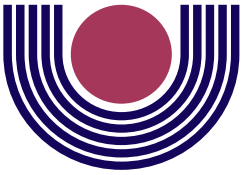

Unioeste - Universidade Estadual do Oeste do Paraná
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
Colegiado de Ciência da Computação
Curso de Bacharelado em Ciência da Computação

Engenharia de Linha de Produtos de Software Ágil: Um estudo exploratório utilizando o ecossistema Ruby

Samuel Blum Vorpagel

CASCAVEL
2015

Samuel Blum Vorpagel

Engenharia de Linha de Produtos de Software Ágil: Um estudo exploratório utilizando o ecossistema Ruby

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação, do Centro de Ciências Exatas e Tecnológicas da Universidade Estadual do Oeste do Paraná - Campus de Cascavel

Orientador: Prof. Ivonei Freitas da Silva

CASCAVEL
2015

Samuel Blum Vorpagel

ENGENHARIA DE LINHA DE PRODUTOS DE SOFTWARE ÁGIL: UM ESTUDO EXPLORATÓRIO UTILIZANDO O ECOSSISTEMA RUBY

Monografia apresentada como requisito parcial para obtenção do Título de Bacharel em Ciência da Computação, pela Universidade Estadual do Oeste do Paraná, Campus de Cascavel, aprovada pela Comissão formada pelos professores:

Prof. Ivonei Freitas da Silva
Colegiado de Ciência da Computação,
UNIOESTE

Prof. Victor Francisco Araya Santander
Colegiado de Ciência da Computação,
UNIOESTE

Prof. Elder Schemberger
Colegiado de Engenharia de Computação, UTFPR

Cascavel, 10 de março de 2016

...E não somente isso, mas também gloriemo-nos nas tribulações; sabendo que a tribulação produz a perseverança, e a perseverança a experiência, e a experiência a esperança... Romanos 5 3-4

AGRADECIMENTOS

Primeiramente agradeço a Deus pela saúde, forças, inspirações e toda ajuda nesses anos de curso. Sei que sem Ele não seria nada e sou extremamente grato. Em momentos de dificuldades Ele me estendeu a mão, em momentos de dúvidas Ele me acalmou e iluminou, em momentos de felicidades Ele esteve ao meu lado comemorando.

Agradeço a minha família por todo apoio emocional e financeiro nesses anos de curso. Em momentos de dificuldade, medos e anseios sempre estiveram ao meu lado. Mesmo que em certo momento os nossos caminhos tenham se afastado meu coração não se afastou, e sempre que possível nos encontraremos em uma esquina qualquer.

Agradeço aos meus amigos pelas ajudas em trabalhos e estudo em prova. Também sou grato por todas as vezes que pudemos sair, mesmo que raro, para jogar uma sinuca ou mesmo tomar uma cerveja ao final do dia dando uma distraída da rotina de trabalho e estudo.

Agradeço a minha namorada Mayara Gall, por todo apoio emocional que me deu em todos esses anos. Você faz minha vida mais feliz e meus dias mais especiais. Mesmo distante saiba que cada simples mensagem me traz uma alegria enorme.

Agradeço aos meus professores pelo conhecimento adquirido durante a graduação, em especial ao meu orientador Ivonei Freitas da Silva que durante esses anos auxiliou no desenvolvimento deste trabalho e suportou muitos erros cometidos por esse orientando.

Por fim sou grato a UNIOESTE por todo auxilio, bolsas, laboratórios, computadores e pesquisas que possibilitaram meu carecimento na graduação e o desenvolvimento deste trabalho.

Lista de Figuras

Lista de Quadros

Lista de Tabelas

Lista de Abreviaturas e Siglas

ASD	*Agile Software Development*
AT	*Acceptance testing*
CRUD	***C**reate, **R**ead, **U**pdate and **D**elete*
DSL	*Domain Specific Language*
FODA	*Feature Oriented Domain Analysis*
HTML	*HyperText Markup Language*
HTTP	*Hypertext Transfer Protocol*
MVC	*Model View Controller*
REST	*Representational State Transfer*
RoR	Ruby on Rails
SPL	*Software Product Line*
TDD	*Test Driven Development*

Sumário

Resumo

Linha de Produtos de Software Ágil representa o compromisso entre a agilidade no processo e o reúso de software. O ecossistema da linguagem Ruby tem sido utilizado em contextos de desenvolvimento ágil de software, porém, com raros exemplos em Linha de Produtos. Ruby tem algumas ferramentas e características que podem endereçar Linha de Produtos de Software. Este trabalho apresenta quais as ferramentas e características do ecossistema Ruby podem ser adotados para implementar uma Linha de Produtos de Software Ágil.

Palavras-chave: Linha de Produto de Software Ágil, Ecossistema Ruby e Ruby on Rails.

Capítulo 1

Introdução

Durante os últimos anos vem surgindo diversos estudos mesclando as metodologias ágeis com a `Linha de Produto de Software` (SPL - *Software Product Line*), agregando a flexibilidade oriunda das metodologias ágeis e a vantagem econômica provinda da reutilização e customização oferecidas por SPL [Ghanam 2012].

A proposta de Yaser Ghanam [Ghanam 2012] integra a abordagem ágil com SPL utilizando práticas ágeis, tais como *Test Driven Development* (TDD), refatoração e desenvolvimento iterativo e incremental. Essa metodologia pode ser adotada por projetos com recursos e orçamento limitados, já que ela possui um processo mais leve comparado a outras abordagens de SPL Ágil [Bayer et al. 1999].

A SPL não necessita ou depende de linguagem, *framework* e outras ferramentas específicas, mas se beneficia muito das ferramentas que automatizam a implementação de variabilidade em uma SPL, bem como, ferramentas para derivação de produtos [Ghanam 2012]. Através de alguns estudos preliminares é possível observar que algumas características valorizadas em SPL [Gacek e Anastasopoules 2001], como estão contidas tanto no `Ruby` [Flanagan e Matsumoto 2008] quanto no *framework* `Ruby on Rails` (RoR) [Hansson 2014]. O `Ruby` contém nativamente suporte a orientação a objetos, orientação a aspectos, modulação e metaprogramação [Fuentes 2013] e o RoR possui como características geradores automáticos e semiautomáticos, convenções de desenvolvimento e fácil gerenciamento de dependências.

Mesmo que RoR não tenha sido criado especificadamente para SPL, essas suas características o mostra com um potencial interessante para atendê-la. O RoR também possui algumas características interessantes voltadas ao desenvolvimento ágil como códigos enxutos, suporte

a TDD e fácil configuração, [Stella, Jarzabek e Wadhwa 2008], além dos pacotes de distribuição `RubyGems` e o constante incentivo da comunidade ao desenvolvimento utilizando TDD [Fuentes 2013], elevando assim, o ecossistema[1] `Ruby` a um potencial ainda maior quando se visa o método proposto por Yaser Ghanam.

O RoR é utilizado principalmente em projetos que necessitam ser experimentados ou de retorno rápido ao cliente final. Esses projetos necessitam de ferramentas que permitam lançar seus produtos rapidamente no mercado [Geer 2006]. Utilizar o processo de Yaser Ghanam em conjunto com o RoR pode mitigar os problemas relacionados aos projetos com recursos e orçamento limitados e retorno rápido aos clientes.

Estudos realizados por Yaser Ghanam mostraram que, empregando as técnicas de SPL, a economia gerada é notável e, quando utilizando as técnicas de metodologia ágil as empresas conseguem reduzir os custos que teriam na elaboração de uma SPL com um processo mais pesado. Além disso, desenvolvedores que optaram por RoR relataram que houve um aumento significativo em produtividade conseguindo até mesmo desempenhar o papel de equipes inteiras de desenvolvimento em outras linguagens [Geer 2006].

1.1 Motivações e Justificativa

O uso do ecossistema `Ruby` é cada vez mais presente, principalmente para soluções *web* pelo seu *framework* de desenvolvimento *web* `Ruby on Rails` [Fuentes 2013]. SPL também vem sendo cada vez mais visada, porém, por conta de seu custo elevado e de sua complexidade, empresas de pequeno e médio porte relutam a utilizar. Além disso, SPL se contrapõem as metodologias ágeis comumente adotadas em empresas de pequeno e médio porte [Ghanam 2012] pois se utilizam de um processo pesado na etapa de levantamento de requisitos e não é comum uma SPL adotar práticas das metodologias ágeis como entrega contínua.

Muitos estudos vem surgindo a fim de estabelecer uma SPL Ágil, unindo assim, o melhor de cada abordagem. Na grande maioria das vezes estes estudos utilizam linguagens como `Java`, `C` e `C++`.

Estudos iniciais [Flanagan e Matsumoto 2008, Gacek e Anastasopoules 2001] mostraram

[1] O Termo **ecossistema** nesse trabalho é utilizado para definir todo e qualquer software, *framework*, biblioteca, ferramenta que fazem relação direta ou indiretamente com o `Ruby`.

que Ruby tem potencial para SPL e para metodologias ágeis. Outros estudos apontam que desenvolvedores Ruby são mais produtivos que desenvolvedores em outras linguagens como Java e .NET [Stella, Jarzabek e Wadhwa 2008, Geer 2006].

A linguagem Ruby é reconhecida por sua facilidade de aprendizado, auto índice de testabilidade, extensibilidade e reusabilidade e o grande número de ferramentas e funcionalidade existentes juntamente sua forte ligação com TDD o torna interessante para o método estudado [Ghanam 2012] pela forma com que ele utiliza os testes de aceitação (AT) e o TDD.

A principal motivação do trabalho é verificar se o ecossistema Ruby permite o desenvolvimento de SPL ágil, para isso, o processo adotado no trabalho é o proposto por Yaser [Ghanam 2012] que até o momento é o que mais se aproxima das metodologias ágeis, como pode ser visto na Figura 1.1.

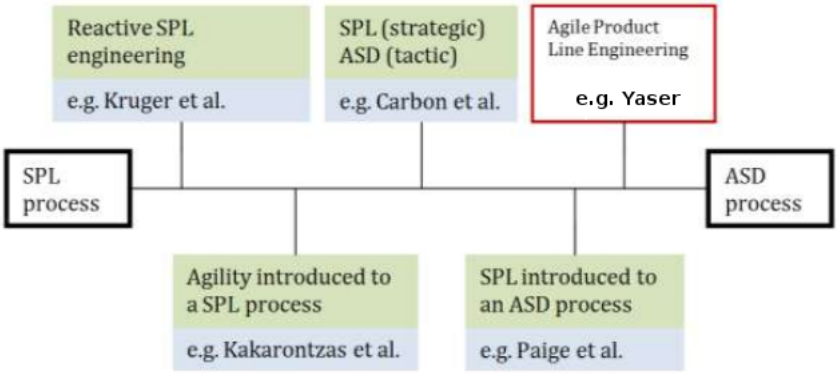

Figura 1.1: Comparação entre os diversos processos de LP, Adaptado de [Ghanam 2012]

1.1.1 Problemas

O ecossistema Ruby e o seu framework Ruby on Rails não foram planejados para a construção de SPL ou mesmo de SPL ágil. Assim é necessário explorar a viabilidade técnica em relação a existência de recursos no ecossistema Ruby presentes na SPL ágil, em especial no método proposto por Yaser [Ghanam 2012]. Explorar essas características é o primeiro passo para viabilizar o desenvolvimento de SPL ágeis em um contexto industrial, que adote o ecossistema Ruby.

Para explorar se o ecossistema Ruby e o *framework* Ruby on Rails oferecem os recursos para implementação de SPL ágil a seguinte questão foi definida:

Q: É possível com o ecossistema `Ruby` realizar o processo de SPL ágil descrito por Yaser?

1.2 Objetivos e resultados esperados

O principal objetivo deste trabalho verificar se o é possível realizar uma iteração do prcesso descrito por Yaser, e assim, fornecer um estudo exploratório implementando uma SPL ágil utilizando o *framework* `Ruby on Rails`. A documentação deste estudo pode servir como um modelo prático para a indústria que deseja implementar linha de produto utilizando o ecossistema `Ruby` e o *framework* `Ruby on Rails` por meio de um processo leve, evitando assim, perder as características de metodologias ágeis já utilizadas. Além disso, essa documentação fornecerá *insights* para futuras pesquisas.

O resultado deste trabalho é um estudo de caso prático que implementa uma SPL ágil utilizando o processo de Yaser [Ghanam 2012] e a linguagem `Ruby` com o seu *framework* de desenvolvimento web `Ruby on Rails`.

1.3 Organização do Documento

O capítulo 2 contém os principais conceitos para o andamento do trabalho, a fundamentação teórica. Neste capítulo estão expostos conceitos como: O que é linha de produto de software, `Ruby`, motivos para utilizar o `Ruby`, o que é desenvolvimento ágil e algumas de suas técnicas de desenvolvimento.

O capítulo 3 apresenta o processo de SPL Ágil adotado para esta monografia, apresentando todas as etapas e um resumo do processo em si. A implementação do capítulo 3 se encontra no capítulo 4, no qual é definido e documentado um estudo exploratório.

O documento é finalizado com o capítulo 5, onde são apresentadas as conclusões e resultados obtidos bem como problemas encontrados e sugestões para futuros avanços no tema.

Capítulo 2

Fundamentação Teórica

A primícia deste trabalho é a verificação do ecossistema `Ruby` em uma abordagem de SPL ágil. Para isso, é necessário uma base teórica em diversos aspectos das metodologias ágeis, SPL e também sobre características e ferramentas do ecossistema `Ruby`. Neste capítulo serão descritas de forma sucinta as técnicas e ferramentas utilizadas no decorrer do trabalho.

2.1 Desenvolvimento Ágil de Software

O Desenvolvimento Ágil de Software (ASD do inglês - *Agile Software Development*) é um conjunto de metodologias de desenvolvimento de software iterativo que prioriza a satisfação do cliente e desenvolvedores [Willi et al. 2001].

Os praticantes de ASD defendem que o desenvolvimento iterativo incentiva valores como a comunicação entre todos os envolvidos, *feedback* do cliente, iterações menores e contínuas e o desenvolvimento *just-in-time* [Willi et al. 2001].

A principal vantagem do ASD é sua capacidade de adaptar-se às mudanças de requisitos tão comuns nos projetos de software, principalmente nas fases iniciais quando é mais visível a incerteza do usuário a cerca do projeto [Highsmith e Cockburn 2001]. Em contrapartida, uma das maiores desvantagens da metodologia ágil é a dificuldade de reusar componentes, visto que é comum desenvolver apenas o mínimo para o funcionamento de um determinado produto não pensando em possíveis variações do mesmo.

Nas subseções abaixas serão apresentadas algumas características importantes da ASD para este trabalho.

2.1.1 Testes Contínuos

Testes contínuos são uma das atividades comuns em ASD na qual para toda a implementação são criados diversos testes descrevendo os comportamentos esperados. Os testes permitem a detecção rápida de erros diminuindo o custo de corrigi-los [Highsmith e Cockburn 2001]. Outra vantagem é a garantia de que o software está coberto por uma suíte de testes e as futuras mudanças realizadas na aplicação estão asseguradas por essa suíte. Em caso de erros ou mudanças de comportamento por parte da aplicação os testes devem falhar e mostrar os erros.

Os testes contínuos são um dos principais artefatos do TDD, que tem como objetivo favorecer a refatoração constante dos códigos e melhorar o design e a arquitetura. No TDD o desenvolvimento é guiado por testes, sendo assim, primeiro são desenvolvidos os testes para só então desenvolver o código que valide esses testes [Baraúna 2010].

2.1.2 Testes Automatizados

Os testes automatizados são executados por uma ou mais máquinas podendo assim, ser executado constantemente e a cada alteração do sistema para verificar se ouve alguma quebra de integridade. Com testes automatizados é possível descobrir erros no software de forma mais rápida e, consequentemente, as medidas para remediá-lo também serão aplicadas de forma mais rápida.

São utilizados diversos tipos de testes na suíte de testes automatizados, como por exemplo, testes de aceitação e de unidade [Baraúna 2010]. A forma mais comum de utilizar testes automatizados é inserindo entradas e verificando se a saída é a esperada.

2.1.3 Testes de Unidade

Testes de unidade visam cobrir pequenas partes do sistema, denominada unidade, e garantir que cada unidade funcione como esperado. Uma unidade é definida como uma pequena parte do sistema e o teste de unidade visa verificar se o comportamento ou a interação desta unidade está correto.

As metodologias ágeis defendem que quanto menor a distância do teste para o desenvolvimento, melhor [Beck 2003]. O TDD traz isso ao extremo tornando o desenvolvimento baseado nos testes. O ciclo do TDD visa garantir uma alta cobertura de testes e uma melhor modula-

ridade [Beck 2003] e com os testes automatizados é possível ter uma melhora significativa na qualidade do código por conta da refatoração constante [Aniche 2013].

2.1.4 Teste de Aceitação

Os testes de aceitação tradicionalmente consistem de uma série de documentos escritos em uma linguagem natural, já as especificações executáveis são descritas em linguagens, geralmente em inglês, que variam de formal [Fuchs 1992] até próximas a linguagem natural [FIT 2007].

Em ASD são utilizadas linguagens muito próximas do natural, geralmente em inglês (*english-like*) para definir testes, cenário [Kaner 2003], histórias [Kerievsky 2010] ou testes de aceitação [Perry 2006].

O comportamento de um sistema pode ser determinado por casos de teste ou cenários preferencialmente feitos em conjunto com especialistas de domínio, clientes e usuários. Os testes e cenários, quando escritos em conjunto com cliente ou especialista de domínio, podem ser utilizados para a validação junto ao cliente e se automatizados é possível previamente verificar se o comportamento corresponde ao esperado. Testes de aceitação geralmente utilizam ferramentas como `RSpec` [Chelimsky et al. 2010] para serem escritos e executados.

2.1.5 Refatoração

Em Fowler [Fowler 1997], refatoração é definida como o processo de mudança de um sistema de software de forma que não altere o seu comportamento externo e gere uma melhora a sua estrutura interna.

A refatoração tem como principal objetivo a melhora do *design* do código, já escrito previamente, tornando-o mais legível, testável, reutilizável e fácil na manutenção. Práticas comuns na refatoração são a extração de métodos de um segmento de código, renomeação de variáveis ou métodos, criação ou abstração de métodos ou classes, modificação de código para atender algum padrão de projeto específico entre outras práticas.

Para que a refatoração não altere o comportamento do código geralmente é utilizado testes executáveis a fim de garantir que o comportamento anterior mantenha-se após as mudanças no código.

2.2 Linha de Produtos de Software

SPL é uma família de produtos de software que compartilham um conjunto comum de *feature*[2], possuindo variações devido as necessidades de cada cliente [Clements e Northrop 2002].

O alto nível de reutilização entre sistemas, acentuado em SPLs, torna a SPL vantajosa economicamente e a flexibilidade para certas variações dentro das *features* reutilizáveis faz com que a customização em massa seja possível [Clements e Northrop 2002].

Com o reaproveitamento das *features* existem diversos benefícios em médio e longo prazo, tais como diminuição do custo de manutenção e agilidade para entregar novos produtos [Clements e Northrop 2002].

Nas subseções seguintes serão descritas algumas características da SPL importantes para este trabalho.

2.2.1 Variabilidade

A variabilidade, em um contexto de sistema de software, é a percepção que os componentes de uma arquitetura de software podem variar por uma série de fatores, como por exemplo, as necessidades de clientes ou simplesmente a estratégia de negócio. Pohl [Pohl, Böckle e Linden 2005], trata a variabilidade de uma linha de produto por uma série de pontos de variação, um conjunto de variantes para cada possível ponto de variação e possíveis restrições.

A variabilidade em um sistema pode ocorrer em duas formas, por lógica de negócio ou por meio de requisitos não-funcionais. O cenário ou mesmo o fluxo de trabalho que um usuário deve seguir na aplicação é determinado pela lógica de negócio. A usabilidade, portabilidade, segurança entre outros é determinado pelos requisitos não-funcionais [Clements e Northrop 2002].

2.2.2 Reúso de Software

O reúso de software é a construção de aplicações de software utilizando artefatos utilizados na construção de outras aplicações ou software [Frakes e Fox 1995], além disso, diversas pesquisas buscam estender esse termo, "reúso de software", para aspectos como o

[2]O termo *Feature* remete a uma funcionalidade em um sistema de software. Essa funcionalidade pode ser traduzida para requisito funcional, não funcional ou até mesmo uma combinação de ambos. [Clements e Northrop 2002]

dimensionamento para reutilização [Bieman e Kang 1995], a implementação de reutilização [Prieto-Diaz 1996], gerenciamento de ativos reutilizáveis [Henninger 1997], busca e recuperação de ativos reutilizáveis [Frakes e Pole 1994] e outros. Diversas pesquisas vão além da reutilização de código e incluem outros artefatos para reutilizar como documentos de projeto, casos de uso e casos de testes [Mohagheghi 2004].

Os sistemas de software em um determinado domínio tendem a ter um maior potencial de reutilização pois resolvem problemas semelhantes, mas esse potencial já é grande mesmo em domínios distintos, como cita Yaser [Ghanam 2012] ao comparar sistema de gerenciamento de arquivos em diversos sistemas operacionais.

Durante sua pesquisa, Yaser [Ghanam 2012] cita alguns benefícios do reúso tais como:

- A entrega de novos produtos mais rápido.
- Redução dos custos de desenvolvimento e manutenção.
- Existe melhora na qualidade dos artefatos existentes.
- Estimativas de projeto mais precisa.

Níveis de reúso

Existem diversos níveis que podem ocorrer reutilização, como as cópias de um fragmento de código ou classe, para utilizar em outro contexto [Ghanam 2012].

Em um nível maior, a reutilização pode acontecer em nível de componente. Existem técnicas de engenharia de software para desenvolvimento baseada em componentes [Councill e Heineman 2001] as quais permitem o desenvolvimento e composição de software de forma flexível.

2.2.3 Engenharia de Domínio e Aplicação

A fase de Engenharia de Domínio e a fase de Engenharia de Aplicação fornecem o processo geral para construir uma SPL [Pohl, Böckle e Linden 2005]. Uma das imagens mais tradicionais que representa esse processo pode ser visto na figura 2.1.

Durante a fase de engenharia de domínio é feita uma análise detalhada para especificar o escopo e similaridades e variabilidades da SPL. Definir o escopo é definir o limite da família de

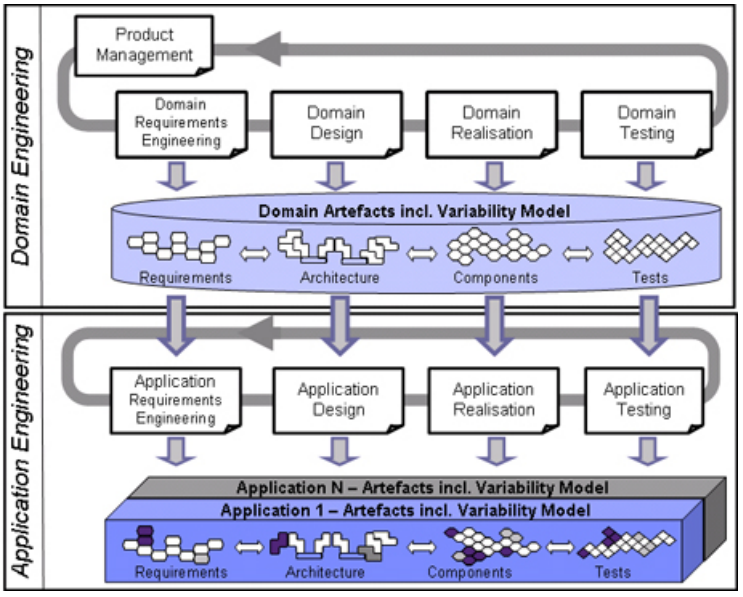

Figura 2.1: Engenharia de Domínio e Engenharia de Aplicação [Pohl, Böckle e Linden 2005]

produto deixando evidente os produtos que se encaixam e não encaixam na SPL. Ao analisar a engenharia de requisitos dos membros da família de software é possível verificar as variabilidades. É nesse ponto que são determinados os pontos de variação[3] e as variabilidades[4], da documentação, da implementação e dos testes. Outra decisão que cabe a fase de engenharia de domínio é a de determinar quais serão os artefatos reutilizados e como terão que ser projetados para que possam ser reutilizados futuramente [Pohl, Böckle e Linden 2005].

A fase de engenharia de aplicação, que tradicionalmente ocorre após a engenharia de domínio, aproveita todos os artefatos colhidos na fase de engenharia de domínio e os deriva como uma aplicação. O objetivo desta fase é, utilizando técnicas da SPL, trabalhar em uma aplicação específica. O conhecimento colhido nessa fase realimenta a fase da engenharia de domínio criando um ciclo entre esses dois processos. O processo de engenharia de requisito e design também são executados para um contexto de domínio [Pohl, Böckle e Linden 2005].

2.2.4 Derivação de Produto

A derivação de um produto consiste na construção de produtos individuais utilizando os artefatos produzidos pela engenharia de domínio. Esse processo tradicionalmente acontece na fase da engenharia de aplicação explicada na seção 2.2.3.

[3]Ponto de variação: Local onde ocorre variação.
[4]Variabilidade: Diferentes construções de uma determinada *feature* em um ponto de variação.

Durante a derivação de produto, a construção dos produtos pode ser feita de duas formas, por montagem ou por configuração [Deelstra, Sinnema e Bosch 2005]. Na montagem, a união de um subconjunto de artefatos é utilizado para criar um produto único, os artefatos utilizados são provindos da base de artefatos já existente. Na configuração o produto é criado seguindo uma série de parâmetros e conjunto de valores de configuração para instanciar o produto [Bosch e Högström 2001], porém, nem sempre configuração é o suficiente para realizar a derivação de um produto [Thao, Munson e Nguyen 2008].

Yaser [Ghanam 2012] utiliza uma combinação destas práticas, utilizando a configuração com conceitos de gerenciamento de variabilidade contido no método de montagem como detalhado nos Capítulos 8 e 9 de seu trabalho.

2.3 Ecossistema Ruby

O `Ruby` é uma linguagem de programação interpretada, fortemente tipada e dinâmica. `Ruby` nasceu com a proposta de ser simples priorizando o desenvolvedor em relação a máquina, sendo legível e agradável de se programar. Foi criada por Yukihiro Matz Matsumoto em 1993 e publicada em 1995 sendo uma combinação das linguagens `Perl`, `Smalltalk`, `Eiffel`, `Ada` e `Lisp` [Flanagan e Matsumoto 2008].

Em `Ruby` tudo que utilizamos são objetos, mesmo os tipos primitivos são objetos isso permite que os desenvolvedores alterem o comportamento de uma classe ou até mesmo de um objeto específico em tempo real.

Uma característica interessante de `Ruby` é que com os traços de linguagens funcionais ele herda recursos poderosos destas linguagens como *lambdas* e *closures* [Flanagan e Matsumoto 2008].

A linguagem `Ruby`, em sua forma pura, é muito utilizada na criação de *scripts* de leitura e processamento de arquivos, *builds* automatizados e *deploys*. `Ruby` conta com diversos *frameworks* para os mais diversos tipos de finalidades, dentro deles, o que mais se destaca pela sua repercussão e adoção, é o `Ruby on Rails` um *framework* para desenvolvimento web que visa livrar o programador de tarefas massantes como a criação das CRUD[5] [SOUZA 2013] e o gerenciamento de bibliotecas externas graças o trabalho em conjunto com o `Bundler`

[5] Acrônimo de ***C**reate*, ***R**ead*, ***U**pdate* e ***D**elete*

[Arko André].

As funções criadas em Ruby possuem características incomuns em outras linguagens como por exemplo, suportar, tanto como argumento quanto para retorno, blocos ou até mesmo funções, boa parte devido à característica de interpretar tudo como objeto.

2.3.1 Características do Ruby importantes para SPL

Por padrão, o Ruby possui diversas características e funcionalidades que o tornam interessantes no contexto de SPL como Mixins, bloco, Módulos e suporte a Metaprogramação.

Os Módulos possuem a função de agregar uma série de classes, funções e métodos a fim de poder distribuir com mais facilidade. Sua ideia em certos aspectos assemelha-se aos pacotes presentes no Java.

Uma extensão dos Módulos são os Mixins. Os Mixins são recursos do Ruby para o compartilhamento de código sem a necessidade de herança. Com sua utilização é possível uma classe usufruir de características provinda de diversas outras classes, objetos ou métodos.

Lucas Souza [SOUZA 2013] cita algumas vantagens de utilizar Módulos e Mixins em lugar da herança.

- Módulos podem ser gerenciados em tempo de execução através do método include, enquanto herança é definido no momento da escrita da classe;
- Módulos são mais fáceis de testar unitariamente de maneira isolada;
- Módulos nos permite utilizar Metaprogramação para definir Domain Specific Languages (DSL) [Maximilien et al. 2007];
- Módulos são classes que não podem ser instanciadas, eles existem apenas para adicionar funcionalidades para classes já existentes.

A Metaprogramação permite que o desenvolvedor não fique limitado a abstração que a linguagem oferece, permitindo até mesmo criar DSLs próprias estendendo o poder da linguagem de programação. Outra característica da Metaprogramação é permitir modificações de comportamento em tempo de execução.

2.3.2 Características do Ruby importante para ASD

A linguagem Ruby possui características interessantes para o desenvolvimento seguindo a filosofia ASD como o suporte nativo ao TDD, sintaxe simples de fácil leitura e escrita contendo ainda diversas Gems com a capacidade de realizar atividades do ASD [Berube 2007].

As *Gems* são "bibliotecas" e *frameworks* que adicionam diversas ferramentas e funcionalidades para o Ruby. Dentre os mais famosos é essencial citar o Ruby on Rails, RSpec e Cucumber.

Uma característica que torna o uso das Gems prático é a forma com que elas são organizadas e gerenciadas, boa parte deste trabalho é feito pelo Bundler em conjunto com o Gemfile que por meio de comandos é possível atualizar e instalar todas as dependências de Gems do projeto.

A comunidade do Ruby é conhecida pelo uso constante do TDD [Baraúna 2010], raramente é visto um projeto *open source* escrito em Ruby sem uma suíte de testes bem definida. Essa característica é acentuada pelos diversos *frameworks* existentes para testes.

A principal característica de Ruby que o torna interessante para ASD seja a sua filosofia. Ao criá-la, Yukihiro Matsumoto [Flanagan e Matsumoto 2008] projetou-a sob o objetivo de ser uma linguagem de fácil e rápido desenvolvimento que proporcionasse ao desenvolvedor felicidade e satisfação ao utilizá-la. Outra característica filosófica é a sua flexibilidade que permite que o desenvolvedor adapte qualquer parte da linguagem para sua real necessidade, isso implica que, do tipo mais primitivo ao de maior nível, o programador possa fazer as modificações que achar necessário, tanto em comportamento ou simplesmente notação para uma padronização própria.

2.3.3 Ruby on Rails

O Ruby on Rails (RoR) é um *framework* para desenvolvimento web que se utiliza da estrutura MVC (Model View Controler) [Fuentes 2013] o qual não se baseia na criação de páginas webs e nas interações entre essas páginas mas sim em recursos. Esses recursos são baseados no REST (*Representational State Transfer*). Os recursos são tudo o que uma aplicação pode servir aos usuários [Fuentes 2013] e, em um contexto de SPL, podem ser interpretados como as *feature*.

Os Modelos criados no Ruby on Rails utilizam o componente ActiveRecord, ele

é um componente que faz o mapeamento de estruturas relacionais para objetos (*Objetct-Relational-Mapping*). Outra característica do `ActiveRecord` é o uso da biblioteca `Arel` capaz de transformar métodos em estruturas SQL facilitando na hora de consultas e manipulações no banco de dados.

Outro componente importante presente nos modelos é o `ActiveModel` que possui diversas ferramentas para manipulação de atributos, facilitando bastante a integração com formulários ou até mesmo traduções de atributos.

A camada de controle é responsável pela comunicação entre as camadas, gerenciamento das sessões, *cookies* entre outros. Assim como o modelo, o controle possui diversos componentes como o `ActionDispatch` e o `ActionController`. O `ActionDispatch` é um componente de menor nível responsável pelo *parsing* dos cabeçalhos HTTP e outras atividades, já o `ActionController` permite que o desenvolvedor trate das requisições com os modelos e regras adequadas à sua situação.

A camada de apresentação (View) é responsável por dar a resposta visual ao usuário final, no caso do RoR geralmente feito para web via páginas HTML. Para isso ser possível ele conta com o componente `ActionView` [Fuentes 2013].

2.3.4 RubyGEM

`RubyGEM` é o formato de distribuição de "bibliotecas"do `Ruby`. No fundo, `RubyGEM` é um pacote compactado contendo diversos códigos `Ruby`, *scripts* e outros arquivos que podem ser acoplados em projetos.

Esses arquivos podem ser distribuídos por meio de repositórios `git`[6], arquivos locais ou mesmo servidores destinados para esse fim. Além disso, esses arquivos são facilmente gerenciados pelo `Bundler`. As `RubyGEMs` tem como característica a possibilidade de estendê-las e modificá-las, além do fator reúso [Guide - RubyGems].

2.3.5 Cucumber

O `Cucumber` é uma ferramenta de testes de aceitação, podendo assim automatizar os testes de comportamento de software. Os testes criados no `Cucumber` não são especificações

[6]Git é um sistema de controle de versão distribuído e um sistema de gerenciamento de código fonte

de classes ou métodos como em outros *frameworks* e sim de uma funcionalidade do sistema [Baraúna 2010].

O desenvolvimento *outside-in*[7] é padrão no `Cucumber`, uma vantagem deste estilo de desenvolvimento é que ao iniciar o desenvolvimento pela camada mais externa para a mais interna do projeto, apenas é implementado o necessário evitando eventuais funcionalidade ou código que jamais seriam utilizados.

Com o `Cucumber` é possível criar uma documentação em linguagem natural em diversas linguagens. Essa linguagem faz com que o código possa ser escrito e entendido por desenvolvedores, testadores analistas, clientes, usuários entre outros, auxiliando assim o entendimento do projeto por todas as partes envolvidas.

Segundo o autor Hugo Baraúna [Baraúna 2010], o uso do `Cucumber` agrega valor nos seguintes pontos:

- Estimula a conversa com os *stakeholders* de modo a entender melhor os requisitos do sistema.
- Une a especificação do software junto aos testes automatizados, permitindo que essa documentação sempre fique atualizada em relação ao comportamento real do software.
- Faz com que a referência da documentação do sistema fique tão importante quanto o código do software e mais simples de ser lida e compreendida em comparação ao código.

O `Cucumber` permite a construção de especificações ágeis (Seção 2.1.1), unindo-as com testes automatizados, diminui o risco das especificações ficarem obsoletas [Baraúna 2010].

2.3.6 RSpec

O `RSpec` é uma biblioteca de testes de unidade sob a abordagem BDD[8] escrito em `Ruby`[Baraúna 2010] possuindo uma DSL simples que lembra a língua inglesa.

Este *framework* é muito utilizado em aplicações `Ruby on Rails` mas nada impede de ser utilizada no `Ruby` ou em outras linguagens. Existem diversos módulos para o `RSpec` que permitem expressar histórias, cenários e expectativas [Vieira 2012].

[7] Sequência de desenvolvimento iniciado na camada mais externa regredindo até a camada mais interna

[8] ***B**ehavior **D**riven **D**evelopment [Baraúna 2010]* derivada do TDD enfatizando o comportamento em lugar do Teste

2.3.7 Bundler

O `Bundler` é uma ferramenta para o ecossistema `Ruby` que permite a instalação e gerenciamento de `RubyGEMs`, controlando até mesmo as diversas versões de uma `RubyGEM` [Arko André].

As diversas `RubyGEMs` do projeto podem ser descritas em um arquivo chamado de `Gemfile` e instalado por meio do comando `bundle install`. Isso permite que toda a equipe de desenvolvimento utilize a mesma versões de cada biblioteca (quando especificado versão).

Capítulo 3

Processo de construção de uma SPL Ágil

Embora existam diversas divergências em praticamente todas as etapas do desenvolvimento de projetos utilizando ASD e SPL que vão da forma de realizar levantamento de requisitos até a forma que são entregue os produtos, existem estudos com o intuito de mesclar ambos, agregando assim a flexibilidade e as respostas rápidas provinda das metodologias ágeis sem perder as vantagens econômicas, customização e reúso de componentes da SPL [Ghanam 2012]. Com a união de ambas as técnicas é possível reduzir as desvantagens de cada uma aproveitando boa parte de suas vantagens.

Existem diversas pesquisas e propostas de técnicas de SPL Ágil [Silva et al. 2011], mas justamente pelas suas discrepâncias dificilmente essas técnicas conseguem usufruir de todo o potencial de uma SPL e ao mesmo tempo ser leve como uma abordagem inteiramente ágil. As propostas da SPL Ágil variam de acordo com seus graus de fidelidade com SPL ou com ASD. Algumas propostas mantêm-se mais conservadoras, como por exemplo a proposta por Kruger [Krueger 2006], e outras buscam ser mais leves, como a proposta por Yaser [Ghanam 2012], veja figura 1.1.

Durante esse trabalho será utilizado a técnica proposta por Yaser, veja figura 3.1. Essa técnica, até a publicação deste trabalho, é a que mais se aproxima da metodologia ágil de desenvolvimento, veja figura 1.1. Ela integra diversas técnicas da ASD com SPL, tais como Behaviour-Driven Development (BDD), refatoração e desenvolvimento iterativo e incremental. Ela pode ser adotada por projetos com recursos e orçamento limitados, já que o processo é leve se comparado a outras abordagens de SPL [Schmid e Widen 2000], principalmente para empresas que já possuam alguma familiaridade com ASD.

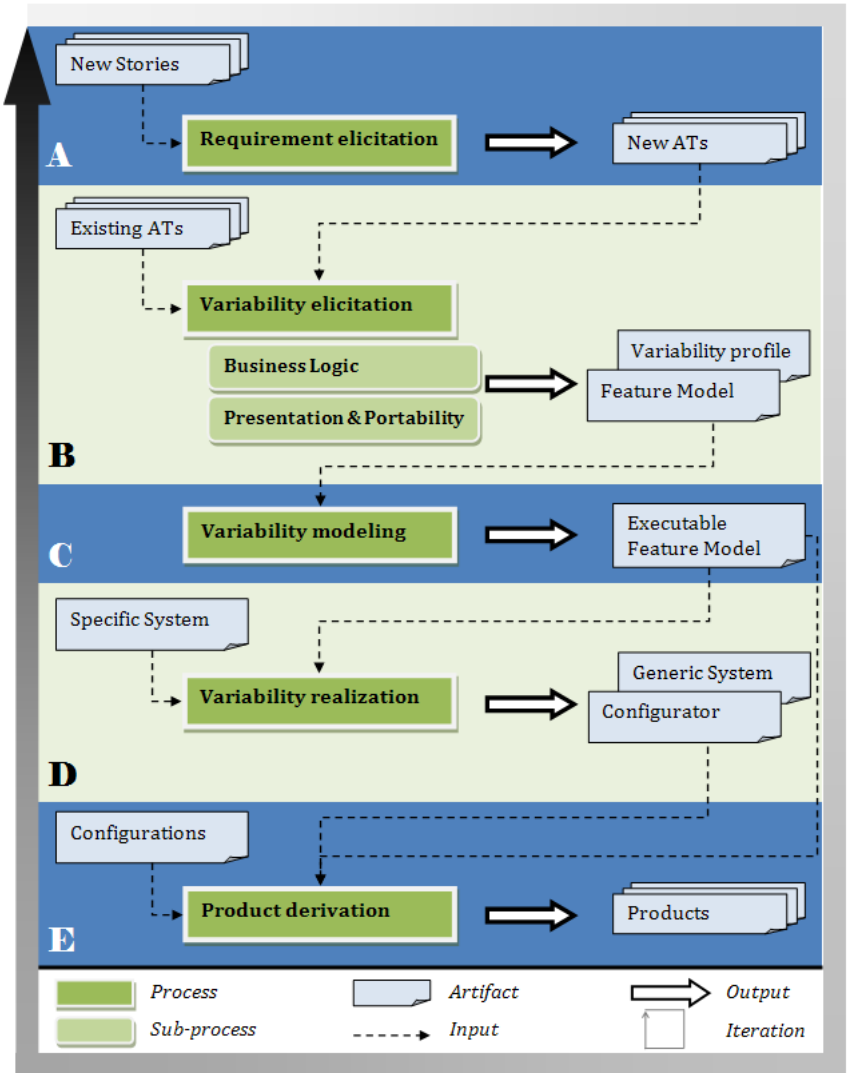

Figura 3.1: Processo proposto por Yaser [Ghanam 2012]

3.1 Resumo geral do processo

O método proposto por Yaser basicamente substitui toda a documentação pesada que uma SPL necessita por testes de aceitação. Isso é possível pela característica dos testes de aceitação, quando bem executado, de documentação viva[9], garantindo a fidelidade da documentação com o código e a garantia que essa documentação nunca se tornará obsoleta.

Como essa abordagem é reativa, não necessariamente para todo o produto criado será desenvolvido uma SPL, por isso, o primeiro produto de uma futura SPL é criado por meio de

[9]Documentação viva é um termo comumente usado pela comunidade `Ruby` para evidenciar o fato dos testes serem uma documentação sempre atualizada e referenciando realmente o que o código faz, diferente da documentação tradicional que, se não atualizada, torna-se obsoleta.

um processo natural de ASD. A única necessidade é de que esse produto realmente utilize das técnicas recomendadas para o desenvolvimento ágil, como por exemplo a criação de testes de aceitação [Ghanam 2012].

Após o primeiro produto inicia-se o desenvolvimento realmente guiado pelo método, o qual, divide-se em cinco etapas, onde as duas primeiras são etapas voltadas para o levantamento de requisitos e detecção de variabilidade, as duas próximas voltadas ao desenvolvimento das variabilidades e por fim, a última etapa voltada a derivação de produtos.

Cada uma das etapas do processo, possui entradas, saídas e algumas atividades que devem ser desempenhadas para alcançar o objetivo de cada etapa. Por causa dessa característica é possível avaliar uma ferramenta, método ou linguagem por proporcionar (ou não proporcionar) meios de desempenhar cada uma das atividades por meio das entradas de cada etapa e por gerar (ou não gerar) as saídas esperadas da etapa.

Nas próximas seções serão descritas cada uma das cinco etapas do processo, acompanhado de suas respectivas entradas e saídas, além das atividades a serem desenvolvidas na etapa. Essas etapas são utilizadas no Capítulo 4 para desenvolver a SPL Ágil com o ecossistema Ruby.

3.2 Etapa A – Levantamento de novos requisitos

A primeira etapa do processo é o levantamento de novos requisitos por meio de histórias de usuários (Figura 3.2). Esse levantamento não se distingue da forma que tradicionalmente é feito em uma abordagem ágil [Ghanam 2012]. Para essa etapa é muito importante a participação do usuário, dando a possibilidade do mesmo descrever os comportamentos desejados.

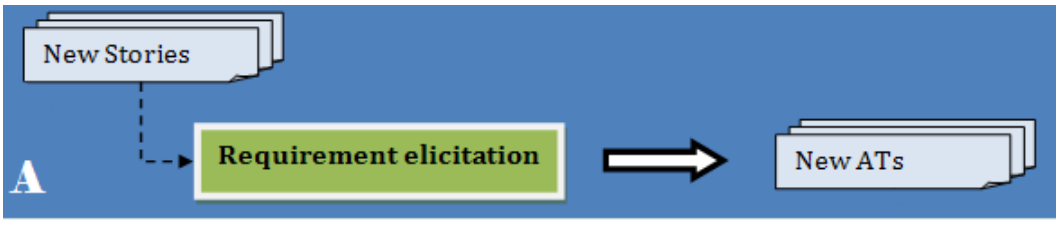

Figura 3.2: Primeira etapa do processo de SPL Ágil, adaptado de [Ghanam 2012]

Em cada nova iteração do processo os novos requisitos são coletados sobre a forma de histórias de usuário [Kerievsky 2010] então traduzidos, preferencialmente com auxílio do usuário para testes de aceitação [Reppert 2004].

Quadro 3.1 Modelo de História de Usuário, onde "| |" significa opcional

Como um <papel>, eu quero <meta/desejo> | de modo que
<benefício> |

Todos os testes coletados na iteração acompanhado dos testes de produtos anteriores são adicionados em uma base e servirão de entrada para a etapa de levantamento de variabilidades.

3.2.1 Atividades da etapa e artefatos de entrada e saída

A primeira etapa do processo possui apenas um artefato de entrada, que são as histórias de usuário. Histórias de usuários é apenas uma das diversas formas de se capturar requisitos dos clientes e usuários. Esse método de extração de requisitos é cada vez mais comum por ser adotado pelas metodologias ágeis de desenvolvimento [Kerievsky 2010].

Não existe um único modelo para a descrição de uma história de usuário, podendo variar de acordo com as necessidades de cada projeto. Uma forma padrão de criar histórias de usuário que se tornou referência foi apresentada por Cohn [Cohn 2004] e pode ser visto no Quadro 3.1.

Essas histórias são transformadas em testes de aceitação por meio de um processo de extração de requisitos. É válido atentar que não necessariamente uma história de usuário tornar-se-á apenas um teste de aceitação, dependendo da complexidade e da quantidade de artefatos envolvidos uma história pode gerar diversos testes.

A saída da etapa é o conjunto de testes, novos e de produtos anteriores. Esses testes além de descrever os critérios de aceitação e os comportamentos das *features* do projeto, também servem de documentação, eles são executáveis e sua obsolescência só ocorre as *features* correspondentes deixem de existir no projeto.

3.3 Etapa B – Elicitação de Variabilidade

Para iniciar essa etapa, presume-se que a primeira etapa do processo já esteja concluída e os novos testes de aceitação gerados para que, em conjunto com os testes de aceitação de iterações anteriores, possam ser analisados. Além disso, qualquer fonte de variabilidade, variantes e restrições necessárias sejam adicionadas ao perfil de variabilidade. Este perfil variabilidade

pode ser modelado, por exemplo, utilizando uma árvore de *features*.

O objetivo dessa etapa é descobrir, por meio dos testes de aceitação, as variabilidades na linha de produto (Figura 3.3). O início de uma etapa de gestão de variabilidade é encontrar os pontos de variação e variantes dentre os requisitos disponíveis. Em uma SPL tradicional essa etapa é realizada através de documentação intensa de forma proativa durante a fase de engenharia de domínio. Isso pode ser impraticável ou mesmo contrário quando se tratar de metodologias ágeis de desenvolvimento. Essa documentação intensa é substituída por testes de aceitação e diferentemente de uma abordagem mais clássica, é utilizado uma abordagem reativa.

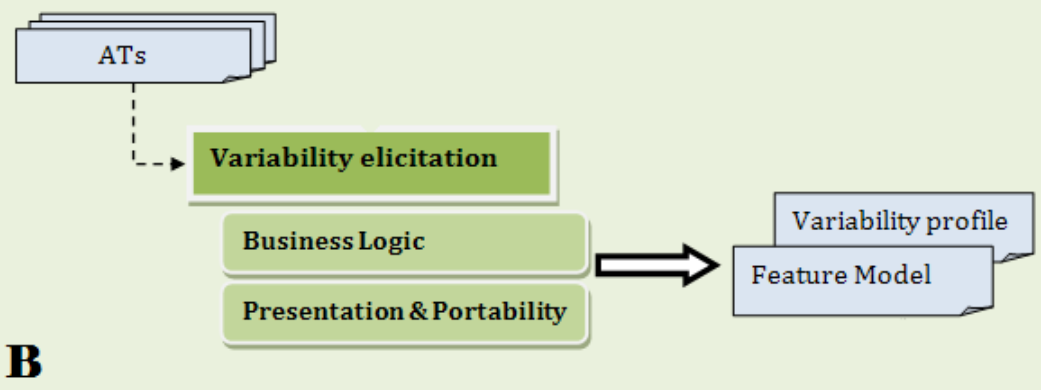

Figura 3.3: Segunda etapa do processo de SPL Ágil, adaptado de [Ghanam 2012]

As variabilidades procuradas nessa etapa podem ser divididas em variabilidades de regras de negócio, e variabilidade de apresentação ou portabilidade. As variabilidades em regras de negócio são descritas em um perfil de variabilidade e também passam a compor o diagrama de *feature*. Futuramente as variabilidades de regras de negócio são transformadas em testes de aceitação executáveis.

Variabilidades de apresentação e portabilidade geralmente remetem aos requisitos não funcionais, e é difícil de descrevê-los em forma de teste de aceitação [Melnik, Read e Maurer 2004]. Caso exista essa dificuldade em algum dos pontos de variabilidade é necessário adicioná-lo ao perfil de variabilidades acompanhado de um documento de problema implicação, como pode ser visto no quadro 3.2.

Com essa etapa concluída, espera-se ter em mãos artefatos que facilitem as modelagens, implementações e derivações de produtos, além de facilitar futuras adições de variabilidade. Esses artefatos devem evoluir em cada iteração dando a real noção de como está a SPL.

Quadro 3.2 Modelo de Problema e Implicação, traduzido de [Ghanam 2012]

História de usuário: A adição pretendida ou alteração do sistema existente, tal como descrito pelo cliente. O cliente pode ser interno ou externo à organização.

Problema: Uma razão explicando por que o sistema atual não pode satisfazer a história de usuário. Uma única história de usuário geralmente resulta em um conjunto de questões.

Implicação: Uma ação que precisa ser tomada, a fim de resolver os problemas resultantes de uma determinada história de usuário.

3.3.1 Atividades da etapa e artefatos de entrada e saída

A entrada da segunda etapa são todos os testes de aceitação existentes na linha de produto. Esses documentos possuem critérios de validação para os requisitos de cada uma das *features*.

Os testes de aceitação possuem papel duplo nessa etapa. O primeiro papel é o de documentação, já que os mesmos devem conter descrições e exemplos reais da funcionalidade de cada *feature*. O segundo papel é o de validar cada uma das *features*, dando segurança de integridade das funcionalidades e fidelidade aos requisitos originais ao desenvolvedor que for criar e implementar os pontos de variação do projeto.

A atividade desta etapa, elicitação de variabilidade, utiliza-se dos testes de aceitação para encontrar artefatos em comum e artefatos variantes, e descrevê-los em um documento de perfil de variabilidade e no diagrama de *feature*. Yaser [Ghanam 2012] em seu processo, defende que provocar as variabilidades e consequentemente evoluir o perfil de variabilidade é feito em seis passos, como segue abaixo:

1. O primeiro sistema é construído num processo de ASD normal, para satisfazer as exigências do cliente em questão, sem especular futuras variações.

2. Um modelo de *feature* inicial do sistema é produzido usando teste de aceitação. O modelo de *feature* inicial é uma simples decomposição de uma determinada *feature* para os diferentes cenários. Isso não significa necessariamente que o modelo deve contemplar as restrições.

3. Após a demanda de um sistema semelhante por um novo cliente, o modelo de *feature*

existente é disponibilizado para o novo cliente, acompanhado dos testes de aceitação.

4. O novo cliente escolhe os testes de aceitação que atendam suas necessidades. O conjunto de teste de aceitação escolhido representa uma instância de *feature*.

5. Se os testes de aceitação atualmente disponíveis não satisfazem as necessidades do cliente, o cliente deve definir um conjunto de alterações que podem incluir, substituir ou remover um conjunto de testes de aceitação.

6. Com base no conjunto de alterações produzidas em 5, o modelo de *feature* é atualizado.

O documento de perfil de variabilidade são os próprios testes de aceitação e podem acompanhar algum documento auxiliar definido pela equipe. Já para o modelo de *feature*, pode-se utilizar o clássico diagrama FODA (*Feature-Oriented Domain Analysis*) [Ghanam 2012]. Ambos os documentos são a saída da etapa, e servirão de apoio para a etapa posterior do processo.

3.4 Etapa C – Modelagem das variabilidades

Para um contexto de SPL é importante que todas as variabilidades existentes na sua linha de produto sejam rastreadas. Essa etapa do processo possui esse objetivo, rastreando cada *feature* em um modelo de *feature* executável, como pode ser visto na Figura 3.4.

Figura 3.4: Terceira etapa do processo de SPL Ágil, adaptado de [Ghanam 2012]

Depois de ter construído o perfil de variabilidade em etapas anteriores, é necessário tomar medidas adicionais para tornar a variabilidade visível para as diferentes partes do sistema.

A árvore de *feature* nativamente não fornece um meio de rastrear os requisitos, relacionando os códigos e sua respectiva *feature* no diagrama. Em uma SPL tradicional, esse problema é resolvido usando camadas de códigos intermediários, contendo as exigências de design e de artefato, assegurando assim, a coerência entre a implementação e o modelo.

Essa rastreabilidade entre implementação e os requisitos trazem diversas vantagens, como por exemplo o aumento da compreensão do *software*, validação da integridade da implementação, análise do impacto de inserção de um novo trecho de código e reutilização [Antoniol et al. 2002].

Yaser [Ghanam 2012], no seu processo mostra que essa camada pode ser substituída por testes de aceitação executáveis. Uma das vantagens da utilização de testes de aceitação executáveis é que com a evolução da SPL é evitado que o relacionamento entre modelo e artefatos de códigos sejam quebrados ou mesmo se tornarem obsoletos.

Ainda segundo Yaser [Ghanam 2012], as duas características mais interessantes de testes de aceitação executáveis para o modelo de *feature* executável são:

Serem provenientes de histórias de usuários e possuírem linguagem legível para os *stakeholders*, servindo assim de documentação para as especificações de uma *feature*. E serem executáveis e podendo assim testar com exatidão o comportamento das *feature* e as validarem.

Segundo Yaser [Ghanam 2012], a vinculação de testes de aceitação executáveis em uma *feature* trás as seguintes consequências:

1. Selecionar uma *feature* na fase de derivação de produtos, implica a sua inclusão acompanhado de todos os seus testes de aceitação.

2. Testes de aceitação executáveis herdarão todas as dependências e restrições inicialmente instituídas em seu nó pai.

3.4.1 Atividades da etapa e artefatos de entrada e saída

A entrada para essa etapa são os testes de aceitação (perfil de variabilidade) e o modelo de *feature*. Durante essa etapa é realizada a atividade de modelagem da variabilidade, isto é, tornar os testes de aceitação da etapa anterior executáveis e automatizados.

Além disso são criados testes de aceitação para as folhas da árvore de *feature* possibilitando adicionar restrições. Com essas adições é possível verificar a compatibilidade ou incompatibilidade de duas ou mais *features* quando selecionadas no mesmo conjunto. A saída dessa etapa são os testes de aceitação executáveis vinculados com suas respectivas *features* da árvore.

3.5 Etapa D – Realização da variabilidade

Esta etapa utiliza uma abordagem reativa e sistemática para introduzir variabilidade. Esta abordagem é de natureza *bottom-up* e orientada aos testes. As variabilidades são inseridas na linha de produto por meio de refatoração, como visto na Figura 3.5.

Além disso, para cada produto é criado uma série de testes de unidade a fim de que, de forma automática, seja possível validar e verificar a consistência de cada um.

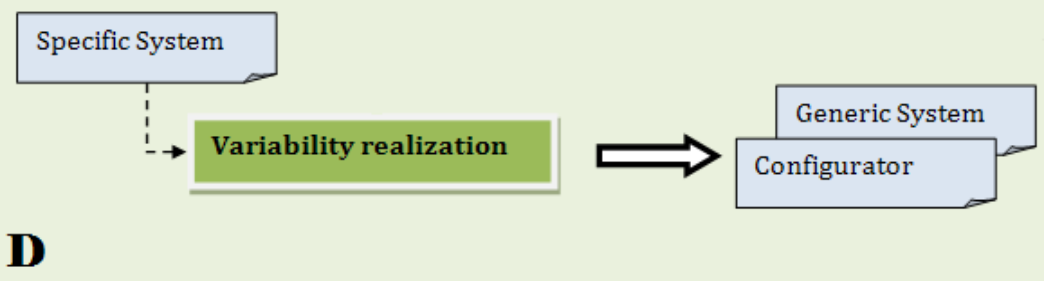

Figura 3.5: Quarta etapa do processo de SPL Ágil, adaptado de [Ghanam 2012]

3.5.1 Atividades da etapa e artefatos de entrada e saída

A entrada desta etapa são os testes de aceitação executáveis da entrada anterior com a especificação de cada um dos produtos. Para cada um dos produtos é criado uma bateria de testes de unidade, com o objetivo de validar e verificar a consistência da implementação.

O resultado desta etapa é a especificação dos produtos em forma de testes de unidade e o sistema implementado de forma com que todos os testes, de aceitação e unidade, desta e das etapas anteriores sejam satisfeitas.

Essa etapa possibilita que para toda e qualquer nova inserção ou remoção de funcionalidade da linha de produto, seja realizada um bateria de testes para verificar se os produtos continuam com o comportamento anterior, modelado nas etapas anteriores.

3.6 Etapa E – Derivação dos produtos

A derivação de diversos produtos utilizando apenas uma base de código é fundamental para a SPL. O objetivo é tornar essa etapa o mais automática possível, conforme a Figura 3.6.

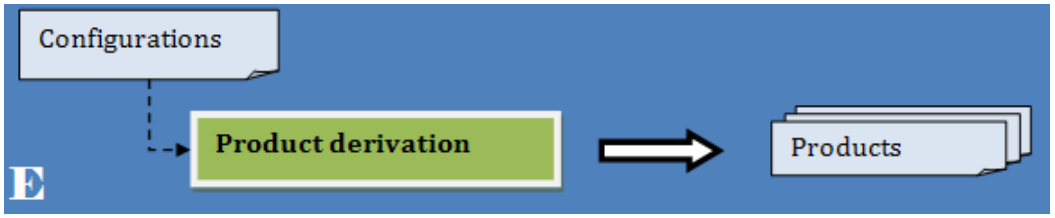

Figura 3.6: Última etapa do processo de SPL Ágil, adaptado de [Ghanam 2012]

Essa etapa tem como objetivo derivar (ou instanciar) diversos produtos diferentes, com diversas customizações e necessidades de diversos clientes, partindo de uma única base de código, esse é um aspecto essencial da SPL. A automatização do processo de derivação é fundamental para sua eficiência, ainda mais em um contexto de customização em massa.

Segundo Yaser, no contexto de SPL, o processo de derivação utiliza, pelo menos, os seguintes componentes:

1. Um sistema de base genérica **S**, que inclui todos os componentes reutilizáveis e que podem ser utilizados para a criação de diferentes produtos.

2. Um conjunto de configurações **C** de personalizações desejadas.

3. Um configurador **E** que com **S** e **C**, permitem ligar as variações e variantes de acordo com o perfil de variabilidade.

3.6.1 Atividades da etapa e artefatos de entrada e saída

Os artefatos de entrada desta etapa são as configurações (*features* desejadas pelos clientes), toda a camada desenvolvida na etapa anterior, código e testes de unidade e o modelo de *feature* para facilitar a visualização.

O processo de derivação, segundo Yaser [Ghanam 2012], acontece da seguinte forma: Os clientes selecionam as funcionalidades em formato de testes de aceitação. Esses testes de aceitação são executados e é verificado o relatório dos testes. Com base na informação recolhida dos testes, as partes relevantes são extraídos do núcleo e então, por último, esse novo sistema é compilado e construído a fim de ser entregue aos clientes.

O resultado final desta etapa são os produtos gerados e entregues, para cada cliente, apoiados por diversas camadas de testes, validando assim, sua consistência e seus critérios de aceitação.

Capítulo 4

Estudo Exploratório Descritivo utilizando o Ecossistema Ruby

Neste capítulo é apresentado o protocolo de pesquisa e a execução do estudo exploratório descritivo. Aqui são definidos as questões de pesquisa, resultados esperados, desenvolvimento do projeto, avaliação e a conclusão.

Essa estrutura auxilia na compreensão do objetivo de pesquisa a fim de guiar as ações de acordo com os dados obtidos.

4.1 Protocolo

A definição deste protocolo foi realizada com base na estrutura definida por Claes Wohlin e Aybüke Aurum [Wohlin e Aurum 2014]. Essa estrutura pode ser observada na Figura 4.1.

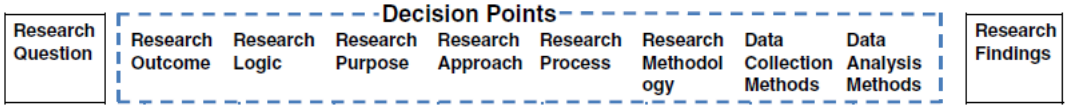

Figura 4.1: Estrutura de um projeto de pesquisa [Wohlin e Aurum 2014]

Essa Figura 4.1 apresenta os principais pontos de uma pesquisa, são eles, Tipo da Pesquisa, Lógica da Pesquisa, Propósito da Pesquisa, Abordagem da Pesquisa, Metodologia da Pesquisa, Método de Coleta de Dados e Método de Análise de Dados.

O objetivo desta pesquisa é verificar a possibilidade de desenvolver SPL Ágil no ecossistema Ruby, caraterizando o Tipo da Pesquisa como aplicada [Wohlin e Aurum 2014]. Em uma pesquisa aplicada usa-se pesquisas básicas, no caso estudos sobre SPL Ágil, para resolver um problema específico. Para verificar a possibilidade de desenvolver SPL Ágil no ecossistema

Ruby são criadas hipóteses e teorias, caracterizando assim a Lógica desta Pesquisa como indutiva [Wohlin e Aurum 2014].

Esta pesquisa tem como propósito realizar um estudo Exploratório [Wohlin e Aurum 2014], isso se deve ao fato das pesquisas unindo Ecossistema Ruby e SPL Ágil serem escassos. Outro Propósito da pesquisa é ser Descritiva [Wohlin e Aurum 2014], servindo assim como apoio para futuras pesquisas e dando a liberdade para o autor de focar apenas no resolver e não no buscar o melhor caminho [Wohlin e Aurum 2014].

Segundo Gil [Gil 2010], uma pesquisa exploratória desenvolve, esclarece e modifica os conceitos e ideias com o objetivo de identificar problemas mais precisos e hipóteses que possam ser utilizados em trabalhos posteriores. O processo de pesquisa exploratória é a etapa de partida para as demais investigações e geralmente parte de um tema bem genérico. O resultado final de uma pesquisa exploratória é o esclarecimento dos problemas possibilitando outras investigações na sequência.

A Abordagem utilizada nesta pesquisa é interpretativista onde o próprio pesquisador é o participante. Esse tipo de abordagem pode levar a subjetividade mas é um passo necessário para guiar o desenvolvimento de novas pesquisas no futuro [Wohlin e Aurum 2014].

O processo de pesquisa utilizado é o Qualitativo [Wohlin e Aurum 2014], pois essa pesquisa não mede eficácia ou compara o ecossistema `Ruby` com outros ecossistemas, linguagens e *frameworks* . Em cada uma das etapas do processo de SPL Ágil é realizada uma série de avaliações, Veja a seção 4.2.4. Essas avaliações são feitas utilizando três marcadores, satisfeito, não satisfeito e parcialmente satisfeito em uma tabela, Veja Tabela 4.1.

A Metodologia de Pesquisa utilizada é *Design Science*. O objetivo do *Design Science* é desenvolver artefatos que permitam uma solução satisfatória de problemas práticos [Lacerda et al. 2013]. Ele possui como objetivo desenvolver conhecimento e artefatos [Aken 2004]. O `Design Science` pode ser dividido em Identificação do Problema, Definição dos Resultados Esperados, Desenvolvimento do Projeto, Avaliação e Conclusão.

Além do *Design Science* ser a Metodologia de Pesquisa, ele é utilizado na etapa de coleta e avaliação de dados, pois ele contempla ambas as tarefas. A tarefa de coletas de dados é realizada na etapa de Desenvolvimento do Projeto, já a tarefa de Avaliação dos Dados é realizada na etapa de Avaliação do *Design Science*.

Nas Seções abaixas é apresentado o Desenvolvimento guiado pelo *Design Science* e ao final, a conclusão do estudo.

4.2 *Design Science*

Nesta Seção é apresentada cada etapa prevista no *Design Science*. A primeira etapa é a Identificação do problema, responsável por auxiliar o autor da pesquisa no entendimento do problema [Lacerda et al. 2013]. A segunda etapa é a Definição dos Resultados Esperados, na qual o autor elabora possíveis hipóteses [Lacerda et al. 2013].

A terceira etapa é o Desenvolvimento do Projeto. Essa é a etapa em que são realizadas as coletas de dados e as tentativas de levantar as hipóteses apontadas [Lacerda et al. 2013]. A última etapa, Avaliação, é responsável por analisar os dados coletados e validar ou não as hipóteses, bem como, responder as perguntas iniciais [Lacerda et al. 2013].

Outras etapas podem ser adicionadas ao *Design Science* mas essas foram suficientes para o projeto.

4.2.1 Identificação do Problema

É muito comum pesquisas em SPL para linguagens como `Java` e `C#` [Ghanam 2012, Mendonca, Branco e Cowan 2009, Kästner et al. 2009] e também para os seus respectivos *frameworks*, porém, não é tão comum essa pesquisa para linguagens consideradas ágeis, como é o caso do `Ruby` e `Python` [Python]. Com isso em mente podemos chegar ao questionamento:

Q: É possível com o ecossistema `Ruby` realizar o processo de SPL ágil descrito por Yaser (Capítulo 3)?

O estudo exploratório utiliza como objeto o ecossistema `Ruby` descrito no capítulo 2.3 e o método proposto por Yaser descrito no capítulo 3. A fim de analisar e validar o estudo é implementado uma SPL Ágil no domínio de *Blogs*. Um *Blog* (contração do termo inglês *web log*) é um site que permite a inserção de artigos, também chamados de *post*. Dependendo do *blog*, os artigos podem ser editados por várias pessoas. Ainda dependendo do *blog*, pessoas podem comentar os artigos gerando interação entre escritor e leitor.

A Figura 4.2 representa o domínio simplificado de *blog* em um contexto de SPL. Nela é possível ver as relações entre cada uma das *features* com suas restrições e inclusões. O método de SPL Ágil utilizado cria os produtos de forma reativa, portanto, essa árvore é gerada incrementalmente.

Questões de Pesquisa

Cada etapas do processo de SPL Ágil utilizado possui artefatos de saída, de entrada e atividades. Por exemplo, na primeira etapa os artefatos de entrada são as histórias de usuário, que por meio de uma atividade de elicitação de requisitos são gerados como saída ATs referentes às histórias de usuário.

Como cada uma das etapas possui tais atributos, é possível desmembrar os questionamentos anteriores em 6 questionamentos, e esses 6 questionamentos, em algumas questões mais específicos como segue abaixo:

1. É possível criar um "primeiro produto" no ecossistema `Ruby` da forma descrita por Yaser (Seção 3.1)?

2. Há ferramentas, funcionalidades ou técnicas do ecossistema `Ruby` que apoiam a Elicitação de Requisitos?

   (a) Histórias de Usuário podem servir de parâmetro para o desenvolvimento em `Ruby`?

   (b) Há ferramentas, funcionalidades ou técnicas que permitam a construção de testes de aceitação para `Ruby`?

3. Há ferramentas, funcionalidades ou técnicas do ecossistema `Ruby` que apoiam a Elicitação de Variabilidade?

   (a) Há ferramentas, funcionalidades ou técnicas do ecossistema `Ruby` que permitam definir o Perfil de Variabilidade de uma SPL Ágil?

   (b) Há ferramentas, funcionalidades ou técnicas do ecossistema `Ruby` que permitam gerar o Modelo de *Feature* de uma SPL Ágil?

4. Há ferramentas, funcionalidades ou técnicas do ecossistema `Ruby` que apoiam a Modelagem de Variabilidade?

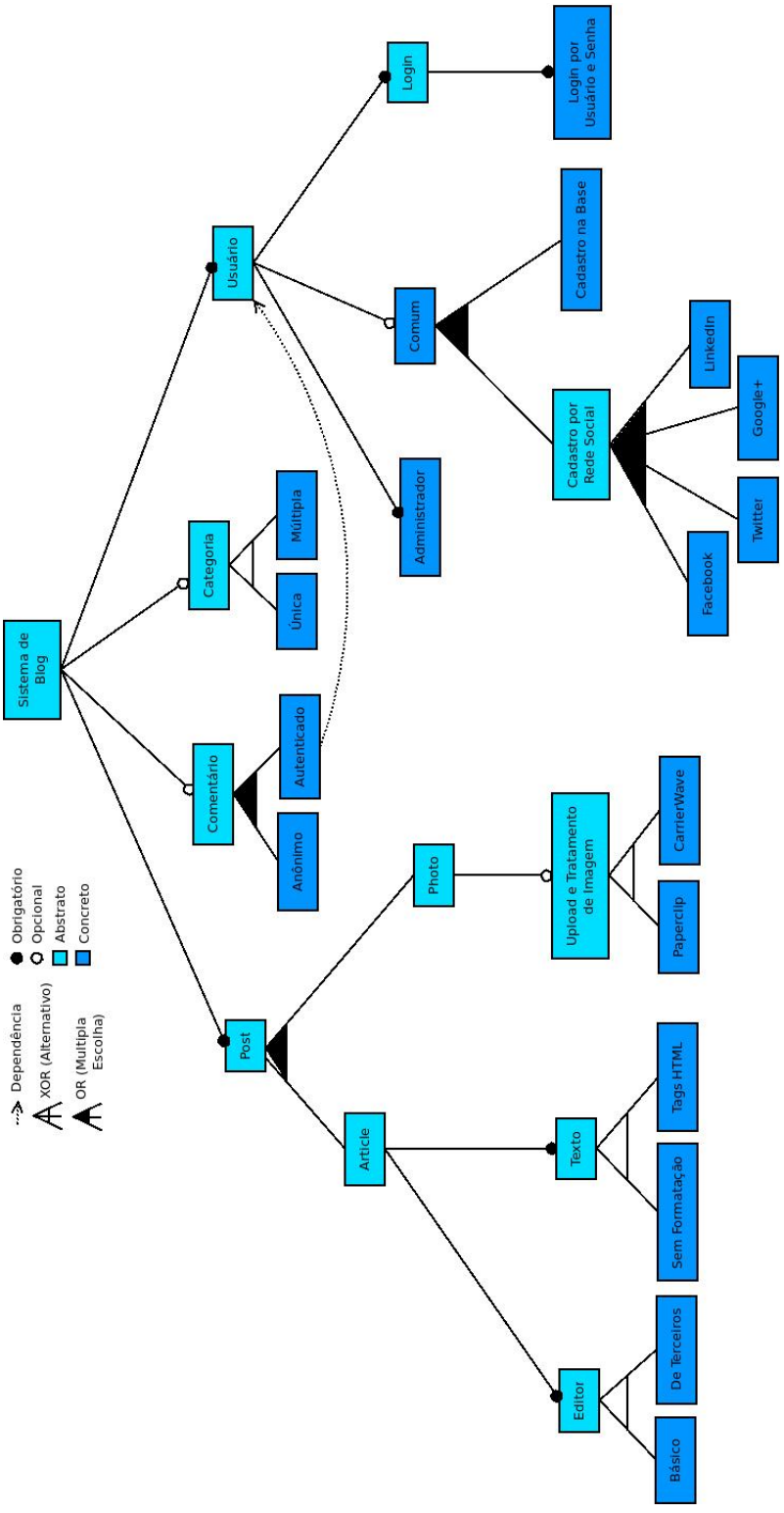

Figura 4.2: Árvore de *feature* para o domínio de *Blogs*

(a) Há ferramentas, funcionalidades ou técnicas do ecossistema `Ruby` que permitam gerar o Modelo de *Feature* executável de uma SPL Ágil?

5. Há ferramentas, funcionalidades ou técnicas do ecossistema `Ruby` que apoiam o processo de Realização de Variabilidade que transforma um conjunto de funcionalidades de um sistema específico em um conjunto genérico de funcionalidades?

   (a) Há ferramentas, funcionalidades ou técnicas do ecossistema `Ruby` que permitam o gerenciamento de um conjunto de *Feature*?

6. Há ferramentas, funcionalidades ou técnicas do ecossistema `Ruby` que apoiam derivação de um produto?

   (a) Há ferramentas, funcionalidades ou técnicas do ecossistema `Ruby` que permitam a seleção de um conjunto de *Feature* gerenciada?

   (b) Existe forma de representar um produto no ecossistema `Ruby`?

4.2.2 Definição dos resultados esperados

O processo de SPL Ágil adotado (ver capítulo 3) consiste em iterações de TDD documentadas por meio de testes de aceitação, utilizando algumas técnicas oriundas da SPL, como por exemplo a derivação. O ecossistema `Ruby` possui diversas ferramentas voltadas ao TDD, as quais pode atender o processo.

No ecossistema `Ruby` é possível criar pacotes (GEMs). Nesses pacotes é possível embarcar uma aplicação inteira por meio de *plugins*, é esperado poder adicionar cada *feature* em uma GEM mantendo-as isoladas uma das outras.

Por fim, o ecossistema `Ruby` também possui ferramentas de gerenciamento automático de GEMs, a mais conhecida é o *Bundler*. O *Bundler* pela característica de gerenciar pacotes deve auxiliar na etapa de derivação de produto.

4.2.3 Desenvolvimento do Projeto

Nesta seção são apresentados os artefatos desenvolvidos. Para responder as perguntas anteriores foram desenvolvidos produtos no domínio de um Blog. Cada um dos produtos está separado nas subseções seguintes.

Primeiro Produto

Conforme visto na seção 3.1 no primeiro produto não é aplicado o processo descrito durante o capítulo 3, e sim, um processo comum de desenvolvimento ágil. Esse processo comum de desenvolvimento ágil variará de acordo com a empresa ou equipe de desenvolvimento, em geral, a única necessidade é o uso de um processo BDD, evitando assim retrabalho para as próximas etapas.

O primeiro passo para a criação do produto em um processo BDD é obtenção dos requisitos ou histórias de usuário. Essas histórias de usuário são mapeadas para testes de aceitação [Reppert 2004] seguindo o fluxo de um desenvolvimento ágil guiado por BDD.

É comum que as histórias de usuários sejam descritas em cartões ou mesmo simples documentos de texto, o que não é impedido em se tratando do ecossistema `Ruby`, porém, existe a ferramenta de testes `Cucumber` que permite a descrição das histórias de usuário já como descrição de testes de aceitação. Um exemplo de história de usuário já em formato `Cucumber` pode ser visto no quadro 4.1.

Para o desenvolvimento do primeiro produto em `Ruby`, podemos seguir por dois caminhos distintos. O primeiro caminho segue o desenvolvimento tradicional de um programador `Ruby on Rails`, utilizando-se da arquitetura do *framework* bem como de todas as suas ferramentas. Essa primeira abordagem não se preocupa com o desenvolvimento para o reúso e sim em agilidade no desenvolvimento.

O segundo caminho, o desenvolvedor deixa a arquitetura tradicional do `Ruby on Rails` e passa a adotar uma arquitetura própria que melhor se adapta para cada *feature* do projeto. Nessa abordagem as *features* do projeto são empacotadas em `RubyGEMs` para a distribuição e desenvolvimento para reúso. Deve-se criar uma forma de integrar essas *features* com os projetos `Ruby on Rails`, uma forma de integrar será apresentado nas próximas seções.

O primeiro produto do estudo de caso será desenvolvido utilizando ambas as abordagens demonstrando assim as diferenças entre ambas alternativas. Ambos os produtos serão baseados apenas na História de Usuário da tabela 4.1, referente a um domínio de blogs. Como no primeiro produto não é aplicado de fato o processo de SPL Ágil, ele será feito da forma mais simples possível, apenas com intuito de demonstração.

O próximo passo do processo de desenvolvimento seguido neste trabalho, desenvolvimento

Quadro 4.1 História de Usuário para o primeiro produto

Eu como usuário desejo poder inserir, editar, visualizar e
excluir artigos contendo um título e um texto.

Quadro 4.2 Descrição da criação de artigos em `Cucumber`

Funcionalidade: Criar um artigo

Cenário Deve preencher todos os campos requiridos e salvar o artigo com sucesso
Dado que eu estou na página de criação de artigo
Quando eu preencher todos os campos
E clicar em "Create Article"
Então deve-se receber a mensagem "Article was successfuly created."

BDD, é descrever o comportamento de nossa aplicação. Para a descrição do comportamento será utilizado a `GEM Cucumber`, com ela é possível descrever as histórias de usuários e os comportamentos das *features*. O processo de descrever a história de usuário, bem como, os seus comportamentos, são realizados de forma manual e preferencialmente com o apoio do usuário. A história de usuário e os comportamentos do primeiro produto pode ser visto no quadro 4.2.

Desenvolvimento tradicional em `Ruby on Rails`

O primeiro passo para o desenvolvimento desse sistema é a criação do projeto. Tradicionalmente a criação de projetos em `Ruby on Rails` é feita utilizando o comando `rails new nomedoprojeto`. Para o exemplo, `rails new blog`. Como resultado criamos um diretório contendo a estrutura básica do projeto.

Para criarmos a *feature* artigo, com o adicionar, remover, editar e visualizar pode-se utilizar um gerador de CRUD[10] do *framework* chamado `scaffold`, dessa forma temos, `rails generate scaffold article title:string body:text`.

Esse comando além de gerar o modelo, controlador e a visão da *feature* artigo ele gera um arquivo para a manipulação do banco de dados chamado de `migrations`. Para executar esses arquivos existe o comando `rake db:migrate`.

Com isso já temos nosso projeto pronto atendendo aos critérios do teste de aceitação, bas-

[10] Acrônimo para as quatro operações básicas utilizadas em banco de dados relacional, **C**reate, **R**ead, **U**pdate e **D**elete

tando o comando `rails server` para iniciarmos o servidor já rodando o nosso projeto, como pode ser visto na Figura 4.3.

Figura 4.3: Cenário de inserção de uma nova postagem com sucesso

Desenvolvimento utilizando uma arquitetura separada

Assim como no desenvolvimento tradicional, o primeiro passo é criar o projeto. Como o objetivo é criar as *features* de forma isolada do projeto principal no *framework* `Ruby on Rails`, elas serão criadas em formato de `RubyGEM`. Para esse exemplo será criado uma `Rails Engine` com o comando `rails plugin new article –mountable`. Com isso criamos um diretório que conterá todos os códigos para a *feature* de criação de artigos do blog.

O segundo passo é adicionarmos o comando `isolate_namespace Article` para podermos estender os modelos, visões e controladores desta `GEM` para o nosso projeto. Feito isso, é possível seguir praticamente os mesmos passos de um desenvolvimento tradicional. Utilizaremos o comando `rails generate scaffold article title:string body:text` para criação do modelo controlador e as visões de artigo.

Com nossa `GEM` finalizada, podemos criar nosso projeto exatamente igual ao desenvolvimento tradicional, `rails new blog`. A diferença é que em vez de criarmos a *feature* artigo chamaremos a `RubyGEM` *article* criada. No projeto criado existe um arquivo chamado `Gemfile`, nesse arquivo é onde ficam adicionadas as chamadas para as `RubyGEMs` necessárias no projeto. Essa chamada variará de acordo com o local onde a `RubyGEM` se encontra. Como nesse projeto se encontra em um diretório local o comando fica da seguinte forma `gem 'article', path: => '../article'`.

Após adicionar a chamada ao arquivo `Gemfile` deve-se executar o responsável por gerenciar as `RubyGEMs` do projeto, no caso o `Bundler`, com o comando `bundle install`.

Quadro 4.3 História de Usuário para o segundo produto

Eu como usuário desejo poder inserir, editar, visualizar e
excluir artigos contendo um título, texto e um resumo.

Quadro 4.4 Descrição da criação de artigos com resumo em `Cucumber`

Funcionalidade: Criar um artigo com resumo

Cenário Deve preencher todos os campos requiridos e salvar o artigo com sucesso
Dado que eu estou na página de criação de artigo
Quando eu preencher todos os campos
E clicar em "Create Article"
Então deve-se receber a mensagem "Article was successfuly created."

Automaticamente ele instala a nossa *feature* no projeto e com ela uma tarefa de importação das manipulações de banco de dados que a *feature* exige. Para executarmos a tarefa de importação é necessário executar o comando `rake article:install:migrations`, com isso podemos utilizar o comando `rake db:migrate` para fazermos as manipulações necessárias no banco de dados.

A última coisa que precisamos fazer é adicionar as rotas, para isso adicionaremos o comando `mount Article::Engine => "/"` no arquivo de configuração de rotas do nosso projeto. Agora temos nosso primeiro produto finalizado bastando executar o comando `rails server` para inicializarmos o nosso servidor.

Segundo Produto

Diferente do primeiro produto, no segundo produto já é aplicado o processo de SPL ágil. O primeiro passo, conforme descrito no capítulo 3 é o levantamento de requisitos, recolher as histórias de usuários e transformar em testes de aceitação. A história de usuário para o segundo produto pode ser vista no Quadro 4.3 e o respectivo teste de aceitação pode ser visto no Quadro 4.4.

A segunda etapa do processo é a elicitação de variabilidade. Nessa etapa são verificados os pontos comuns e variantes bem como os efeitos das novas histórias de usuário na base já existente. Neste exemplo, temos a *Feature* `Artigo`, com a chegada da nova história de usuário ela ganhou um novo artefato, o resumo, porém, essa mudança não pode afetar na base anterior

Quadro 4.5 Descrição da sentença `Então` deve-se receber a mensagem "Article was sucessfuly created."do Criar Artigo.

```
it /Então deve-se receber a mensagem "Article was sucessfuly created."/ do
    Article :create, title: "Um Título", body: "Um texto bem longo.", resume: "Um texto"
    expect([flash:success]).to be_present
end
```

tornando-a uma variação opcional de `Artigo`, conforme pode ser visto na Figura 4.4.

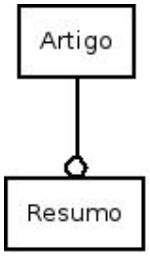

Figura 4.4: *Feature Model* do `Artigo`

A terceira etapa do processo é a automatização do perfil de variabilidade. O perfil de variabilidade são os próprios testes de aceitação. Nessa etapa é criado para cada uma das sentenças (Exemplo: `Então` deve-se receber a mensagem "Article was sucessfuly created.") testes automatizados. Um exemplo de teste utilizado no trabalho pode ser visto no Quadro 4.5.

Com todos os testes automatizados é possível iniciar a quarta etapa do processo. Essa etapa pode ser dividida em duas partes. A primeira é implementar as histórias de usuário utilizando-se da refatoração e também de testes de unidade quando necessário. A segunda etapa é a de criar mecanismos para que a *Feature* possa ser inserida em um produto da maneira mais automática e causando o menor impacto possível.

Primeiro passo é criar o `rails plugin` para o artigo com resumo. O comando utilizado para isso é `rails plugin new article_resume –moutable`. Será criado um novo diretório e dentro dele existe um arquivo denominado `Gemfile`. É nesse arquivo que adicionaremos a dependência com o primeiro projeto, geralmente `gem 'article'` mas isso depende de onde está armazenado o projeto anterior. Adicionado a dependência é necessário rodar o `Bundler` e executar o `rake article:install:migrations`.

O próximo passo é implementar as modificações. Como a modificação é uma simples adição de um campo resumo no projeto ela pode ser implementada adicionando uma modificação ao escopo do projeto anterior, para isso, é necessário rodar o comando `rails generate`

Quadro 4.6 Configuração do primeiro produto

```
gem "article"
route 'mount Article::Engine => "/"'
after_bundle do
    rake ("article:install:migrations")
    rake ("db:migrate")
end
```

Quadro 4.7 Configuração do segundo produto

```
gem "article_resume"
route 'mount ArticleResume::Engine => "/"'
after_bundle do
    rake ("article_resume:install:migrations")
    rake ("db:migrate")
end
```

`migration add_resume_to_articles resume:text` e executar o comando `rake db:migrate`, feito isso, basta apenas adicionar o campo no formulário `html` e adicionar o resume nos parâmetros do controlador.

Esse é apenas um exemplo, existem várias outras formas de criar essa variação. É possível criar ela na mesma GEM criada no primeiro produto, ou ainda utilizando `template`. Além do ferramental, poderia ainda se utilizar de técnicas como reflexão, herança ou tantas outras possíveis em `Ruby` e viáveis para SPL. Fazendo dessa forma, o resultado desta etapa já é uma GEM contendo o segundo produto, bastando criar o arquivo de configuração.

Para criar esse arquivo de configuração, usaremos nesse exemplo o `rails template`. Ele nada mais é do que um arquivo `Ruby` com alguns comandos. A sua vantagem é que o `rails` é capaz de recebê-los de parâmetro na criação de um novo projeto. O `rails template` do primeiro produto pode ser visto no Quadro 4.6, já o do segundo produto pode ser visto no 4.7.

A última etapa do processo é a derivação, por utilizar `rails template` e tratar cada *feature* como uma GEM essa etapa é facilitada. Para derivarmos o primeiro produto apenas precisamos criar um projeto `rails` passando como parâmetro o `template`. Isso é feito com o comando `rails new nome_da_aplicacao -m template_do_primeiro_produto.rb"` para o segundo produto mesma coisa, apenas alterar para a segunda configuração. Uma das maiores vantagens deste método é que além da

facilitação na hora de derivar essa derivação pode ser feita utilizando um *script* em um servidor.

4.2.4 Avaliação

Organizamos a avaliação por meio das respostas de cada uma das perguntas anteriores com base no desenvolvimento feito na seção anterior. Todas as perguntas possui o intuito de que em conjunto respondam a pergunta principal.

1. **É possível criar um "primeiro produto" no ecossistema `Ruby` da forma descrita por Yaser (Seção 3.1)?**

   R: Sim, como visto na seção 4.2.3 é possível criar o primeiro produto utilizando o ecossistema `Ruby`, ainda é possível criar de diversas formas dependendo das necessidades do desenvolvedor.

2. **Há ferramentas, funcionalidades ou técnicas do ecossistema `Ruby` que apoiam a Elicitação de Requisitos?**

   R: Sim, o padrão de desenvolvimento do `Ruby` é utilizando o BDD, e o BDD como base utiliza as histórias de usuário para o levantamento de requisito. Tudo isso pode se confirmar tanto no desenvolvimento de ambos os produtos.

   (a) **Histórias de Usuário podem servir de parâmetro para o desenvolvimento em `Ruby`?**

   R: Sim, histórias de usuário são entradas padrões do BDD e o BDD é largamente utilizado pela comunidade `Ruby`, além de termos utilizado durante o desenvolvimento de todo o projeto.

   (b) **Há ferramentas, funcionalidades ou técnicas que permitam a construção de testes de aceitação para `Ruby`?**

   R: Sim, o ecossistema `Ruby` conta com diversas ferramentas para testes de aceitação, como por exemplo o `Cucumber` e o `RSpec` utilizados nesse trabalho.

3. **Há ferramentas, funcionalidades ou técnicas do ecossistema `Ruby` que apoiam a Elicitação de Variabilidade?**

R: Sim, os testes de aceitação são utilizados para buscar as distinções entre *features* e assim gerar as variabilidades.

(a) **Há ferramentas, funcionalidades ou técnicas do ecossistema `Ruby` que permitam definir o Perfil de Variabilidade de uma SPL Ágil?**

R: Sim, os próprios testes de aceitação são usados para isso.

(b) **Há ferramentas, funcionalidades ou técnicas do ecossistema `Ruby` que permitam gerar o Modelo de *Feature* de uma SPL Ágil?**

R: Não, não encontramos no ecossistema `Ruby` ferramentas que possam ser utilizadas para gerar e gerir automaticamente o gráfico do modelo de *feature*, porém, dificilmente alguma outra linguagem não criada especificamente para SPL terá uma ferramenta para isso. Durante o projeto foi utilizado um editor de diagramas, DIA [Dia], para realizar o modelo de *feature*.

4. **Há ferramentas, funcionalidades ou técnicas do ecossistema `Ruby` que apoiam a Modelagem de Variabilidade?**

R: Sim, encontramos várias ferramentas de comportamento e de testes no ecossistema `Ruby` que podem ser utilizadas para modelar a variabilidade.

(a) **Há ferramentas, funcionalidades ou técnicas do ecossistema `Ruby` que permitam gerar o Modelo de *Feature* executável de uma SPL Ágil?**

R: Sim, o `Cucumber` em conjunto com o `RSpec` foi utilizado para isso.

5. **Há ferramentas, funcionalidades ou técnicas do ecossistema `Ruby` que apoiam o processo de Realização de Variabilidade que transforma um conjunto de funcionalidades de um sistema específico em um conjunto genérico de funcionalidades?**

R: Sim, o `rails` possui uma ferramenta chamada `rails plugins` que pode ser utilizada para empacotar uma funcionalidade em uma GEM, além disso, o `Ruby` possui suporte a diversas técnicas utilizadas na SPL como, por exemplo, reflexão, herança e lambdas.

(a) **Há ferramentas, funcionalidades ou técnicas do ecossistema `Ruby` que permitam o gerenciamento de um conjunto de *Feature*?**

R: Sim, o ecossistema `Ruby` possui um gerenciador de pacotes chamado de `Bundler` e ele pode ser utilizados para gerenciá-las cada uma das *features* empacotadas em GEMs.

6. **Há ferramentas, funcionalidades ou técnicas do ecossistema `Ruby` que apoiam derivação de um produto?**

R: Sim, é possível derivar um produto por meio de arquivos templates. Existem outras forma de derivar produtos no ecossistema `Ruby` como, por exemplo, utilizando o `rake`, porém, provavelmente essa seja a mais simples.

(a) **Há ferramentas, funcionalidades ou técnicas do ecossistema `Ruby` que permitam a seleção de um conjunto de *Feature* gerenciada?**

R: Sim, é possível agrupar diversas GEMs em um arquivo Gemfile, permitindo o `Bundler` gerenciá-las, além disso, é possível criar arquivos de *template* para facilitar a seleção de GEMs.

(b) **Existe forma de representar um produto no ecossistema `Ruby`?**

R: Sim, um produto pode ser criado por meio do comando `rails new produto` e ainda é possível passar como parâmetro um template para a sua criação.

4.3 Conclusão

Com os questionamentos da Seção 4.2.4 devidamente respondidos, é possível responder ao questionamento principal:

Q: É possível com o ecossistema `Ruby` realizar o processo de SPL ágil descrito por Yaser (Capítulo 3)?

Sim, é possível criar uma SPL ágil utilizando o processo descrito por Yaser e utilizando as ferramentas descritas na etapa de desenvolvimento. Isso não quer dizer que seja a única forma ou a melhor e sim que existe uma forma. As ferramentas e um resumo de cada uma das etapas pode ser visto na Tabela 4.1

Tabela 4.1: Ferramentas utilizadas para cada uma das etapas do processo

Etapa	**Interna**	**Externa**	**Conclusão**
Elicitação de requisitos	Cucumber		Etapa satisfeita
Elicitação de Variabilidade	Cucumber	DIA	Etapa parcialmente satisfeita
Modelagem da Variabilidade	Cucumber e RSpec		Etapa satisfeita
Realização da Variabilidade	Template, rake, Bundler, GEM, Rails Plugin e Rails		Etapa Satisfeita
Derivação de produto	Bundler, Rails, rake e Rails plugin		Etapa Satisfeita

Um ponto de destaque foi a necessidade de modificar a arquitetura tradicional de desenvolvimento `Rails` colocando todas as *features* do projeto em `Ruby Gems` isso pode acarretar transtornos ou custos adicionais em projetos maiores com estágios de desenvolvimentos mais avançados.

Tradicionalmente um projeto `Rails` possui uma arquitetura MVC amarrada ao próprio projeto, então fazem parte do projeto todas os controladores, todos os modelos e todas as visões. Ao separar da forma como feita neste trabalho, é isolada o controlador, modelo e visão de cada *feature* em uma `Ruby Gem` e no projeto é incluso as `Ruby Gems` necessárias.

A maior vantagem detectada foi a facilidade de derivar produtos por conta do `Rails Template`. Com um simples comando e um simples arquivo de configuração é possível gerar todo um projeto de Software com as bases de projetos anteriores.

Alguns relatos da literatura [Deelstra, Sinnema e Bosch 2005] apontam que a fase mais complexa de uma SPL é a sua derivação, porém, utilizando o ecossistema `Ruby` notamos que foi relativamente simples. Claro que não foram realizadas derivações mais complexas e de produtos mais complexos pois não era o objetivo da pesquisa.

Capítulo 5

Conclusão

Nas seções abaixo apresentamos nossas conclusões sobre o trabalho, sobre o Ecossistema `Ruby` e sobre o método de desenvolvimento de SPL Ágil utilizado. Ao final apresentamos possíveis trabalhos futuros.

5.1 Sobre o Projeto

É possível desenvolver SPL Ágil utilizando o ecossistema Ruby por conta de ferramentas como `Rails`, `Bundler`, `Rake`, `Rails Plugin`, `Cucumber`, `RSpec`, `Gemfile`, `Rails Template`, entre outros. Existem outras técnicas e ferramentas que poderiam ser exploradas, porém, não coube no escopo do trabalho.

O maior problema da forma com que foi desenvolvida a SPL Ágil neste trabalho é a necessidade de modificar a arquitetura tradicional de desenvolvimento `Rails` colocando todas as *features* do projeto em `Ruby Gems` isso pode acarretar transtornos ou custos adicionais em projetos maiores com estágios de desenvolvimentos mais avançados.

A maior vantagem detectada foi a facilidade de derivar produtos por conta do `Rails Template`. A derivação de produtos tende a ser a parte mais complexa de uma SPL [Deelstra, Sinnema e Bosch 2005] mas no ecossistema Ruby é possível realiza-la com um comando e um arquivo de configuração é possível gerar um projeto de Software utilizando como bases projetos anteriores.

5.2 Ecossistema Ruby

O Ecossistema Ruby apresentou diversas ferramentas e funcionalidades muito interessantes para SPL Ágil. Seus geradores, facilitaram muito a tarefa de desenvolvimento e foi possível preservá-los nas criações das *features* graças a utilização do Rails Plugin.

O maior problema apresentado pelo ecossistema Ruby foi em seu *framework* Rails. Ele foi criado com a premissa convenção maior que configuração, fazendo com que dificulte muito o desenvolvimento fora da convenção, e no caso, uma SPL Ágil não é a convenção deles. Esse problema foi solucionado utilizando as ferramentas de compartilhamento de código da própria comunidade, Ruby Gem.

5.3 Método de SPL Ágil utilizado

Yaser [Ghanam 2012] ao criar seu método encontro no TDD uma forma de garantir consistência e qualidade de código. Porém, acreditamos que esse método ainda necessite ser testado em larga escala em projetos com muitos produtos distintos, pois sentimos que durante o trabalho, rastrear requisitos apenas utilizando testes pode ser perigoso, pois as ferramentas de testes em mercado não são preparadas para isso e conforme aumentar o número de *features* aumentará a complexidade de rastreá-las.

5.4 Trabalhos Futuros

Uma possível sequência no trabalho seria verificar a viabilidade de utilizar essa técnica de SPL Ágil e esse ecossistema em um ambiente de desenvolvimento, pois é possível que a visão obtida nesse trabalho seja alterada quando confrontada em um escopo de desenvolvimento maior.

Outro trabalho possível seria a comparação das ferramentas utilizadas nessa pesquisa com outras ferramentas mais tradicionais se tratando de SPL. Por fim, observamos que outro possível trabalho futuro é verificar como o ecossistema Ruby se comportaria quando utilizado em outras metodologias de desenvolvimento de SPL Ágil e também de SPL tradicional.

Referências Bibliográficas

[Aken 2004]AKEN, J. E. v. Management research based on the paradigm of the design sciences: the quest for field-tested and grounded technological rules. *Journal of management studies*, Wiley Online Library, v. 41, n. 2, p. 219–246, 2004.

[Aniche 2013]ANICHE, M. Test-driven development: Teste e design no mundo real. *Casa do Código*, 2013.

[Antoniol et al. 2002]ANTONIOL, G. et al. Recovering traceability links between code and documentation. *Software Engineering, IEEE Transactions on*, IEEE, v. 28, n. 10, p. 970–983, 2002.

[Arko André]ARKO ANDRé, T. L. T. M. S. S. A. M. S. G. *Bundler: The best way to menage a Ruby application's gems*. Consultado na INTERNET: http://bundler.io/, 2015.

[Baraúna 2010]BARAúNA, H. *Cucumber e RSpec: Construa aplicações Ruby com testes e especificações*. 1. ed. São Paulo: Casa do Código, 2010.

[Bayer et al. 1999]BAYER, J. et al. Pulse: a methodology to develop software product lines. In: ACM. *Proceedings of the 1999 symposium on Software reusability*. [S.l.], 1999. p. 122–131.

[Beck 2003]BECK, K. *Test-driven development: by example*. [S.l.]: Addison-Wesley Professional, 2003.

[Berube 2007]BERUBE, D. *Practical Ruby Gems*. [S.l.]: Apress, 2007.

[Bieman e Kang 1995]BIEMAN, J. M.; KANG, B.-K. Cohesion and reuse in an object-oriented system. In: ACM. *ACM SIGSOFT Software Engineering Notes*. [S.l.], 1995. v. 20, n. SI, p. 259–262.

[Bosch e Högström 2001]BOSCH, J.; HÖGSTRÖM, M. Product instantiation in software product lines: A case study. In: *Generative and Component-Based Software Engineering*. [S.l.]: Springer, 2001. p. 149–163.

[Chelimsky et al. 2010]CHELIMSKY, D. et al. *The RSpec book: Behaviour driven development with Rspec, Cucumber, and friends*. [S.l.]: Pragmatic Bookshelf, 2010.

[Clements e Northrop 2002]CLEMENTS, P.; NORTHROP, L. *Software product lines: practices and patterns*. [S.l.]: Addison-Wesley Reading, 2002.

[Cohn 2004]COHN, M. *User stories applied: For agile software development*. [S.l.]: Addison-Wesley Professional, 2004.

[Councill e Heineman 2001]COUNCILL, B.; HEINEMAN, G. T. Definition of a software component and its elements. *Component-based software engineering: putting the pieces together*, p. 5–19, 2001.

[Deelstra, Sinnema e Bosch 2005]DEELSTRA, S.; SINNEMA, M.; BOSCH, J. Product derivation in software product families: a case study. *Journal of Systems and Software*, Elsevier, v. 74, n. 2, p. 173–194, 2005.

[Dia]DIA. Consultado na INTERNET: http://dia-installer.de/, 2015.

[FIT 2007]FIT. *Fit: Framework for Integrated Test*. 2007. Consultado na INTERNET: http://fit.c2.com, 2015.

[Flanagan e Matsumoto 2008]FLANAGAN, D.; MATSUMOTO, Y. *The ruby programming language*. [S.l.]: "O'Reilly Media, Inc.", 2008.

[Fowler 1997]FOWLER, M. *Refactoring: Improving the Design of Existing Code*. [S.l.]: Addison-Wesley, Boston, 1997.

[Frakes e Fox 1995]FRAKES, W. B.; FOX, C. J. Sixteen questions about software reuse. *Communications of the ACM*, ACM, v. 38, n. 6, p. 75–ff, 1995.

[Frakes e Pole 1994]FRAKES, W. B.; POLE, T. P. An empirical study of representation methods for reusable software components. *Software Engineering, IEEE Transactions on*, IEEE, v. 20, n. 8, p. 617–630, 1994.

[Fuchs 1992]FUCHS, N. E. Specifications are (preferably) executable. *Software Engineering Journal*, IET, v. 7, n. 5, p. 323–334, 1992.

[Fuentes 2013]FUENTES, V. *Ruby on Rails: Coloque sua aplicação web nos trilhos*. [S.l.]: Casa do Código, 2013.

[Gacek e Anastasopoules 2001]GACEK, C.; ANASTASOPOULES, M. Implementing product line variabilities. In: ACM. *ACM SIGSOFT Software Engineering Notes*. [S.l.], 2001. v. 26, n. 3, p. 109–117.

[Geer 2006]GEER, D. Will software developers ride ruby on rails to success? *Computer*, IEEE, v. 39, n. 2, p. 18–20, 2006.

[Ghanam 2012]GHANAM, Y. *An agile framework for variability management in software product line engineering*. Tese (Doutorado) — University of Calgary, 2012.

[Gil 2010]GIL, A. C. Métodos e técnicas de pesquisa social. In: *Métodos e técnicas de pesquisa social*. [S.l.]: Atlas, 2010.

[Guide - RubyGems]GUIDE - RubyGems. Consultado na INTERNET: http://guides.rubygems.org/, 2015.

[Hansson 2014]HANSSON, D. H. *Agile Development: Hallmarks of Success*. 2014. Consultado na INTERNET: http://rubyonrails.org/, 2015.

[Henninger 1997]HENNINGER, S. An evolutionary approach to constructing effective software reuse repositories. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, ACM, v. 6, n. 2, p. 111–140, 1997.

[Highsmith e Cockburn 2001]HIGHSMITH, J.; COCKBURN, A. Agile software development: The business of innovation. *Computer*, IEEE, v. 34, n. 9, p. 120–127, 2001.

[Kaner 2003]KANER, C. The power of'what if... and nine ways to fuel your imagination: Cem kaner on scenario testing. *Software Testing and Quality Engineering*, v. 5, p. 16–22, 2003.

[Kästner et al. 2009]KÄSTNER, C. et al. Featureide: A tool framework for feature-oriented software development. In: IEEE. *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*. [S.l.], 2009. p. 611–614.

[Kerievsky 2010]KERIEVSKY, J. *Storytesting*. 2010. Consultado na INTERNET: http://industrialxp.org/storytesting.html, 2015.

[Krueger 2006]KRUEGER, C. W. New methods in software product line development. In: IEEE. *Software Product Line Conference, 2006 10th International*. [S.l.], 2006. p. 95–99.

[Lacerda et al. 2013]LACERDA, D. P. et al. Design science research: método de pesquisa para a engenharia de produção. *Gestão & Produção*, SciELO Brasil, v. 20, n. 4, p. 741–761, 2013.

[Maximilien et al. 2007]MAXIMILIEN, E. M. et al. *A domain-specific language for web apis and services mashups*. [S.l.]: Springer, 2007.

[Melnik, Read e Maurer 2004]MELNIK, G.; READ, K.; MAURER, F. Suitability of fit user acceptance tests for specifying functional requirements: Developer perspective. In: *Extreme programming and agile methods-XP/Agile Universe 2004*. [S.l.]: Springer, 2004. p. 60–72.

[Mendonca, Branco e Cowan 2009]MENDONCA, M.; BRANCO, M.; COWAN, D. Splot: software product lines online tools. In: ACM. *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. [S.l.], 2009. p. 761–762.

[Mohagheghi 2004]MOHAGHEGHI, P. *The impact of software reuse and incremental development on the quality of large systems*. Tese (Doutorado) — Norwegian University of Science and Technology, 2004.

[Perry 2006]PERRY, W. E. *Effective Methods for Software Testing: Includes Complete Guidelines, Checklists, and Templates*. [S.l.]: John Wiley & Sons, 2006.

[Pohl, Böckle e Linden 2005]POHL, K.; BÖCKLE, G.; LINDEN, F. V. D. Software product line engineering. *Springer*, Springer, v. 10, p. 3–540, 2005.

[Prieto-Diaz 1996]PRIETO-DIAZ, R. Reuse as a new paradigm for software development. In: *Systematic Reuse: Issues in Initiating and Improving a Reuse Program*. [S.l.]: Springer, 1996. p. 1–13.

[Python]PYTHON. Consultado na INTERNET: https://www.python.org/, 2015.

[Reppert 2004]REPPERT, T. Dont just break software, make software: How story-test-driven-development is changing the way qa, customers, and developers work. *Better Software*, New York, p. 18–23, July 2004.

[Schmid e Widen 2000]SCHMID, K.; WIDEN, T. Customizing the pulsetm product line approach to the demands of an organization. In: *Software Process Technology*. [S.l.]: Springer, 2000. p. 221–238.

[Silva et al. 2011]SILVA, I. F. da et al. Agile software product lines: a systematic mapping study. *Software: Practice and Experience*, Wiley Online Library, v. 41, n. 8, p. 899–920, 2011.

[SOUZA 2013]SOUZA, L. Ruby: Aprenda a programar na linguagem mais divertida.[sl]: Casa do código, 2013. *ISBN*, v. 335782455, p. 35, 2013.

[Stella, Jarzabek e Wadhwa 2008]STELLA, L. F. F.; JARZABEK, S.; WADHWA, B. A comparative study of maintainability of web applications on j2ee,. net and ruby on rails. In: IEEE. *Web Site Evolution, 2008. WSE 2008. 10th International Symposium on*. [S.l.], 2008. p. 93–99.

[Thao, Munson e Nguyen 2008]THAO, C.; MUNSON, E. V.; NGUYEN, T. N. Software configuration management for product derivation in software product families. In: IEEE. *Engineering of Computer Based Systems, 2008. ECBS 2008. 15th Annual IEEE International Conference and Workshop on the*. [S.l.], 2008. p. 265–274.

[Vieira 2012]VIEIRA, N. *howto - Guia RÃ¡pido de RSpec*. 1. ed. [S.l.]: Independente, 2012.

[Willi et al. 2001]WILLI, R. et al. *Documentation for jnicklas/Capybara*. 2001. Consultado na INTERNET:http://www.agilemanifesto.org, 2015.

[Wohlin e Aurum 2014]WOHLIN, C.; AURUM, A. Towards a decision-making structure for selecting a research design in empirical software engineering. *Empirical Software Engineering*, Springer, p. 1–29, 2014.