidades reais, respeitar colaboradores 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.

---

<sup>1</sup> <<https://www.kernel.org/>>

<sup>2</sup> <<http://agilemanifesto.org/>>



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, 2013](#)).

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, 2013](#)).

## 2.3 Padrões de 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:

- **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<sup>3</sup>, 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.

O terceiro padrão é o que melhor se encaixa ao contexto desse trabalho, já que o Mezero 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.

---

<sup>3</sup> Integrated Development Environment, um ambiente integrado para desenvolvimento de software

- 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 terminá-la é importante para manter colaboradores motivados, e conforme ganham mais experiência 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 reportá-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<sup>4</sup> 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-las 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.

## 2.4 Evolução 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. Empresas, que possuem parte dos seus lucros associados diretamente, ou não, há sistemas de software, precisam evolui-los, seja para adequa-los à mudanças no ambiente onde estão inseridos, ou para mante-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

Por outro lado, a manutenção desses sistemas é difícil, consome bastante tempo e recursos. Tarefas como adicionar novas funcionalidades, suporte a novos dispositivos de

---

<sup>4</sup> SCM - Source Code Management - GIT, SVN, Baazar, Mercurial, entre outros

hardware, correção de defeitos, entre outros, se tornam mais difíceis conforme o sistema cresce e envelhece (GODFREY; TU, 2000).

Acima foram mencionados os termos manutenção e evolução de software. Na maioria das vezes essas palavras aparecem juntas na literatura, e 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.

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.

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 média 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 outros 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.

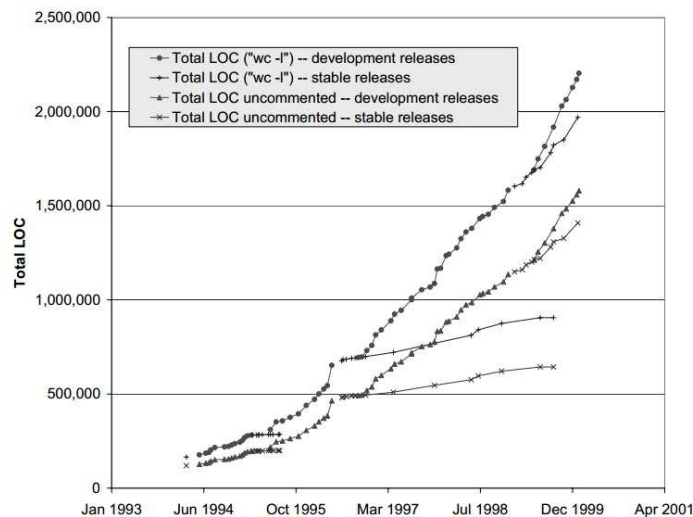


Figura 1 – Evolução do sistema operacional Linux, extraído de (GODFREY; TU, 2000)

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 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 Arquitetura de Software

O desenvolvimento de um sistema de software não é uma tarefa simples por conta da complexidade envolvida no processo. Além de lidar com a complexidade inerente ao problema a ser resolvido, devemos nos preocupar em como o software resolve esse mesmo problema. Por esse motivo muitos softwares fracassam, seja por custar muito acima do orçamento, estarem incompletos ou não solucionarem os problemas como deveriam. Assim um software além de resolver o problema, deve resolvê-lo da forma esperada, satisfazendo atributos de qualidade (GERMOGLIO, 2010).

Conforme a complexidade dos softwares aumentam, surgem 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).

Neste capítulo serão apresentados aspectos da arquitetura, relevantes durante a evolução de um sistema de software. Na seção 4.1 são apresentados os principais conceitos sobre arquitetura do ponto de vista da engenharia de software. Já a seção 4.2 expõe elementos da arquitetura do Ruby on Rails, arcabouço utilizado para o desenvolvimento da plataforma Mezero.

### 3.1 Arquitetura na Engenharia de Software

Desde a primeira referência em um relatório técnico intitulado *Software Engineering Techniques* (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 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 destruída junto ao código no seu repositório.

## 3.2 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 o Rails para construir 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 manutenção 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**presentational **State** **T**ransfer. Propõe princípios (não exclusivos ao REST) que definem como Web Standards como HTTP e URIs devem ser usados. Os princípios são:

- Todos os recursos devem ter um identificador. Um conceito comum para identificador na web é a URI;
- Utilize links (hipermídia) para referenciar recursos;
- 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;
- 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;
- 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;

### 3.2.1 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 Mezero, 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.

### 3.2.2 Padrão Arquitetural MVC

O padrão 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.

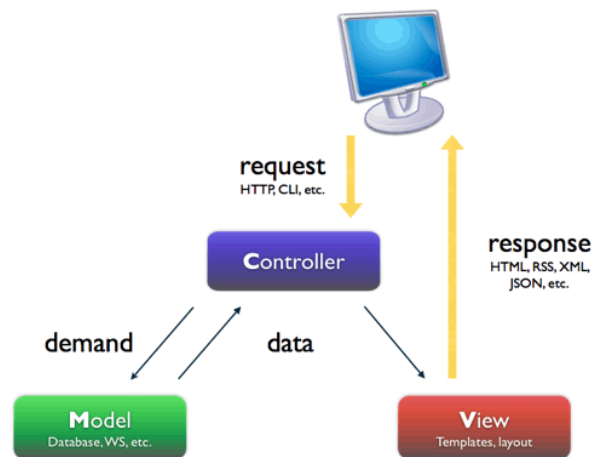


Figura 2 – Representação do padrão MVC

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.

### 3.2.3 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.

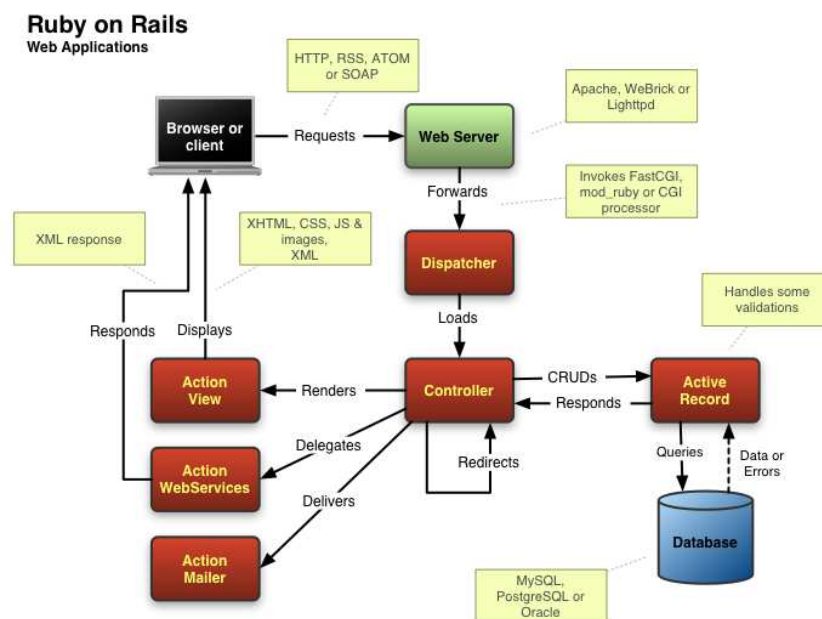


Figura 3 – Interação entre os componentes do Rails. Extraído de (MEJIA, 2011)

**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;

**Action Pack 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;

**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;

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

**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;

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





## 4 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, 2013). 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, elas auxiliam a equipe durante o processo de desenvolvimento. Porém, 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.

### 4.1 Concepção do Projeto Mezuro

A partir da necessidade de extrair métricas de código-fonte e interpretar seus valores, foi desenvolvida uma plataforma chamada Mezuro <sup>1</sup>. 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

---

<sup>1</sup> <<http://mezuro.org/>>

Qualipso<sup>2</sup>. 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, foram encontrados alguns projetos com propósitos similares(MEIRELLES et al., 2010), entre eles:

- **FlossMetrics** (Free/Libre Open Source Software Metrics) é um projeto que utiliza metodologias e ferramentas existentes para fornecer uma extensa base de dados com informações sobre desenvolvimento de software livre;
- **Ohloh** é um website que fornece um conjunto de web services e uma plataforma online com o objetivo de construir uma visão geral sobre o desenvolvimento de software livre;
- **Qualoss** (Quality in Open Source Software) é uma metodologia para automatizar a medição de qualidade de projetos de software livre, utilizando ferramentas para analisar o código-fonte e informações sobre seu repositório;
- **SQO-OSS** (Software Quality Assessment of Open Source Software) fornece um conjunto de ferramentas de análise e benchmarking de projetos de software livre.
- **QSOS** (Qualification and Selection of Open Source Software) é uma metodologia baseada em quatro passos: definição de referência utilizada, avaliação do software, qualificação de usuários específicos, e por último, seleção e comparação de software;
- **FOSSology** (Advancing Open Source Analysis and Development) é um projeto que fornece uma base de dados aberta com informações sobre licenças de software;
- **HackyStat** é um ambiente de análise, visualização e interpretação de processos de desenvolvimento e dados de software.
- **CodeClimate** é um software que auxilia a inserção de qualidade em códigos-fonte escritos na linguagem Ruby por meio da análise 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<sup>3</sup>.

---

<sup>2</sup> Quality Platform for Open Source: <<http://qualipso.icmc.usp.br/>>

<sup>3</sup> Representa o grau de especialização de uma classe para desempenhar papéis em um contexto. Quanto menos responsabilidades tiver uma classe, mais coesa ela será.

- **Alta complexidade total em classes** - Mesmo que os métodos da classe sejam simples, se a classe for muito grande é sinal que ela possibilitando o monitoramento de características específicas do software tem muitas responsabilidades.

Em particular, o projeto Mezuro surgiu antes do CodeClimate<sup>4</sup>, 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 providas pelos coletores de métricas: Analizo<sup>5</sup>, CheckStyle<sup>6</sup> e CVSanaly<sup>7</sup>.

O Mezuro utiliza o Kalibro Metrics<sup>8</sup> 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. O 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 web service através de um protocolo conhecido como SOAP<sup>9</sup>, baseado em XML<sup>10</sup> para o formato de mensagens. Por meio de requisições SOAP, o Mezuro acessa os *end-points*<sup>11</sup> do Kalibro Service. Em resumo, isso permite ao Mezuro provê aos seus usuários as seguintes funcionalidades:

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

---

<sup>4</sup> <<http://codeclimate.com>>

<sup>5</sup> <<http://anzalizo.org>>

<sup>6</sup> <<http://checkstyle.sourceforge.net/>>

<sup>7</sup> <<http://tools.libresoft.es/cvsanaly>>

<sup>8</sup> <<http://kalibro.org>>

<sup>9</sup> Simple Object Access Protocol

<sup>10</sup> Linguagem de Marcação Extensível

<sup>11</sup> Métodos disponibilizado pelo web service, onde cada método define uma funcionalidade

## 4.2 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<sup>12</sup> 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 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<sup>13</sup>. O funcionamento dos plugins é baseado no paradigma de orientação a eventos. O núcleo do Noosfero dispara um evento durante sua execução e os plugins interessados nesse evento saberão como tratar esse evento. Os eventos que são disparados pelo Noosfero são chamados de “hotspots”.

## 4.3 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

---

<sup>12</sup> Licença de software GNU Affero General Public License

<sup>13</sup> <<http://noosfero.org/Development/Plugins>>

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. O Noosfero, porém, ainda está nas versões 1.8 e 2 do Ruby e do Rails, respectivamente. Como apresentado na Seção 3.2.1, o Rails encontra-se na quarta versão, com mais recursos e melhorias em relação à segunda versão. 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<sup>14</sup> 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 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 9, 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 pelos desenvolvedores do Mezuro como desnecessários para uma ferramenta de monitoramento de código-fonte. Entre esses recursos estão: blog, fórum, upload de arquivos, CMS, 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 evoluía, 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 4 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

---

<sup>14</sup> <<http://rubyonrails.org/>>

vez utiliza coletores de métricas (Analizo, CheckStyle e CVSAAnly) para executar suas funcionalidades.

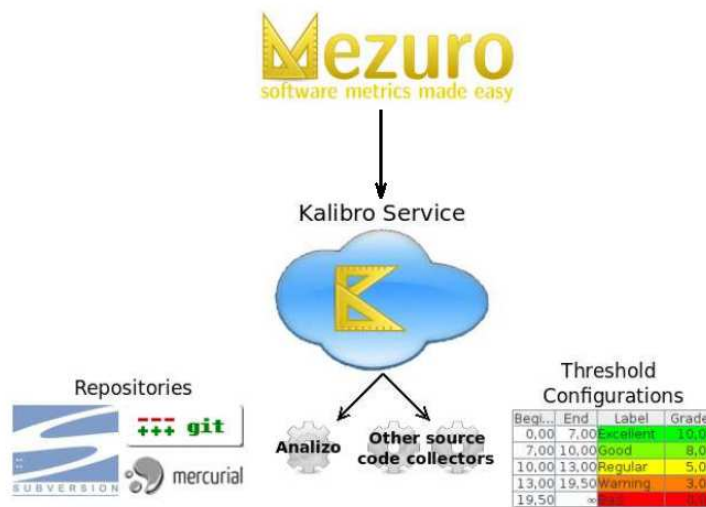


Figura 4 – Design de alto-nível do Mezuro. Editado de (MEIRELLES et al., 2010)

Sem as restrições impostas pelo desenvolvimento do Noosfero, uma importante decisão arquitetural tomada pela equipe do Mezuro foi o desenvolvimento de uma *gem* (para Ruby, equivalente a uma *Lib* 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 em projetos desenvolvidos com esse arcabouço e necessitam utilizar serviços do Kalibro, bastam instalar e utilizar a *gem*, sem a necessidade de reimplementar a interface do serviço.

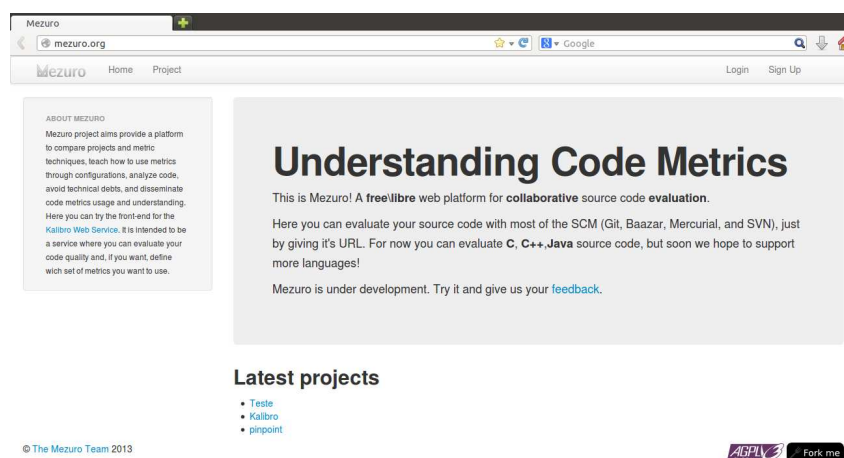


Figura 5 – Tela principal do Mezuro. Disponível em <<http://mezuro.org/>>

Dentro da evolução do Mezuro, nesta primeira fase trabalho, como uma forma

de conhecer a tecnologia e aprender a colaborar com o Mezuro, desenvolvemos a funcionalidade de inserção de repositórios nos projetos cadastrados para serem monitorados no Mezuro. De acordo com o diagrama da Figura 9 é observado que um projeto possui diversos repositórios, os quais podem ser adicionados e removidos de um projeto, além de editados e visualizados. No caso da visualização é feito um processamento para que o usuário analise as métricas extraídas do código presente nesse repositório. Essa funcionalidade, desenvolvida ao decorrer deste trabalho, se encontra atualmente disponível na primeira versão do Mezuro como plataforma independente, lançada no mês de novembro de 2013, ilustrada na Figura 5.

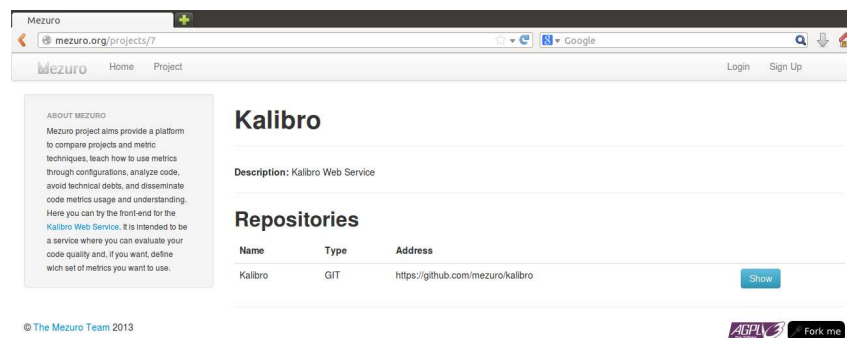


Figura 6 – Tela de visualização de um projeto

Ao clicar em algum projeto disponível no Mezuro, sua tela visualização é carregada. Nela são exibidas todos os repositórios relacionados a esse projeto. Na Figura 6, por exemplo, é exibida a tela do monitoramento do código do Kalibro cadastrado no Mezuro, assim como um único repositório associado a ele.

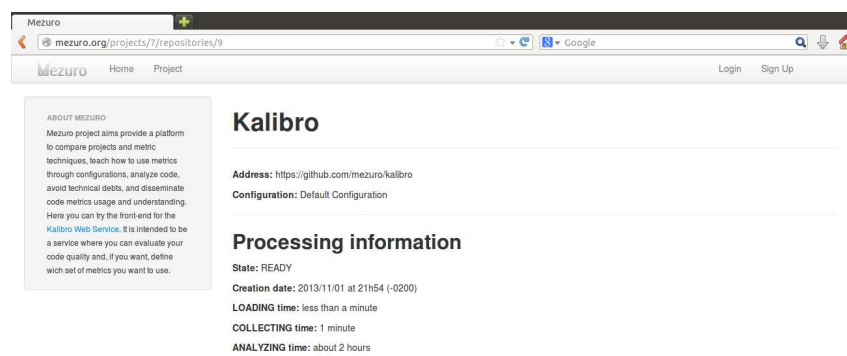


Figura 7 – Tela de informações do repositório

Ao clicar em visualizar um repositório, é feito um processamento desse, e a tela de detalhes do mesmo é carregada. A Figura 7 e 8 são os resultados do processamento do único repositório associado ao projeto Kalibro monitorado pelo Mezuro. A Figura 7 exibe

## 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 8 – Métricas do repositório após processamento

informações gerais do processamento do repositório, como a configuração, estado, data de criação e tempo de processamento. A Figura 8 mostra o conjunto de métricas relacionadas à configuração do repositório processado (uma configuração *default*, por exemplo), assim como os intervalos e interpretação de cada uma dessas métricas.

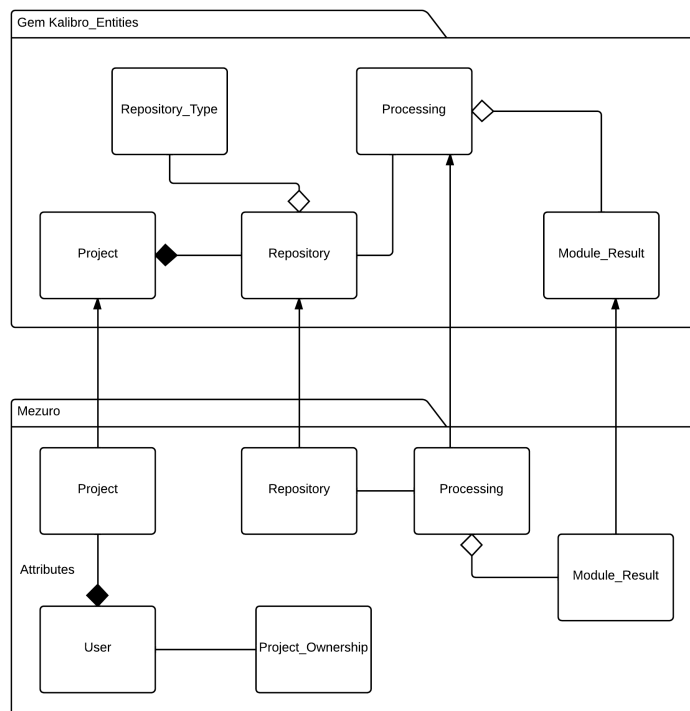


Figura 9 – Diagrama de classes simples do Mezuro

Por fim, a Figura 9 apresenta um diagrama de classes que representa as 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<sup>15</sup>, Subversion<sup>16</sup>, Bazaar<sup>17</sup>

<sup>15</sup> <<http://git-scm.com/>>

<sup>16</sup> <<http://subversion.apache.org/>>

<sup>17</sup> <<http://bazaar.canonical.com/en/>>



e Mercurial<sup>18</sup>. A classe *Processing* representa o processamento de um repositório para obter os resultados das métricas extraídas de acordo com a configuração selecionada. E finalmente, a classe *Module-Result* representa o resultado do processamento de um repositório. 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. Terminar de migrar todas essas funcionalidades, que já estavam implementadas no Mezuro Plugin, é o ponto em que está esse processo de evolução do Mezuro.

---

<sup>18</sup> <<http://mercurial.selenic.com/>>



## 5 Considerações Finais

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-a 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 a 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.

A contribuição do principal autor deste trabalho, com a funcionalidade de “Manter Repositórios”, é um resultado dessa nova comunidade de software livre formada a partir da decisão de tornar a plataforma Mezuro uma aplicação independente. 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.

### 5.1 Cronograma

Para o curto e médio prazo, no escopo deste trabalho, há atividades planejadas para o futuro do Mezuro. Até agora, o Mezuro evoluiu em sua arquitetura, migrando de plugin para aplicação independente com o Rails 4. Na segunda fase deste trabalho, vamos colaborar com novas funcionalidades, que também demandará uma etapa de pesquisa

complementar ao que fizemos até agora. As atividades planejadas são:

1. Complementação das pesquisas sobre software livre e evolução de software;
2. Pesquisa sobre visualização de software, formas visuais das métricas de código-fonte;
3. Colaboração com finalização da migração das funcionalidades do Mezuro Plugin para o novo Mezuro;
4. Incorporar formas visuais das métricas ao Mezuro (visualizações gráficas);
5. Implementar suporte a novas linguagens (e.g Ruby);
6. Escrita do TCC.

Atividade	Dez 2013	Jan 2014	Fev 2014	Mar 2014	Abr 2014	Mai 2014	Jun 2014
1	•	•					
2		•	•	•			
3	•	•	•				
4				•	•	•	
5						•	•
6	•	•	•	•	•	•	•

Tabela 3 – Cronograma para atividades do TCC2

É importante enfatizar que, após as leituras preliminares e uma avaliação inicial do Mezuro Plugin para a definição do tema e passos para este trabalho, ao final de agosto de 2013, este trabalho começou a ser desenvolvido em setembro de 2013, desde o aprendizado das tecnologias, além dos estudos teóricos para a elaboração deste texto. Em 2 meses, conseguimos colaborar efetivamente com o Mezuro ao desenvolver toda a parte de “gestão de repositório” de um projeto cadastrado no Mezuro, ou seja, um item central entre as funcionalidades do mesmo. Avaliamos que, durante o TCC 1, já percorremos a curva de aprendizado necessária para contribuir com o projeto de Mezuro, para, num primeiro momento, finalizarmos a migração para uma aplicação independente e, posteriormente, inserirmos novas funcionalidades, em especial no contexto de visualização de software, como uma das partes da área de evolução de software que será pesquisada durante o TCC 2 (7 meses, conforme o cronograma apresentado na Tabela 3).

## .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 fornecer 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 ficassem 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.





# Referências

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 31.

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 31.

CORBUCCHI, H. *Métodos ágeis e software livre: um estudo da relação entre estas duas comunidades*. 2011. Citado 2 vezes nas páginas 22 e 23.

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 32.

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 11 e 28.

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 31.

GERMOGLIO, G. *Fundamentos da arquitetura de software*. 2010. Citado 3 vezes nas páginas 31, 32 e 36.

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 9, 26, 29 e 30.

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 4 vezes nas páginas 9, 39, 40 e 44.

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 pag, v. 33, 2009. Citado na página 39.

MEIRELLES, P. R. M. *Monitoramento de métricas de código-fonte em projetos de software livre*. Tese (Doutorado) — Instituto de Matemática e Estatística – Universidade de São Paulo (IME/USP), 2013. Citado 3 vezes nas páginas 21, 23 e 39.

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 9 e 37.

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 2 vezes nas páginas 28 e 29.

QUALIPSO Project - O Projeto Qualipso. 2009. Disponível em: <<http://qualipso.icmc.usp.br/>>. Citado na página 40.

RAYMOND, E. The cathedral and the bazaar: Musings on linux and open source from an accidental revolutionary, sebastapol. CA: *O'Reilly BC Associates*, 1999. Citado na página 21.

SHAW, M.; GARLAN, D. Software architecture: perspectives on an emerging discipline. Prentice Hall, 1996. Citado na página 31.

TERCEIRO, A. S. de A. Caracterização da complexidade estrutural em sistemas de software. 2012. Citado na página 22.