Universidade Federal de Pernambuco

Centro de Informática

Graduação em Ciência da Computação

Desenvolvimento de ferramenta open source para automação de versionamento de release

---

Autor:
Airton Sampaio de Sobral
asds@cin.ufpe.br

Orientador:
Carlos Ferraz
cagf@cin.ufpe.br

Recife, 2016

Airton Sampaio de Sobral

Desenvolvimento de ferramenta open source para automação de versionamento de release

Trabalho de Conclusão de Curso apresentado à Universidade Federal de Pernambuco – UFPE, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Universidade Federal de Pernambuco

Orientador: Carlos Ferraz

Recife
2016

Airton Sampaio de Sobral

Desenvolvimento de ferramenta open source para automação de versionamento de release

Trabalho de Conclusão de Curso apresentado à Universidade Federal de Pernambuco – UFPE, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Trabalho aprovado. Recife, 21 de dezembro de 2016:

Prof. Carlos Ferraz
Orientador

Prof. Fernando Castor
Avaliador

Recife
2016

Agradecimentos

Primeiramente, gostaria de agradecer ao professor Carlos Ferraz por ter me incentivado a fazer o trabalho de graduação para a finalização da minha graduação. Não é fácil focar em um curso de graduação quando em paralelo você tem uma empresa para gerir. Sem o apoio de Carlos, eu não teria dado seguimento ao trabalho.

Em segundo lugar, gostaria de agradecer aos meus colegas de trabalho que participaram da concepção do Versi: Igor Leão e Guilherme Cavalcanti.

Em terceiro lugar, gostaria de agradecer a Marina Haack, por me incentivar a fazer o trabalho de graduação, e a Alexandre Cisneiros por me auxiliar na formatação do relatório.

Resumo

Apesar da recepção positiva da cultura DevOps, na prática, ela não é nada fácil de ser implantada. Muita automação se faz necessária para que os desenvolvedores tenham controle sobre o processo de geração de *release* e de *deploy*, além das mudanças de processos para que haja a comunicação necessária entre um time onde a divisão entre desenvolvedor e operacional passa a ficar cada vez menos visível.
Neste trabalho de graduação pretende-se desenvolver uma ferramenta open source de automação denominada Versi, escrita na linguagem Ruby, com o intuito de prover facilidade ao desenvolvedor para versionar um novo release. Esta ferramenta será integrada com o controle de versionamento Git e será capaz de rodar em ambiente de Integração Contínua.

Palavras-chave: git; release; semantic; versioning; ferramenta; automação; DevOps

Abstract

Despite the positive reception of the ”DevOps”culture, in practice, it is not easy to implement. A lot of automation becomes necessary for developers to have control over the release and deployment generation process, along with process changes so that there is the necessary communication between a team where the division between developer and operational becomes less and less visible.
This graduation work intends to develop an automation open source tool denominated Versi, written in the Ruby language, in order to aid developers to versionate new releases. This tool will be integrated with GIT version control and will be able to run in a Continuous Integration environment.

Palavras-chave: git; release; semantic; versioning; tool; automation; DevOps

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

1.1 Contexto

Desde o começo do século, as empresas de tecnologia já sentiam a necessidade de mudanças entre área de desenvolvimento e de operacional de infra-estrutura. As duas áreas, por padrão, possuem mentalidades distintas: A de desenvolvimento quer realizar mudanças no sistema e colocar no ar o mais rápido possível para agregar valor ao produto, enquanto que o operacional deseja sempre realizar o menor número de modificações para garantir a estabilidade e confiabilidade em produção [1]. Devido a esta dualidade, a colaboração entre as duas áreas se torna complicada [1], reduzindo a velocidade de entrega de novas funcionalidades e consequentemente impactando o crescimento do produto.

Em meados de 2008 uma palestra realizada por Patrick Debois de título "Agile Operations and Infrastructure: How Infra-gile Are You?" começou a discussão a respeito do que seria necessário mudar para resolver (ou diminuir) o problema [1]. Foi assim que surgiu o movimento cultural denominado DevOps, que apesar de ter vários significados, possui um objetivo primário: Através de uma série de práticas, reduzir o tempo entre realizar uma modificação em um sistema e esta modificação ser colocada em produção, com garantia de qualidade [2].

Apesar da popularidade rapidamente adquirida do movimento DevOps, sendo classificado como "em ascensão" pela Gartner Hype Cycle for Application Development em 2013 [2], não é nada fácil aplicar suas práticas. É necessário muito esforço em várias vertentes: processos, estruturação, automação e cultura. Todas essas vertentes vão deixar mais tênue a linha que divide o desenvolvimento do operacional, principalmente para dar mais liberdade ao desenvolvedor para realizar o *deploy* e acompanhar a performance do seu código em produção [2]. Para dar essa liberdade ao desenvolvedor se faz necessário um ferramental que proporcione automação, de tal forma que seja possível para um desenvolvedor, que não possui os mesmos conhecimentos de infra-estrutura como um operador, a criação de um *release*, realização do seu *deploy* e monitoração do sistema.

1.2 Cenário

O desenvolvimento deste trabalho de graduação foi realizado diante dos desafios de uma startup de Recife (Pernambuco - Brasil) chamada In Loco Media. Utilizando um modelo de organização semelhante ao do Spotify, a In Loco Media divide seus times de desenvolvimento em grupos chamados de *squads*, que funcionam como times multi-funcionais, ou seja, com pessoas de várias áreas (desenvolvedores, designers, UI, UX) focadas em

um único produto/objetivo. Esta abordagem traz grandes vantagens, sendo a principal o nível de independência e foco entre as *squads*. Mas traz também o grande desafio de fornecer ferramentas e processos para que as *squads* consigam controlar todo o ciclo de desenvolvimento (desde o código-fonte até o serviço em produção) sem depender de outras *squads* ou times operacionais.

Neste cenário os desenvolvedores possuem uma grande proficiência em Ruby e Java, sendo os maiores serviços escritos nestas duas linguagens. Além disso, eles utilizam Linux e Mac OSX (Unix) para desenvolvimento e Linux em sistemas em produção.

1.3 Motivação

O processo de versionamento de *release*, em geral, é realizado manualmente, podendo ocasionar em diversas inconsistências e atrapalhar o processo de identificação de *releases* problemáticos. O problema mais comum é o de esquecer de realizar o tagueamento do *release*, principalmente quando este foi apenas uma simples correção. Esquecendo de realizar o tagueamento, vários *releases* não vão ser registrados, complicando o processo de escolher o último *release* para dar *rollback*. Outro problema que pode acontecer é o desenvolvedor esquecer de subir a tag para o servidor do GIT, gerando inconsistência entre as tags locais e as do servidor.

Além de querer garantir que estes problemas não aconteçam, existe uma segunda motivação: Reduzir o número de tarefas do desenvolvedor. Isto porque o desenvolvedor já vai ter mais tarefas devido ao processo de descentralização das atividades de *release* e *deploy* do operacional, propostos pela cultura DevOps [2].

1.4 Objetivo

Neste trabalho de graduação pretende-se desenvolver uma ferramenta open source de automação denominada Versi, escrita na linguagem Ruby, com o intuito de prover facilidade ao desenvolvedor para versionar um novo *release*. Esta ferramenta será integrada com o controle de versionamento Git e será capaz de rodar em ambiente de Integração Contínua.

1.5 Estrutura do trabalho

Este trabalho está dividido em 5 capítulos, incluindo este capítulo de introdução. No Capítulo 2, serão apresentados conceitos importantes para o melhor entendimento da ferramenta, além de outras ferramentas que possuem a funcionalidade de versionamento de *release*. Em seguida, o Capítulo 3 descreverá a fase de concepção da ferramenta, explicando qual foi o racional para cada decisão e quais foram os requisitos que adotamos

como essenciais. O Capítulo 4 explicará a implementação do Versi. Por fim, o Capítulo 5 expõe as conclusões obtidas após a concepção e desenvolvimento realizados.

2 Conceitos e Ferramentas existentes

2.1 Conceitos

2.1.1 Release

No contexto de desenvolvimento de software, o termo *"release"* possui múltiplas acepções, sendo as mais comuns:

- O ato de lançar uma nova versão de um software em um determinado ambiente (*staging*, *production*)
- Uma nova versão de um código-fonte que foi selecionada para para ser lançada.

Neste trabalho, será utilizada a última acepção. É importante notar que, apesar de selecionado o *release*, não necessariamente ele está pronto para ir ao ar em produção. Em geral existem fases para um *release*, estando cada uma mais próxima de uma versão estável: Alpha, Beta, Release Candidate, Gold [3].

No contexto de sistema de controle de versão, pode-se gerar uma tag para indicar o exato ponto de um *release*. Cada *release* possui um nome, que pode ser puramente sequencial: 1, 2, 3, 4; ou pode ser mais complexo e significativo: 3.1.3-beta. Neste trabalho utilizaremos a expressão "geração de *release*" para designar a geração de tag que identifica um *release*. Note que na maioria dos casos se faz necessária a geração de artefatos (como binários) a partir do código-fonte. A etapa de geração de artefatos é comumente conhecida como "processo de *build*" e pode ser executada localmente ou em um servidor/serviço a parte. Em cenários mais automatizados, utiliza-se o processo de Integração Contínua, onde existe um serviço a parte, responsável por integrar o código, executar a suíte de testes e gerar todos os artefatos necessários [3].

2.1.2 Semantic Versioning

Um padrão bastante comum de se nomear *releases*, principalmente na comunidade ruby, é o "Semantic Versioning". Criado por Tom Preston-Werner (Co-founder do Gravatar e Github), o padrão consiste em manter a versão com 3 partes, separadas por pontos: MAJOR.MINOR.PATCH; onde MAJOR é um número que deve ser incrementado quando houver mudanças não retrocompatíveis (ou seja, quando for gerado um *Major* Release), MINOR é um número que deve ser incrementado quando houver mudanças retrocompatíveis relacionadas a novas funcionalidades (ou seja, quando for gerado um *Minor* Release) e o PATCH é um número que deve ser incrementado quando houverem correções de bugs

retrocompatíveis (ou seja, quando for gerado um *Patch* Release). Além disso, sufixos podem ser adicionados para indicar um release candidate, versão beta, etc.. [4]. Exemplos de versões utilizando Semantic Versioning:

- 0.0.1-pre1
- 0.2.0
- 3.2.13-beta

Este padrão além de prover uma boa visibilidade sobre a evolução do sistema, é especialmente útil para APIs e bibliotecas, pois permite a visibilidade de modificações retrocompatíveis.

2.1.3 Integração Contínua

A Integração Contínua representa uma mudança de paradigma. Sem Integração Contínua, seu software está no estado "quebrado" até que alguém prove que ele funciona, geralmente durante uma fase de teste ou de integração. Com a Integração Contínua, o software é provado que funciona a cada nova mudança, e você sabe o momento que ele quebra e pode corrigi-lo imediatamente [3].

As equipes que utilizam a integração contínua de forma eficaz são capazes de desenvolver um *software* muito mais rápido, e com menos *bugs*, do que equipes que não utilizam. Os *bugs* são detectados muito mais cedo no ciclo de entrega, quando eles são mais baratos de resolver, representando uma significativa economia de custos e tempo. Por isso, ela é considerada uma prática essencial, talvez tão importante quanto usar o controle de versão [3].

Em geral, o processo de Integração Contínua é realizado em um servidor ou serviço a parte, onde cada commit recebido pelo servidor Git remoto inicia o processo de realizar as etapas de *build* e testes.

2.1.4 Deploy

O termo *deploy*, no contexto de desenvolvimento de *software*, significa a ação de disponibilizar um sistema para uso. O *deploy* é realizado em uma série de etapas sequenciais, com o propósito de atualizar um sistema, ou colocar um sistema inteiramente novo no ar [3]. As etapas de um *deploy* vão depender da arquitetura do sistema e da infra-estrutura utilizada.

2.2 Ferramentas existentes

2.2.1 RMT

RMT (Release Management Tool) é uma ferramenta útil para ajudar a liberar novas versões do seu *software*. É possível definir o padrão de nome de versão que você deseja usar (por exemplo, Semantic Versioning), onde você deseja armazenar a versão (por exemplo, em um arquivo changelog ou como uma tag do sistema de controle de versionamento) e uma lista de ações que devem ser executadas antes ou depois da criação de uma nova versão [5]. Construída em PHP, a ferramenta possui integração com os sistemas de controle de versionamento mais utilizados: Git e SVN; e configuração para a execução de comandos antes e depois da geração de tag. Os pontos negativos são a necessidade de conhecimento em PHP para realizar modificações e extensões na ferramenta (linguagem cada vez menos utilizada segundo o [6] e com nula adesão na In Loco Media) e a necessidade de etapas interativas (que requerem um humano para responder perguntas) durante a geração do *release*. Este último ponto é importante, pois impede a execução da ferramenta em um ambiente automatizado como o de Integração Contínua.

2.2.2 GitVersion

GitVersion olha para o seu histórico Git e descobre automaticamente a versão a ser utilizada, utilizando o padrão Semantic Versioning [7]. Construída em C# (.NET Framework), a ferramenta possui integração com o sistema de controle de versionamento Git e plataformas de Integração Contínua como Jenkins, Bamboo e GitLab CI. As grandes desvantagens são a dependência do framework .NET e o suporte fraco para Linux/Unix, ainda com muitos bugs [7].

3 Concepção

3.1 Mindset

Antes de começar a implementação da ferramenta, muito foi se foi questionado a respeito da sua real necessidade, pois reinventar a roda é sempre custoso e acaba atrasando o que de fato agrega valor para a empresa. Então o processo de concepção foi iniciado com o trabalho de verificar o que realmente seria necessário termos em nossa ferramenta, e o que poderia ser reaproveitado ou utilizado de outras ferramentas. Das ferramentas encontradas ambas ferramentas utilizam o padrão Semantic Versioning, se encaixando no padrão já utilizado hoje na In Loco Media, porém ambas pecam na impossibilidade ou dificuldade em se fazer uma customização ou extensão. Com o RMT, em especial, teriam-se sérios problemas para utilizar em um ambiente de Integração Contínua, resultando em uma difícil tarefa de modificar um código fonte em PHP. Além disso, ambas adicionam dependências pesadas e desnecessárias ao *stack* utilizado na In Loco (PHP ou .NET Framework).

Tinha-se em mente também que a linguagem mais adequada para a implementação da ferramenta seria Ruby, tanto pela sua versatilidade quanto pelo conhecimento técnico aprofundado do time que iria desenvolvê-la. Além disso, a tecnologia Ruby está em quase todo *stack* da In Loco Media, oferecendo menos barreiras para ser utilizada localmente e em um ambiente de Integração Contínua.

Um grande fator levado em consideração foi a escolha de abrir o código-fonte como um projeto open source, cultura proveniente da comunidade Ruby que a startup está inserida. Isto também se deve ao fato de a startup utilizar bastante tecnologias *open source* em seu *stack*.

O público alvo da ferramenta é o de desenvolvedores, e por isso foi escolhido que ela fosse uma CLI (*Command Line Interface*) que pudesse ser utilizada em qualquer plataforma Unix/Linux.

Para não ferir o propósito de facilitar, foi decidido que a ferramenta deveria ser rápida o suficiente para o seu *overhead* se tornar imperceptível ao desenvolvedor/serviço que a utiliza.

3.2 Requisitos essenciais

A partir do mindset, tem-se os seguintes requisitos:

- Ser *open source*

- Ser implementado em Ruby
- Ser uma CLI
- Ser modular
- Funcionar em plataformas Unix e Linux
- Ser rápida

Alem destes, será necessário:

- Integração com Git
- Integração com o padrão Semantic Versioning
- CLI não-interativa

As integrações são necessárias devido ao fato da In Loco Media utilizar tais ferramentas/padrões. E por último, possuir uma CLI não-interativa, se deve ao fato de o objetivo é de que essa ferramenta possa rodar em um ambiente de Integração Contínua.

3.3 Fluxos

3.3.1 Introdução

Nesta seção será mostrado como são os fluxos atuais para o versionamento de release, e como seriam os novos fluxos com a utilização da ferramenta construída. Os fluxos vão estar relacionados a 2 cenários onde são criadas novas tags de release:

1. Um *bug* foi encontrado e se faz necessário a realização de uma correção rápida. Após realizada a correção, ela precisa ser adicionada a *branch* principal, e uma nova tag de *release* precisa ser criada para identificar as novas modificações.
2. Após um conjunto de funcionalidades terem sido implementadas, o código está pronto para ser adicionado a *branch* principal e uma nova tag de *release* precisa ser criada para identificar as novas funcionalidades.

O primeiro cenário será chamado de "cenário de *hotfix*" (ou CH) e o segundo será chamado de "cenário de *release*" (ou CR). Quando mencionado que um *pull-request* é aprovado pela Integração Contínua, significa que o código fonte passou por todas as etapas da Integração Contínua (*build* e teste) sem apresentar problemas. Para designar uma etapa de um cenário, será utilizado o padrão <nome do cenário><número da etapa>. Exemplo: CR1, que corresponde a primeira etapa do cenário de *release*.

Duas branchs serão mencionadas com frequência: *master* e *develop*. A primeira é utilizada para guardar o código mais recente, que está em produção. A segunda é utilizada durante o desenvolvimento, e pode conter modificações que ainda não estão prontas para ir para produção.

3.3.2 Fluxo atual

Para o fluxo atual, serão colocadas em negrito as etapas que possuem um potencial risco de falha humana. Em seguida serão mostrados os problemas que podem ser causados por essas falhas.

Atualmente tudo é feito de forma manual, ou seja, em ambos os cenários, uma pessoa é responsável por fazer todas as etapas.

Para o cenário de hotfix, tem-se o seguinte fluxo:

1. Uma nova *branch* é criada, com o prefixo "hotfix/", partindo da *branch master*
2. A correção é realizada na *branch*
3. A *branch* é submetida e um *pull-request* é criado, com a *branch master* como alvo
4. Ao ser aprovado o *pull-request* (por um revisor e pela Integração Contínua), a *branch hotfix* é mesclada com a *branch master*
5. Após a mesclagem com a *branch master*, o ciclo de Integração Contínua é ativado novamente. Caso todas as etapas sejam bem sucedidas, o fluxo continua.
6. **O nome do novo *release* é calculado (atualizando-se as tags locais e calculando manualmente qual será o novo *release*, seguindo o padrão Semantic Versioning)**
7. Uma nova tag então é criada para marcar o exato ponto do *release*
8. **A nova tag é submetida para o servidor Git**

Para o cenário de release, tem-se o seguinte fluxo:

1. **O nome do novo *release* é calculado (atualizando-se as tags locais e calculando manualmente qual será o novo *release*, seguindo o padrão Semantic Versioning)**
2. Uma nova *branch* é criada, no formato "release/<nome do release>" partindo-se da branch *develop*
3. A *branch* é submetida e um *pull-request* é criado, com a *branch master* como alvo

4. Ao ser aprovado o *pull-request* (por um revisor e pela Integração Contínua), a *branch release* é mesclada com a *branch master*

5. Após a mesclagem com a *branch master*, o ciclo de Integração Contínua é ativado novamente. Caso todas as etapas sejam bem sucedidas, o fluxo continua.

6. **O nome do novo *release* é calculado (atualizando-se as tags locais e calculando manualmente qual será o novo *release*, seguindo o padrão Semantic Versioning)**

7. Uma nova tag então é criada para marcar o exato ponto do *release*

8. **A nova tag é submetida para o servidor Git**

Nos passos CH6, CR1 e CR6, o desenvolvedor está propenso ao erro de utilizar um nome de *release* que já existe, ou de pular um número de *release*. Para isso basta que ele esqueça de atualizar suas tags locais com as de produção antes de gerar a nova. Este erro será chamado de ERR1. Nos passos CH8 e CR8, o desenvolvedor está propenso ao simples erro de esquecer de submeter a tag para o servidor Git, gerando inconsistências entre as tags locais e do servidor, além de não propagar o exato *commit* do *release*, atrapalhando o processo de se voltar para a última versão estável. Este erro será chamado de ERR2 No passo CR6 o desenvolvedor pode ser levado a colocar no nome do *release* o mesmo nome da *branch* que foi criada no passo CR1, porém entre a criação da *branch* de *release* e o momento de mesclá-la com a *master*, pode ter sido criado um *release* que alterou a última versão criada. Este erro será chamado de ERR3. O fluxo abaixo demonstra o ERR3:

Considere 4 desenvolvedores B1, B2, B3 e B4 que trabalham no projeto DELTA, atualmente no *release* 1.0.0.

1. B1 começa o processo de criação de um *release*, criando a *branch* release/1.1.0

2. B1 cria o *pull-request* da *branch* release/1.1.0, mirando na *branch master*

3. B2 é notificado que existe um pequena funcionalidade que foi esquecida e precisa ir ao ar o mais rápido possível

4. B2 cria um *hotfix* partindo da *branch master* e realiza a implementação

5. B2 cria o *pull-request* do *hotfix*

6. B3 inicia a análise do *pull-request* do *release* feito por B1

7. B4 inicia a análise do *pull-request* do *hotfix* feito por B2

8. O *pull-request* do *hotfix* realizado por B2 é aprovado pelo revisor (B4) e pela Integração Contínua

9. B2 mescla a sua *branch* do *hotfix* com a *branch master*

10. B2 cria a tag de *release* 1.1.0 e sobe para o servidor Git

11. O *pull-request* do *release* realizado por B1 é aprovado pelo revisor (B3) e pela Integração Contínua

12. B1 mescla sua *branch* de *release* (release/1.1.0) com a *branch master*

13. **B1 cria a tag de *release* 1.1.0 e tenta subir para o servidor Git, recebendo uma mensagem de erro dizendo que a tag já existe**

No passo 13, duas coisas podem acontecer: B1 pode analisar o que houve, descobrindo o motivo de a tag já existir e aí então tomar uma decisão, ou, ele pode tentar sobrescrever a tag que já existe no servidor Git utilizando *flags* para forçar a sobrescrita (`git push --force`). No primeiro caso, haverá uma perda de tempo significante, e no segundo haverá uma sobrescrita de *release*, impactando no histórico do projeto, e gerando inconsistências entre os repositórios locais dos desenvolvedores (B1 e B2 ficariam com a tag 1.1.0 apontando para lugares diferentes).

3.3.3 Novo fluxo

No novo fluxo, o desenvolvedor não irá se preocupar em ter que analisar os últimos *releases* para decidir qual será o nome do próximo. Isto ficará a cargo da ferramenta criada neste trabalho. Apesar disto, será visível qual tag foi criada e qual era a anterior, para que o desenvolvedor saiba o que aconteceu.

Para o novo fluxo, serão colocadas em negrito as etapas que forem novas ou que existirem modificações em relação a como era feito antes. Em seguida elas serão explicadas.

Para o cenário de *hotfix*, tem-se o seguinte fluxo:

1. Uma nova *branch* é criada, com o prefixo "hotfix/", partindo da *branch master*

2. A correção é realizada na *branch*

3. A *branch* é submetida e um *pull-request* é criado, com a *branch master* como alvo

4. Ao ser aprovado o *pull-request* (por um revisor e pela Integração Contínua), a *branch hotfix* é mesclada com a *branch master*

5. **Após a mesclagem com a *branch master*, o ciclo de Integração Contínua é ativado novamente. Caso todas as etapas sejam bem sucedidas, irá ser chamada a ferramenta, que irá gerar o nome da próxima versão, taguear e subir para o servidor Git**

Na etapa CH5 a Integração Contínua irá identificar que a mesclagem foi realizada entre a *branch hotfix* e a *master*, e fará o trabalho de executar a ferramenta para que ela gere o nome da próxima versão e suba para o servidor Git. As informações de geração da tag de *release* vão estar nos logs das Integração Contínua, que podem ser acessados a qualquer momento pelo desenvolvedor. É importante ressaltar que nesse fluxo é possível que se faça tanto uma versão MINOR, quanto uma versão PATCH, o que deve ser detectado pela ferramenta através das mensagens do *commit* ou do nome da *branch*.

Para o cenário de *release*, tem-se o seguinte fluxo:

1. **Uma nova *branch* é criada, no formato "release/<nome do release>" partindo-se da *branch develop*, utilizando um comando da ferramenta**
2. A *branch* é submetida e um *pull-request* é criado, com a *branch master* como alvo
3. Ao ser aprovado o *pull-request* (por um revisor e pela Integração Contínua), a *branch release* é mesclada com a *branch master*
4. **Após a mesclagem com a *branch master*, o ciclo de Integração Contínua é ativado novamente. Caso todas as etapas sejam bem sucedidas, irá ser chamada a ferramenta, que irá gerar o nome da próxima versão, taguear e subir para o servidor Git**

Na etapa CR1, a criação da branch release será realizada com o auxílio da ferramenta desenvolvida neste trabalho. Na etapa CR4 ocorre a criação automática da tag de *release*, evitando potenciais erros humanos. É importante ressaltar que nesse fluxo é possível que se faça tanto uma versão MAJOR, quanto uma versão MINOR, o que deve ser detectado pela ferramenta através das mensagens do *commit* ou do nome da *branch*.

3.4 Especificação

Nesta seção será detalhada quais serão os comandos disponíveis pela ferramenta, o que eles fazem, quais os seus parâmetros e quando devem ser utilizados de acordo com os fluxos descritos em 3.3.3.

3.4.1 Comando generate

O comando *"generate"* será o responsável pela geração da tag de um novo *release*. Esta tag será criada localmente e então enviada para o servidor Git.

Ele deve ter os seguintes parâmetros opcionais:

- –type ou -t
  Este parâmetro indica qual o tipo de *release* será criado. Caso não informado, o

tipo de *release* será inferido a partir dos últimos *commits*.
Os valores válidos para esse parâmetro são: major, minor ou patch.
Exemplos:
`--type major`
`-t patch`

- –name ou -n
  Este parâmetro indica qual será o nome da tag do *release*. O propósito deste parâmetro é para o caso de se querer utilizar uma tag específica que esteja fora do padrão.
  Exemplos:
  –name “obliteration”
  -n “cia”

- –prefix ou -p
  Este parâmetro indica qual será o prefixo da tag do *release*. Caso não informado, o prefixo da tag de *release* será vazio.
  Exemplos:
  –prefix “v”
  -p “v.”

- –suffix ou -s
  Este parâmetro indica qual será o sufixo da tag do *release*. Caso não informado, o sufixo da tag de *release* será vazio. O sufixo será adicionado com um “-” no início para que fique no padrão do Semantic Versioning (3.5.6-beta por exemplo).
  Exemplos:
  –suffix “beta”
  -s “gama”

- –message ou -m
  Este parâmetro indica qual será a mensagem gravada na tag do Git. Caso não informada, a mensagem será extraída dos nomes dos
  últimos *commits*.
  Exemplos:
  –message “Solves the P=NP problem”
  -m “Fixes the world poverty”

- –remote ou -r
  Este parâmetro indica qual servidor remoto do Git será enviada a tag. Caso não informado o servidor remoto *“origin”* será utilizado.
  Exemplos:

–remote delta
-r backup

- –debug ou -d
  Este parâmetro indica se o comando será executado em modo de *debug*. Neste modo apenas o nome da tag do *release* será gerado, mas a tag em si não será criada nem localmente nem no servidor Git. Este parâmetro não requer nenhum argumento extra. Este comando é útil para testes.
  Exemplos:
  –debug
  -d

Este comando deve ser executado nas etapas CH5 e CR4 descritas no novo fluxo (3.3.3). Em ambos casos, não será necessário nenhum parâmetro para o comando.

3.4.2 Comando branch

O comando *"branch"* será responsável pela criação de uma nova *branch*, contendo o nome da próxima tag de *release*.

Ele deve ter os seguintes parâmetros:

- –type ou -t
  Este parâmetro indica qual o tipo de *release* será utilizado para criar o nome da *branch*
  Os valores válidos para esse parâmetro são: major, minor ou patch.
  Exemplos:
  –type major
  -t patch
  Este parâmetro é obrigatório

- –debug ou -d
  Este parâmetro indica se o comando será executado em modo de *debug*. Neste modo apenas o nome da *branch* será gerado, mas a *branch* em si não será criada. Este parâmetro não requer nenhum argumento extra. Este comando é útil para testes.
  Exemplos:
  –debug
  -d
  Este parâmetro é opcional

3.5 Licença

A licença escolhida foi a "MIT License", uma licença permissiva curta e simples com condições que apenas exigem a preservação de direitos autorais e avisos de licença. Trabalhos licenciados, modificações e trabalhos maiores podem ser distribuídos sob diferentes termos e sem código fonte [8].

A ideia é restringir o menos possível o desenvolvimento de novas ferramentas que possam surgir desta.

4 Implementação

4.1 Visão geral

Para a implementação de uma CLI, são necessárias várias etapas entre o comando ser inserido pelo usuário, e ele ser executado. Na figura 1 é possível visualizar as etapas.

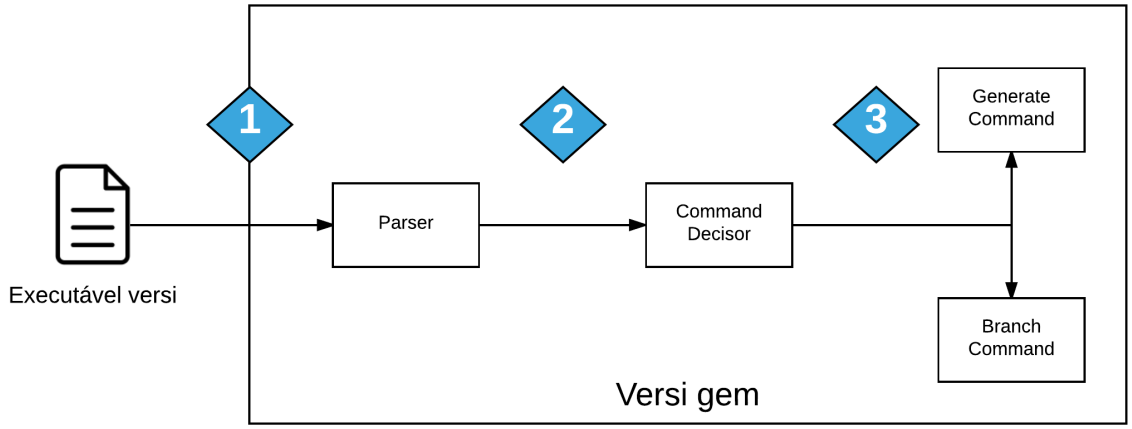

Figura 1 – Visão geral das etapas da Ferramenta

1. Após o comando ser inserido pelo usuário, um executável (versi) será acionado e enviará o comando para a gem Versi
2. O comando será analisado, subdividido e verificado sintaticamente por um *Parser*
3. Os dados provenientes do comando são analisados para determinar qual implementação deve ser utilizada e encaminhar os dados para a mesma

Nas seções seguintes, será explicada a implementação de cada um dos módulos (Executável, *Parser*, *Command Decisor*, *Generate Command* e *Branch Command*).

4.2 Executável versi

Como ruby é uma linguagem interpretada, não se faz necessário ter um executável com o código fonte compilado de uma gem. Por conta disto, o executável serve apenas como uma fachada para simplificar o ponto de entrada da CLI. Ele conta com apenas três linhas de código: uma para que o arquivo, por padrão, seja executado como um arquivo ruby,

outra para importar a gem “versi” e a última que inicia através do método “run” a classe principal do CLI (a classe Versi). Abaixo pode ser visualizado o código fonte do executável e sua simplicidade:

```
#!/usr/bin/env ruby
require "versi"

Versi.run
```

No executável não se envia os dados inseridos pelo usuário, porque os mesmos podem ser acessados por uma array global do ruby chamado “ARGV”. Este array é preenchido automaticamente pelo Ruby, contendo as strings que formam o comando inserido pelo usuário.

4.3 Parser e Command Decisor

O *Parser* é o responsável por processar o comando inserido, junto com as opções e fornecer isso de forma prática para a aplicação que irá utilizá-los. Fazer um *parser* para uma CLI não é nada trivial, existem muitos casos a serem tratados, e muitas funcionalidades necessárias para que a etapa fique a prova de erros.

O *Command Decisor* é o responsável por direcionar o fluxo da aplicação para a implementação correta, a partir das informações recebidas na etapa de *parsing*.

No começo da implementação já estava bem clara a necessidade de uma outra ferramenta ou *framework* que abstraísse estas duas partes, pois ambas dariam muito trabalho de criar, testar e manter. Graças a vastidão de gems disponíveis na comunidade Ruby, foi um trabalho relativamente simples escolher a que melhor se encaixa no nosso caso de uso. Em especial, tiveram 3 opções que mais chamaram a atenção:

1. Thor: Thor é uma ferramenta simples e eficiente para a construção de auto-documentação utilitárias de linha de comando. Ele remove a dor de opções de linha de comando de análise, escrevendo "USAGE:"banners, e também pode ser usado como uma alternativa para a ferramenta de compilação Rake. A sintaxe é Rake-like, por isso deve ser familiar para a maioria dos usuários Rake [9]

2. Clamp: "Clamp"é um *framework* mínimo para utilitários de linha de comando. Ele lida com coisas chatas, como analisar a linha de comando e gerar ajuda, para que você possa fazer o seu comando realmente fazer coisas [10]

3. Commander: A solução completa para executáveis de linha de comando Ruby. O Commander preenche a lacuna entre outras bibliotecas relacionadas ao terminal que você conhece e adora (OptionParser, HighLine), ao mesmo tempo que fornece muitos recursos novos e uma API elegante [11]

Todas as 3 opções são ótimas: A gem thor é a mais antiga, mais utilizada e oferece mais funcionalidades e mais contribuidores. Já as gems clamp e commander, possuem APIs que se diferenciam pela elegância, porém possuem menos contribuidores (cerca de 1/10 da gem thor). Devido a sua robustez e constante atividade dos mantenedores, a gem thor foi a primeira opção. Após algum tempo trabalhando com ela, foi notado que o seu suporte a subcomandos possui muitos *bugs* [12]. Isto porque ele não foi construído para se trabalhar com subcomandos, e o suporte a tal funcionalidade foi adicionado posteriormente. Por conta disso, decidiu-se utilizar a gem clamp, que possui um bom suporte a subcomandos.

4.4 Comandos

4.4.1 Dependências

Nesta seção serão descritas as dependências utilizadas na implementação dos dois comandos propostos pela ferramenta Versi.

4.4.1.1 Clamp

A gem clamp [10] é a responsável por abstrair o *Parser*, *Command Decisor*, e por fornecer uma estrutura simples e robusta para a implementação de uma CLI. Sem ela, muito tempo seria perdido implementando a estrutura necessária para se ter uma CLI funcional.

Com apenas um arquivo, como o mostrado abaixo, é possível ter uma CLI funcional:

```
require 'clamp'

Clamp do
  option "--loud", :flag, "say it loud"
  option ["-n", "--iterations"], "N", "say it N times" do |s|
    Integer(s)
  end

  def execute
    # Implementation
  end
end
```

4.4.1.2 Semantic

Semantic é uma gem que provê uma pequena classe de Ruby para auxiliar no armazenamento, análise e comparação de strings de versão do padrão Semantic Versioning [13]. Ela é utilizada na ferramenta Versi auxiliar no incremento de versões.

4.4.1.3 Interactor

A gem interactor fornece suporte para o padrão interactor. Os interactors são usados para encapsular a lógica de negócios da sua aplicação. Cada interactor representa uma coisa que seu aplicativo faz. [14]. A ferramenta Versi utiliza interactors para organizar cada etapa de cada comando. Os interactors facilitam a forma como código é trabalhado e testado. Cada interactor é implementado como uma classe que implementa o método call, como pode ser visualizado no código abaixo:

```
require 'interactor'

class AuthenticateUser
  include Interactor

  def call
    if user = User.authenticate(context.email, context.password)
      context.user = user
      context.token = user.secret_token
    else
      context.fail!(message: "authenticate_user.failure")
    end
  end
end
```

Além dos interactors, é possível criar organizers, que são conjuntos de interactors que rodam sequencialmente. Este último é especialmente útil para organizar o fluxo de uma funcionalidade grande que pode ser particionada em outras menores.

4.4.2 Classes auxiliares

4.4.2.1 Classe System

A classe System foi criada para oferecer uma facilidade no uso de comandos que vão rodar no sistema. Nesta classe existem dois métodos estáticos: “call” e “call!”.

O método “call” recebe uma string de comando, executa no sistema e retorna uma struct com 3 atributos: stdout, stderr e exit_status. O stdout é um array contendo as linhas enviadas para o STDOUT do sistema. O stderr é um array contendo as linhas enviadas para o STDERR do sistema. E o exit_status é o código de saída do comando, onde, em geral, 0 significa que o comando foi executado com sucesso. Neste método, se houver algum erro de execução, nenhuma exceção do ruby será lançada.

O método “call!” é similar ao método “call”, porém caso o comando retorne um status diferente de 0, será lançada a exceção System::CommandError.

4.4.2.2 Classe RescueInteractor

Por padrão, caso uma exceção seja lançada dentro de um interactor, ele interrompe sua execução e lança a exceção para o escopo de onde ele foi iniciado. A classe RescueInteractor foi criada com o intuito de modificar esse comportamento. Nela, quando uma exceção é lançada, o erro é tratado, sua execução é interrompida, o erro é armazenado em seu contexto e seu estado é modificado para o estado "fail". A grande diferença é que caso os interactors estejam rodando em sequência através de um Interactor::Organizer, e ocorrer uma exceção, o interactor que estiver rodando irá falhar graciosamente, permitindo que os interactors anteriores possam dar *rollback*.

4.4.2.3 Classe GitInterface

A classe GitInterface serve como interface para a execução de comandos Git. Ela foi criada para isolar todo o acesso externo da aplicação referente a ferramenta de controle de versionamento Git.

4.4.3 Comando generate

O comando *generate* é composto de 3 etapas, onde a implementação de cada etapa é um interactor. Além disso, são possíveis 2 fluxos, o fluxo normal e o fluxo de *debug*. Nas próximas seções serão explicadas cada uma dessas etapas e cada um dos fluxos.

4.4.3.1 Interactor BuildReleaseTag

Nesta etapa são realizados os seguintes passos:

1. As tags (git) locais são atualizadas com as tags (git) remotas
2. A última tag de *release* é identificada
3. O nome da próxima tag de *release* é gerado
4. O nome da próxima tag e a mensagem da próxima tag são encapsulados em um objeto da classe GitTag para que essas informações possam ser utilizadas nas próximas etapas

O passo 1 é trivial: basta executar um comando Git para baixar as tags remotas. No passo 2 cada uma das tags locais são convertidas em objetos da classe Semantic::Version, para que sejam comparados e se chegue a tag de versão mais recente. O passo 3 envolve uma série de abordagens até se chegar ao nome do próximo *release*. Abaixo estão listadas quais são essas as abordagens. Caso uma abordagem falhe, a próxima será executada, até que se ache em uma resposta.

- Utilizar a opção *"name"*, quando fornecida pelo usuário, como o nome do próximo *release*

- Utilizar a opção *"type"* (que pode ser major, minor ou patch), quando fornecida pelo usuário, para calcular qual será o nome do próximo *release*. Esta tarefa é realizada através da gem "semantic" que provisiona a classe Semantic::Version, que por sua vez possui métodos para incrementar qualquer uma das partes que compõem uma versão no padrão Semantic Versioning

- Utilizar o nome do último *commit*, caso ele seja um *commit* de uma operação *"merge"* do Git, para inferir qual é o tipo de *release*, baseado no nome da *branch* que foi mesclada. Caso a *branch* mesclada tenha o prefixo "hotfix/" e a mesma tenha a palavra *"feature"* no nome (case insensitive), o tipo "minor" será utilizado. Caso a *branch* mesclada tenha o prefixo "hotfix/" e a mesma tenha não possua palavra *"feature"* no nome (case insensitive), o tipo "patch" será utilizado. Caso a *branch* mesclada tenha o prefixo "release/", o tipo minor será utilizado. Por fim, caso a *branch* mesclada tenha o prefixo "major-release/", o tipo "major" será utilizado

Caso nenhuma abordagem funcione, um erro será lançado (UnknownReleaseTypeError) informando a impossibilidade de prosseguir a execução do programa. Caso alguma abordagem funcione, o nome da próxima tag de *release* é calculada, adicionando o prefixo e o sufixo que podem ser informados pelo usuário através das opções *"prefix"* e *"suffix"* do comando *generate*. Na figura 2 pode ser visualizado o fluxo lógico do terceiro passo.

No quarto passo, uma instância da classe GitTag é criada, contendo o nome da tag e a mensagem. Caso a mensagem da tag seja informada, através da opção *"message"*, esta deve ser utilizada; se não, a mensagem será gerada automaticamente no formato "Release <nome do release>".

4.4.3.2 Interactor BuildLocalGitTag

Nesta etapa a tag gerada anteriormente é persistida localmente, através do comando de anotação de tag do Git, no formato: `tag -a '<nome da tag>' -m '<mensagem>'`.

Neste interactor existe um método de *rollback*, para caso uma etapa posterior falhe, a tag criada seja deletada.

4.4.3.3 Interactor PushGitTag

Nesta etapa a tag gerada anteriormente é persistida no servidor remoto, através do comando push do Git, no formato: `git push <nome do servidor remoto> <nome da tag>"`.

4.4.3.4 Fluxo normal

No fluxo normal, as 3 etapas descritas anteriormente são executadas sequencialmente, com o objetivo final de criar a nova tag de *release* localmente e no servidor remoto.

4.4.3.5 Fluxo debug

No fluxo de *debug*, apenas a primeira etapa é executada, com o objetivo apenas de simular qual nome de tag seria criado. Este fluxo é útil para testar a ferramenta de forma manual, sem criar tags locais nem remotas.

4.4.4 Comando branch

O comando *branch* é composto de 2 etapas, onde a implementação de cada etapa é um interactor. Além disso, são possíveis 2 fluxos, o fluxo normal e o fluxo de *debug*. Nas próximas seções serão explicadas cada uma dessas etapas e cada um dos fluxos.

4.4.4.1 Interactor BuildReleaseBranchName

Nesta etapa são realizados os seguintes passos:

1. O interactor BuildReleaseTag é executado, para obter-se qual será o nome do próximo *release*
2. O nome da *branch* é criado, utilizando o nome do próximo *release* e um prefixo. O prefixo é obtido através do tipo de *release* que o usuário inseriu através da opção *"type"*. Caso o usuário insira o tipo "major", o prefixo será "major-release", se não, o prefixo será *"release"*

4.4.4.2 Interactor CreateReleaseBranch

Nesta etapa o nome da *branch* gerado anteriormente é utilizado para se criar uma nova *branch* local através do comando *checkout* do Git, no formato: `git checkout -b <nome da branch>`.

4.4.4.3 Fluxo normal

No fluxo normal, as 2 etapas descritas anteriormente são executadas sequencialmente, com o objetivo final de criar a nova *branch* local para se trabalhar em um novo *release*.

4.4.4.4 Fluxo debug

No fluxo de *debug*, apenas a primeira etapa é executada, com o propósito de verificar-se qual nome a *branch* teria caso fosse criada.

4.5 Aspectos complementares

4.5.1 Log

Para ter-se um *log* visível para o usuário, foi utilizada a classe Logger, nativa do ruby. Sua instância é criada estáticamente e utilizada por toda a aplicação. Abaixo pode-se visualizar um exemplo de linhas geradas pelo *logger* no terminal:

```
Versi@[2016-12-16 21:50:04]: Updating local git tags..
Versi@[2016-12-16 21:50:08]: Your latest release tag is "0.0.0"
Versi@[2016-12-16 21:50:08]: Your next release tag will be "0.1.0"
Versi@[2016-12-16 21:50:08]: No tag will be created, because we are in debug mode
```

O formato de *log* escolhido permite a visualização do que foi gerado durante a execução, e quando foi executado.

4.5.2 Rollback

Um trabalho especial foi realizado para garantir que a ferramenta Versi não deixe o usuário em um estado inconsistente. Isto foi feito através da implementação de etapas como interactors, que por sua vez permitem a implementação de um método *rollback* para garantir que em caso de falha a ferramenta possa se recuperar e desfazer o que foi feito.

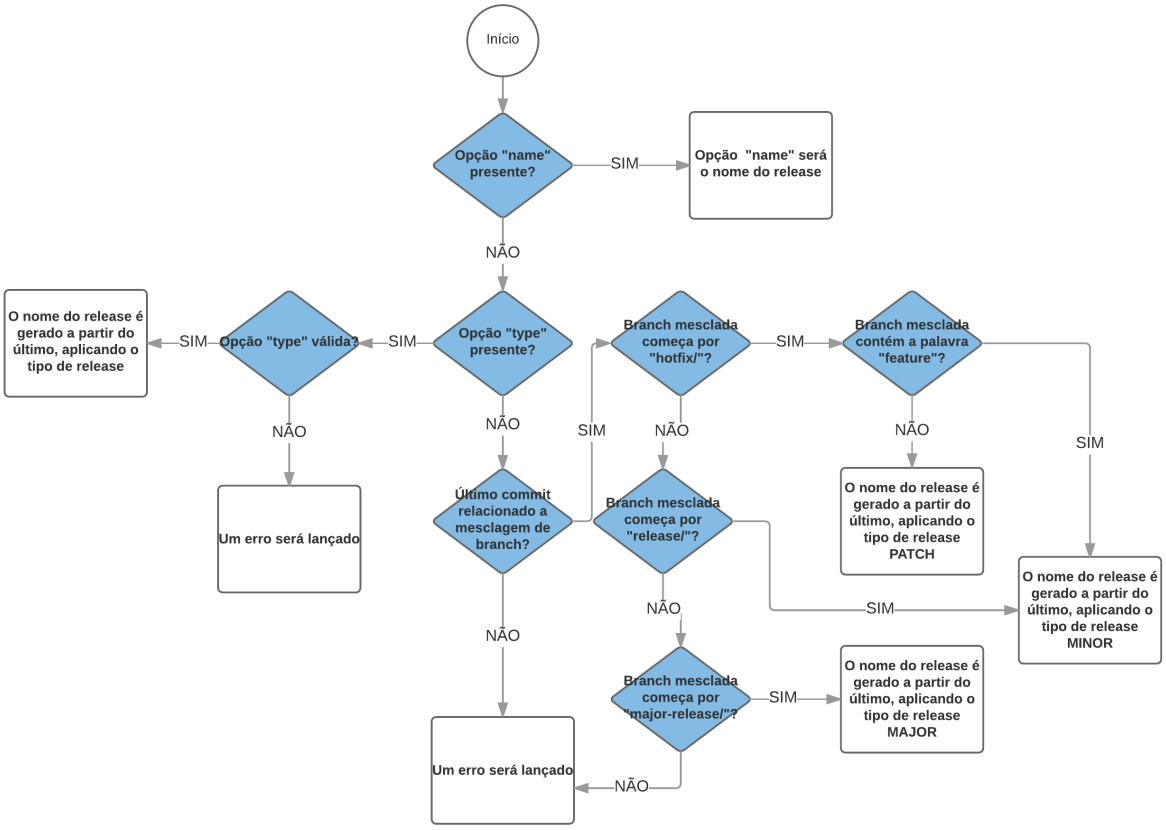

Figura 2 – Fluxo lógico do terceiro passo do Interactor BuildReleaseTag

5 Conclusão

Com este trabalho foi possível construir uma ferramenta simples que provê facilidades para a manipulação de versões no Git no padrão Semantic Versioning. Esta ferramenta é um exemplo de automação que pode ser realizada no dia-a-dia do desenvolvedor, provendo velocidade e menor suscetibilidade a erros. É um exemplo também de projeto que tem o potencial de auxiliar desenvolvedores de várias empresas, e de se tornar cada vez mais robusto com o engajamento da comunidade *open source*. Além disso ela contribui para a mudança de paradigma proposto pelo movimento DevOps, trazendo o desenvolvedor para perto da criação de *releases* e *deploys*.

No desenvolvimento foi possível sentir os problemas que surgem ao se desenvolver uma CLI, e foi possível aprender como que ela funciona internamente, sendo de grande utilidade para o desenvolvimento de futuras CLIs.

O objetivo do trabalho foi alcançado com sucesso, e proporcionou a experiência necessária para que o time de automação da In Loco Media conseguisse dar início ao desenvolvimento do seu próprio *toolbelt* (conjunto de ferramentas de automação).

Muito trabalho ainda precisa ser feito na ferramenta Versi: ela é pouco customizável, e não cobre alguns cenários de falha, como o caso de o usuário não ter Git instalado por exemplo. Mas a ferramenta já é bastante funcional, e está em uso na In Loco Media.

A ferramenta está disponível em [15].

Referências

1 HüTTERMANN, M. *DevOps for developers*. New York: Springer-Verlag New York, 2012. ISBN 9781430245698.

2 BASS, L.; WEBER, I.; ZHU, L. *Devops: A software architect's perspective*. United States: Addison-Wesley Educational Publishers, 2015. ISBN 9780134049847.

3 HUMBLE, J.; FARLEY, D. *Continuous delivery: Reliable software releases through build, test, and deployment automation*. United States: Addison-Wesley Educational Publishers, 2010. ISBN 9780321601919.

4 PRESTON-WERNER, T. *Semantic Versioning 2.0.0*. 2013. Disponível em: <http://semver.org/>.

5 PRODON, L. et al. *Liip/RMT*. 2016. Disponível em: <https://github.com/liip/RMT>.

6 BV, T. software. *TIOBE software PHP*. 2016. Disponível em: <http://www.tiobe.com/tiobe-index/php/>.

7 ULSBERG, A.; PARK, G.; BERGER, P. *GitTools/GitVersion*. 2016. Disponível em: <https://github.com/GitTools/GitVersion>.

8 INITIATIVE, O. S. *The MIT license (MIT)*. Disponível em: <https://opensource.org/licenses/MIT>.

9 MICHAELS-OBER, E.; KATZ, Y. *Erikhuda/thor*. 2016. Disponível em: <https://github.com/erikhuda/thor>.

10 WILLIAMS, M. *Mdub/clamp*. 2016. Disponível em: <https://github.com/mdub/clamp>.

11 GILDER, G. *Commander-rb/commander*. 2016. Disponível em: <https://github.com/commander-rb/commander>.

12 MICHAELS-OBER, E.; KATZ, Y. *Erikhuda/thor*. 2016. Disponível em: <https://github.com/erikhuda/thor/search?q=subcommand&type=Issues&utf8=\%E2\%9C\%93>.

13 LINDSEY, J. *Jlindsey/semantic*. 2016. Disponível em: <https://github.com/jlindsey/semantic>.

14 IDEA, C. *Collectiveidea/interactor*. 2016. Disponível em: <https://github.com/collectiveidea/interactor>.

15 SOBRAL, A. *Tonidas/versi*. 2016. Disponível em: <https://github.com/tonidas/versi>.