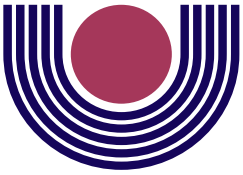

Unioeste - Universidade Estadual do Oeste do Paraná
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
Colegiado de Ciência da Computação
Curso de Bacharelado em Ciência da Computação

Utilizando Padrões de Projeto como Mecanismo para Implementar Variabilidades de Software: Um Estudo em Ruby

Rafael Fiori Kruger

CASCAVEL
2015

RAFAEL FIORI KRUGER

UTILIZANDO PADRÕES DE PROJETO COMO MECANISMO PARA IMPLEMENTAR VARIABILIDADES DE SOFTWARE: UM ESTUDO EM 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

RAFAEL FIORI KRUGER

UTILIZANDO PADRÕES DE PROJETO COMO MECANISMO PARA IMPLEMENTAR VARIABILIDADES DE SOFTWARE: UM ESTUDO EM 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 e avaliadores:

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

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

Prof. Elder Elisandro Schemberger
Colegiado de Engenharia de Computação, UTFPR

Cascavel, 8 de março de 2016

AGRADECIMENTOS

Primeiramente agradeço a Deus por ter me guiado diante das dificuldades, pela saúde e forças que me proporcionou em todos os momentos.

Agradeço ao meu orientador Ivonei, pelo seu suporte, sua paciência, pelas suas correções e incentivos no decorrer dos últimos dois anos, tanto na elaboração deste trabalho quanto no desenvolvimento de projetos de iniciação científica.

Agradeço aos meus pais, pelo incentivo, apoio emocional e financeiro, atenção e suporte constantes durante todo o meu período de graduação. Em cada falha e em cada conquista, eles estiveram a meu lado me incentivando ou comemorando juntos.

Também agradeço aos amigos que fiz no decorrer do curso, que estiveram juntos em todos os momentos da graduação, seja em momentos de estudo e de trabalho, de distração e de festas, de debates sobre os mais diversos assuntos.

Aos demais colegas e professores que conheci durante o período de graduação, que direta ou indiretamente fizeram parte de minha formação pessoal, profissional e científica.

À UNIOESTE por ter me dado aportunidade de realizar os projetos de iniciação científica com bolsa, pela estrutura de salas de aula e laboratórios, pela oportunidade de estágio e pela oportunidade de participar de cursos de línguas e testes de proficiência.

Lista de Figuras

Lista de Tabelas

Lista de Quadros

Lista de Abreviaturas e Siglas

BDD	Behavior Driven Development
CRUD	Create, Read, Update, Delete
DSL	Domain-Specific Language
DSR	Design Science Research
GQM	Goal Question Metric
IU	Interface de Usuário
LPS	Linha de Produtos de Software
RoR	Ruby on Rails
SGBD	Sistema Gerenciador de Banco de Dados
SPL	Software Product Line
TDD	Test Driven Development
UI	User Interface

Sumário

Resumo

Gerenciar variabilidades envolve o estudo de um domínio de aplicação objetivando identificar características comuns e variáveis entre sistemas similares, mas não idênticos, que pertencem ao domínio. Este estudo compreende o processo de engenharia de domínio, o qual é parte do paradigma de linha de produtos de *software* (LPS). Neste contexto, os requisitos e funcionalidades de domínio são entendidos como *features*. As *features* podem possuir código comum a vários produtos, como também podem possuir pontos de variação, isto é, formas diferentes de serem realizadas.

Para que seja possível implementar as variabilidades e comunalidades presentes nas *features*, são utilizados os mecanismos de implementação de variabilidades. Padrões de projeto são utilizados como mecanismo para implementar variabilidades. Os padrões de projeto *Observer*, *Template Method*, *Strategy* e *Decorator* são bem adaptados para implementar variabilidades que podem ser habilitadas ou desabilitadas em tempo de execução.

Este trabalho consistiu em um estudo da viabilidade técnica de utilizar estes padrões de projeto para implementar variabilidades em `Ruby`. Para isto, foi necessário definir características específicas sobre como os padrões de projeto possibilitam a implementação de variabilidades. Os seguintes pontos de análise foram considerados para isso: os tipos *features*, de variabilidade, de unidades de código, de escopo de variabilidade e tempo de vinculação existentes. Após definir estas características, implementações na linguagem `Ruby` foram realizadas. Informações foram coletadas das implementações e analisadas em paralelo com as características identificadas na teoria. Conclui-se que é viável tecnicamente implementar variabilidades com estes padrões de projeto utilizando a linguagem `Ruby`.

Palavras-chave: Linha de Produtos de Software; Ruby; Padrões de Projeto; Implementação; Variabilidade

Capítulo 1

Introdução

Neste capítulo são introduzidos os temas abordados com objetivo de facilitar o entendimento das motivações, justificativa, problema e objetivos deste trabalho.

1.1 Contextualização

Ao longo dos últimos anos, diversas pesquisas surgiram na área de linha de produtos de *software* (LPS), do inglês *software product lines* (SPL) [Clements e Northrop 2001], também chamada de família de produtos de *software*. Este é um paradigma da engenharia de *software* que objetiva mapear e criar uma base de características comuns e características variáveis entre os diferentes produtos de um domínio específico de aplicação. Por domínio de aplicação entende-se o segmento de mercado que a linha de produtos de *software* foca.

Em uma LPS existem vários artefatos, os quais podem ser no nível de documentação, testes, código de programação, requisitos funcionais e não funcionais e outras características pertinentes para o desenvolvimento de *software*. Neste contexto, o conjunto de artefatos que define uma característica específica dentro do domínio de aplicação é denominado de *feature*. Uma *feature* também é considerada uma abstração dos requisitos [Böckle, Pohl e Linden 2005].

Para exemplificar, suponha uma LPS no domínio de *blogs*. O domínio de aplicação refere-se a todos os possíveis produtos de *blogs*. Através desta LPS no domínio de *blogs*, é possível derivar produtos que possuam *features* em comum, tais como gerenciamento de *posts*, gerenciamento de usuário e login. Porém, é possível também adicionar *features* que são variáveis, isto é, *features* que alguns produtos gerados por esta linha de produto não necessariamente possuem, ou que são feitos de formas diferentes entre eles, como por exemplo o cadastro de usuários,

gerenciamento de comentários ou categorias. Em um contexto mais geral, uma *feature* pode ser, por exemplo, o gerenciamento de usuário ou o gerenciamento de comentários, sendo características específicas existentes dentro do domínio de *blogs*.

Esta característica da LPS de identificar o que é comum e o que é variável traz muitos benefícios. Ao identificar os artefatos comuns, permite-se que estes artefatos sejam reutilizados em massa e, ao identificar o que é variável, permite-se que os produtos sejam customizados de acordo com as necessidades ou desejos dos clientes, utilizando uma base de *features* de onde são derivados os produtos. Desta forma, a linha de produtos de *software* traz vantagens econômicas de longo prazo [Clements e Northrop 2001].

A variabilidade para a LPS é a habilidade de mudar ou customizar um sistema [Bosch, Gurp e Svahnberg 2000]. Variabilidade pode ser os aspectos comuns entre todos os sistemas, pode ser uma característica comum a alguns sistemas mas não para todos (incluindo formas diferentes de desenvolver uma característica), e também pode ser específica de um produto [Rommes, Schmid e Linden 2007].

Assim como na engenharia de requisitos, onde os requisitos devem ser corretamente identificados, na LPS as *features* e suas variabilidades e comunalidades devem ser identificadas para que possam ser gerenciadas.

Construir um sistema de forma a facilitar o gerenciamento da variabilidade implica que mudanças futuras serão facilitadas. É possível indentificar antecipadamente alguns tipos de variabilidade e construir o sistema de forma que este facilite o desenvolvimento destas variabilidades. Porém, infelizmente sempre há alguns tipos de variabilidade as quais não é possível identificar antecipadamente [Bosch, Gurp e Svahnberg 2000].

As *features* que contêm variabilidade possuem pontos de variação. Entende-se que estes pontos são os locais onde a variabilidade vai ser habilitada. Os pontos de variação são locais onde podem existir diferentes formas de habilitar algum recurso. *Features* e pontos de variação são subdivididos e classificados por tipos.

Um ponto de variação pode ser aberto ou fechado [Bosch, Svahnberg e Gurp 2005]. Um ponto de variação é dito ser aberto quando novas variantes podem ser adicionadas e as antigas podem ser removidas. Por outro lado, um ponto de variação é fechado quando não é possível adicionar novas variantes. O ponto de variação aberto pode ser melhor descrito utilizando o

termo “escopo de variabilidade” [Chang, Her e Kim 2005]. O escopo é classificado em binário, seleção e aberto. É “binário” quando somente podem existir duas variantes em um ponto de variação. O escopo é denominado “seleção” quando três ou mais variações previamente conhecidas estão disponíveis em um ponto de variação. O escopo é chamado de “aberto” quando podem existir qualquer quantidade de variantes conhecidas e desconhecidas em um ponto de variação [Amin, Mahmood e Oxley 2011].

Existem denominações também para os tipos de *feature* existentes, as quais são as seguintes [Anastasopoulos e Gacek 2001]:

- Obrigatório: quando a *feature* deve estar presente em todo produto;
- Opcional: quando a *feature* é um complemento independente que pode ser incluído ou não;
- Alternativo: quando uma *feature* é incluída de forma a substituir outra feature;
- Mutualmente Inclusivo: quando uma *feature* somente pode ser incluída se outra *feature* específica for incluída também e vice-versa;
- Mutualmente Exclusivo: quando uma *feature* somente pode ser incluída se outra *feature* específica não for incluída e vice-versa.

Os tipos de variabilidade existentes são os seguintes [Rommes, Schmid e Linden 2007]:

- Comunalidade: característica comum a todos os produtos da LPS;
- Variabilidade: característica que pode ser comum para alguns produtos, mas não para todos;
- Específica de Produto: característica que pode ser parte de somente um produto;

A distinção entre o reuso na engenharia de domímio da linha de produtos de *software* e entre algumas das várias outras técnicas de reuso, tais como *frameworks* [Apel et al. 2013], componentes [Apel et al. 2013] ou bibliotecas de classes [Apel et al. 2013] é que os vários artefatos e *features* componentes da LPS contém variabilidade explícita [Rommes, Schmid e Linden 2007]. Isto é, ao observar os componentes e *features* existentes

no sistema, suas características permitem que o cliente saiba exatamente o que é comum e o que pode variar entre os sistemas.

Desta forma, é necessário diferenciar o desenvolvimento de *software* **com reuso** do desenvolvimento de *software* **para reuso**. No desenvolvimento com reuso, os componentes ou artefatos são reutilizados de forma ocasional e este reuso não é planejado sistematicamente. No desenvolvimento para reuso, tudo o que pode ser reutilizado é identificado previamente, para então ser desenvolvido de forma que posteriormente possa ser reutilizado [Sommerville 2008].

Tendo isso posto, uma LPS possui estas duas formas de reuso, as quais são organizadas de uma forma que sistematiza o desenvolvimento como um todo. Na LPS, o reuso será feito em larga escala. As *features* que podem ser reutilizadas são identificadas previamente para que possam ser efetivamente reutilizadas posteriormente. Esta forma sistemática de construção da engenharia de LPS garante a organização e gerenciamento do reuso, de modo que qualquer modificação que necessite ser feita em um artefato será replicada automaticamente aos artefatos correspondentes em todos os produtos que o possuem, diferentemente do que ocorre em outras técnicas de reuso [Rommes, Schmid e Linden 2007].

Por conseguinte, o paradigma de engenharia de linha de produtos de *software* possui dois processos distintos em sua composição [Rommes, Schmid e Linden 2007] [Böckle, Pohl e Linden 2005].

O primeiro processo, denominado engenharia de domínio, possui foco na análise de domínio e seu desenvolvimento é para o reuso, isto é, deve prever tudo o que pode ser reutilizado, de forma a gerar uma base de artefatos comuns para a derivação de produtos individuais no processo posterior [Rommes, Schmid e Linden 2007]. Nesta etapa, os componentes do domínio são descritos por casos de uso e por modelos de *features*.

O segundo processo, denominado de engenharia de aplicação, possui foco na geração e construção dos vários produtos com características comuns a partir da base definida no processo anterior [Rommes, Schmid e Linden 2007].

A engenharia de domínio trata de diversos artefatos utilizados para a modelagem de domínio, para o gerenciamento de reuso e para a realização de variabilidades, tais como a documentação, requisitos funcionais e não funcionais, testes e código de programação. Todos estes artefatos serão utilizados de alguma forma no processo de engenharia de aplicação para a gera-

ção de um produto de *software*. No entanto, o interesse aqui é referente aos artefatos de código de programação da engenharia de domínio.

Neste contexto, para que seja possível desenvolver os artefatos de código de programação, é necessário utilizar mecanismos que possibilitem gerar artefatos reusáveis e artefatos que possuem variabilidade. São diversos os mecanismos existentes, que podem ser subdivididos em mecanismos voltados à ferramentas e mecanismos voltados à linguagens de programação [Apel et al. 2013]. Várias técnicas de reuso, incluindo a engenharia de linha de produtos de *software*, utilizam destes mecanismos para habilitar o reuso e a variabilidade em seus artefatos de código.

Este trabalho possui foco na implementação de variabilidades de *software*, uma das etapas da engenharia de domínio. É abordado o mecanismo de padrões de projeto, o qual é voltado a linguagens de programação. Mais especificamente, este mecanismo é analisado quanto as suas características para implementação de variabilidades de *software* em uma LPS. Os seguintes padrões de projeto são alvos de estudo deste trabalho: *Template Method*, *Decorator*, *Strategy* e *Observer*. Estes padrões foram identificados na literatura como sendo mecanismos de tempo de execução bem adaptados para implementação de variabilidades [Apel et al. 2013]. Após implementados, possibilitam a adição ou remoção de *features* em tempo de execução. Desta forma, estes mecanismos adequam-se às necessidades deste trabalho.

A análise do mecanismo de padrões de projeto objetiva identificar como é realizada a manipulação da variabilidade e de *features* através dos padrões *Template Method*, *Decorator*, *Strategy* e *Observer*. Esta análise é feita para cada um dos padrões de projeto que é objeto de estudo deste trabalho, e a análise os classifica quanto:

- aos tipos de variabilidade que é possível obter em relação à sua finalidade e também ao seu efeito;
- ao seu tempo de vinculação, isto é, qual momento o padrão de projeto é capaz de habilitar a variabilidade;
- ao escopo de variabilidade que o padrão de projeto possui;
- a quais artefatos o padrão de projeto opera para habilitar a variabilidade (atributos, métodos, classes, objetos);

- aos tipos de *features* que o padrão de projeto pode habilitar.

Após esta análise, é adotada uma linguagem de programação para realizar implementações dos padrões de projeto, e a linguagem `Ruby` foi escolhida para esta finalidade. São implementadas variabilidades de *software* a partir de requisitos obtidos de um domínio de aplicação utilizando o mecanismo de padrões de projeto para estruturar e guiar a implementação destas variabilidades. O domínio de aplicação utilizado é o domínio de blogs. A intenção foi criar uma base de *features* através da implementação dos padrões de projeto citados anteriormente.

Questões acerca da derivação de produtos e execução dos mesmos, etapas da engenharia de aplicação, não são consideradas neste trabalho, bem como o gerenciamento de variabilidade de requisitos, de artefatos de projeto e de testes. A discussão e estudo realizados neste trabalho são exclusivamente acerca do gerenciamento de variabilidades relativos à fase de implementação da engenharia de domínio, referindo-se à geração de código fonte ou à tradução de um projeto em componentes de *software* reusáveis e que possuíssem variabilidades também.

A coleta e análise de dados, tanto das características, identificadas em teoria, destes padrões de projeto que são voltadas à implementação de variabilidades, quanto das características obtidas das implementações dos padrões, é estruturadas tendo como base a abordagem do *Goal Question Metric* (GQM) [Basili, Caldiera e Rombach 1994] (ver seção 3.1.8 para detalhes desta metodologia), na qual são definidas questões de pesquisa e métricas respectivas à estas questões, de forma que as questões são respondidas pelas métricas.

Como são coletados dados tanto da teoria quanto das implementações dos padrões de projeto, ao final é possível comparar a análise dos padrões de projeto realizada anteriormente com os dados coletados das implementações, a fim de obter, na análise final, um estudo da viabilidade técnica de implementar os padrões de projeto citados anteriomente no contexto de linha de produtos de *software* utilizando a linguagem de programação selecionada para estas implementações. A análise dos padrões de projeto e de suas implementações em `Ruby` segue os parâmetros adotados em [Anastasopoulos e Gacek 2001] e em [Amin, Mahmood e Oxley 2011].

A linguagem de programação `Ruby`, que foi adotada como linguagem base para implementar os padrões de projeto, é uma linguagem de programação interpretada, com tipagem dinâmica e forte. Contempla os paradigmas de programação orientado a objetos, funcional e imperativo [Flanagan e Matsumoto 2008].

Além disso, possui recursos como as *Gems*, para a geração de bibliotecas; os *Mixins*, para o desenvolvimento modularizado de código; blocos, possibilitando a execução dinâmica de código; e a metaprogramação e reflexão, que possibilitam uma programação que atua sobre outro programa [Flanagan e Matsumoto 2008].

O `Ruby` também dispõe de um *framework* para a *web*, chamado de `Rails`. Este *framework* é construído sobre as premissas de simplicidade, reusabilidade, extensibilidade, velocidade, testabilidade, produtividade e portabilidade. `Rails` utiliza convenções ao invés de configurações, de forma que seguir a convenção se torna mais simples do que realizar configurações [Ruby-On-Rails 2015].

O `Rails` possui um recurso chamado *ActiveRecord*, o qual facilita o trabalho com banco de dados, fornecendo um gerador de *Create, Read, Update, Delete* (CRUD) dinâmico, possibilitando que a modelagem do banco de dados e a geração de consultas seja realizada de forma dinâmica, conforme o desenvolvimento da aplicação. Além disso, por ser um *framework* para a *web*, possui ferramentas para construção de interfaces de usuário e utiliza o servidor *WEBrick* por padrão [Ruby-On-Rails 2015].

1.2 Motivações e Justificativa

Assim como ocorre na engenharia de requisitos, onde se necessita identificar e descrever as funcionalidades corretas que deverão ser implementadas, em uma LPS deve-se identificar e descrever as funcionalidades corretas para a implementação de artefatos reutilizáveis, identificando os requisitos que possam gerar possíveis pontos de variabilidade em produtos futuros. Os requisitos, comunalidades e variabilidades identificados devem satisfazer as principais metas de negócios da empresa [Kang, Park e Sugumaran 2009].

Neste contexto de linha de produtos de *software*, após a descrição dos requisitos com variabilidade, existe a preocupação de como proceder para implementar código com variabilidades dentro dos artefatos de *software*. Para solucionar este problema, foram estudados e criados diversos mecanismos de implementação de variabilidades no decorrer dos últimos anos, levando em consideração as características de tecnologias e linguagens de programação existentes ou que vieram a surgir durante este período. Cada um destes mecanismos possui um contexto de aplicação que abrange os objetos que são foco da solução do mecanismo dentro do código, as

características de linguagens de programação ou outras tecnologias que são exploradas para fornecer a solução, onde e quando devem ser aplicados e quais suas limitações.

Existem diversos trabalhos analisando aspectos de mecanismos de implementação e manipulação de variabilidades no contexto de linha de produtos de *software* [Amin, Mahmood e Oxley 2011] [Anastasopoulos e Gacek 2001] [Bosch, Gurp e Svahnberg 2000] [Fritsch, Lehn e Strohm 2002] [Gurp e Savolainen 2006].

Em [Anastasopoulos e Gacek 2001], a análise dos mecanismos, a nível de código, é feita em relação aos tipos de variabilidade que é possivel obter nas etapas de *interface*, implementação e inicialização e em qual momento o mecanismo é capaz de habilitar a variabilidade, indicando os requisitos e restrições de uso quanto a um conjunto específico de linguagens de programação. A análise dos mecanismos também é realizada em relação a alguns critérios de validação de mecanismos de implementação de variabilidades. Os critérios avaliados neste artigo foram *separation of concerns*, traceabilidade e escalabilidade.

Em [Amin, Mahmood e Oxley 2011], o objetivo é classificar os mecanismos orientados a objetos, a nível de código, em termos de tipo de variabilidade, escopo, tipos de *feature*, quais artefatos são afetados pela solução proposta pelo mecanismo e em qual momento o mecanismo é capaz de habilitar a variabilidade.

Entretanto, estes trabalhos apenas citam padrões de projeto como um mecanismo possível para implementar variabilidades, mas não realizam uma análise específica sobre alguns dos vários padrões de projeto existentes.

Padrões de projeto tem sido amplamente disseminados e utilizados na comunidade de desenvolvedores de *software* e é uma técnica clássica de reuso [Apel et al. 2013] [Gamma et al. 1995]. Revisões da literatura apontaram que alguns dos padrões de projeto clássicos [Gamma et al. 1995] existentes são adequados para serem utilizados como mecanismos para implementação de variabilidades [Apel et al. 2013], onde estes padrões são os seguintes: *Template Method*, *Decorator*, *Strategy* e *Observer*. Além disso, são mecanismos que podem habilitar ou desabilitar as *features* em tempo de execução. Estes fatos acabaram por motivar a utilização destes padrões de projeto como mecanismo para a condução do desenvolvimento dos artefatos de *software* com variabilidade realizado neste trabalho.

Baseando-se nos trabalhos que analisam os aspectos dos mecanismos de variabilida-

des, existem outros vários trabalhos que analisam aspectos destes mecanismos de variabilidades de *software* utilizando `Ruby` como linguagem base e como variável de influência na implementação destes mecanismos, demonstrando um interesse crescente em validar `Ruby` como uma linguagem viável para a implementação de variabilidades [Günther 2009] [Günther e Sunkle 2009-A] [Jesus 2013] [Günther e Sunkle 2009-B] [Günther e Sunkle 2010] [Günther e Sunkle 2012].

Além disso, estes trabalhos apontam que a natureza dinâmica, as facilidades da metaprogramação e a simplificação sintática presentes em linguagens de *scripting* fazem deste tipo de linguagem uma alternativa adequada para implementação de variabilidades. De acordo com estes trabalhos, entre as linguagens de *scripting* existentes, `Ruby` mostra-se a com alta práticidade de utilização por conta de características como modificações sintáticas, tipagem de dados e facilidade de leitura de código [Günther 2009] [Jesus 2013]. Alia-se a isto a praticidade, rapidez no desenvolvimento de *software* e produtividade que seu *framework* para a web `Rails` fornece [Jesus 2013], sendo um *framework* de ampla divulgação atualmente, colaborando fortemente para que a linguagem `Ruby` ocupe o décimo lugar no índice *Tiobe* (atualização de janeiro de 2016), onde este índice refere-se à uma classificação das linguagens de programação mais utilizadas [Tiobe 2016].

É possível desenvolver em `Ruby` linguagens de domínio específico, do inglês *domain-specific language* (DSL), para gerar representações mais acuradas de uma solução proposta para uma área específica [Günther 2009]. Isto foi feito utilizando notações e as abstrações para representar o conhecimento e os conceitos do domínio. Como o conhecimento de domínio é colocado dentro da linguagem de programação, o programador que a utiliza tem melhor conhecimento do domínio [Günther 2009].

Em continuação ao trabalho mencionado anteriormente, foi definido o mecanismo de implementação de variabilidades orientado à *feature*, que foi desenvolvido utilizando uma DSL no contexto de linha de produtos de *software* [Günther e Sunkle 2009-A] [Günther e Sunkle 2009-B] [Günther e Sunkle 2012]. Em adição a estes trabalhos, em [Günther e Sunkle 2010] o mecanismo de metaprogramação foi implementado como extensão do mecanismo orientado à *feature*, aproveitando-se das características dinâmicas presentes em `Ruby`, como forma de possibilitar e facilitar a execução de mudanças no código em tempo de

execução.

1.2.1 Problema

Como já citado anteriomente, nos trabalhos que avaliam os mecanismos de implementação e manipulação de variabilidades de *software* não há uma avaliação específica para o mecanismo de padrões de projeto em relação à implementação de variabilidades, apesar de alguns dos trabalhos citarem e caracterizarem este mecanismo como possível para fazer isto.

Além disso, em [Anastasopoulos e Gacek 2001] não há uma avaliação sobre quais mecanismos podem ou não serem utilizados quando a linguagem `Ruby` é utilizada como linguagem base, apesar de o artigo apresentar esta avaliação para outras linguagens, tais como Java, C++, Smalltalk e Delphi.

Em nossas pesquisas anteriores, realizadas no contexto de iniciação científica, realizamos um estudo de viabilidade técnica de implementar na linguagem `Ruby` os mecanismos presentes nos trabalhos supracitados, analisando, identificando e mapeando quais mecanismos podem ou não ser utilizados nesta linguagem, bem como os que não são efetivos, considerando, para realizar esta avaliação, a interseção entre as características necessárias para o desenvolvimento de uma linha de produtos de *software* juntamente com as características presentes na linguagem `Ruby` e em cada um dos mecanismos estudados.

Nestas nossas pesquisas anteriores, as avaliações presentes em [Anastasopoulos e Gacek 2001] e [Amin, Mahmood e Oxley 2011] foram utilizadas como parâmetro para a coleta de dados e avaliação dos mecanismos implementados em `Ruby`. O mapeamento destes mecanismos em relação aos resultados encontrados é mostrado a seguir: Agregação/Delegação: possível; Herança: possível; Parametrização: possível; Sobrecarga: inefetivo; Propriedades: não possível; Programação Orientada a Aspecto: possível; Carregamento Dinâmico de Classes: possível; Bibliotecas Estáticas: possível; Bibliotecas de Ligação Dinâmica: possível; Frames: possível; Genéricos: inefetivo; Compilação Condiconal: não possível.

Porém, em nossos trabalhos anteriores, o mecanismo de padrões de projeto não foi implementado em `Ruby` ou avaliado. Em [Anastasopoulos e Gacek 2001] há uma breve explicação sobre este mecanismo, porém não possui uma avaliação mais detalhada deste mecanismo em

específico neste trabalho, de forma que não havia um conjunto de informações que pudessem ser utilizadas como base de referência para a coleta de dados realizada em nossas pesquisas, necessitando de uma pesquisa que identificasse as características deste mecanismo e quais dos vários padrões de projeto existentes são adequados para a implementação de variabilidades de *software*, para que posteriormente a avaliação deste mecanismo pudesse ser elaborada considerando a linguagem `Ruby` como variável de influência.

Desta forma, considerando desenvolvedores em `Ruby` que adotam ou desejam adotar os padrões de projeto *Template Method*, *Decorator*, *Strategy* e *Observer* como mecanismo de implementação de variabilidade, ainda há necessidade de verificar a viabilidade técnica de implementar variabilidades utilizando estes padrões de projeto na linguagem `Ruby`.

1.2.2 Questão de Pesquisa

Sabendo que padrões de projeto foram estudados e projetados para possibilitarem o reuso, levando em consideração características da orientação a objetos [Gamma et al. 1995], e sabendo que `Ruby` é uma linguagem que possui fortes características da orientação a objetos, é possível realizar a seguinte questão: é viável tecnicamente implementar variabilidades de *software* considerando o cenário de utilizar os padrões de projeto *Template Method*, *Decorator*, *Strategy* e *Observer* em `Ruby`?

No contexto desta questão, estudar a viabilidade técnica é um termo mais abstrato, que refere-se a verificar quais elementos de variabilidade cada um destes padrões de projeto contempla e a aderência destes padrões em relação à implementação de variabilidades de *software* na linguagem `Ruby`.

Supondo que seja possível utilizar as várias outras características presentes em `Ruby`, tais como a metaprogramação, programação funcional, blocos, *Mixins* e *Gems* (ver seção 2.3 para informações detalhadas sobre estas características), de forma a auxiliar a implementação de padrões de projeto, ou ao menos oferecer formas alternativas de realizar estas implementações, e sabendo que as soluções propostas por padrões de projeto podem ser codificadas em várias linguagens de programação distintas, pois os padrões descrevem somente o projeto e não trechos de código em si [Apel et al. 2013], a viabilidade técnica será mensurada tendo por base os seguintes pontos de análise:

- pode ser possível que as características da linguagem `Ruby` influenciem na implementação das variabilidades utilizando os padrões de projeto *Template Method*, *Decorator*, *Strategy* e *Observer* em `Ruby`;
- caso as características de `Ruby` influenciem, identificar como isto ocorre;
- identificar se afeta positivamente ou negativamente a implementação e uso das variabilidades;
- é possível que ocorram mudanças nos tipos de variabilidades, nos tipos de *features*, no escopo, nas unidades de código e no tempo de vinculação que teoricamente são atendidos por cada um dos padrões de projeto que são alvo de estudo deste trabalho.

1.3 Objetivos

Este trabalho possui um objetivo geral que é refinado em objetivos específicos. Uma lógica indutiva é utilizada, permitindo coletar informações e características específicas identificadas em cada padrão de projeto implementado, sob vários contextos e pontos de vista, que respondem, em seu conjunto, à questão mais geral de pesquisa [Aurum e Wohlin 2014].

Este trabalho almeja obter um estudo da viabilidade técnica do uso de padrões de projeto como mecanismo para implementar variabilidades em `Ruby`. Para isto ele segue um processo dividido em etapas (objetivos específicos).

As etapas deste processo são compostas da seguinte forma:

- (i) análise dos padrões de projeto *Template Method*, *Decorator*, *Strategy* e *Observer* com relação à implementação de variabilidades;
- (ii) definição do escopo que será utilizado como base de requisitos para as implementações;
- (iii) projetar o desenvolvimento de artefatos de *software* que possuem variabilidade utilizando os padrões de projeto citados anteriomente;
- (iv) implementação destes padrões de projeto em `Ruby` juntamente com a coleta de dados;

- (v) comparação das informações obtidas na análise dos padrões com as informações obtidas das implementações.

Na Figura 1.1 é possível obter uma visão geral e observar o fluxo deste processo.

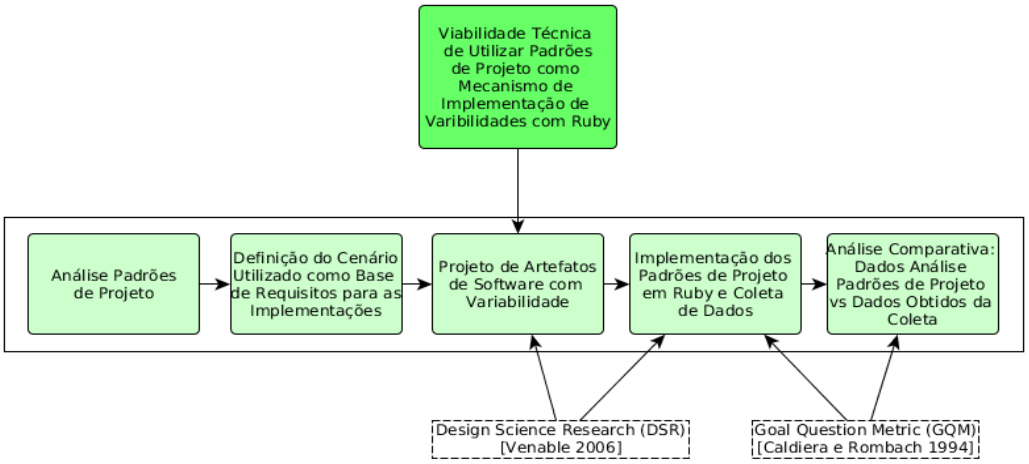

Figura 1.1: Visão geral e fluxo do processo (objetivos específicos) necessário para atingir o objetivo geral

Na Figura 1.1 são observadas cinco etapas que compõem o objetivo final. Estas etapas podem ser interpretadas também como objetivos específicos, os quais devem ser realizados para que o objetivo geral seja atingido.

No primeiro objetivo específico é necessário analisar, utilizando a descrição de cada um destes padrões de projeto, como a implementação de variabilidades pode ser realizada em cada um dos padrões. Esta análise leva em consideração o tipo de variabilidade em relação à funcionalidade e em relação ao efeito da variação, além de identificar seu escopo, seu tempo de vinculação, os tipos de *features* que podem ser habilitados e quais são os artefatos alvo no código (métodos, classes, atributos, objetos). Esta análise segue os critérios adotados em [Amin, Mahmood e Oxley 2011] e em [Anastasopoulos e Gacek 2001] e, neste ponto, não leva em consideração a interferência de uma linguagem de programação específica. Esta análise é uma revisão da teoria que identifica nestes padrões de projeto os pontos de análise definidos para responder à questão de pesquisa.

O segundo objetivo específico refere-se à definição de um domínio de aplicação para que possa servir de base de requisitos para as implementações. Neste trabalho, o domínio será o de

blogs. Inspirando-se por características existentes em sites de construção de *blogs* e considerando a etapa de engenharia de domínio existente no paradigma de LPS, objetiva-se a construção de uma base de *features*, onde as *features* possuam variabilidades e comunalidades (ver a subseção 3.1.9 para informações detalhadas deste modelo de *features*).

O terceiro objetivo específico é referente ao projeto dos artefatos de *software* que são implementados posteriormente utilizando a metodologia do *Design Science Research* (DSR) [Venable 2006] (ver subseção 3.1.7 para detalhes sobre o DSR). Implementar variabilidades em `Ruby` é o problema, porém também deve-se considerar que isto é feito utilizando padrões de projeto previamente definidos. Cada padrão de projeto possui uma descrição e um contexto de uso e a linguagem `Ruby` possui características que impactam no projeto dos artefatos de código desenvolvidos em `Ruby`, sendo necessário levar estas características em consideração no projeto dos artefatos. O domínio de aplicação definido anteriormente é o que fornece contexto para o projeto dos artefatos.

O quarto objetivo específico é a implementação dos padrões *Template Method*, *Decorator*, *Strategy* e *Observer* utilizando a linguagem de programação `Ruby`, com foco na implementação de variabilidades, e coletar informações acerca destas implementações. Nesta etapa o modelo de *features* elaborado anteriormente é utilizado como base de requisitos para estas implementações. De acordo com a metodologia do DSR, conhecimento sobre as implementações será obtido ao realizar as implementações [Venable 2006]. Será possível então mapear este conhecimento com as questões e respectivas métricas definidas no GQM.

Por fim, o quinto objetivo específico é uma análise realizando um paralelo com as informações obtidas na análise dos padrões de projeto com as informações coletadas das implementações do padrões em `Ruby`. O GQM é utilizado no trabalho como uma ferramenta que facilita a definição das questões de pesquisa e suas respecticas métricas, a qualificação e interpretação dos dados obtidos [Basili, Caldiera e Rombach 1994] (ver subseção 3.1.8 para detalhes sobre o GQM). Desta forma, será possível identificar semelhanças, diferenças e possíveis relações existentes entre as informações da análise dos padrões de projeto e das implementações.

1.4 Resultados Esperados

Do ponto de vista teórico, os resultados deste estudo de viabilidade identificam características relacionadas à implementação de variabilidades utilizando os padrões de projeto *Template Method*, *Decorator*, *Strategy* e *Observer*.

Do ponto de vista prático, os resultados demonstram se estes padrões possuem variações quanto aos tipos de *features*, variabilidades, tempo de vinculação, escopo de variabilidade e em quais artefatos os padrões operam quando a linguagem de programação `Ruby` está sendo utilizada como variável de influência para implementar as variabilidades.

1.5 Visão Geral do Documento

A seguir, os capítulos presentes neste trabalho são sumarizados, apresentando-se brevemente os assuntos tratados em cada um.

Capítulo 2, a fundamentação teórica necessária é apresentada, explicando os conceitos, temas e ferramentas utilizados. São apresentados os principais conceitos e processos do paradigma de linha de produtos de *software*. São apresentadas as definições de variabilidade de *software* e de *features* no contexto de LPS, além da definição e classificação dos tipos de mecanismos de implementação de variabilidades de *software* e em que etapas do desenvolvimento de uma linha de produtos elas podem ocorrer, as vantagens e desvantagens da utilização de linha de produtos de *software*.

É apresentada uma introdução textual à linguagem de programação `Ruby`, trazendo seu histórico e explicando seus principais conceitos, os paradigmas que contempla e suas características gerais. O *framework* `Rails` é apresentado explicando seus principais conceitos, características e convenções.

São descritos os principais trabalhos relacionados, realizando um paralelo com o presente trabalho.

Capítulo 3, é apresentado o protocolo de pesquisa definido para guiar e estruturar a pesquisa, abrangendo as etapas de definição das questões de pesquisa, estruturação da pesquisa, desenvolvimento, coleta de dados e manipulação dos dados obtidos neste trabalho. Também é apresentada a definição do cenário que foi utilizado como base de requisitos para as implemen-

tações.

Capítulo 4, é apresentada uma introdução ao mecanismo de padrões de projeto e sua utilização como mecanismo para implementação de variabilidades. Segue uma análise de padrões de projeto em específico com foco na implementação de variabilidades, sendo apresentada a definição das questões e métricas a serem mapeadas da teoria, tendo por base a metodologia do GQM. Os padrões de projeto abordados são os seguintes: *Template Method*, *Decorator*, *Strategy* e *Observer*. O modelo de coleta de dados baseado no GQM é utilizado para mapear os dados necessários para a análise. Ao final, um sumário de comparação é apresentado.

Capítulo 5, uma avaliação é realizada para as implementações dos padrões de projeto *Template Method*, *Decorator*, *Strategy* e *Observer* em `Ruby`. Inicia-se mostrando como a avaliação foi planejada e qual escopo foi abordado e o projeto dos artefatos de *software* que foram desenvolvidos tendo como base a metodologia do DSR.

Também é apresentada a definição das questões e suas respecticas métricas a serem mapeadas através da coleta de dados, tendo por base a metodologia do GQM.

É apresentado os resultados da coleta de dados, fornecendo uma análise e discussão acerca destes resultados, tendo como foco quais elementos de variabilidade foram possíves obter e a aderência destes padrões em relação à liguagem `Ruby`.

Ao final, um sumário de comparação é apresentado, realizando um comparativo com o sumário apresentado no final do Capítulo 4.

Capítulo 6, é apresentada a conclusão do trabalho, retomando o contexto do trabalho e do estudo realizado, apresentando as contribuições do trabalho e respondendo a questão primária de pesquisa. É mostrado também como este trabalho pode ser refinado ou complementado através de trabalhos futuros.

Capítulo 2

Fundamentação Teórica e Ferramentas

Neste capítulo, os conceitos e ferramentas utilizados neste trabalho são apresentados para facilitar o entendimento da proposta desenvolvida neste trabalho.

Para informações mais detalhadas e profundas acerca de cada conceito e ferramenta apresentados, solicita-se a consulta da bibliografia utilizada.

2.1 Linha de Produtos de Software

Em contraposição às metodologias clássicas de desenvolvimento de *software* onde a análise de requisitos é feita normalmente para um único produto e o sistema é todo construído especificamente para este produto, em LPS a análise é feita sobre o domínio de aplicação, de forma que os artefatos construídos possam ser sistematicamente reutilizados, possibilitando posterior derivação em vários produtos similares na etapa de engenharia de aplicação através da combinação dos artefatos desenvolvidos, buscando as similaridades e possíveis variabilidades entre os requisitos do domínio [Rommes, Schmid e Linden 2007]. Por domínio de aplicação entende-se como sendo o segmento de mercado que a linha de produtos foca.

A diferença básica entre as duas formas de desenvolvimento de *software* é a mudança de foco na etapa de análise de requisitos [Rommes, Schmid e Linden 2007].

Uma LPS é uma família de produtos de *software* que possui meios para que as *features* e artefatos comuns aos produtos da família possam ser gerenciados e compartilhados, de forma a permitir uma margem de variabilidade para satisfazer diferentes necessidades dos clientes ou de um segmento de mercado [Clements e Northrop 2001].

Para melhor compreender como a linha de produtos faz o reuso de seus artefatos e fornece a

variabilidade que possibilita a customização em massa, na Figura 2.1 pode ser visto um exemplo de linha de produtos. A figura mostra um configurador de uma linha de produto de sanduíches, exemplificando uma abordagem de linha de produtos que pode ser encontrada na indústria de *fastfood*. É um exemplo não muito usual, porém é válido e ilustrativo, de forma que facilita o entendimento do conceito geral de linha de produtos.

Algumas lojas de sanduíches não vendem tipos predefinidos de sanduíches, mas permitem que o cliente escolha entre um conjunto de opções para criar seu próprio sanduíche de acordo com seu gosto pessoal. O conjunto de escolhas possíveis é grande porém finito, incluindo diferentes tipos de pães, coberturas principais e secundárias, bem como vários tipos de molhos e temperos. O produtor não pega pedidos arbitrários e começa a pensar no que se deve comprar e em como preparar os ingredientes para cada pedido. Ao invés disso, o produtor define um conjunto de ingredientes e prepara as partes antecipadamente, tais como tomates cortados, carnes pré cozidas e molhos que possam ser combinados [Apel et al. 2013].

Algumas companhias fornecem um configurador semelhante ao da Figura 2.1. Os clientes não podem escolher arbitrariamente, mas são guiados por um configurador que possui descrições sobre opções pré-definidas. Desta forma, o espaço de possíveis sanduíches que podem ser fabricados é grande, e provavelmente irá oferecer uma opção adequada e adaptada para a vasta maioria dos clientes [Apel et al. 2013].

Em ambos modelos de negócio, o produtor sempre estará utilizando uma base de ingredientes igual e fornecendo uma ampla gama de variabilidade de sanduíches ao cliente.

Como é possível notar através deste exemplo, cada produto derivado da base da linha de produtos possui alguma semelhança com os outros produtos derivados dessa mesma base, sendo que essa semelhança refere-se à utilização de componentes iguais ou da forma de construção do produto. É importante entender, tal como pode ser observado na Figura 2.1, que as *features* referem-se a uma descrição mais genérica de uma característica desejada a um produto, e as variabilidades referem-se a comportamentos específicos que são desejados para esta característica. Em outros termos, a *feature* descreve o que se deseja no produto, e as variabilidades descrevem como será esta característica desejada no produto.

Tendo estes conceitos em mente, compreende-se que uma base de *features* permite reuso ao compartilhar componentes pré-projetados e a forma de estruturar os produtos quando se está

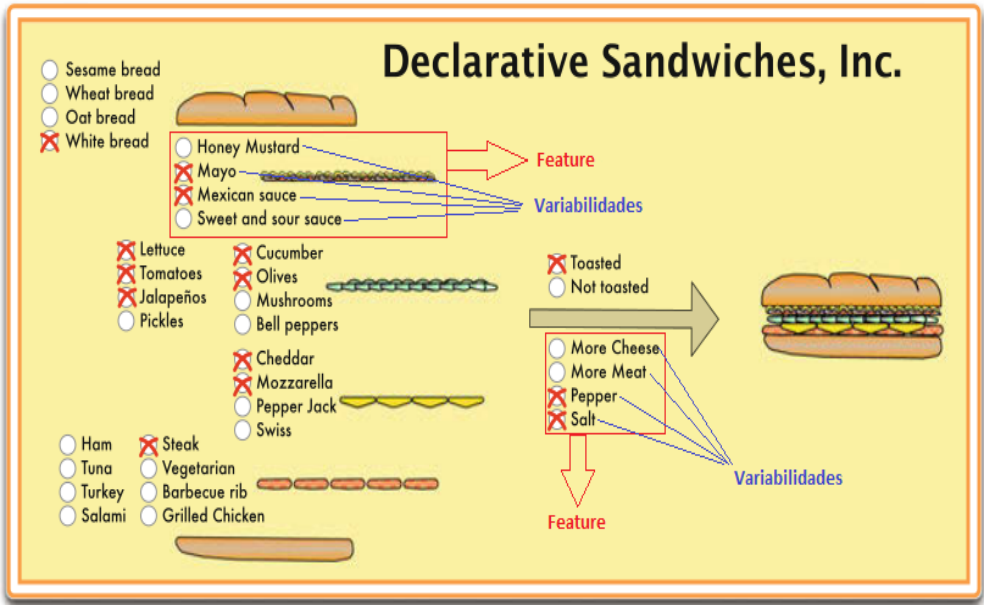

Figura 2.1: Exemplo de linha de produtos [Apel et al. 2013]

derivando produtos da base, e fornece customização em massa ao fornecer variabilidades com comportamentos e características específicos que atendem aos desejos ou às necessidades do cliente.

Desta forma, a comunalidade existente entre sistemas de um mesmo domínio de aplicação e a reutilização em massa fazem as LPSs economicamente viáveis, enquanto que a variabilidade faz a customização em massa possível, de forma que seja econômico atender ao mercado massivo e diferentes necessidades [Böckle, Pohl e Linden 2005].

Para o cliente, o termo "customização em massa" geralmente significa possuir um produto individualizado, enquanto que para as empresas significa investimentos maiores em tecnologias, resultando em um produto com desenvolvimento mais caro e com menores margens de retorno à empresa. A combinação entre customização em massa e presença de uma base comum permite a reutilização desta base tecnológica comum e, ao mesmo tempo, trazer ao mercado produtos para diferentes necessidades, de forma que é possível economizar na produção dos produtos padrão e dos produtos individualizados [Böckle, Pohl e Linden 2005].

2.1.1 Definição de Feature

Existe pouco consenso na literatura acerca da definição do termo *feature*. A definição adotada neste trabalho é que uma *feature* é um conjunto de funcionalidades, características, atributos e comportamentos definidos pelos requisitos do usuário ou cliente [Rommes, Schmid e Linden 2007] [Apel et al. 2013]. A criação das *features* gera vários artefatos, o quais podem ser casos de uso, diagramas e outros modelos, código da aplicação, requisitos funcionais, requisitos não funcionais e outros documentos de projeto que venham a ser criados [Clements e Northrop 2001].

As *features* e os artefatos geralmente são desenvolvidos para um contexto de aplicação específico. Uma das motivações para o surgimento e desenvolvimento do paradigma de linha de produtos de *software* foi a percepção de que os requisitos dos clientes raramente são os mesmos. Aproveitando-se desta informação, uma LPS é projetada de forma a fornecer meios para que as comunalidades e as diferenças entre artefatos e *features* sejam identificados e considerados no projeto da LPS, de forma que as comunalidades possam ser reutilizadas várias vezes, e as diferenças sejam desenvolvidas como possíveis pontos de variação, ou, em outros termos, pontos onde exista variabilidade de comportamentos para uma determinada *feature* [Böckle, Pohl e Linden 2005].

Desta forma, assim como ocorre na engenharia de requisitos, onde se necessita identificar e descrever as funcionalidades corretas que deverão ser implementadas, em uma LPS a chave para o sucesso é identificar e descrever as funcionalidades corretas para a implementação de artefatos reutilizáveis, identificando os requisitos que possam gerar possíveis pontos de variabilidade em produtos futuros. Os requisitos, comunalidades e variabilidades identificados devem satisfazer as principais metas de negócios da empresa [Kang, Park e Sugumaran 2009].

Tipos de Feature

Saber o tipo das *features* é importante para a implementação porque, dado um tipo de *feature*, um certo mecanismo de implementação de variabilidades pode ser recomendado ou não [Anastasopoulos e Gacek 2001].

Os tipos de *features* são os seguintes [Anastasopoulos e Gacek 2001]:

- Obrigatório: quando a *feature* deve estar presente em todos os produtos do domínio;

- Opcional: quando a *feature* é um complemento independente que pode ser incluído ou não;
- Alternativo: quando uma *feature* é incluída de forma a substituir outra feature;
- Mutualmente Inclusivo: quando uma *feature* somente pode ser incluída se outra *feature* específica for incluída também e vice-versa;
- Mutualmente Exclusivo: quando uma *feature* somente pode ser incluída se outra *feature* específica não for incluída e vice-versa.

2.1.2 Definição de Variabilidade

A variabilidade para a LPS é a habilidade de mudar ou customizar um sistema [Bosch, Gurp e Svahnberg 2000]. Variabilidade pode ser as diferentes formas que uma funcionalidade pode ser implementada, pode ser as diferentes formas de realizar a documentação do sistema baseado nas funcionalidades que são incluídas no produto de *software*, como também pode ser as diferenças existentes entre sistemas que possuem um mesmo propósito, de forma que estas diferenças podem ir desde formas diferentes de realizar a implementação de uma funcionalidade como na existência de requisitos que podem existir em sistema e não em outro. Além disso, aspectos comuns entre sistemas também são entendidos como uma forma de variabilidade.

Construir um sistema de forma a facilitar o gerenciamente da variabilidade implica que mudanças futuras serão facilitadas. É possível indentificar antecipadamente alguns tipos de variabilidade e construir o sistema de forma que este facilite o desenvolvimento destas variabilidades. Porém, infelizmente sempre há alguns tipos de variabilidade as quais não é possível identificar antecipadamente [Bosch, Gurp e Svahnberg 2000].

Tipos de Variabilidade

Quando se manipula variabilidades em uma LPS, é necessário distinguir três tipos principais:

- Comunalidade: uma característica (funcional ou não funcional) pode ser comum a todos os produtos da linha de produtos. Esta característica, chamada de comunalidade, é

implementada como parte da base [Rommes, Schmid e Linden 2007].

- Variabilidade: a característica pode ser comum para alguns produtos, mas não para todos. Isto inclui formas diferentes de desenvolver uma característica. Isto deve, então, ser explicitamente modelado como uma possível variabilidade e deve ser implementado de uma forma que permita habilitá-la somente nos produtos selecionados [Rommes, Schmid e Linden 2007].

- Específica de Produto: uma característica pode ser parte de somente um produto. Tais especialidades, muitas vezes não são exigidos pelo mercado em si, mas são devido à preocupações de clientes individuais. Mesmo que estas variabilidades não sejam integradas à base, a base deve ser capaz de suportá-las [Rommes, Schmid e Linden 2007].

Durante o ciclo de vida da linha de produtos, uma variabilidade específica pode mudar seu tipo. Por exemplo, uma caracterśitica específica de um produto pode se tornar uma variabilidade. Por outro lado, uma comunalidade pode se tornar uma variabilidade também [Rommes, Schmid e Linden 2007].

Enquanto comunalidades e variabilidaes são manipuladas principalmente na engenharia de domínio, partes específicas de produtos são gerenciadas exclusivamente na engenharia de aplicação [Rommes, Schmid e Linden 2007]. Isso é mostrado na Figura 2.2.

O gerenciamento de variabilidades é uma preocupação da engenharia de linha de produtos de *software*. Isto cobre todo o ciclo de vida da LPS. Inicia com os primeiros passos do escopo, cobrindo todo o caminho até implementação e testes e finalmente indo para a evolução. Intrinsicamente, a variabilidade é relevante para todos os artefatos presentes no desenvolvimento de *software* [Rommes, Schmid e Linden 2007].

2.1.3 Locais de Ocorrência de Variabilidade

No uso comum da linguagem, o termo variabilidade refere-se à habilidade ou tendência à mudança. O tipo da variabilidade de interesse não ocorre por acaso, mas é trazida para um propósito [Böckle, Pohl e Linden 2005]. Quando um item ou uma propriedade de um item do mundo real possui variabilidade, é chamado de *sujeito de variabilidade*. Quando há uma instanciação de um sujeito de variabilidade, é chamado de *objeto de variabilidade*

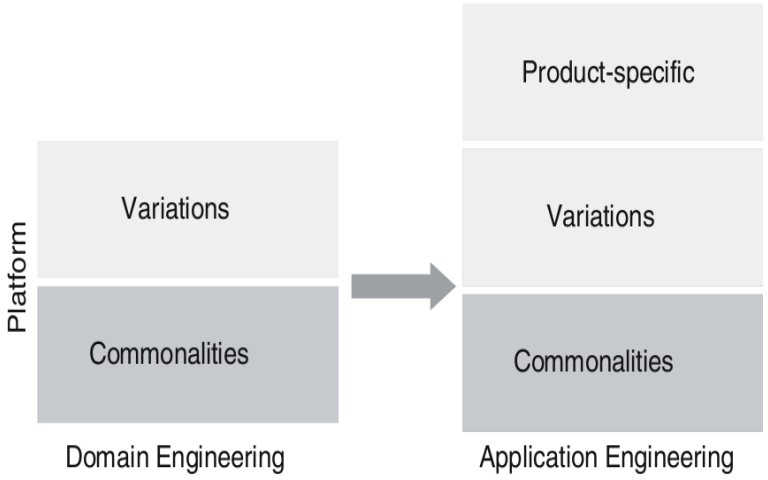

Figura 2.2: Relação entre diferentes tipos de variabilidade e os processos de engenharia de domínio (esquerda) e engenharia de aplicação (direita) [Rommes, Schmid e Linden 2007]

[Böckle, Pohl e Linden 2005]. Os pontos de variação são os locais onde a variabilidade irá ocorrer.

A variabilidade pode ser externa, onde a variabilidade das *features* de domínio são visíveis ao cliente e ele pode escolher as variações que deseja, e também pode ser interna, onde a variabilidade dos artefatos e *features* de domínio não são visíveis ao cliente e cabem aos desenvolvedores determinar quando serão utilizadas [Böckle, Pohl e Linden 2005]. Na Figura 2.3 observa-se uma pirâmide que ilustra os diferentes níveis de variabilidade, fazendo relação com as variabilidades interna e externa.

Como observado na Figura 2.3, a variabilidade pode existir nos requisitos, na arquitetura, nas implementações e nos testes. Isto é, em todos estes locais é possível ter os chamados pontos de variação. Isto ocorre porque ao considerar as comunalidades e as variabilidades no projeto da LPS, é possível obter vários níveis de abstração de variabilidade em relação aos artefatos e *features* compartilhados (por exemplo: variabilidade de requisitos, projeto, arquitetura, testes, implementação, compilação, documentação, etc) [Böckle, Pohl e Linden 2005].

Outro fato observado na Figura 2.3 é que quanto mais abaixo se está na pirâmide (menor nível de abstração), menos poder de escolha o cliente possui no processo de geração do produto. Esta é a variabilidade interna, pois os desenvolvedores são quem determinam quando e onde utilizar determinadas variações.

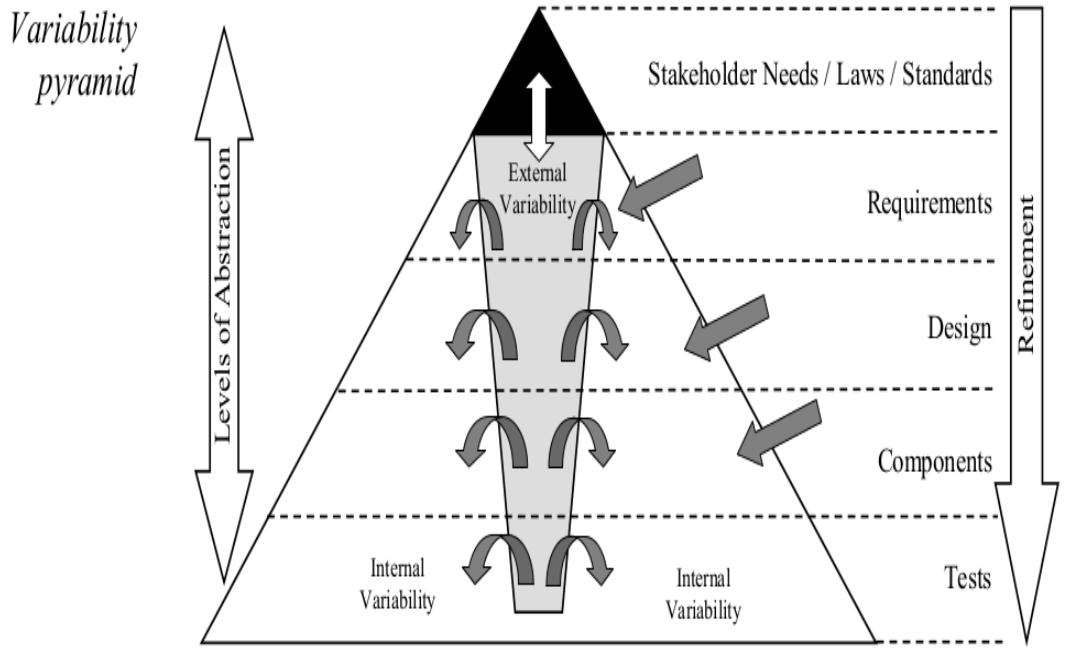

Figura 2.3: Dimensão da variabilidade nos diferentes níveis de abstração [Böckle, Pohl e Linden 2005]

Em contrapartida, a variabilidade externa refere-se às variabilidades visíveis ao clientes, isto é, as variações que ele pode escolher. Quanto mais acima na pirâmide (maior nível de abstração), mais poder de escolha o cliente possui no processo de geração do produto. É importante observar que a variabilidade externa impacta diretamente na definição das variabilidades internas. No topo da pirâmide são definidos os requisitos imutáveis, que são os padrões e leis a serem seguidos pela empresa e que devem estar presentes no produto.

Por fim, os desenvolvedores devem relacionar a variabilidade definida no modelo de *features* com os artefatos especificados em outros modelos, documentos textuais e código [Böckle, Pohl e Linden 2005].

2.1.4 Processos do Desenvolvimento da Linha de Produtos de Software

A distinção entre o reuso na engenharia de linha de produtos de *software* e entre algumas das várias outras técnicas de reuso, tais como Geradores de Programas, Aplicações Configuráveis, Empacotamento de sistemas legados, Padrões de Arquitetura, *Frameworks*, Bibliotecas de Programas e Sistemas orientados a serviços, por exemplo, é que os vários artefatos e *features*

componentes da LPS contêm variabilidade explícita [Rommes, Schmid e Linden 2007].

Na visão do cliente, variabilidade explícita significa que ele tem acesso e pode escolher as variações que deseja para o seu produto. Na visão do desenvolvedor, significa programar formas diferentes para realizar uma mesma tarefa. E na visão do projetista, significa identificar o que pode variar, porque varia e porque está inserindo determinada variabilidade no projeto.

Desta forma, é necessário diferenciar o desenvolvimento de *software* **com reuso** do desenvolvimento de *software* **para reuso**. No desenvolvimento com reuso, os componentes ou artefatos são reutilizados de forma ocasional e este reuso não é planejado sistematicamente. No desenvolvimento para reuso, tudo o que pode ser reutilizado é identificado previamente, para então ser desenvolvido de forma que posteriormente possa ser reutilizado [Sommerville 2008].

Considerando isto, uma LPS possui estas duas interpretações de reuso, as quais são organizadas de uma forma que sistematiza o processo como um todo pois, na LPS, o reuso será feito em larga escala. As *features* que podem ser reutilizadas são identificadas previamente para que possam ser efetivamente reutilizadas posteriormente. Esta forma sistemática de construção da engenharia de LPS garante a organização e gerenciamento do reuso, de modo que qualquer modificação que necessite ser feita em uma *feature* será replicada às *features* correspondentes em todos os produtos que a possuem, diferentemente do que ocorre em outras técnicas de reuso [Rommes, Schmid e Linden 2007].

Por conseguinte, o paradigma de engenharia de linha de produtos de *software* possui dois processos distintos para análise e geração de produtos com comunalidades: a *engenharia de domínio*, onde o desenvolvimento é **para o reuso**, e a *engenharia de aplicação*, onde o desenvolvimento é **com reuso** [Rommes, Schmid e Linden 2007]. Assim, as etapas de análise de domínio da engenharia de domínio e de geração de produtos da engenharia de aplicação podem ser estudadas e realizadas de forma distinta e sequencial. A Figura 2.4 ilustra as diferenças e semelhanças entre os dois processos.

O primeiro processo, a *engenharia de domínio*, uma análise de domínio é realizada objetivando desenvolver e estrututurar uma base de artefatos comuns. A base de artefatos desenvolvida nesta etapa é utilizada para a derivação de produtos individuais no processo posterior [Rommes, Schmid e Linden 2007].

A infraestrutura da linha de produtos de *software* não compreende somente o código do

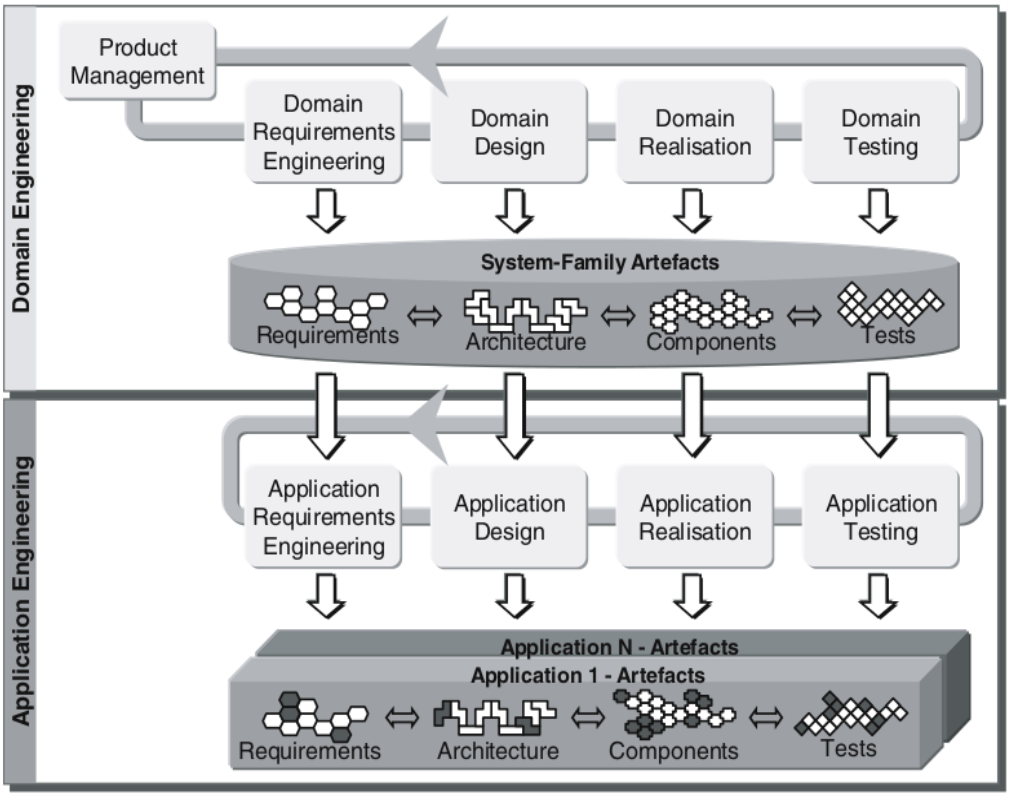

Figura 2.4: Modelos de engenharia de aplicação e engenharia de domínio [Rommes, Schmid e Linden 2007]

produto, mas todos os artefatos que são relevantes através do ciclo de vida do desenvolvimento de *software*. Os artefatos abrangem todos os requisitos, a arquitetura, implementação e testes, podendo existir variabilidade em todos estes artefatos [Rommes, Schmid e Linden 2007].

Nesta etapa, os componentes do domínio são descritos por casos de uso e por modelos de *features*. Como cada conjunto único formado por artefatos relacionados, tais como os requisitos, implementação, documentação e testes para uma dada funcionalidade é chamado de *feature*, a base de artefatos é chamada também de base de *features*.

O processo seguinte, a *engenharia de aplicação*, é o que efetivamente constrói os produtos utilizando os artefatos da base desenvolvida na engenharia de domínio, tal como pode ser observado na Figura 2.4 na etapa de engenharia de aplicação, onde os produtos derivados da base utilizam apenas alguns dos artefatos da base (os artefatos utilizados são representados como

os pontos preenchidos nas etapas de requisitos, arquitetura, componentes e testes; logo, os não utilizados são os pontos brancos). A engenharia de aplicação é dirigida pela infraestrutura da linha de produtos, o qual contém as funcionalidades básicas necessárias para a geração de um produto [Rommes, Schmid e Linden 2007].

Explorar a variabilidade modelada na base permite a derivação de produtos individualizados, mantendo as ligações corretas entre os artefatos. Quando um novo produto é desenvolvido, um projeto de acompanhamento é criado. Os vários artefatos básicos para o novo produto são instanciados para a iniciação do produto, onde cerca de noventa por cento do novo produto é reutilização de artefatos da base e somente cerca de dez por cento deve ser desenvolvido nos passos seguintes [Rommes, Schmid e Linden 2007].

2.1.5 Tipos de Variabilidade em Componentes de Software

No contexto da LPS, os artefatos de código são os componentes de *software* desenvolvidos para a base de *features*. Um componente, neste contexto, refere-se a um elemento independente que encapsula funcionalidades, e a construção de um produto depende da combinação de uma série de componentes.

A variabilidade nos componentes de *software* pode ser classificada por sua finalidade. Neste contexto, a variabilidade pode ser classificada como positiva, negativa, opcional, alternativa, funcional e de plataforma/ambiente [Sharp 2000].

No tipo positivo funcionalidades são adicionadas, enquanto que no tipo negativo funcionalidades são removidas. No tipo opcional, código é incluído, enquanto que no tipo alternativo o código é substituído. Na variabilidade funcional, a funcionalidade muda. Na variabilidade de plataforma ou ambiente, basicamente, a plataforma ou o ambiente mudam [Sharp 2000].

Estas classificações são utilizadas para cada um dos parâmetros de variação adotados. Por parâmetro de variação entende-se como sendo o que está variando (atributos, métodos, classes, lógica, fluxo, interface e persistência de dados).

A variabilidade de componentes é classificada também por seu efeito. Neste contexto, a variabilidade pode ser classificada como variabilidade de atributos, variabilidade de lógica, variabilidade de fluxo, variabilidade de persistência e variabilidade de interface de componentes [Chang, Her e Kim 2005].

Um atributo é definido como sendo uma entidade abstrata para armazenar valores. Na etapa de implementação, os atributos são realizados em forma de constantes, variáveis e estruturas de dados. A variabilidade de atributos ocorre quando há pontos de variação no conjunto de atributos pertencentes a uma função. Formas comuns de variações são em relação à quantidade diferente de atributos utilizados e os tipos de dados dos atributos. Não é considerado variabilidade de atributos quando dois tipos são compatíveis através do uso de coerção ou outra forma de conversão de tipos, e quando uma variação de um valor estático ocorre [Chang, Her e Kim 2005].

A lógica descreve um algoritmo ou o fluxo procedural de uma função. A variabilidade de lógica ocorre quando há pontos de variação no algoritmo ou no procedimento lógico de uma função, tais como quando há variações no tratamento de exceções e quando há variações nos efeitos colaterais da função. Não é considerado variabilidade de lógica em declarações simples tais como *if-then-else* e *switch-case* [Chang, Her e Kim 2005].

O fluxo descreve uma sequencia de invocações de métodos que são executados em uma função. A variabilidade de fluxo ocorre quando há pontos de variação na sequencia dos métodos invocados por uma função, tais como a mudança de ordem de execução ou a quantidade de métodos utilizados [Chang, Her e Kim 2005].

A persistência descreve os valores armazenados dos atributos de um componente em um local de armazenamento permanente, de forma que o estado do componente possa sobreviver entre as sessões do sistema. Geralmente, um componente possui várias classes, e estas classes possuem atributos que podem ser persistidos. Estes atributos podem ser armazenados em um local de armazenamento secundário, tais como arquivos ou disco rígido e tabelas relacionais de bases de dados. A variabilidade de persistência ocorre quando há pontos de variação no esquema físico ou na representação dos atributos que podem ser persistidos em um local de armazenamento secundário [Chang, Her e Kim 2005].

Em interfaces de componentes há uma descrição de um protocolo ou contrato para componentes com os quais um componente cliente ou um programa interagem. Uma interface consiste de uma ou mais assinaturas de métodos, isto é, somente uma definição ou abstração do que os métodos que implementam as interfaces realmente fazem. A interface é um elemento essencial de componentes de *software* porque uma interface é definida separadamente dos componentes. A variabilidade de interfaces ocorre quando há pontos de variação nas assinaturas de méto-

dos de uma interface, ocorrendo por exemplo quando é necessário seguir uma convenção para os nomes de métodos ou quando um sistema legado impõe que certos nomes sejam utilizados [Chang, Her e Kim 2005].

Escopo de Variação

Os três tipos básicos de escopo de variação são o binário, o de seleção e o escopo aberto [Chang, Her e Kim 2005].

O escopo binário de um ponto de variação possui apenas duas possíveis variantes previamente conhecidas. O ponto de variação deste escopo pode conter somente uma das duas variantes [Chang, Her e Kim 2005].

O escopo de seleção de um ponto de variação possui três ou mais possíveis variantes previamente conhecidas. O ponto de variação deste escopo pode conter somente uma das várias variantes [Chang, Her e Kim 2005].

O escopo aberto de um ponto de variação possui qualquer número de variantes previamente conhecidas e pode possuir variantes adicionais ainda não conhecidas, as quais podem ser adicionados depois. O ponto de variação deste escopo pode conter qualquer quantidade necessária do conjunto das variantes [Chang, Her e Kim 2005].

Tempo de Vinculação

O chamado **tempo de vinculação** de um mecanismo refere-se tanto ao momento ao qual uma variante foi atribuída a um ponto de variação [Bosch, Svahnberg e Gurp 2005], quanto ao último momento durante o desenvolvimento que uma variação pode ser ligada a um ponto de variação, sendo que o tempo de vinculação da variabilidade depende do mecanismo usado para implementar a variabilidade [Bosch, Deelstra e Sinnema 2009] [Amin, Mahmood e Oxley 2011]. Desta forma, os tempos de vinculação no nível de implementação são os seguintes [Anastasopoulos e Gacek 2001]:

- Tempo de Compilação: a variabilidade é resolvida antes ou durante a compilação do programa;
- Tempo de Ligação: a variabilidade é resolvida durante a ligação de módulos ou de bibliotecas;

- Tempo de Carregamento: a variabilidade é resolvida depois da compilação, quando o programa é iniciado;
- Tempo de Execução: a variabilidade é resolvida durante o tempo de execução do programar;
- Tempo de Atualização ou Tempo de Pós-Execução: a variabilidade é resolvida durante as atualizações do programa ou depois de sua execução.

Parâmetros de Variação

Depois de identificar os tipos de variabilidade e os tempos de vinculação é necessário definir quais são os parâmetros de variação [Sharp 2000], isto é, identificar exatamente o que está variando, e os pontos de variação, que são onde as variações ocorrem [Anastasopoulos e Gacek 2001]. Identificar os parâmetros de variação refere-se ao processo de identificar os tipos de *features*, variabilidade, escopo, unidades de código e o tempo de vinculação.

Os principais parâmetros de variação estão nas interfaces e nas implementações correspondentes às interfaces [Sharp 2000]. Outro parâmetro de variação possível é a inicialização de módulos [Anastasopoulos e Gacek 2001]. Nestes contextos ocorrem os tipos de variabilidade (finalidade e efeito) possíveis, o escopo de variabilidade possível, os tipos de *features* possíves e os tempos de vinculação possíveis, os quais foram explicados anteriormente. Desta forma, dada uma variabilidade, é possível identificar no código fonte o que está variando e o local onde está variando [Sharp 2000].

2.1.6 Representação De Modelos de Features

Na etapa de engenharia de domínio, a análise de domínio descreve componentes e requisitos do domínio através de casos de uso e por modelos de *features*.

Os modelos de *features* são representados através de relacionamentos entre *features*. Várias formas de representação existem, porém a mais utilizada e intuitiva é a baseada em notação gráfica.

Até o momento, no decorrer do texto, foram apresentadas várias definições ou interpretações do termo “variabilidade”. A notação gráfica demonstrada na Figura 2.5 objetiva simplificar o

conceito de variabilidade para que ela possa ser representada graficamente de uma maneira mais simples. Um exemplo baseado nesta notação gráfica pode ser vista na Subseção 3.1.9.

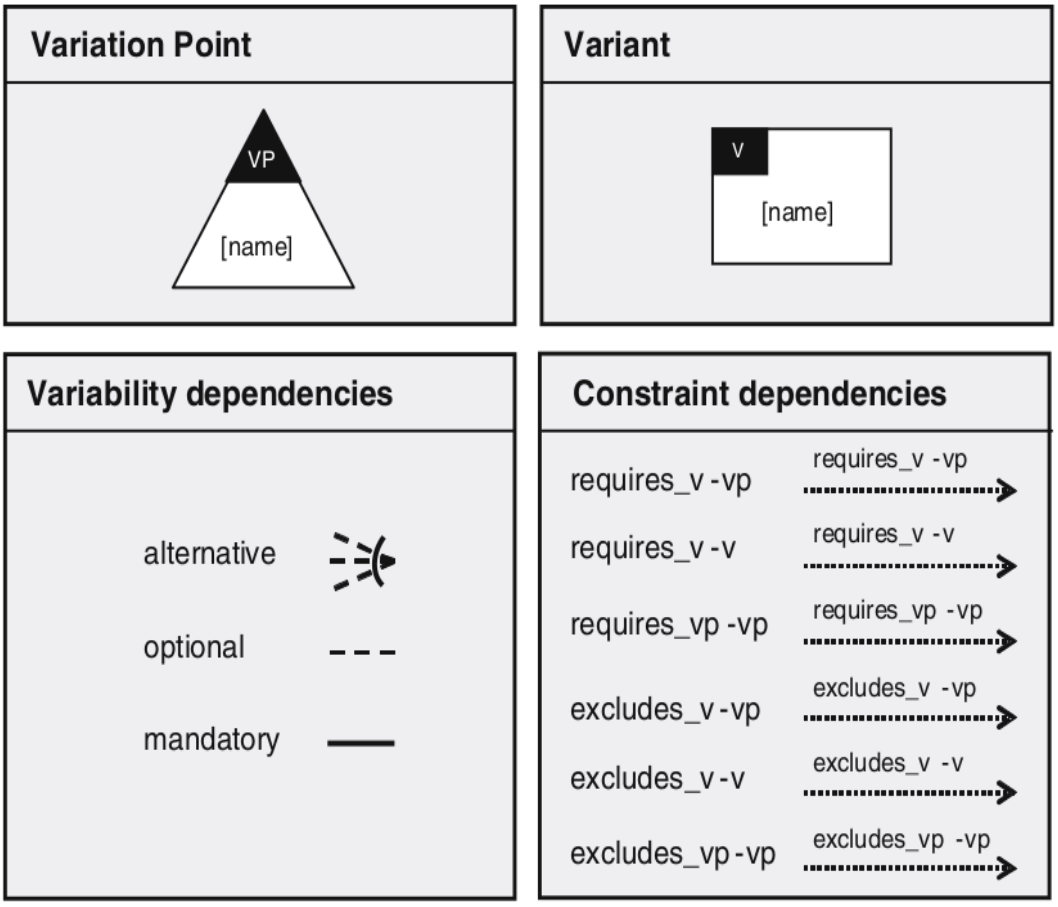

Figura 2.5: Notação gráfica para modelos de *features* [Rommes, Schmid e Linden 2007]

Os elementos observados na Figura 2.5 possuem o seguinte significado [Rommes, Schmid e Linden 2007]:

- Ponto de Variação: descreve onde estão as diferenças existentes nos sistemas finais;
- Variações: são as diferentes possibilidades existentes para satisfazer um ponto de variação;
- Dependências de Variabilidade: é utilizado para denotar as diferentes escolhas que são possíveis para preencher um ponto de variação. A notação pode incluir cardinalidade indicando quantas variações podem ser selecionadas ao mesmo tempo;

- Restrições de Dependências: descrevem restrições entre certas seleções de variantes. Existem duas formas de restrições. Uma é a exigência, onde a seleção de uma variante específica pode exigir a seleção de outra variante. A outra é exclusão, onde a seleção de uma variante em específico pode proribir a seleção de outra variante.

Os pontos de variação possuem dependências de variabilidade de *features*. Estas podem ser obrigatórias ou opcionais, sendo que as opcionais podem possuir múltiplas escolhas [Böckle, Pohl e Linden 2005].

As dependências opcionais ocorrem quando uma *feature* pode, mas não necessita ser parte da aplicação da linha de produtos.

As dependências obrigatórias definem que uma *feature* deve ser escolhida para uma aplicação se, e somente se, o ponto de variação associado for parte da aplicação.

As dependências de múltipla escolha agrupam um conjunto de *features* que são relacionadas através das dependências opcionais com um mesmo ponto de variação e definem um alcance para o conjunto de *features* opcionais para serem selecionadas nesse grupo.

2.1.7 Motivações Para o Uso da Abordagem de Linha de Produtos de Software

Uma das consequências do uso do paradigma de linha de produtos de *software* é a diminuição de custos e o tempo de mercado, os quais estão intimamente relacionados, uma vez que essa abordagem caracteriza-se pelo reuso em larga escala. O reuso possui maior custo-benefício do que o desenvolvimento de um produto desde o início em algumas ordens de magnitude. Desta forma, custos de desenvolvimento e o tempo de mercado podem ser dramaticamente reduzidos pela utilização de linha de produtos [Rommes, Schmid e Linden 2007].

A Figura 2.6 relaciona a diferença entre o tempo de mercado da engenharia de linha de produtos de *software* e a metodologia clássica de desenvolvimento, através da qual a análise e o desenvolvimento são para um produto específico.

Mas para que seja possível aproveitar estes benefícios, é necessário um alto investimento inicial em termos de tempo, esforço e recursos financeiros para a construção de artefatos reutilizáveis, adaptação da organização para uma nova abordagem na produção de seus produtos, dentre outras modificações [Rommes, Schmid e Linden 2007].

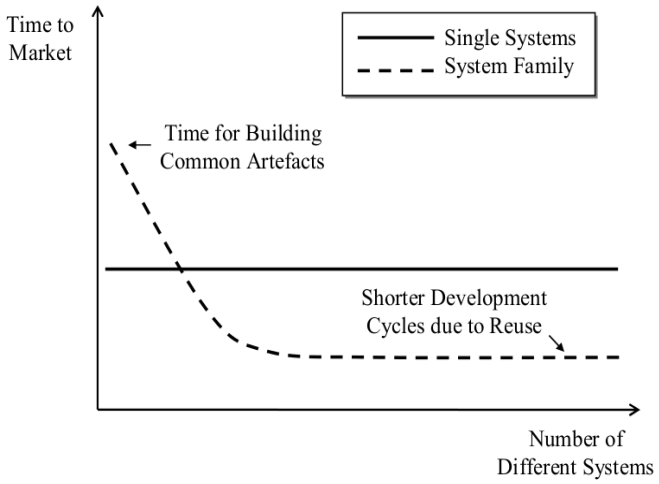

Figura 2.6: Tempo de mercado ao utilizar engenharia de linha de produtos de software [Böckle, Pohl e Linden 2005]

Entretanto, em média a partir do terceiro produto construído utilizando o conceito de linha de produtos, o alto investimento inicial começa a trazer retorno financeiro, igualando-se em relação aos custos de manutenção com o método usual de desenvolvimento, entendendo-se por método usual aquele em que o reuso, de modo geral, não é sistematicamente planejado. Depois do terceiro produto, a linha de produtos é mais lucrativa, menos custosa para a manutenção e a geração de novos produtos é muito mais rápida e barata em relação ao método clássico de desenvolvimento [Rommes, Schmid e Linden 2007].

A Figura 2.7 ilustra de forma comparativa os recursos referentes a tempo e recursos financeiros necessários para cada uma das abordagens.

Observando a Figura 2.7 é possível perceber que na engenharia de *software* clássica, onde a análise é feita para produtos específicos, caso estivessem sendo desenvolvidos quatro diferentes produtos, significaria ter quatro diferentes projetos.

Utilizando o paradigma de LPS, haveria um processo de engenharia de domínio, no qual haveria somente um projeto que criaria uma base com todas as partes comuns entre os produtos, para que os quatro produtos fossem adaptados ou derivados dessa base no processo de engenharia de aplicação [Böckle, Pohl e Linden 2005].

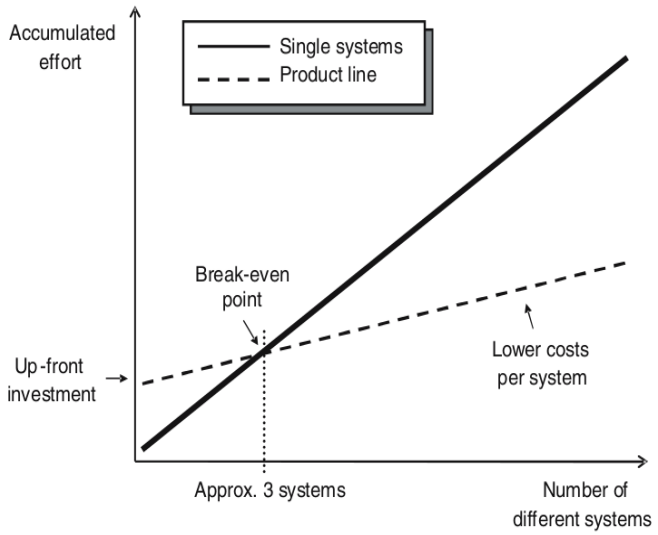

Figura 2.7: Economia de recursos com o uso da engenharia de linha de produtos de software [Rommes, Schmid e Linden 2007]

A base também determina características específicas que distinguem os quatro diferentes produtos como também os desejos do cliente por produtos mais individualizados. Por exemplo, se um cliente deseja alguma funcionalidade ou alguma customização do *software* que não esteja presente na base, isto pode ser desenvolvido para ser adicionado ao produto final deste cliente, sendo que esta funcionalidade ou customização poderá ser incluída na base. Resumidamente, criar a base significa preparar-se para a customização em massa [Böckle, Pohl e Linden 2005].

A estratégia para a análise de domínio é, primeiramente, focar no que é comum em todos os produtos, para depois focar no que é diferente. Fazendo isso, consegue-se flexibilidade, tornando possível reutilizar os artefatos comuns em várias aplicações e provendo a customização em massa. A flexibilidade é uma pré-condição para a customização em massa. Isso também significa que é possível predefinir quais opções possíveis para o produto devem ser desenvolvidas [Böckle, Pohl e Linden 2005].

Desenvolver a aplicação utilizando bases comuns significa planejá-las proativamente para o reuso, construir partes reutilizáveis e reusar o que foi construído para ser reutilizado [Böckle, Pohl e Linden 2005].

2.1.8 Vantagens e Desvantagens do Uso da Abordagem de Linha de Produtos de Software

Com a utilização de LPS, muitas empresas relataram reduções no número de defeitos em seus produtos, corte de custos e de tempo de produção em um fator de 10 ou mais. No entanto, os custos em termos de tempo, de esforço e econômicos para iniciar o desenvolvimento de uma linha de produtos de *software* são altos [Schmid e Verlage 2002].

As principais vantagens do uso de linha de produtos de *software* são quanto ao reuso de artefatos, evitando o trabalho de refazer as mesmas tarefas para diferentes produtos e permitindo a produção em larga escala e customização em massa, a facilidade de manutenção obtida por manter uma base comum para vários produtos, onde uma alteração na base facilitará a replicação da alteração para todos os produtos derivados desta base e garantirá a alta qualidade dos produtos ao diminuir o risco de problemas [Clements e Northrop 2001].

Essa replicação em cadeia de alterações pode ocorrer tanto de forma automática, manual ou programada, dependendo se a alteração é crítica ou necessária e dependendo também do tipo de sistema.

Isto implica na redução nos custos de desenvolvimento, manutenção e geração de novos produtos, mais eficiência no uso dos recursos humanos, aumento da satisfação do cliente, garantido a presença de mercado, reduzindo o tempo de mercado e fazendo com que o retorno a longo prazo seja muito grande. Também são observados ganhos nas etapas de desenvolvimento da linha de produtos de *software*, como por exemplo nas etapas de testes, planejamento e definição de processos [Clements e Northrop 2001].

Suas desvantagens são quanto ao grande investimento inicial em termos econômicos, de tempo e de esforço (pesquisa, conhecimento, trabalho) necessários para que a base da linha de produtos de *software* seja estabelecida, uma vez que o retorno financeiro somente ocorrerá a longo prazo [Clements e Northrop 2001].

2.2 Mecanismos de Implementação de Variabilidades de Software

Existem diversos mecanismos que possibilitam a realização de variabilidades de *software*. Os mecanismos podem ser voltados às linguagens de programação e voltados às ferramentas de

gerenciamento de *features*.

Os mecanismos são voltados à linguagem de programação quando utilizam as características presentes em alguma linguagem de programação para implementar as *fetaures*, impactando posteriormente na forma de realizar as derivações de produtos. Nesta abordagem, tanto a implementação das *features* quanto o gerenciamento da variabilidade localizam-se no código fonte. Desta forma, o desenvolvedor possui uma compreensão facilitada sobre a linha de produto e sua implementação como um todo. Entretanto, dependendo da mecanismo de implementação utilizado, os limites e o gerenciamento das *features* tendem a desaparecer no código [Apel et al. 2013]. Alguns exemplos de mecanismos voltados às linguagens de programação são os seguintes:

- **Agregação/Delegação:** é um mecanismo orientado a objeto que teoricamente habilita os objetos a suportar qualquer funcionalidade ao encaminhar solicitações que ele normalmente não podem satisfazer para objetos chamados de *objetos de delegação*, os quais fornecem os serviços ou funcionalidades requisitados [Anastasopoulos e Gacek 2001];

- **Herança:** pode ser utilizada para a funcionalidade básica ou comum para classes hierarquicamente superiores e as especializações para classes hierarquicamente inferiores [Anastasopoulos e Gacek 2001];

- **Parametrização:** representa o *software* reusável como uma biblioteca de componentes parametrizados. O comportamento do componente é determinado pela atribuição de diferentes valores dos parâmetros [Anastasopoulos e Gacek 2001];

- **Sobrecarga:** reutiliza um nome existente, mas de forma a fazê-lo operar em diferentes tipos. Este nome ou símbolo pode ser atribuído para funções, procedimento e operadores [Anastasopoulos e Gacek 2001];

- **Reflexão:** é a habilidade de um programa de manipular os dados como algo que representa o estado do programa durante sua própria execução [Anastasopoulos e Gacek 2001];

- **Padrões de Projeto:** pode ser explorado no contexto de uma linha de produto uma vez que muitos deles identificam aspectos do sistema que podem variar e fornecem soluções

para gerenciar a variação. Mecanismos orientados à objeto juntamente com parametrização são repetidamente empregados para a implementação de padrões de projeto. O código correspondente a um padrão de projeto é, em muitos casos, entrelaçado com o resto do código e espalhado sobre diversos componentes [Anastasopoulos e Gacek 2001];

- **(Delphi) Propriedades:** propriedades em Delphi são atributos de um objeto. Uma propriedade associa ações específicas com a leitura ou modificação de seus dados. Propriedades fornecem controle sobre o acesso aos atributos de um objeto, e permitem que os atributos sejam computados [Anastasopoulos e Gacek 2001];

- **Programação Orientada a Aspectos:** é um mecanismo que habilita a modularização das chamados *crosscutting concerns*, denominados de aspectos, como também permite a integração dos pontos de junção. Pontos de junção são locais nos sistemas que são afetados por um ou mais *crosscutting concerns*. O processo de integração dos pontos de junção envolve descrever como um *crosscutting concern* afeta o código em um ou mais pontos de junção. A programação orientada a aspecto auxilia o programador ao separar claramente os componentes dos aspectos, provendo mecanismos que tornam isto possível de abstrair e compô-los para produzir o sistema como um todo [Anastasopoulos e Gacek 2001];

- **Carregamento Dinâmico de Classes:** é um padrão onde todas as classes são carregadas em memória no mesmo momento em que se necessita delas [Anastasopoulos e Gacek 2001];

- **Bibliotecas Estáticas:** contêm um conjunto de funções externas que podem ser ligadas a uma aplicação depois que ela foi compilada. O código da aplicação e da biblioteca são carregados no mesmo espaço de memória [Anastasopoulos e Gacek 2001];

- **Bibliotecas de Ligação Dinâmica:** são bibliotecas que são carregadas dentro das aplicações quando necessário, em tempo de execução. Elas podem ser úteis para a seleção de funcionalidades variantes [Anastasopoulos e Gacek 2001];

- **Frames:** provê meios para maximizar a reusabilidade de código através da definição e uso de entidades adaptáveis chamadas *frames*. O objetivo é formar conjuntos hierárquicos de reutilização destas entidades. *Frames* são arquivos fonte equipados com diretivas

semelhantes à pré processadores, os quais permitem os *frames* pai copiarem e adaptarem seus *frames* filhos. No topo de cada conjunto hierárquico de *frames* encontra-se um *frame* de especificação correspondente, o qual coleta código dos *frames* que estão abaixo na hierarquia e fornece, após ser processado, um módulo pronto para compilar ou uma fonte de aplicação. Os desenvolvedores podem selecionar um conjunto de *frames* específico para suas necessidades [Anastasopoulos e Gacek 2001];

- **Genéricos:** refere-se a um mecanismo para parametrizar classes e funções. Uma classe para coleções é um exemplo onde o mecanismo para tipos genéricos pode ser utilizado [Amin, Mahmood e Oxley 2011];

- **Compilação Condicional:** habilita o controle sobre os segmentos de código a serem incluídos ou excluídos de uma compilação de um programa. Diretivas marcam no código os locais que variam [Anastasopoulos e Gacek 2001];

- **Programação Orientada a Feature:** é uma abordagem baseada em composições para construir linha de produtos de *software* onde estas composições endereçam diretamente ao conceito de *features*. A ideia é decompor o projeto e o código do sistema nas *features* que ele fornece. Desta forma, a estrutura de um sistema alinha-se com suas *features*, idealmente, com um módulo ou componente por *feature*. Para esta finalidade, novas construções de linguagem são necessárias para expressar quais partes de um programa contribuem para quais *features* e para encapsular o código das *features* em unidades modulares que podem ser compostas [Apel et al. 2013].

Os mecanismos são voltados às ferramentas de gerenciamento de *features* quando utilizam uma ou mais ferramentas externas para implementar ou representar as *features* em código ou para controlar o processo de derivação de produtos. Esta abordagem favorece uma separação clara entre a implementação das *features* e entre o gerenciamento das variabilidades e *features* e derivação de produtos. Esta separação simplifica a estrutura do código, porém se faz necessário que os desenvolvedores e ferramentas localizem e compreendam múltiplos artefatos em diferentes locais. Ao conseguir realizar isto, a derivação de produtos é facilitada [Apel et al. 2013]. Alguns exemplos de mecanismos voltados às ferramentas de gerenciamento de *features* são os seguintes:

- **Sistema de Controle de Versão:** rastreia mudanças no código fonte e em outros artefatos para facilitar o desenolvimento colaborativo [Apel et al. 2013];

- **Pré Processadores:** é uma ferramenta para manipular o código fonte antes da compilação. Um pré processador popular é o cpp da linguagem C. Ele fornece diretivas para alinhar arquivos, para definir macros, e para remover fragmentos de código baseado em condições definidas pelo usuário [Apel et al. 2013];

- **Visões sobre Código:** dadas as ligações de rastreamento, este é um mecanismo para emular a chamada *separation of concerns* ao fornecer uma visão sobre os códigos fonte. Uma das motivações chave para a *separation of concerns* é que os desenvolvedores podem encontrar todo o código pertencente a uma *feature* em um único lugar, sem ter distrações com outros segmentos de código não relacionados [Apel et al. 2013];

- **Derivação de Produto Integrada:** uma integração de todas as fases de desenvolvimento pode ajudar, especialmente, durante a engenharia de aplicação e derivação de produto, quando se deseja derivar um produto para clientes e requisitos específicos. Uma ferramenta de derivação de produtos integrada pode fornecer suporte de duas formas: checando a validação de seleções de *features* e automatizando a derivação de produto. Se os modelos de *features* não forem integrados, os desenvolvedores estarão sozinhos para descobrir como configurar os parâmetros e como eles interagem, ou em como montar os componentes, ou em quais *plugins* são compatíveis [Apel et al. 2013];

- **Sistemas de Construção:** é uma ferramenta responsável por agendar e executar todas as tarefas relacionadas à construção, as quais podem incluir geradores em execução, compilação de códigos fonte, execução de testes, e criação e cópia de unidades que podem ser entregues. Com um sistema de construção, os desenvolvedores documentam e automatizam o processo de construção. Sistemas de contrução são parte do gerenciamento de configuração de *software* [Apel et al. 2013].

Todos estes mecanismos permitem a implementação e manipulação de código reutilizável e de código que contêm variabilidade.

2.2.1 Classificação dos Mecanismos de Implementação de Variabilidades de Software

Existem outros três tipos básicos abstratos de mecanismos para desenvolver a variabilidade. Estes tipos básicos de mecanismos são abstrações e possuem natureza classificativa. Sua representação refere-se ao nível de implementação. Estas mecanismos são denominadas de **adaptação**, **substituição** e **extensão** [Rommes, Schmid e Linden 2007]. A Figura 2.8 demonstra graficamente os três tipos básicos de mecanismos.

No mecanismo de **adaptação**, há somente uma única implementação disponível para o componente, mas ele oferece *interfaces* para ajustar seu comportamento. O mecanismo de herança é um mecanismo adaptativo. Dado a classe e sua implementação, uma subclasse é introduzida mudando alguns dos comportamentos padrões conforme seja necessário para a aplicação [Rommes, Schmid e Linden 2007].

No mecanismo de **substituição**, muitas implementações de um componente estão disponíveis. Cada implementação adere à especificação do componente tal como descrito na arquitetura. Um gerador de código é um exemplo de mecanismo de substituição. Ao ler algum tipo de especificação de alto nível, como um modelo ou um *script*, ele gera o código solicitado para um certo componente, ou até mesmo para um produto inteiro [Rommes, Schmid e Linden 2007].

O mecanismo de **extensão** necessita de uma arquitetura que ofereça interfaces que permitam a adição de novos componentes. Os componentes adicionados podem ser ou não específicos de um produto. A diferença deste mecanismo com o de substituição é que agora somente *interfaces* genéricas estão disponíveis para adicionar componentes, permitindo que diferentes tipos de componentes sejam adicionados. Com a substituição, a *interface* especifica exatamente o que o componente deve fazer, e somente como isto é feito varia. Além disso, com o mecanismo de extensão inúmeros componentes podem ser adicionados utilizando a mesma *interface*, enquanto que com a substituição um único componente é substituído por outro [Rommes, Schmid e Linden 2007].

Um exemplo de mecanismo de extensão são os *plugins*. Um *plugin* pode ser uma biblioteca de métodos. Em `Ruby`, as `Gems`, que são as bibliotecas `Ruby`, podem ser consideradas como sendo um tipo de mecanismo de extensão.

Os tipos básicos de técnicas aqui citados, por si só, não são capazes de habilitar a variabili-

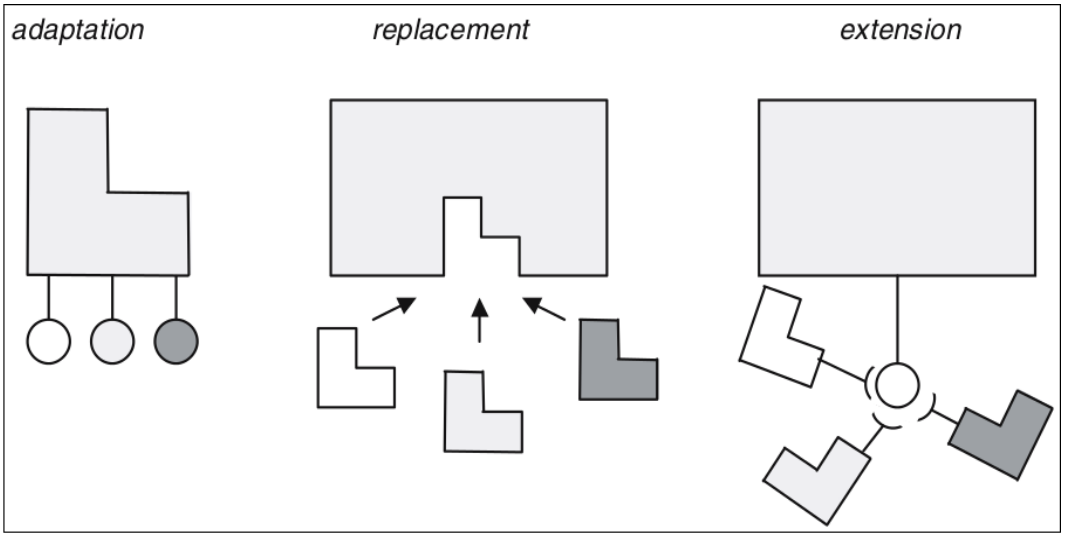

Figura 2.8: Três tipos básicos de técnicas para a realização de variabilidade em uma arquitetura [Rommes, Schmid e Linden 2007]

dade em LPS. Características mais específicas precisam ser levadas em consideração para que a implementação obtenha sucesso, tais como as características da linguagem de programação utilizada (paradigmas suportados, recursos, facilidade de implementação, desempenho, dentro outros), as características das ferramentas que serão utilizadas para trabalhar com os artefatos desenvolvidos e os requisitos que serão trabalhados.

Para que a variabilidade seja efetivamente habilitada, deve-se utilizar os mecanismos de implementação de variabilidades, os quais são aplicados no ponto de variação.

2.2.2 Padrões de Projeto

Alguns padrões de desenvolvimento de *software* evoluíram levando em consideração como a variabilidade pode ser separada e dissociada, e desta forma são considerados bem adequados para implementações de variabilidade [Apel et al. 2013]. Entre estes padrões, estão os bem conhecidos padrões de projeto, dos quais alguns são conhecidos por conterem características da programação orientada a objetos, tais como os padrões *Observer*, *Template Method*, *Strategy* e *Decorator* [Gamma et al. 1995]. Estes quatro padrões de projeto também são conhecidos por serem bem adaptados à implementação de variabilidades [Apel et al. 2013].

Padrões de projeto, em sua definição mais geral, são descrições de objetos colaborativos e classes que são customizadas para resolver um problema de projeto comum em um contexto

particular. Cada padrão de projeto sistematicamente nomeia, explica e avalia um importante e recorrente projento em sistemas orientados a objetos [Gamma et al. 1995].

Os padrões de projeto são classificados de acordo com o tipo de problema para o qual possuem uma solução. As classificações são as seguintes: padrões de criação, padrões estruturais e padrões de comportamento [Gamma et al. 1995].

O padrão *Decorator* descreve um mecanismo baseado em delegação para extender flexivelmente um objeto, adicionando a ele novos comportamento [Apel et al. 2013]. Em outros termos, ele adiciona responsabilidades a um objeto dinamicamente [Gamma et al. 1995].

O padrão *Observer* define uma dependência um-para-muitos entre objetos de forma que quando um objeto muda seu estado, todos os objetos *observers* deste objeto são notificados e atualizados automaticamente [Gamma et al. 1995].

O padrão *Strategy* define uma família de algoritmos, encapsula cada um, e os faz serem intercambiáveis. Dessa forma, este padrão permite que o algoritmo varie independentemente de quem o use [Gamma et al. 1995].

O padrão *Template Method* define o esqueleto de um algoritmo em uma operação, deixando alguns passos para as subclasses. Desta forma, este padrão permite que as subclasses alterem ou redefinam alguns passos do algoritmo sem que a estrutura do algoritmo seja alterada [Gamma et al. 1995].

Uma explicação mais detalhada sobre cada um desses padrões de projeto, envolvendo a análise da utilização de cada padrão como mecanismo para implementar variabilidades, é realizada no Capítulo 4.

2.3 Linguagem de Programação Ruby

A linguagem `Ruby` é relativamente nova no conjunto das linguagens de programação. Seu lançamento oficial ocorreu em 1995 [Ruby-Lang 2015], mas o crescimento da importância da linguagem e da quantidade de usuários da linguagem se deu somente ao longo dos últimos dez anos, principalmente por conta do surgimento do *framework* `Rails` em 2004, o qual automatiza e facilita muitas das tarefas necessárias para o desenvolvimento de sistemas para a *web* utililizando a linguagem `Ruby` [Ruby-On-Rails 2015]. Ao utilizar o *framework* `Rails` no `Ruby`, podemos chamá-lo também de `Ruby on Rails` (RoR).

Seu criador, o japonês Yukihiro Matsumoto, projetou e desenvolveu a linguagem `Ruby` com o objetivo de fazer a programação ser mais rápida e fácil, de forma que o programador ficasse feliz e satisfeito ao programar com a linguagem [Flanagan e Matsumoto 2008]. Matsumoto baseou-se em várias linguagens que ele conhecia para projetar a linguagem `Ruby`, aproveitando-se das principais e melhores características de cada uma. Dentre essas linguagens que inspiraram o criador, podemos destacar Python, Perl, Smalltalk, Eiffel, Ada e Lisp, sendo que `Ruby` assemelha-se principalmente à Python, com sua gramática semelhante às liguagens C/C++ e Java [Flanagan e Matsumoto 2008].

`Ruby` é uma linguagem de programação, mas foi concebida com muitas características de uma linguagem de *script*. É interpretado e suporta vários paradigmas, tais como a programação funcional, orientação à objetos, paradigma imperativo e reflexivo. O criador desejava uma linguagem com mais recursos que Perl e ao mesmo tempo que tivesse mais orientação à objetos que Python, de forma que a programação funcional e a programação imperativa ficassem em equilíbrio [Ruby-Lang 2015].

`Ruby` possui tipagem de dados forte e dinâmica. Na forma mais simplificada possível, tipagem de dados forte significa que as operações dependem do tipo das variáveis para serem executadas com sucesso, e tipagem dinâmica significa que os tipos das variáveis podem ser alterados durante a execução do programa [Flanagan e Matsumoto 2008].

`Ruby` é uma linguagem que contempla o paradigma orientado a objetos. Tudo é tratado como objeto em `Ruby`, desde um valor como o inteiro "3", uma `string` qualquer e até mesmo o valor `nil`, que é utilizado para representar o valor nulo, ou em outros termos, a falta de valor. Todos os dados possuem suas próprias propriedades e ações, sendo que a orientação à objetos define as propriedades como variáveis de instância e as ações como métodos. Não existem tipos primitivos, como observado em outras linguagens, todo valor ou informação é um objeto, sendo que os objetos são instâncias de classes [Flanagan e Matsumoto 2008]. Todas as regras que se aplicam aos objetos se aplicam a tudo em `Ruby`, objetivando facilitar o uso da linguagem.

"`Ruby` é visto como uma linguagem flexível, uma vez que permite aos seus utilizadores alterar partes da Linguagem. Partes essenciais do `Ruby` podem ser removidas ou redefinidas à vontade. Partes existentes podem ser acrescentadas. O `Ruby` tenta não restringir o programador" [Ruby-Lang 2015].

A linguagem possui um recurso denominado `closure`, que refere-se a um objeto que é tanto uma função invocável quanto variável de ligação para esta função. O `closure` é subdividido em `procs` e `lambdas`, onde os `procs` são objetos de blocos e comportam-se como blocos, enquanto que os `lambdas` são objetos de blocos mas possuem um comportamento levemente diferente e se comportam mais como um método do que como um bloco. Os blocos, por sua vez, são pedaços executáveis de código e podem possuir parâmetros e podem ser passados como parâmetros. Diferentemente dos métodos, os blocos não possuem nomes e somente podem ser invocados indiretamente através de um método de iteração [Flanagan e Matsumoto 2008].

A linguagem possui o conceito de módulos, os quais são coleções nomeadas de métodos, constantes e variáveis de classes. Módulos são similares a classes, mas ao contrário destas, não podem ser instanciados ou possuir classes derivadas, não possuindo assim um hierarquia de módulos de herança. Os módulos podem ser usados como `namespaces` e como `mixins` [Flanagan e Matsumoto 2008].

Os `namespaces` são delimitadores abstratos que fornecem um contexto para um conjunto de identificadores, termos técnicos e nomes armazenados por ele, permitindo a desambiguação de itens que possuem o mesmo nome mas estão em *namespaces* diferentes. O significado de um nome pode variar de acordo com o `namespace` a qual pertence. Eles são utilizados quando a programação orientada à objetos não é necessária e se deseja agrupar métodos relacionados [Ruby-Lang 2015].

Com os `mixins`, quando uma classe inclui um `mixin`, a classe implementa a `interface` caso haja e inclui todos os atributos e métodos, ao invés de herdá-los, evitando os problemas causados pela herança múltipla. `Ruby` suporta somente herança simples, os `mixins` então possibilitam uma forma de emular heranças múltiplas em um formato mais simples e também que a aplicação possa ser modularizada [Flanagan e Matsumoto 2008].

A modularização pode ser entendida como uma separação de funcionalidades, que podem ser ativadas ou não e podem ser modificadas para gerar uma forma diferente de cumprir o mesmo objetivo.

`Ruby`, como já citado anteriormente, é também uma linguagem reflexiva. A reflexão, também chamada de introspecção, significa simplesmente que um programa pode examinar seu

estado e sua estrutura [Flanagan e Matsumoto 2008].

A reflexão e o conceito de blocos presentes no `Ruby` fazem dele uma linguagem ideal para a metaprogramação. Este paradigma possui a função de criar programas ou *frameworks* que por sua vez criam ou manipular outros programas (ou a si próprios) assim como seus dados, através de uma linguagem denominada metalinguagem. Quando uma linguagem de programação possui a habilidade de ser sua própria metalinguagem, ela é dita reflexiva. Esta característica ajuda o programador a escrever programas de forma a criar e modificar métodos dinamicamente, isto é, em tempo de execução [Flanagan e Matsumoto 2008].

`Ruby` oferece ferramentas em seu ecossistema que suportam a programação orientada a testes, do inglês *Test Driven Development* (TDD) e programação orientada ao comportamento, do inglês *Behavior Driven Development* (BDD), as quais podem ser muito úteis para evitar vários tipos de erros ao codificar [Ruby-Lang 2015]. Ao programar e testar a aplicação ao mesmo tempo, evita-se a geração e propagação de erros e o retrabalho para a correção de erros, além de que é possível gerar testes para garantir que o produto atenda aos requisitos do usuário.

O BDD é uma prática de desenvolvimento de *software* que trabalha com uma iteração pequena para o *feedback*, onde se aplica consistentemente o TDD para toda nova *feature* que está sendo explorada e desenvolvida.

O TDD é baseado em três passos básicos e em ciclos de repetições destes passos [Caelum 2016]. No primeiro passo é escrito o primeiro teste antes mesmo da lógica da aplicação existir, sendo que cada funcionalidade deve iniciar com a criação de um teste. Este teste precisa inevitavelmente falhar porque ele é escrito antes da funcionalidade a ser implementada, pois se ele falha, então a funcionalidade ou melhoria proposta é óbvia [Caelum 2016]. Ao escrever o teste antes da lógica, o TDD torna o desenvolvedor focado nos requisitos antes do código, que é uma sutil mas importante diferença. Isto aumenta a confiança que se está testando a coisa certa, e que o teste somente irá passar nos casos intencionados. Por outro lado, não exatamente garante confiança total.

O segundo passo é o ponto em que a lógica da aplicação está pronta para que o teste previamente criado passe. O importante é que o código escrito deve ser construído somente para passar no teste. Nenhuma funcionalidade (muito menos não testada) deve ser predita ou permitida em qualquer ponto. Esta lógica deve ser desenvolvida da forma mais simples possível

eliminando complexidades desnecessárias fazendo com que a evolução do código ocorra de forma segura [Caelum 2016].

O último passo é relativo à refatoração, isto é, a melhoria do código, limpando o código conforme necessário. Neste ponto são removidas duplicações, múltiplas responsabilidades e o código fica cada vez mais próximo de sua versão final. Ao final deste passo, o ciclo reinicia, retornando ao primeiro passo e escrevendo um novo teste para a funcionalidade, sendo que cada funcionalidade possui um ciclo de desenvolvimento de testes [Caelum 2016].

Desta forma, o fluxo básico para seguir o TDD é o seguinte: escrever um caso de teste e o vê-lo falhar; escrever o código que implementa o caso de teste, executar o teste e vê-lo passar.

Um dos recursos mais fortes de `Ruby` são as suas `Gems`. Uma `Gem` é uma biblioteca, um conjunto de arquivos é métodos `Ruby` reusáveis, etiquetada com um nome e uma versão (via um arquivo chamado de `gemspec`). As `gems` ficam armazenadas em um servidor, e quando há necessidade de utilizar uma `gem` em específico, basta chamá-la dentro do código `Ruby` e ela será carregada e instalada no código da aplicação automaticamente. Depois, basta utilizar seus métodos para executar a funcionalidade desejada [Ruby-Lang 2015].

Em adição às características principais citadas, `Ruby` também possui tratamento de exceções, é altamente portável entre sistemas operacionais, possui gerenciamento automático de memória e possui um sistema de *threading* independente do sistema operacional e de plataforma [Ruby-Lang 2015].

Além de ser uma linguagem de compreensão e aprendizado relativamente fácil, ela possui código enxuto, isto é, os comandos possuem nomes pequenos, de forma a facilitar a escrita e leitura de código [Ruby-Lang 2015].

O `Ruby` atualmente se encontra em sua versão 2.3, a qual será utilizada para o desenvolvimento das implementações deste trabalho.

2.3.1 Framework Rails

O `Ruby` dispõe de um *framework* para a *web*, chamado de `Rails`. Este *framework* é construído sobre as premissas de simplicidade, reusabilidade, extensibilidade, velocidade, testabilidade, produtividade e portabilidade [Ruby-On-Rails 2015].

`Rails` utiliza o padrão de arquitetura *Model View Controller* (MVC), assumindo que este

padrão é o mais correto e impondo, de certa forma, que o programador utilize este padrão. A justificativa para isso é que a utilização deste padrão facilita o desenvolvimento de aplicações [Hansson, Ruby e Thomas 2013].

`Rails` utiliza convenções ao invés de configurações, de forma que seguir a convenção se torna mais simples do que realizar configurações [Ruby-On-Rails 2015]. Ao utilizar as conveções do `Rails`, otimiza-se o que sempre é usado na construção de sistemas, permitindo ao desenvolvedor focar nas regras de negócio.

Estas convenções e a agilidade que elas proporcionam no momento de desenvolver aplicações podem ser vistas através de seus geradores, sendo o mais conhecido o *scaffold*, que gera toda a estrutura da aplicação com modelo, controlador, visão, testes e configurações padrão. Existem também geradores específicos para modelos, controladores, visões, estrutura de plugins, dentre vários outros. Os geradores proporcionam agilidade porque automatizam tarefas repetitivas do processo de desenvolvimento de *software* para a *web*. Em `Rails`, esta automatização faz parte de um conceito denominado *Don't Repeat Yourself* (DRY) [Hansson, Ruby e Thomas 2013] [Ruby-On-Rails 2015].

Atualmente, os desenvolvedores RoR tendem a utilizar metodologias ágeis, TDD e BDD. Assim como no `Ruby`, o ecossistema `Rails` disponibiliza ferramentas que possibilitam estes tipos de programação.

O `Rails`, assim como o `Ruby`, utiliza as `gems` como bibliotecas de arquivos e métodos reutilizáveis, sendo essa uma de suas características mais importantes.

O `Rails` possui um recurso chamado `ActiveRecord`, o qual facilita o trabalho com banco de dados. O *ActiveRecord*, ao utilizar metaprogramação, a qual é utilizada para criar *softwares* que tenham a capacidade de escrever código por si mesmo, cria em tempo de execução todo o código necessário para a manipulação das informações de um banco de dados. Ao adicionar uma nova coluna no banco de dados, não é necessário alterar nenhuma linha no código do modelo. O `ActiveRecord`, utilizando metaprogramação, consegue identificar a nova coluna e automaticamente criar todos os métodos e atributos necessários para permitir o acesso a esta nova coluna [Ruby-On-Rails 2015].

Desta forma, o `ActiveRecord` fornece um gerador de *Create, Read, Update, Delete* (CRUD) dinâmico, possibilitando que a modelagem do banco de dados e a geração de consultas

seja realizada de forma dinâmica, conforme o desenvolvimento da aplicação. Outra contribuição do *ActiveRecord* é oferecer uma camada de abstração do banco de dados, permitindo uma descrição de banco de dados que não depende de plataforma ou de um sistema gerenciador de banco de dados (SGBD) em específico. Além disso, `Rails` utiliza o servidor `WEBrick` por padrão [Ruby-On-Rails 2015].

O *framework* `Rails` encontra-se atualmente na versão 4.2, a qual será utilizada para o desenvolvimento das implementações deste trabalho.

2.4 Trabalhos Relacionados

Existem diversos trabalhos analisando aspectos de mecanismos de implementação e manipulação de variabilidades no contexto de linha de produtos de *software*. Aqui serão resumidos os trabalhos mais relevantes em relação a esta pesquisa.

Em [Anastasopoulos e Gacek 2001], a análise dos mecanismos, a nível de código, é feita em relação aos tipos de variabilidade que é possivel obter nas etapas de *interface*, implementação e inicialização e em qual momento o mecanismo é capaz de habilitar a variabilidade, indicando os requisitos e restrições de uso quanto a um conjunto específico de linguagens de programação. São apresentados também vários critérios que podem ser usados para validação de um mecanismo de implementação de variabilidades. Dentre os critérios descritos, somente os critérios de escopo, flexibilidade, eficiência, *separation of concerns*, traceabilidade, escalabilidade, facilidade de introdução e suporte de linguagens de programação foram utilizados na avaliação dos mecanismos realizada neste artigo. Os resultados deste artigo foram obtidos através de observações sobre a prática de uso destes mecanismos em um parceiro industrial. Em relação aos critérios de validação, serão explicitados aqui os critérios relevantes a esta pesquisa:

- Escopo: refere-se à menor entidade possível que pode ser adaptada para ser suportada pelo mecanismo;
- Flexibilidade: refere-se ao tempo de vinculação possível para a adaptação da entidade;
- Suporte de Linguagens de Programação: refere-se à forma que as linguagens de programação existentes podem ser utilizadas para aplicar o mecanismo.

Este artigo assemelha-se a esta pesquisa ao abordar a temática de avaliar a implementações de mecanismos de variabilidades, considerando um conjunto de quatro linguages de programação (C++, Delphi, Java e Smalltalk) agindo como variável de influência nas implementações. Porém, nesta pesquisa o foco é na utilização do mecanismo de padrões de projeto, sendo que padrões de projeto em específico não foram avaliados no artigo. Além disso, nesta pesquisa pretende-se utilizar a linguagem de programação `Ruby` como variável de influência nas implementações.

Em [Amin, Mahmood e Oxley 2011], o objetivo é classificar os mecanismos orientados a objetos, a nível de código, em termos de tipo de variabilidade (finalidade e efeito da variabilidade), escopo, tipos de *feature*, quais artefatos são afetados pela solução proposta pelo mecanismo e em qual momento o mecanismo é capaz de habilitar a variabilidade. Para isso, são analisados seis mecanismos orientados a objetos, fornecendo exemplos de implementação de cada mecanismo na linguagem Java. Os dados analisados foram coletados destes exemplos implementados em Java.

Este artigo assemelha-se a esta pesquisa em relação aos tipos de dados que deseja-se coletar, como também à forma de obter estes dados, que é através da análise da teoria dos mecanismos e depois a análise de implementações. Porém, embora este artigo estude os mecanismos orientados a objetos, não há avaliação do mecanismo de padrões de projeto. Além disso, nesta pesquisa será utilizada a linguagem `Ruby` para avaliar as implementações, ao contrário do artigo, onde é utilizado a linguagem Java.

Baseando-se nos trabalhos que analisam os aspectos dos mecanismos de variabilidades, existem vários trabalhos relacionando mecanismos de implementação de variabilidades de *software* com `Ruby`.

É possível desenvolver em `Ruby` linguagens de domínio específico, do inglês *domain-specific language* (DSL), para gerar representações mais acuradas de uma solução proposta para uma área específica [Günther 2009]. Isto foi feito utilizando notações e as abstrações para representar o conhecimento e os conceitos do domínio. Como o conhecimento de domínio é colocado dentro da linguagem de programação, o programador que a utiliza tem melhor conhecimento do domínio [Günther 2009].

Este tipo de linguagem caracteriza-se por ser especializada em um domínio de aplicação

específico, uma técnica de representação específica ou uma técnica de solução específica, em contraste com as linguagens de propósito geral. Características de `Ruby` como a metaprogramação, simplificação sintática e natureza dinâmica são fatores que pesaram a favor da seleção da linguagem [Günther 2009].

O trabalho do autor tem por objetivo a busca por amenizar dois pontos que considera como problemas presentes na engenharia de *software*, os quais são a grande quantidade de decomposições de *software*, referindo-se às restrições que são impostas através das decomposições à habilidade da engenharia de *software* em representar preocupações específicas de maneira modular, e o problema da linguagem, referindo-se a grande quantidade de linguagens necessárias para o desenvolvimento de *software*, tais como linguagens para banco de dados, comunicação com a *internet* e linguagens para páginas *web* [Günther 2009].

Para resolver estes problemas, o autor desenvolveu uma *Software Product Line Configuration Language* (SPLCL), que realiza a configuração da LPS que desenvolveu, e uma DSL utilizando o mecanismo de programação orientado a *feature*, onde essa DSL define uma linguagem para a construção de uma LPS baseando-se nos conceitos e vocabulário utilizados no paradigma de linha de produtos de *software* [Günther 2009].

Afim de demonstrar detalhadamente como esta DSL baseada no mecanismo de programação orientada a *feature* foi definida, o autor do referido trabalho usa exemplos como o de uma linha de produtos para expressões matemáticas, explicando todo o processo de construção da DSL e de como o mecanismo de programação orientado a *feature* foi habilitado [Günther e Sunkle 2009-A] [Günther e Sunkle 2009-B] [Günther e Sunkle 2012].

Em continuação a estes trabalhos, o autor realizou implementações de entidades de primeira classe, do inglês *first-class entities*, utilizando a metaprogramação de `Ruby`. Estas entidades permitem configuração e adaptação em tempo de execução, tal como adicionar novas *features*, restrições na LPS ou a instanciação de muitas variantes de um produto com diferentes configurações de *features*. Isto foi desenvolvido como uma extensão do mecanismo orientado a *feature* desenvolvido em trabalhos anteriores, aproveitando-se das características dinâmicas presentes em `Ruby` e de sua metaprogramação, como forma de possibilitar e facilitar a execução de mudanças no código e mudanças de configuração em tempo de execução [Günther e Sunkle 2010].

Estes trabalhos, apesar de relacionarem a linguagem de programação `Ruby` com o para-

digma de linha de produtos de *software* e utilizarem com sucesso um conjunto de mecanismos existentes para implementar artefatos de código reutilizáveis e com variabilidade, tais como a programação orientada a *feature* e mecanismos orientados a objetos, não demostram qual a viabilidade de utilizar a linguagem `Ruby` no contexto de LPS. O foco destes trabalhos é utilizar as características presentes em `Ruby` para desenvolver uma DSL e tentar amenizar a grande quantidade de decomposições existentes no desenvolvimento de *software* e também amenizar o problema que a grande quantidade de linguagens necessárias para representar uma solução de *software*.

Outro trabalho, endereçando os temas de LPS e `Ruby on Rails`, explora a variabilidade de Interfaces de Usuário (IU). O grande diferencial da variabilidade, neste contexto, é que a combinação de funcionalidades cria um número exponencial de configurações. Desta forma, projetar uma interface adequada para todas as configurações é inviável do ponto de vista prático [Jesus 2013].

Desta forma, o autor apresenta um novo algoritmo utilizando como base um estudo exploratório para identificar os principais desafios e limitações das abordagens atuais no contexto de variabilidade de Interfaces de Usuário em LPS's. Utilizando como base a implementação de uma LPS baseada em um sistema *web*, que foi desenvolvida utilizando a linguagem de programação `Ruby` e o *framework* `Rails`, foi explorada a utilização de técnicas para a implementação dessa variabilidade neste caso de uso particular em conjunto de algumas modificações feitas nestas técnicas para otimizar este processo [Jesus 2013].

O trabalho citado anteriormente, mesmo relacionando os temas de LPS e `Ruby on Rails`, possui foco no estudo de variabilidade em Interfaces de Usuário. Embora esteja relacionado com este trabalho ao utilizar `Ruby on Rails` para implementação de variabilidades, não apresenta a viabilidade de utilizar `Ruby on Rails` no contexto de implementação de variabilidades em uma LPS.

Capítulo 3

Protocolo de Pesquisa

Neste capítulo é apresentado o protocolo de pesquisa definido para guiar o desenvolvimento deste trabalho, e também o cenário utilizado como base de requisitos para as implementações.

O uso da estrutura deste protocolo fornece suporte e guia a pesquisa à uma melhor compreensão dos objetivos, do que deve ser feito e como fazer, e quais ações tomar a partir dos dados obtidos.

3.1 Definição do Protocolo

A definição do protocolo elaborado para este trabalho tem por base a estrutura proposta por Claes Wohlin e Aybüke Aurum [Aurum e Wohlin 2014], a qual pode ser observada na Figura 3.1.

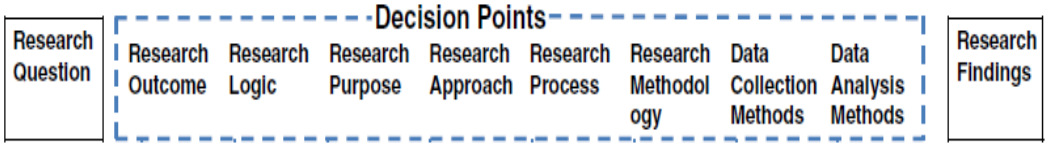

Figura 3.1: Estrutura do Projeto de Pesquisa [Aurum e Wohlin 2014]

Como pode ser visto na Figura 3.1, os principais pontos da pesquisa são as questões de pesquisa, o processo de pesquisa e os resultados. No processo de pesquisa são definidos pontos de decisão acerca do tipo de pesquisa e da forma de condução da mesma. Estes pontos são: Questão de Pesquisa, Contribuição da Pesquisa, Lógica da Pesquisa, Propósito da Pesquisa, Abordagem da Pesquisa, Processo da Pesquisa, Metodologia da Pesquisa, Método de Coleta de Dados, Método de Análise de Dados e Resultados da Pesquisa.

No artigo, o autor fornece algumas opções que servem como exemplo para cada um dos pontos de decisão, sendo que a definição de um ponto de decisão pode influenciar a definição de outro. Não é necessário realizar definições dos pontos de decisão de forma sequencial. Caso um ponto de decisão já seja previamente conhecido, os demais podem ser definidos baseado neste ponto [Aurum e Wohlin 2014]. Além disso, os pontos de decisão apresentados no artigo são apenas exemplos dentro do conjunto de pontos de decisão existente, não necessariamente devendo se limitar a apenas estes. A utilização desta estrutura visa facilitar a definição de cada um dos pontos de decisão ao elaborar um protocolo de pesquisa.

3.1.1 Questão de Pesquisa

A questão de pesquisa é o elemento central da pesquisa, pois ela determina ou influencia muito todo o processo de pesquisa, incluindo a escolha de uma opção adequada para cada ponto de decisão. Uma questão de pesquisa pode ser relacionada a um conjunto de hipóteses, conceitos ou relações entre conceitos [Aurum e Wohlin 2014].

Neste trabalho, a questão central de pesquisa refere-se a obter a viabilidade técnica de implementar variabilidades na linguagem `Ruby` utilizando os padrões de projeto *Template Method*, *Decorator*, *Strategy* e *Observer*. Este objetivo relaciona conceitos como a engenharia de domínio presente na linha de produtos de *software*, a linguagem `Ruby`, gerenciamento de variabilidades e padrões de projeto como mecanismos de implementação de variabilidades. Na Figura 3.2 é possível ver esta relação entre as variáveis de estudo.

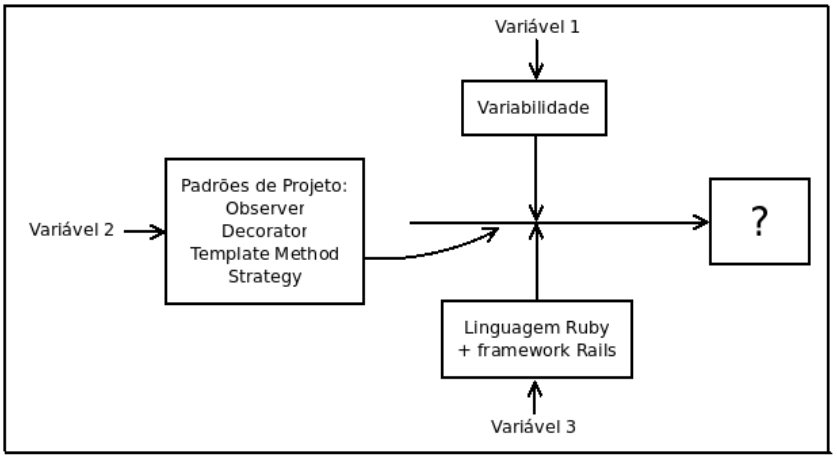

Figura 3.2: Relação entre as variáveis de estudo

Esta questão de pesquisa impactou o restante da pesquisa pois foi necessário que uma análise dos padrões de projeto citados anteriormente fosse realizada em relação ao gerenciamento de variabilidades, para que então fosse possível analisar as respectivas implementações em `Ruby`. Como o gerenciamento de variabilidades é mais adequado e eficiente quando a análise de requisitos e construção de artefatos é realizada sobre o domínio de aplicação [Rommes, Schmid e Linden 2007], para realizar as implementações foi necessário definir um cenário de algum domínio de aplicação, e o domínio selecionado foi o de *blogs*.

3.1.2 Contribuição da Pesquisa

Para que seja possível realizar o estudo de viabilidade técnica proposto por este trabalho, uma pesquisa básica é necessária. A pesquisa de base é aplicada a um problema quando o foco está em entender um problema e não em fornecer uma solução específica para ele [Aurum e Wohlin 2014].

O principal artefato gerado em uma pesquisa de base é o conhecimento obtido através do entendimento do problema [Aurum e Wohlin 2014]. No caso deste trabalho, objetiva-se obter conhecimento sobre os elementos de variabilidade e sobre a aderência de utilizar padrões de projeto como mecanismo para implementar variabilidades em Ruby.

A aplicação de uma pesquisa básica é justificada pois a ênfase está em entender aspectos inerentes à utilização de padrões de projeto para implementar variabilidades e os resultados do uso destes padrões com a linguagem de programação `Ruby`, comparando as características definidas pela teoria dos padrões de projeto com as características obtidas na prática ao implementá-los em `Ruby`.

3.1.3 Lógica da Pesquisa

A lógica da pesquisa é indutiva. Uma pesquisa indutiva baseia-se em argumentos indutivos, isto é, parte de dados mais específicos para chegar a dados mais gerais. A observação e análise do conjunto dos dados específicos permite que o pesquisador realize inferências de conceitos teóricos ou de padrões, permitindo o desenvolvimento de conclusões gerais ou teorias [Aurum e Wohlin 2014].

Neste trabalho, ao combinar os temas de gerenciamento de variabilidades, padrões de pro-

jeto e Ruby neste trabalho, existem algumas características semelhantes entre estes temas, sendo possível inferir conceitos e padrões através da teoria, culminando no surgimento de alguns pontos passíveis de análise, que podem ser interpretados também como questões de pesquisa.

A análise destes pontos de vista justifica a coleta de dados para avaliar a viabilidade técnica de implementar variabilidades em Ruby utilizando padrões de projeto. Os dados coletados das implementações são utilizados para analisar cada ponto de análise e, por consequência, respondem às respectivas questões de pesquisa, confirmando ou não as relações teóricas identificadas inicialmente.

3.1.4 Propósito da Pesquisa

O propósito da pesquisa é descritivo. Uma pesquisa descritiva é aplicado quando se deseja descrever um fenômeno ou características de um problema [Aurum e Wohlin 2014].

Intenciona-se identificar as características dos padrões de projeto que são definidas pela teoria, focando na implementação de variabilidades, e posteriormente identificar se estas características ocorrem na prática ao utilizar Ruby para implementar os padrões. Desta forma, será possível descrever as características do desenvolvimento de artefatos de *software* em Ruby utilizando os padrões de projeto definidos, para que posteriormente estas características sejam analisadas possibilitando obter o estudo da viabilidade técnica de utilizar padrões de projeto como mecanismo para implementar variabilidades em Ruby.

3.1.5 Abordagem da Pesquisa

A abordagem da pesquisa é crítica. Uma pesquisa crítica é utilizada quando se deseja avaliar criticamente as possíveis relações entre os temas abordados na pesquisa, assumindo que não se deve ignorar variáveis que aparentemente não exercem influência nas respostas de questões de pesquisa.

Como foi proposta uma interseção entre as características presentes nos temas de linha de produtos de software, padrões de projeto como mecanismo para implementação de variabilidades e a linguagem Ruby, deseja-se verificar a veracidade destas relações. Variáveis tais como as características da linguagem de programação utilizada e características de implementação presentes em cada um dos padrões de projeto, como também o gerenciamento de variabilidade

presente na etapa de análise de domínio e gerenciamento de variabilidade presente no mecanismo de padrões de projeto são considerados para a etapa de coleta e análise dos dados.

3.1.6 Processo da Pesquisa

O processo de pesquisa é “mixado”. Uma pesquisa mixada é utilizada quando objetiva-se obter informações de cunho prático acerca de um processo ou ferramenta. Um processo mixado envolve a coleta de dados tanto qualitativos quanto quantitativos, para que desta forma os problemas de pesquisa sejam melhor compreendidos [Aurum e Wohlin 2014] [Creswell 2013].

O objetivo é compreender o processo de implementação de variabilidades com padrões de projeto utilizando `Ruby` na perspectiva de um possível programador `Ruby`. Embora os dados coletados sejam qualitativos e quantitativos, o papel dos dados quantitativos será de auxiliar na interpretação e análise dos dados qualitativos.

Considerando que o processo da pesquisa é mixado, a coleta de dados é projetada de forma concorrente, isto é, dados qualitativos e quantitativos são coletados.

3.1.7 Metodologia da Pesquisa

A metodologia de pesquisa é baseada na metodologia do *Design Science Research* (DSR). A metodologia do DSR fornece um modelo dividido em estágios para a construção e refinamento dos artefatos desenvolvidos. No DSR, o objetivo é identificar o problema, sugerir uma forma de resolvê-lo, desenvolver esta sugestão, validar o artefato desenvolvido e obter conclusões acerca do que foi feito, considerando o contexto que o artefato será aplicado. O artefato pode ser um método, uma ferramenta, uma teoria ou regra, documentação, dentre outras coisas. É possível utilizar uma revisão de literatura para desenvolver ou guiar a construção do artefato, bem como utilizar o conhecimento prático de pessoas que trabalham na área para a qual o artefato está sendo desenvolvido. A premissa do uso do DSR é "aprender através do ato de construir", isto é, utilizar o conhecimento para projetar as soluções [Aurum e Wohlin 2014].

A utilização do DSR inicia com a definição de uma classe de problema, isto é, identificar onde está o problema e quais conceitos estão relacionados a ele. O problema a ser abordado pode ser tanto teórico quanto prático. Após isso, é necessário identificar quais objetivos ou metas se fazem necessários para que o problema seja considerado satisfatoriamente resolvido.

Este processo inicial refere-se à etapa de "conscientização", e é uma delineação inicial do problema. Após a etapa de conscientização, é necessário que uma revisão sistemática na literatura seja realizada objetivando estabelecer um quadro de soluções empíricas conhecidas e possíveis [Lacerda et al. 2013].

Após definir a classe de problemas, é necessário caracterizar os artefatos associados à esta classe. No contexto do DSR, o artefato refere-se à organização de componentes no ambiente externo de forma que surtam efeito ou atinjam objetivos em um ambiente externo. Após definidos, os artefatos podem ser classificados como constructos, modelos, métodos ou instanciações [Lacerda et al. 2013].

Os constructos, ou também chamados de conceitos, constituem o vocabulário de um domínio. Eles são basicamente uma conceituação utilizada para descrever os problemas dentro do domínio e para especificar as respectivas soluções [Lacerda et al. 2013].

Os modelos são referentes à conjuntos de proposições ou declarações que expressam as relações entre os constructos. Em atividades de *design*, modelos representam situações como problema e solução. Ele pode ser visto como uma descrição, ou seja, como uma representação de como as coisas são [Lacerda et al. 2013].

Os métodos referem-se à conjuntos de passos, tais como um algoritmo ou orientação, usados para executar uma tarefa. Métodos baseiam-se em um conjunto de constructos subjacentes (linguagem) e uma representação (modelo) em um espaço de solução. Os métodos podem ser ligados aos modelos, nos quais as etapas do método podem utilizar partes do modelo como uma entrada que o compõe [Lacerda et al. 2013]

As instanciações são a concretização de um artefato em seu ambiente. Instanciações operacionalizam constructos, modelos e métodos. No entanto, uma instanciação pode, na prática, preceder a articulação completa de seus constructos, modelos e métodos. Instanciações demonstram a viabilidade e a eficácia dos modelos e métodos que elas contemplam [Lacerda et al. 2013].

O processo de condução de uma DSR é composto por etapas. A Figura 3.3 demonstra visualmente as etapas e o fluxo entre elas.

A etapa de Conscientização observada na Figura 3.3, refere-se à compreensão do problema a ser abordado. O principal resultado desta etapa é a definição e a formalização do problema

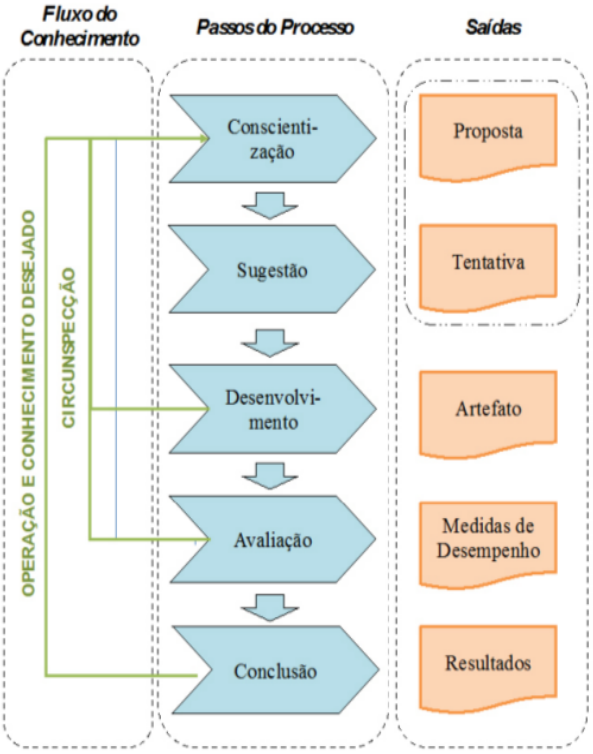

Figura 3.3: Condução da DSR, relacionado etapas e fluxo (Adaptado de: [Lacerda et al. 2013])

a ser solucionado, suas fronteiras (ambiente externo) e as soluções satisfatórias necessárias, os quais compõem uma proposta [Lacerda et al. 2013].

A etapa de Sugestão, refere-se às atividades de desenvolver uma ou mais alternativas de artefato que solucionam o problema. Por consequência, o resultado desta etapa é o conjunto de possíveis artefatos e a escolha de um, ou mais, para serem desenvolvidos, os quais são interpretados como uma tentativa de solução [Lacerda et al. 2013].

A etapa de Desenvolvimento corresponde ao processo de constituição do artefato em si. Neste momento o pesquisador constrói o ambiente interno do artefato, uma vez que os objetivos e o ambiente externo foram caracterizados na Conscientização. Essa construção pode utilizar diferentes abordagens, tais como algoritmos ou protótipos, de forma que o resultado seja um artefato funcional [Lacerda et al. 2013]. Esta etapa não se trata exclusivamente do desenvolvimento de produtos, pois o objetivo da abordagem do DSR é mais amplo, podendo ser a geração de conhecimento que seja aplicável e útil para a solução de problemas, melhoria de sistemas já existentes e, ainda, criação de novas soluções e/ou artefatos [Venable 2006].

Este conhecimento gerado na etapa de Desenvolvimento, embora seja aplicado pontualmente na solução de problemas específicos ou no desenvolvimento de novos artefatos, deve ser generalizável para o que foi definido como Classe de Problemas. Esta generalização permite a construção de um conhecimento útil no sentido pragmático [Lacerda et al. 2013] [Venable 2006].

A etapa de Avaliação refere-se a um processo de verificação do comportamento do artefato no ambiente para o qual foi projetado, em relação às soluções que se propôs alcançar. Se for necessário verificar o desempenho do artefato, uma série de procedimentos deve ser utilizada [Lacerda et al. 2013]. A validação do artefato possui fundamentação na filosofia pragmática [Worren, Moore e Elliott 2002], a qual possui três componentes principais: proposições explícitas e causais tais como se A, então B é provável, dependendo das condições, regras que podem ser usadas para testar a validade destas afirmações causais e declarações explícitas de como os resultados são criados [Worren, Moore e Elliott 2002].

A etapa de Conclusão consiste na formalização geral do processo e sua comunicação às comunidades acadêmica e de profissionais [Lacerda et al. 2013].

No caso deste trabalho, objetiva-se obter conhecimento através da análise da teoria de padrões de projeto e testar a validade da teoria perante a prática da construção de artefatos com padrões de projeto. A parte prática será em relação à construção destes artefatos em `Ruby`.

A classe de problemas é a construção de artefatos reutilizáveis de *software* em `Ruby` utilizando de padrões de projeto, devendo atingir o objetivo de fornecer a variabilidade de *software* também. A linguagem `Ruby` e os padrões de projeto que serão implementados possuem características que favorecem à uma implementação baseada em componentes.

O processo de implementação gera informações de cunho prático, as quais serão avaliadas utilizando as questões e métricas definidas no GQM.

3.1.8 Método de Coleta e Análise de Dados

Para estruturar a coleta de dados, é utilizado o *framework* do GQM para relacionar as questões de pesquisa como os dados que devem ser coletados.

O GQM possui três níveis, sendo que através deste método é possível estabelecer uma estrutura hierárquica de avaliação [Basili, Caldiera e Rombach 1994]. A Figura 3.4 demonstra

visualmente a estrutura do GQM.

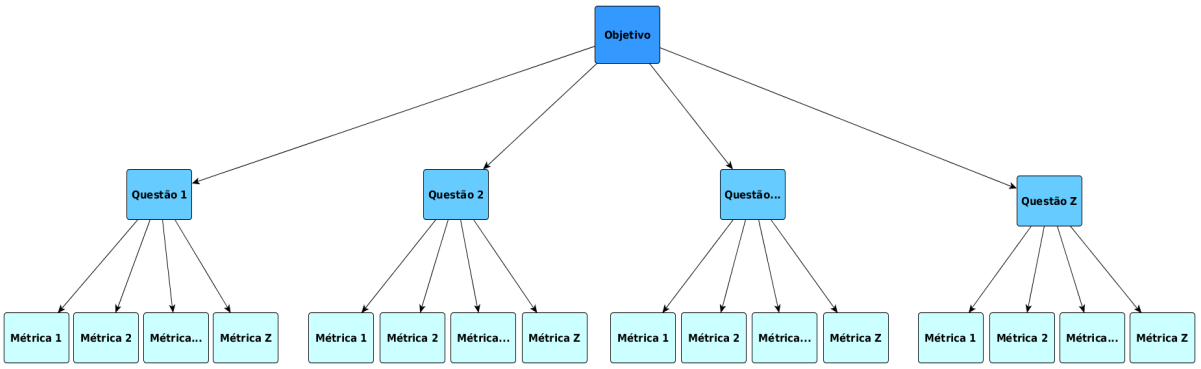

Figura 3.4: Exemplo da estrutura hierárquica da abordagem do GQM

No nível conceitual (nível mais alto da estrutura representada na figura) um objetivo é definido para o objeto de pesquisa, por uma variedade de razões, baseado em vários modelos de características e a partir de vários pontos de vista, relativamente a um ambiente particular. Os objetos de medição podem ser produtos, processos ou recursos [Basili, Caldiera e Rombach 1994].

No nível operacional (nível intermediário da estrutura representada na figura) um conjunto de questões de pesquisa é definido como forma de caracterizar a avaliação ou realização de um objetivo específico a ser cumprido. As questões tentam caracterizar o objeto de medida com respeito a uma característica específica a ser selecionada, e tentam também determinar suas caraterísticas sobre determinado ponto de vista [Basili, Caldiera e Rombach 1994].

No nível quantitativo (nível mais baixo da estrutura representada na figura) um conjunto de métricas ou dados é associado com cada questão de pesquisa de forma a respondê-la de forma quantitativa. As métricas ou dados podem ser objetivos, quando dependem somente do objeto que está sendo medido e não do ponto de vista dos quais vieram, e podem ser subjetivos, quando dependem tanto do objeto que está sendo medido quanto do ponto de vista dos quais vieram [Basili, Caldiera e Rombach 1994].

O resultado da aplicação desta abordagem é a especificação de um sistema de medição que aponta para um conjunto particular de assuntos e para um conjunto de regras para a interpretação dos dados medidos [Basili, Caldiera e Rombach 1994]. Desta forma, o GQM é um *framework* para coleta e análise de dados.

Para a análise dos padrões de projeto, foi definido um modelo baseado no GQM, o qual

pode ser visto na Seção 4.2. Para as implementações, o modelo baseado na abordagem do GQM utilizado neste trabalho está definido na Seção 5.1.3.

3.1.9 Elaboração do Cenário Para as Implementações

Para implementar os mecanismos de padrões de projeto, foi necessário obter requisitos sobre algum domínio de aplicação, de forma que fosse possível obter um cenário com requisitos e *features* e que pudesse contemplar também a variabilidade para realizar as implementações. Desta forma, foi elaborado um modelo de *features*, baseando-se na modelagem realizada no processo de engenharia de domínio existente na LPS, de forma que este modelo pudesse representar os requisitos, as *features* e as variabilidades a serem utilizados para as implementações.

Foi selecionado o domínio de *blogs* para realizar esta modelagem. As principais *features* presentes neste domínio foram identificadas para desenvolver o modelo de *features*, baseando-se em *features* existentes em *sites* de criação de *blogs* tais como ***Blogger*** [Blogger 2015] e ***Wordpress*** [Wordpress 2015]. Este modelo é representado visualmente através da árvore de *features* na Figura 3.5.

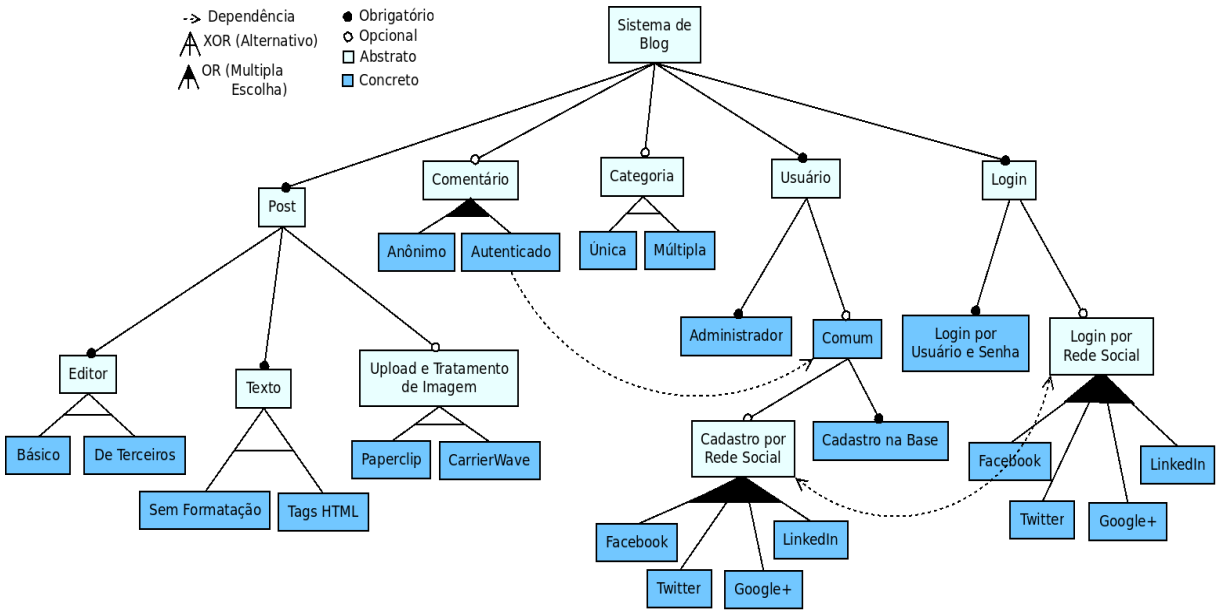

Figura 3.5: Representação visual em formato de árvore do modelo de *features* para o domínio de *blogs*

Cada *feature* abstrata (azul claro), observada nas folhas mais altas da figura, representa uma

necessidade, ou um conjunto de funcionalidades que atendem uma necessidade, incluindo a definição da *feature* e requisitos funcionais, tais como inserção, exclusão, edição e busca.

Cada *feature* concreta (azul escuro), observada nas folhas mais baixas da figura, representa as formas que podem ser implementadas para cada *feature* abstrata à qual está relacionada.

Este modelo e a avaliação não contemplam aspectos como segurança e desempenho. Cada um destes aspectos possuem características, ferramentas e técnicas muito específicas e necessárias para que fossem pudessem ser abordadas no contexto deste trabalho, mas de uma forma que não haveria contribuição no sentido de avaliar implementação de variabilidades em `Ruby` utilizando o mecanismo de padrões de projeto.

O modelo de *features* não contempla requisitos funcionais e não-funcionais acerca da área de banco de dados. Neste ponto, considerar a utilização do *framework* `Rails` mostra-se uma alternativa interessante. O recurso de *Active Record* do `Rails` fornece uma camada de abstração do banco de dados, permitindo uma descrição de banco de dados que não depende de plataforma ou de um sistema gerenciador de banco de dados (SGBD) em específico, além de utilizar o servidor `WEBrick` como padrão. Desta forma, elimina-se a necessidade de trabalhar com variabilidade em banco de dados.

Além disso, não é trabalhada a variabilidade com *interfaces* de usuário. O `Rails` fornece geradores para visões, permitindo a construção de *interfaces* gráficas sem mexer em seu código interno. Desta forma, elimina-se a necessidade de trabalhar com a variabilidade de *interfaces* de usuário.

A utilização do *framework* `Rails` nas implementações elimina a necessidade de tratar destas questões no modelo de *features* apresentado por conter estas características adotadas como *default*, uma vez que os pontos de variação que existem em banco de dados e *interfaces* de usuário não são controlados diretamente pelo mecanismo de padrões de projeto para implementação de variabilidades, o qual deseja-se avaliar a especificação em código desenvolvida em `Ruby`.

Este modelo é capaz de fornecer contexto para cada padrão de projeto que será implementado. O contexto dos padrões de projeto são explicados na Seção 4.2.

Capítulo 4

Padrões de Projeto como Mecanismo para Implementação de Variabilidades

Alguns padrões de projeto evoluíram no sentido de como a variabilidade poderia ser separada e desacoplada. Dentre eles, alguns são padrões de projeto orientados a objetos bem conhecidos, tais como o *Observer*, o *Template Method*, o *Strategy* e o *Decorator* [Gamma et al. 1995]. De acordo com a literatura revisada, estes quatro padrões de projeto são considerados adaptados para realizar implementações e gerenciamento de variabilidades [Apel et al. 2013].

Estes quatro padrões habilitam a variabilidade em tempo de execução, o que é particularmente interessante para este trabalho, uma vez que a linguagem `Ruby`, a qual será utilizada para as implementações, é uma linguagem interpretada.

Por fim, a utilização destes padrões incentiva o desacoplamento e encapsulamento de *features* [Apel et al. 2013].

4.1 Definição de Padrões de Projeto

Padrões de projeto, em sua definição mais geral, são descrições de objetos e classes colaborativos que são personalizados para para resolver um problema de projeto comum em contexto particular [Gamma et al. 1995].

Um padrão de projeto possui quatro elementos essenciais [Gamma et al. 1995]:

- O **nome do padrão** é uma forma de referenciar todo o contexto do padrão, tal como a descrição do problema, sua solução e consequências, utilizando apenas uma ou duas palavras. Ao utilizar um nome para o padrão, permite aos desenvolvedores comunicarem-

se com um alto nível de abstração, sem necessitar repetir todos os detalhes do padrão a cada vez que referenciá-lo;

- O **problema** descreve quando aplicar o padrão, explicando o problema e o seu contexto. Ele deve descrever problemas específicos de projeto, classes ou estruturas de objetos que possuem características de um projeto não flexível. Algumas vezes, o problema irá incluir uma lista de condições que deverão ser atingidas antes, para que então faça sentido aplicar o padrão;

- A **solução** descreve os elementos que compõem o projeto, seus relacionamentos, responsabilidades e colaborações. A intenção da solução é servir como um modelo que pode ser aplicado em várias situações distintas, e desta forma, a solução não descreve um projeto ou implementação único. Desta forma, o padrão fornece uma descrição abstrata de um problema de projeto e como um arranjo de elementos soluciona este problema;

- As **consequências** são os resultados e os conflitos de escolha da aplicação do padrão. Emboras as consequências não sejam frequentemente expressas na descrição das decisões de projeto, elas são críticas para avaliar alternativas de projeto e para entender os custos e benefícios da utilização do padrão. Uma vez que o reuso é um fator determinante em projetos orientados à objetos, as consequências de um padrão incluem o impacto na flexibilidade, extensibilidade e portabilidade de um sistema. Identificar as consequências ajuda a entendê-las e avaliá-las.

Além disso, os padrões de projeto são classificados por dois critérios: o seu propósito e o seu escopo [Gamma et al. 1995].

O propósito demonstra o que o padrão faz. Os padrões podem ter um propósito criacional, estrutural ou comportamental. Padrões de criação preocupam-se com o processo de criação dos objetos. Padrões estruturais trabalham com a composição de classes ou objetos. Padrões comportamentais caracterizam as formas com as quais as classes e objetos interagem e distribuem responsabilidades [Gamma et al. 1995].

O escopo especifica se o padrão aplica-se em primeiro lugar às classes ou aos objetos. Padrões de classes trabalham com o relacionamento entre classes e subclasses. Estes relacionamentos são estabelecidos principalmente através de herança. Padrões de objetos trabalham com

relacionamentos entre objetos, e também são estabelecidos através de herança. Desta forma, os únicos padrões chamados padrões de classes são aqueles que focam no relacionamento entre classes, porém a maioria dos padrões estão no escopo dos objetos [Gamma et al. 1995].

Combinando estes dois critérios utilizados para classificar padrões de projeto, obtem-se uma classificação mais específica do padrão.

Padrões criacionais de classes deixam alguma parte da criação de objetos para as subclasses, enquanto que os padrões criacionais de objetos deixam esta tarefa para outro objeto. Os padrões estruturais de classes utilizam herança para compor as classes, enquanto que os padrões estruturais de objetos descrevem formas de realizar as ligações entre os objetos e montá-los para formarem uma estrutura. Os padrões comportamentais de classes utilizam herança para descrever algoritmos e o fluxo de controle enquanto que os padrões comportamentais de objetos descrevem como um grupo de objetos coopera para realizar uma tarefa que um objeto sozinho não poderia realizar [Gamma et al. 1995].

4.2 Análise dos Padrões de Projeto

Nesta seção, as definições e características dos padrões de projeto *Observer*, *Template Method*, *Strategy* e *Decorator* são apresentadas, trazendo também as características quanto à utilização de cada um destes padrões para implementar variabilidades.

A intenção aqui é de obter uma análise destes padrões de projeto, examinando como estes mecanismos realizam a manipulação da variabilidade no contexto de linha de produtos de *software*.

Para estruturar a coleta e análise dos dados, um modelo baseado nas métricas e questões do GQM foi definido. As questões de pesquisa neste modelo são baseadas na presença dos elementos de variabilidade e nas abordagens que podem ser seguidas para realizar implementações de componentes com variabilidade através dos padrões. As métricas definidas no modelo são dados que são coletados da teoria dos padrões de projeto.

Baseando-se na estrutura proposta pela abordagem do GQM, foram elaborados o escopo e objetivo de coleta de dados, as questões e suas respectivas métricas. O escopo e objetivo são referentes aos elementos de variabilidades que podem ser obtidos por cada padrão e como os padrões podem ser utilizados para implementar variabilidades, de acordo com a literatura. Este

Tabela 4.1: Tabela do modelo de coleta de dados da teoria elaborado com base no GQM

Objetivo	
Propósito	Analisar Padrões de Projeto
Assunto	Teoria dos Padrões de Projeto
Objeto (processo)	Implementação de Variabilidades
Ponto de vista	Programador Interessado em Desenvolver Variabilidades de Software com Padrões de Projeto
Questão 1)	Q1: Como o padrão de projeto 'X' pode ser utilizado para implementar variabilidades?
Métricas	M1.1: Descrever características que podem ser utilizadas M1.2: Por que estas características podem ser utilizadas
Questão 2)	Q2: Quais elementos de variabilidade o padrão de projeto 'X' possui (variabilidade quanto à finalidade, unidades de código, variabilidade quanto ao efeito, escopo de variação, tempo de vinculação, features)?
Métricas	M2.1: Variabilidade quanto à finalidade M2.2: Unidades de código M2.3: Variabilidade quanto ao efeito M2.4: Escopo de variação M2.5: Tempo de vinculação M2.6: Feature

modelo assume um formato semelhante a um questionário, e pode ser visualizado na Tabela 4.1.

Observando a Tabela 4.1, o 'X' presente nas questões refere-se aos padrões de projeto que são avaliados. Como cada um deles é implementado e avaliado individualmente, na avaliação o 'X' assume o nome do padrão de projeto que está sendo avaliado em determinado momento. Os rótulos *Q1* e *Q2* apontam para as questões de pesquisa, e os rótulos *M1.1, M1.2, M2.1, ..., Mi.j* apontam para as métricas que são respectivas às questões de pesquisa.

O nível conceitual foi definido como sendo o objetivo de analisar os padrões de projeto, através da descrição em teoria destes padrões.

No nível operacional foram definidas as questões a serem respondidas, abrangendo questões como quais características do padrão podem ou não ser utilizadas para implementar variabilidades e se os padrões atenderam aos elementos de variabilidade.

No nível quantitativo foram definidas as métricas que se deseja medir e as informações que se deseja coletar, associadas às respectivas questões de pesquisa. Sobre as métricas, foram definidos os elementos de variabilidade, isto é, o que varia e onde varia, para que pudessem ser identificados, através da revisão e análise da teoria, informações sobre vários pontos de análise,

os quais foram os tipos de *features*, os tipos de variabilidade quanto à finalidade e quanto ao efeito, quais unidades do código o padrão de projeto funciona, o escopo de variabilidade e o tempo de vinculação que a solução proposta por cada um destes padrões de projeto podem satisfazer.

Neste contexto, a análise foi feita sem considerar uma linguagem de programação específica. Desta forma, este objetivo possui um posto de vista mais teórico, visando identificar como estes padrões de projeto em específico podem manipular variabilidades.

Ao final é gerada uma tabela que sumariza de forma comparativa as informações de cada padrão de projeto que foram obtidas através da teoria.

4.2.1 Padrão Observer

O *Observer* é um padrão comportamental que descreve uma forma comum para implementar um gerenciamento distribuído de eventos, no qual um objeto denominado "assunto", do inglês *subject*, notifica mudanças em seu estado para todos os seus observadores registrados [Apel et al. 2013]. Desta forma, este padrão define uma dependência um-para-muitos entre os objetos.

Ao particionar um sistema em uma coleção de classes colaborativas, é necessário manter a consistência entre objetos relacionados. Porém, ao alcançar a consistência, as classes estarão fortemente acopladas, de forma que esta união reduz muito a capacidade de reuso. Este problema é o que motiva a utilização deste padrão de projeto [Gamma et al. 1995].

O padrão *Observer* descreve como estabelecer os relacionamentos entre estes objetos relacionados. Os objetos chave neste padrão são o assunto e os observadores. O *subject* pode possuir qualquer número de observadores, e todos estes observadores são notificados a toda vez que o assunto sofre uma mudança de estado. Como resposta, cada observador irá consultar o *subject* para que os seus estados possam ser sincronizados. Este tipo de interação também é conhecida como publicação-assinatura, do inglês *publish-subscribe*. O *subject* é o que publica as notificações e as envia sem necessitar saber quem são seus observadores [Gamma et al. 1995].

Este padrão é utilizado quando uma implementação possui dois aspectos, de forma que um depende do outro. Ao encapsular cada um destes aspectos em objetos separados, é possível realizar variações destes aspectos e reutilizá-los de forma independente. Outra situação em

que é utilizado é quando uma mudança em um objeto faz necessário realizar mudanças em outros objetos, de forma que não é possível saber quantos objetos necessitam ser modificados. Uma outra situação é quando um objeto deve ser capaz de notificar outros objetos sem realizar suposições sobre quem são estes objetos, de forma a evitar que os objetos sejam fortemente acoplados [Gamma et al. 1995].

A estrutura e relacionamentos entre os objetos componentes deste padrão podem ser vistos na Figura 4.1.

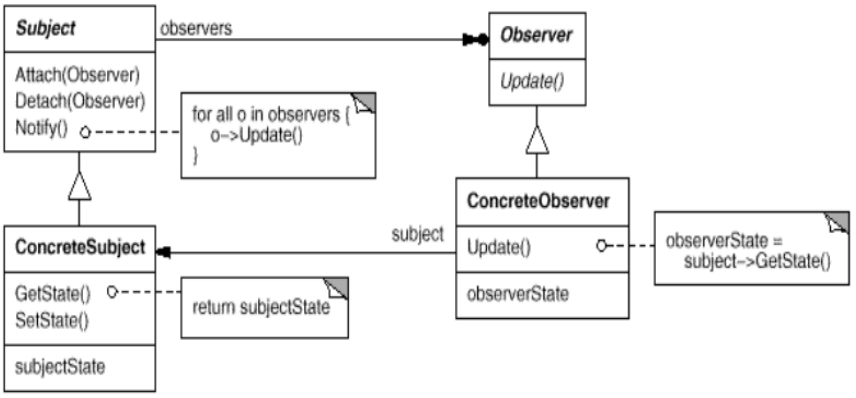

Figura 4.1: Estrutura do padrão de projeto *Observer* [Gamma et al. 1995]

Como pode ser visto na figura, os objetos participantes deste padrão e suas colaborações são os seguintes [Gamma et al. 1995]:

- O *Subject*, o qual conhece seus observadores e fornece uma *interface* para anexar e desanexar objetos observadores. Desta forma, o assunto pode ter um número ilimitado de observadores;

- O *Observer*, o qual define uma *interface* de atualização para os objetos que devem ser notificados de mudanças no assunto;

- O *ConcreteSubject*, o qual armazena os estados que são de interesse dos objetos *ConcreteObserver* e envia as notificações para seus *Observers* quando há uma mudança de estado;

- O *ConcreteObserver*, o qual mantem uma referência ao objeto *ConcreteSubject* e armazena o estado, de forma que o estado seja consistente com o estado do *Subject*. Para isto, implementa uma *interface* de atualização.

A solução proposta pelo padrão *Observer* permite variar os assuntos e os observadores de forma independente, isto é, sem realizar modificações nos *subjects* e nos observadores, sendo possível reusar os *subjects* sem reusar seu observadores e vice-versa [Gamma et al. 1995].

Outra consequência do uso deste padrão é uma abstração do acoplamento entre *subject* e *observer*. Tudo o que o *subject* sabe é que ele possui uma lista de *observers*, porém está referenciando somente a *interface* simples e abstrata da classe *Observer*, isto é, o *subject* não conhece a classe concreta *concreteObserver* de nenhum *observer*, fazendo o acoplamento entre *subject* e *observer* ser mínimo. Desta forma, o *subject* e o *observer* podem pertencer a diferentes camadas do sistema [Gamma et al. 1995].

O *Observer* facilita a comunicação por *broadcast* entre os objetos, isto é, as notificações enviadas pelo *subject* não necessitam especificar quem e quantos são seus receptores, mas são enviadas automaticamente a todos os *observers* interessados que o assinaram [Gamma et al. 1995].

Uma consequência negativa é que podem ocorrer atualizações inesperadas porque os *observers* não possuem conhecimento da existência um do outro, eles não estarão cientes do custo de modificar o *subject*. Uma operação que aparentemente não é prejudicial pode causar uma cascata de atualizações para os *observers* e seus objetos dependentes [Gamma et al. 1995].

No desenvolvimento de uma linha de produtos, o padrão *Observer* facilita a adição ou remoção de *features*, uma vez que as *features* podem ser implementadas como um *observer*. Cada *feature* implementa a *interface* do objeto *observer* e registra a si mesma no *subject* para eventos relevantes. A variabilidade então é alcançada ao registrar ou não registrar os *observers*. O código de diferentes *features* podem ser separados em classes *observers* distintas. Desta forma, é possível adicionar *features* sem modificar a implementação do *subject* [Apel et al. 2013]. No Quadro 5.1 da Subseção 5.1.4 é possível ver um exemplo em `Ruby` de implementação deste padrão.

A Tabela 4.2 mapeia as informações deste padrão para a estrutura de questões e métricas do GQM.

Tabela 4.2: Implementação de variabilidades com o padrão Observer

Padrão Observer	
Questão 1)	
Métricas	M1.1: **Implementar cada *feature* como um *observer*** M1.2: **Como cada *feature* é uma implementação concreta da interface de um *observer*, se uma *feature* for adicionada, ela extenderá comportamentos, e se for removida, não irá alterar o comportamento padrão**
Questão 2)	
Métricas	M2.1: **Positiva, Negativa** M2.2: **Classes** M2.3: **Variabilidade de Lógica** M2.4: **Escopo Aberto** M2.5: **Execução** M2.6: **Obrigatória, Opcional**

Implementar este padrão requer pré planejamento, pois o desenvolvedor necessita decidir antecipadamente onde a futura variação será adicionada, para então preparar o código de forma que este facilite o registro de *observers* e exponha informações relevantes na *interface* do *observer*. As extensões só podem ser adicionados sem modificações invasivas do código base, quando o padrão foi preparado no código de base [Apel et al. 2013].

4.2.2 Padrão Decorator

O padrão estrutural *Decorator* descreve um mecanismo baseado em delegação que flexibiliza a extensão de comportamentos de objetos em tempo de execução.

A motivação para este padrão é que em determinados momentos é necessário adicionar responsabilidades a objetos de forma individual, e não à toda a classe. Uma forma de fazer isso é com herança, mas ao herdar as características de uma classe, estas características são repassadas a todas as instância de subclasses. Esta é uma abordagem inflexível uma vez que não é possível controlar como e quando as características serão adicionadas [Gamma et al. 1995].

Uma abordagem mais flexível é anexar o componente a outro objeto que adiciona as características. Este objeto que faz a anexação é chamado de *decorator*. O *decorator* obedece à *interface* do componente que ele está anexado, e isto faz com que sua presença seja transparente aos clientes do componente [Gamma et al. 1995].

O *decorator* também define como suas subclasses podem extender suas operações, porém elas são livres para adicionar operações para uma funcionalidade específica.

O padrão *Decorator* é utilizado quando se deseja adicionar reponsabilidades a objeto de forma individual, isto é, sem afetar outros objetos, quando se deseja modelar responsabilidades que podem ser retiradas, e quando a extensão de responsabilidades através de subclasses é impraticável, pois algumas vezes é possível obter um grande número de extensões independentes que produzirão uma explosão de subclasses para dar suporte a cada combinação [Gamma et al. 1995].

A estrutura e relacionamentos entre os objetos componentes deste padrão podem ser vistos na Figura 4.2.

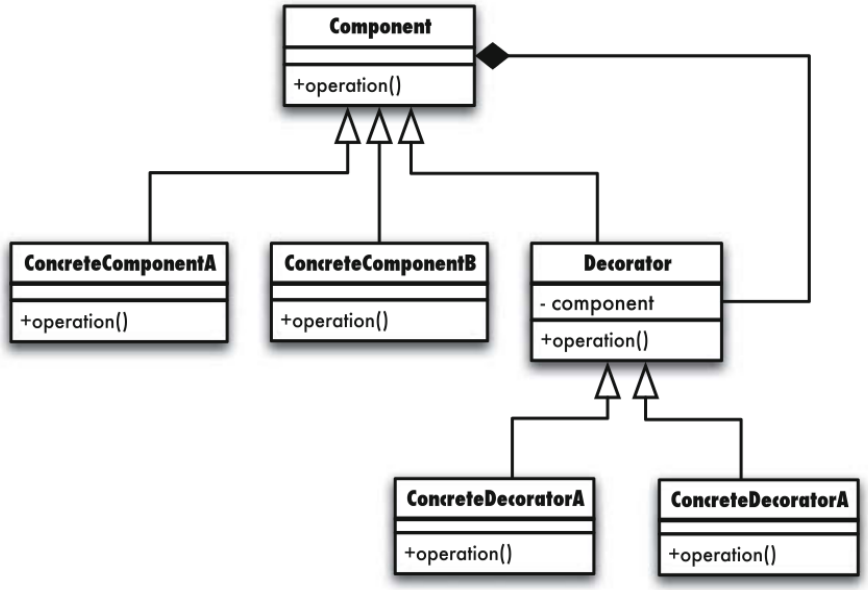

Figura 4.2: Estrutura do padrão de projeto *Decorator* [Apel et al. 2013]

Como pode ser visto na figura, os objetos participantes deste padrão são os seguintes [Gamma et al. 1995]:

- O *Component* define uma interface para os objetos que podem ter responsabilidades adicionadas a eles dinamicamente;
- O *ConcreteComponent* define um objeto para o qual podem ser adicionadas responsabilidades adicionais;

- O *Decorator* mantêm uma referência para um objeto *Component* e define uma interface que se adapta à interface do *Component*;
- O *ConcreteDecorator* é o objeto que efetivamente adiciona as responsabilidades ao componente.

Um objeto do tipo *decorator* encaminha solicitações para um objeto do tipo *component*, podendo realizar ações adicionais antes ou depois de encaminhar estas solicitações.

A aplicação deste padrão em um sistema resulta em uma maior flexibilidade do que a herança estática, pois ao utilizar os objetos *decorators*, as responsabilidades podem ser simplesmente adicionadas e removidas em tempo de execução ao anexá-las ou desanexá-las, contrastando com a herança, na qual é necessário criar uma classe para cada responsabilidade adicional, aumentando muito o número de classes e a complexidade do sistema. Além disso, definir diferentes classes do tipo *Decorator* para uma classe específica do tipo *Component* permite misturar e combinar responsabilidades. Os *decorators* também facilitam para que uma propriedade seja adicionada duas vezes [Gamma et al. 1995].

Outra consequência é que se evita que existam classes cheias de *features* no topo da hierarquia. Ao invés de tentar fazer uma classe complexa e customizável que suporte todas as *features* previsíveis, com o padrão *Decorator* define-se uma classe simples e adiciona as funcionalidades incrementalmente com objetos do tipo *decorator*, fazendo a funcionalidade ser composta de peças simples. Desta forma, não há custo para a aplicação de possuir *features* que não utiliza [Gamma et al. 1995]. No Quadro 5.2 da Subseção 5.1.4 é possível ver um exemplo em `Ruby` de implementação deste padrão.

Um projeto que use o padrão *Decorator* frequentemente resulta em sistemas compostos de vários objetos pequenos que são todos parecidos. Os objetos diferem somente na forma em que estão interconectados, mas não em suas classes ou nos valores de suas variáveis. Embora isto facilite a customização dos sistemas, é difícil entender as interconexões entre objetos [Gamma et al. 1995]. A Tabela 4.3 mapeia estas informações para a estrutura de questões e métricas do GQM.

Tabela 4.3: Implementação de variabilidades com o padrão Decorator

Padrão Decorator	
Questão 1)	
Métricas	M1.1: **Implementar cada *feature* como um *decorator*** M1.2: **Cada *feature* é uma implementação de um *decorator*, sendo que ao adicionar uma *feature* ao componente, ela extenderá comportamentos, e se for removida, não irá alterar o comportamento padrão**
Questão 2)	
Métricas	M2.1: **Positiva** M2.2: **Classes, Métodos** M2.3: **Variabilidade de Lógica, Variabilidade de Fluxo** M2.4: **Aberto** M2.5: **Execução** M2.6: **Opcional, Obrigatório**

No desenvolvimento de uma linha de produtos, o padrão *Decorator* é bem adaptado para implementar *features* opcionais e grupos de *features* nos quais múltiplas *features* podem ser selecionadas [Apel et al. 2013].

4.2.3 Padrão Template Method

O padrão comportamental *Template Method* basicamente define o esqueleto de um algoritmo em uma classe mais abstrata e deixa algumas etapas deste algoritmo para serem definidos em subclasses, de forma que a estrutura do algoritmo não seja modificada. As diferentes subclasses podem fornecer diferentes implementações destas etapas, e desta forma podem modificar o comportamento de um programa [Apel et al. 2013].

A motivação para este padrão é que as vezes é necessário definir um algoritmo em termos de operações abstratas, de forma que as subclasses possam subscrever para fornecer o compartamento concreto. Ao definir estas etapas posteriormente, o *Template Method* ajusta a ordem das etapas e permite que as subclasses variem as etapas para atender suas necessidades.

Desta forma, o padrão *Template Method* deve ser usado quando se deseja implementar as partes que não variam de um algoritmo somente uma vez e deixar as subclasses implementar o compartamento que varia. Também deve ser usado quando o comportamento comum entre as subclasses deve ser fatorado e localizado em uma classe comum para evitar duplicação de

código. Outro uso é para quando se deseja controlar as extensões das subclasses, definindo um *Template Method* que chama "operações de gancho", do inglês *hook operations*, em pontos específicos, de forma a permitir as extensões somente nestes pontos. Uma *hook operation* fornece um comportamento padrão que as subclasses podem extender se necessário, porém, por padrão, uma *hook operation* não é capaz de executar operações [Gamma et al. 1995].

A estrutura e relacionamentos entre os objetos componentes deste padrão podem ser vistos na Figura 4.3.

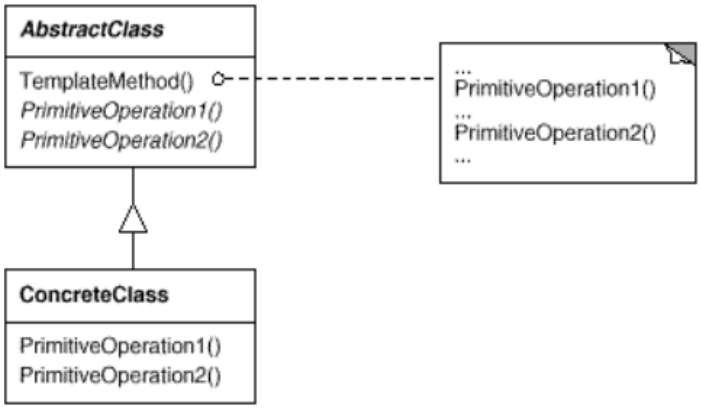

Figura 4.3: Estrutura do padrão de projeto *Template Method* [Gamma et al. 1995]

Como pode ser visto na figura, os objetos participantes deste padrão e suas colaborações são os seguintes [Gamma et al. 1995]:

- A *AbstractClass* define as operações primitivas abstratas que as subclasses do objeto *ConcreteClass* podem modificar para implementar os passos de um algoritmo. Também implementa um *template method* definindo o esqueleto de um algoritmo, o qual faz chamadas às operações primitivas e às operações na *AbstractClass*;

- A *ConcreteClass* implementa efetivamente as operações primitivas para realizar as etapas específicas do algoritmo.

O uso do padrão *Template Method* conduz para uma estrutura de controle invertida, referindo-se a como uma classe pai chama as operações das subclasses e não o inverso [Gamma et al. 1995].

No desenvolvimento de uma linha de produtos, é possível explorar este padrão para implementar comportamentos de *features* alternativas por meio de diferentes subclasses. Especialmente se o algoritmo apresentar diferenças apenas em pequenos detalhes em cada *feature*, é possível compartilhar as partes comuns do algoritmo em uma classe abstrata comum. Desta forma, o padrão *Template Method* faz a separação entre o código das *features* e o código da base [Apel et al. 2013].

O padrão *Template Method* é melhor adaptado a *features* alternativas. Porém *features* opcionais também podem ser implementadas, quando uma implementação padrão é fornecida para cada *Template Method*. Entretanto, o *Template Method* não é adequado para combinar *features* múltiplas por conta das limitações da herança [Apel et al. 2013]. A Tabela 4.4 mapeia estas informações para a estrutura de questões e métricas do GQM.

Tabela 4.4: Implementação de variabilidades com o padrão Template Method

Padrão Template Method	
Questão 1)	
Métricas	M1.1: **Implementar cada *feature* como um *template method*** M1.2: **Cada *feature* sobrescreve os métodos definidos na estrutura do *template method*, sendo que cada *feature* é definida por uma classe que herda de uma classe mais abstrata, podendo assim selecionar comportamentos diferentes para o mesmo método**
Questão 2)	
Métricas	M2.1: **Positiva, Opcional, Alternativa, Funcional** M2.2: **Classes, Métodos** M2.3: **Variabilidade de Lógica, Variabilidade de Fluxo** M2.4: **Seleção** M2.5: **Execução** M2.6: **Obrigatória, Opcional, Alternativa**

No Quadro 5.3 da Subseção 5.1.4 é possível ver um exemplo em `Ruby` de implementação deste padrão.

4.2.4 Padrão Strategy

O padrão *Strategy* define uma família de algoritmos e encapsula cada um deles, fazendo-os ser substituíveis entre eles. Isto permite que o algoritmo varie independentemente da subclasse

que o use [Gamma et al. 1995].

A diferença entre o padrão *Strategy* e o padrão *Template Method* é que o *Strategy* utiliza delegação ao invés de herança. Ao invés de definir um método abstrato para ser subscrito pelas subclasses, o *Strategy* fornece uma *interface* que é implementada pelas subclasses [Apel et al. 2013].

A motivação para este padrão é que as vezes as subclasses ficarão muito grandes e complexas se os vários algoritmos que podem ser necessários durante a execução do programa forem adicionados a elas, dificultando a manutenção. Além disso, diferentes algoritmos serão necessários em tempos diferentes. Não há necessidade de realizar a manutenção de algoritmos que não são utilizados.

Desta forma, o padrão *Strategy* é utilizado quando muitas classes relacionadas diferem somente em seus comportamentos. O *Strategy* fornece uma forma de configurar uma classe com um de muitos comportamentos. Outro uso é quando se necessita de diferentes variantes de um algoritmo. É indicado usar este padrão quando o algoritmo utiliza dados que subclasses não podem saber ou ter acesso, pois este padrão evita expor estrutura de dados específicas e complexas. Por fim, outro uso é quando uma classe define vários comportamentos, os quais parecem ser declarações de múltipla escolha sobre suas operações. Ao invés de utilizar declarações de múltipla escolha, pode-se mover as ramificações condicionais relacionadas para dentro de suas próprias classes *Strategy* [Gamma et al. 1995].

A estrutura e relacionamentos entre os objetos componentes deste padrão podem ser vistos na Figura 4.4.

Como pode ser visto na figura, os objetos participantes deste padrão e suas colaborações são os seguintes [Gamma et al. 1995]:

- O *Strategy* declara uma *interface* comum para todos os algoritmos suportados. Um objeto do tipo *Context* usa esta *interface* para chamar o algoritmo definido por um objeto *ConcreteStrategy*;
- O *ConcreteStrategy* implementa o algoritmo utilizando a *interface* do *Strategy*;
- O *Context* é configurado com um objeto *ConcreteStrategy* e mantém uma referência para um objeto *Strategy*. O *Context* pode definir uma *interface* que permite o *Strategy* acessar

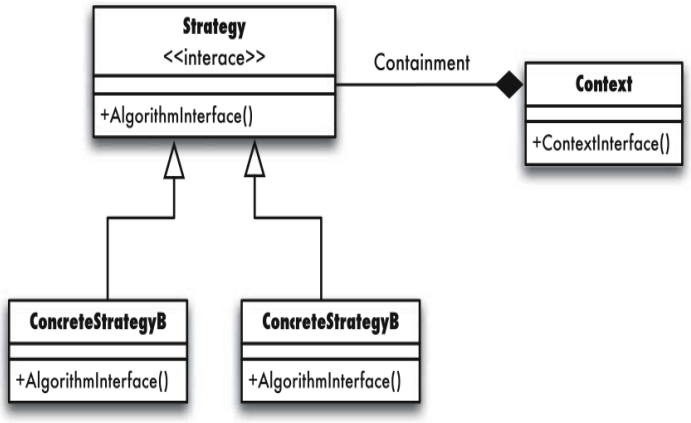

Figura 4.4: Estrutura do padrão de projeto *Strategy* [Apel et al. 2013]

seus dados.

Como consequências, o padrão *Strategy* permite obter uma família de algoritmos ou comportamento relacionados, definida pela hierarquia das classes, de forma que possam ser reutilizados. Fornece também uma alternativa à geração de subclasses feitas através de herança [Gamma et al. 1995].

Os objetos do tipos *Strategy* eliminam declarações opcionais ao ofecerer uma forma alternativa de selecionar comportamentos. Ao encapsular o comportamento em diferentes classes do tipo *Strategy*, as declarações opcionais são eliminadas. Os objetos também podem fornecer diferentes implementações do mesmo comportamento. Além disso, o número de objetos dentro da aplicação é aumentado [Gamma et al. 1995].

No desenvolvimento de uma linha de produtos, o padrão *Strategy* é melhor adaptado para implementar *features* alternativas, considerando que as *features* correspondem a diferentes implementações de métodos. O padrão substitui declarações condicionais não planejadas do código fonte com chamadas aos métodos da *interface* do *Strategy*. Implementar *features* com o *Strategy* encoraja os programadores a encapsular e desacoplar *features* com *interfaces* de uma forma disciplinada. Os desenvolvedores podem especificar precisamente a *interface* para implementações alternativas e variações futuras [Apel et al. 2013].

No Quadro 5.4 da Subseção 5.1.4 é possível ver um exemplo em `Ruby` de implementação deste padrão.

Embora o padrão *Strategy* seja bem adaptado para *features* alternativas, desenvolvedores podem também codificar *features* opcionais. Para este fim, os desenvolvedores fornecem implementações padrão ou modelo de *strategies* para desmarcar a seleção de *features* ou aceitar o valor nulo como parâmetro. Finalmente, combinar múltiplas *features* é possível, se preparado de forma que possa aceitar múltiplos *strategies* e executar todos eles [Apel et al. 2013].

A Tabela 4.5 mapeia as informações obtidas para a estrutura de questões e métricas do GQM.

Tabela 4.5: Implementação de variabilidades com o padrão Strategy

Padrão Strategy	
Questão 1)	
Métricas	M1.1: **Implementar cada *feature* como um *strategie*** M1.2: **Cada *feature* é controlada por uma *strategie* mais abstrata, a qual pode controlar várias *features*. Quando um objeto necessita de um comportamento específico, realiza uma chamada à *strategie* adequada para realizá-lo. Ao adicionar ou remover *strategies*, a variabilidade é alcançada**
Questão 2)	
Métricas	M2.1: **Positiva, Negativa** M2.2: **Classes, Métodos** M2.3: **Variabilidade de Lógica, Variabilidade de Fluxo** M2.4: **Seleção** M2.5: **Execução** M2.6: **Opcional, Alternativa**

4.3 Sumário de Comparação: Teoria Padrões de Projeto

A Tabela 4.6 foi elaborada de forma a sumarizar as características identificadas em cada padrão de projeto analisado.

Tabela 4.6: Tabela dos Elementos de Variabilidade Presentes nos Padrões de Projeto

		Observer	Decorator	Template Method	Strategy
Tipo de Variabilidade (Finalidade)	Positivo	X	X	X	X
	Negativo	X			X
	Opcional			X	
	Alternativo			X	
	Funcional			X	
Unidades de Código	Variáveis				
	Métodos		X	X	X
	Classes	X	X	X	X
Tipo de Variabilidade (Efeito)	Atributo				
	Lógica	X	X	X	X
	Fluxo		X	X	X
	Interface				
Escopo	Binário				
	Seleção			X	X
	Aberto	X	X		
Tipo de Feature	Obrigatório	X	X	X	
	Opcional	X	X	X	X
	Alternativo			X	X
Tempo de Vinculação	Carregamento				
	Ligação				
	Compilação				
	Execução	X	X	X	X
	Pós-Execução				

Análise Padrão Observer

O padrão *Observer* é adequado para *features* obrigatórias e opcionais. Os *subjects* fornecem uma interface padrão onde os *observers* se inscrevem e adicionam funcionalidade. Para retirar funcionalidades, basta cancelar a assinatura de *observers*.

Desta forma, este padrão contempla os tipos positivo e negativo de variabilidade quanto à finalidade. O *Observer* basicamente define como estabelecer relacionamentos entre classes colaborativas, portanto as unidade de código que são alvo dessa solução são as classes.

O tipo de variabilidade, quanto ao efeito, é a variabilidade de lógica, pois os *observers* que são adicionados são interpretados como sendo as diferentes *features* a serem adicionadas. O escopo de seleção é aberto, pois vários *observers* podem ser adicionados ou removidos.

O padrão *Observer* permite que as variabilidades sejam habilitadas em tempo de execução.

Análise Padrão Decorator

O padrão *Decorator* é funcional apenas para *features* opcionais, pois a solução proposta pelo *Decorator* compreende realizar a extensão de funcionalidades de objetos de forma dinâmica, de forma que não há código comum nem código alternativo. Além disso, um objeto pode possuir quantas extensões forem necessárias, possuindo um escopo aberto.

Observando isso, compreende-se que o *Decorator* apenas adiciona funcionalidades, e portanto possui apenas o tipo positivo de variabilidade quanto à finalidade. Entendendo que os *Decorators* são apenas extensões, eles são implementados através de classes, e portanto a unidade de código alvo são as classes. As variabilidades quanto ao efeito que compreende são as variabilidades de lógica e de fluxo, pois o que varia nas extensões são as implementações de métodos.

Este padrão permite que as variabilidades sejam habilitadas no tempo de execução.

Análise Padrão Template Method

O padrão *Template Method* pode ser utilizado principalmente para *features* obrigatórias e alternativas, porquê este padrão é utilizado para fornecer uma implementação padrão de um único algoritmo. Porém, *features* opcionais também podem ser consideradas se existir uma implementação padrão para cada *template method*.

Os tipos de variabilidades quanto à finalidade que este padrão compreende são o positivo, o alternativo e o funcional. Isto porquê quando uma subclasse sobrescreve a implementação padrão do *template method*, ela está adicionando código, modificando a funcionalidade do método ou substituindo o código. Portanto, as unidades de código em que ele opera são as classes e os métodos.

Ao realizar as sobrescritas de métodos, os tipos de variabilidades quanto ao efeito que são possível são a variabilidade de fluxo e variabilidade de lógica. O escopo de variabilidade é de seleção, pois somente uma variante de um algoritmo pode ser selecionada.

Este padrão permite que as variabilidades sejam habilitadas no tempo de execução.

Análise Padrão Strategy

O padrão *Strategy* é bem adaptado para *features* alternativas, uma vez que cada *feature* corresponde a uma implementação diferente de um método. Portanto, as unidades de código em que este padrão opera são os métodos. O *Strategy* também pode ser utilizado para implementar *features* opcionais, porém adaptações técnicas são necessárias.

As variabilidades quanto à sua finalidade que são atendidas pelo *Strategy* são positivas e opcionais. Ao implementar a interface padrão fornecida, está apenas adicionando funcionalidades ou adicionando código. Por este motivo também, o tipo de variabilidade quanto ao seu efeito é somente a variabilidade de fluxo.

O escopo de variabilidade que este padrão possui é o padrão de seleção. Várias implementações da interface padrão podem existir, porém apenas uma pode ser selecionada para ser executada.

Este padrão permite que as variabilidades sejam habilitadas no tempo de execução..

Capítulo 5

Avaliação das Implementações em Ruby

Neste capítulo são apresentados o escopo e planejamento desta avaliação. O projeto dos artefatos desenvolvidos baseando-se na metologia do DSR e a elaboração da estrutura de coleta de dados baseando-se no GQM são apresentados em seguida.

São apresentados e sumarizados também os resultados da coleta de dados sobre as implementações. Uma análise destes resultados é realizada, seguida da avaliação e discussão sobre os dados coletados para cada padrão de projeto implementado em `Ruby`.

5.1 Escopo e Planejamento

Na avaliação realizada neste trabalho, a discussão acerca do gerenciamento de variabilidades foi em relação à fase de implementação da engenharia de domínio, referindo-se à geração de código fonte ou à tradução de um projeto em componentes de *software*. É importante salientar que o gerenciamento de variabilidade de requisitos, de artefatos de projeto e de testes estão fora do escopo desta avaliação, bem como a derivação de produtos através da base de *features* que será desenvolvida (engenharia de aplicação). A estratégia utilizada foi a baseada em componentes, onde cada *feature* é implementada como um elemento independente, e desta forma a implementação não objetiva realizar a etapa de engenharia de aplicação, pois não faz parte dos objetivos implementar e avaliar código que realiza a seleção e ligação entre os componentes e os torna uma aplicação funcional.

O foco é na utilização da linguagem `Ruby` e de seu *framework* `Rails` para realizar implementações dos padrões de projeto *Template Method*, *Decorator*, *Strategy* e *Observer*, para que seja possível coletar de dados acerca das características de implementação que mostrem a

viabilidade técnica de implementar variabilidades em `Ruby` utilizando estes padrões.

Foi realizada uma implementação para cada um destes padrões de projeto, de forma isolada, pois deseja-se obter e avaliar de forma isolada as informações de implementação obtidas a partir cada padrão de projeto implementado em `Ruby`.

A coleta de dados foi elaborada baseando-se na metodologia do GQM, de forma que as questões de pesquisa para este trabalho são baseadas na avaliação dos padrões de projeto realizada no Capítulo 4, na teoria de gerenciamento de variabilidades e de *features* do paradigma de LPS e em características de `Ruby`, e os dados e métricas que respondem às questões definidas na coleta de dados são coletados das implementações.

Para que fosse possível realizar as implementações, foi necessário obter requisitos de algum domínio de aplicação. Estes requisitos foram definidos por um modelo de *features* e contemplam o domínio de *blogs*. Este modelo de *features* possui variabilidades e *features* presentes no domínio de *blogs* e foi definido na Seção 3.1.9.

Para a coleta de dados e avaliação das implementações em `Ruby`, foram coletados dados sobre vários pontos de análise, os quais foram os tipos de *features*, sobre os tipos de variabilidade quanto à finalidade e quanto ao efeito, com quais unidades do código o padrão de projeto funciona, seu escopo de variabilidade e seu tempo de vinculação.

Como foi visto no Capítulo 2, são vários os tipos *features*, de variabilidade, de unidades de código, de escopo de variabilidade e tempo de vinculação existentes. O cenário definido na Seção 3.1.9, a análise dos padrões de projeto feita no Capítulo 4 e algumas características de `Ruby` acabaram por restringir a presença de alguns destes tipos na avaliação das implementações. A Figura 5.1 ilustra as escolhas realizadas em cada um destes pontos de análise.

Como pode ser visto na Figura 5.1, os tipos de *features* considerados foram a obrigatória, a opcional e a alternativa.

Os tipos de variabilidade, quanto à sua finalidade, considerados foram a positiva, a negativa, a opcional, a alternativa e a funcional. Variabilidades de plataforma/ambiente não são obtidas através destes mecanismos e, portanto, não serão considerados.

Os tipos de variabilidade, quanto ao seu efeito, considerados foram a variabilidade de atributo, de lógica, de fluxo e de interface. Variabilidade de persistência não é considerada pois estes padrões de projeto não fornecem formas de gerenciar variabilidades em esquemas físicos

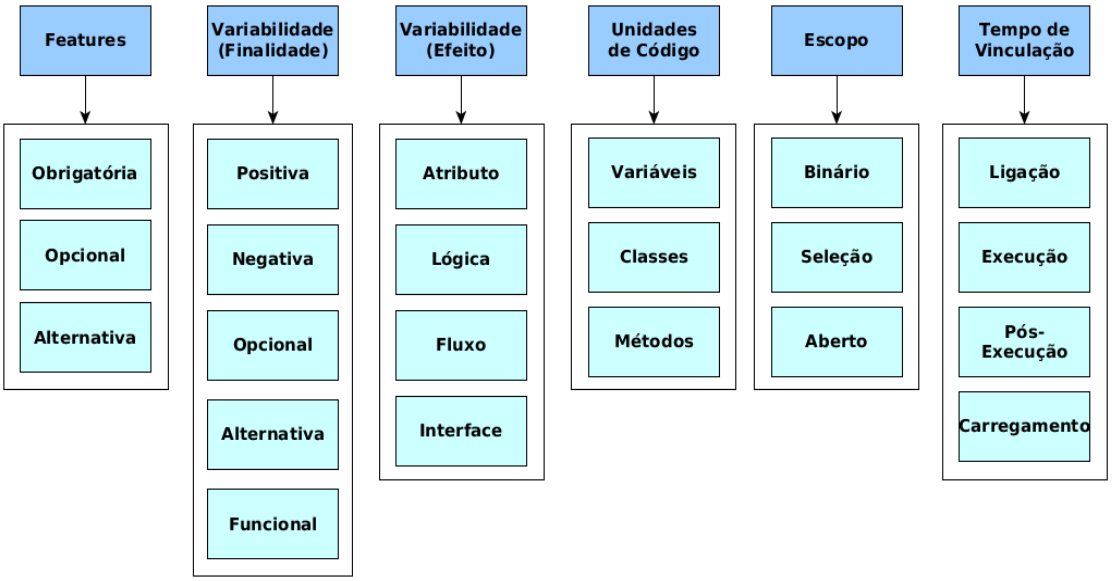

Figura 5.1: Pontos de análise e os tipos selecionados

de banco de dados.

As unidades de código consideradas foram as variáveis, os métodos e as classes.

Os tipos de escopo considerados foram o escopo binário, o escopo de seleção e o escopo aberto.

Os tempos de vinculação considerados foram o tempo de carregamento, de ligação, o tempo de execução e tempo de pós-execução. O tempo de compilação não é considerado porque a linguagem `Ruby` é interpretada, não utilizando de um compilador para poder executar seus códigos fonte.

5.1.1 Projeto de Artefatos Para o DSR

Nesta seção é demonstrado o projeto de artefatos para cada padrão referente ao cenário para as implementações. O projeto de artefatos está relacionado com a etapa de Sugestão do DSR, referindo-se à atividade de desenvolver uma ou mais alternativas de artefato que implementam as variabilidades.

Na sequência, são apresentados diagramas de classe que visam demonstrar os locais onde as variabilidades foram inseridas no código de cada padrão de projeto. As *features* abordadas são as de `Usuário` e `Post`, pois elas contêm as variabilidades que podem ser utilizadas.

Projeto de Artefatos: Observer

Como visto na Seção 4.2.1, as variabilidades podem ser implementadas como *observers* utilizando o padrão *Observer*. A estrutura para as *features* `Usuário` e `Post` podem ser vistas nas Figuras 5.2 e 5.3, respectivamente.

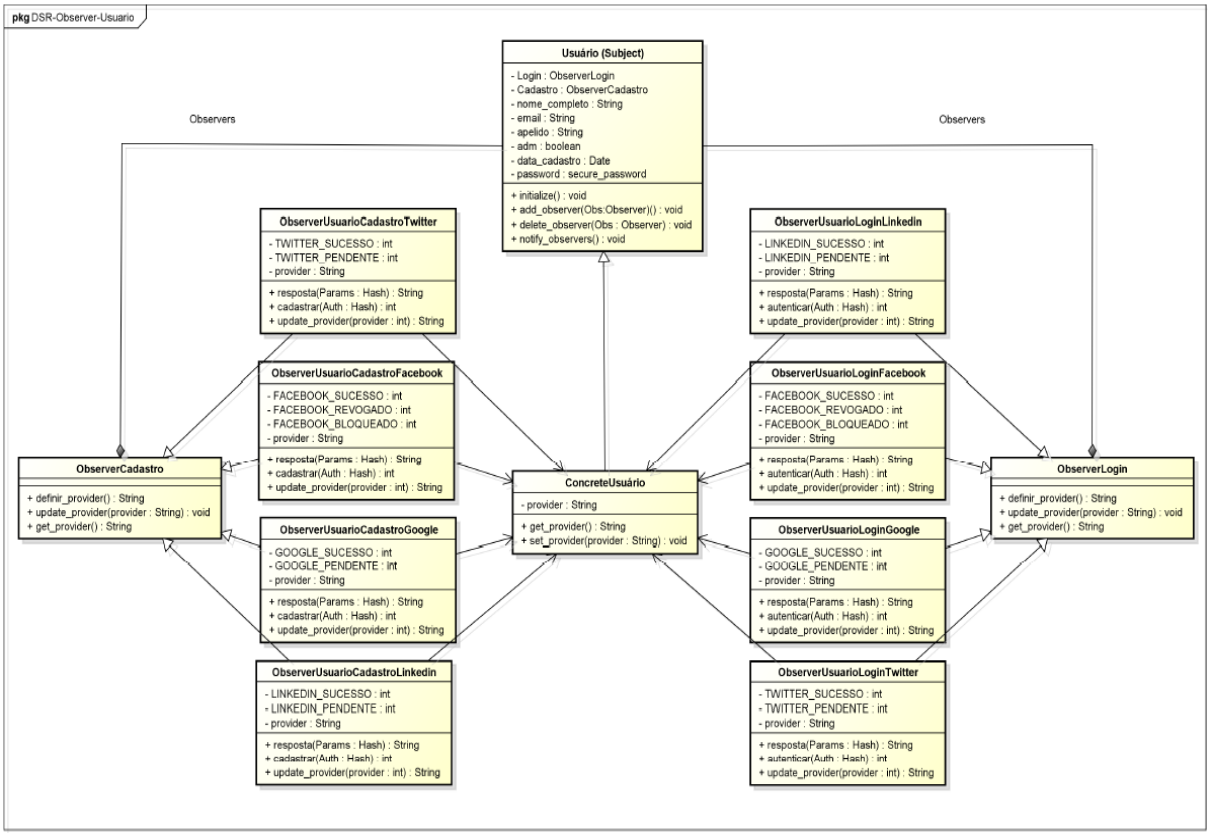

Figura 5.2: Projeto de artefatos do padrão Observer para a *feature* Usuário

Como pode ser visto na Figura 5.2, o *subject* é definido como sendo a classe `Usuário`, a qual possui referências às classes abstratas dos *observers* de login e cadastro. A classe `ConcreteUsuário` armazena os dados que podem ser acessados diretamente pelos *observers*. A variabilidade está na quantidade de *observers* que podem ser anexados ou desanexados das interfaces abstratas de `login` e `cadastro`.

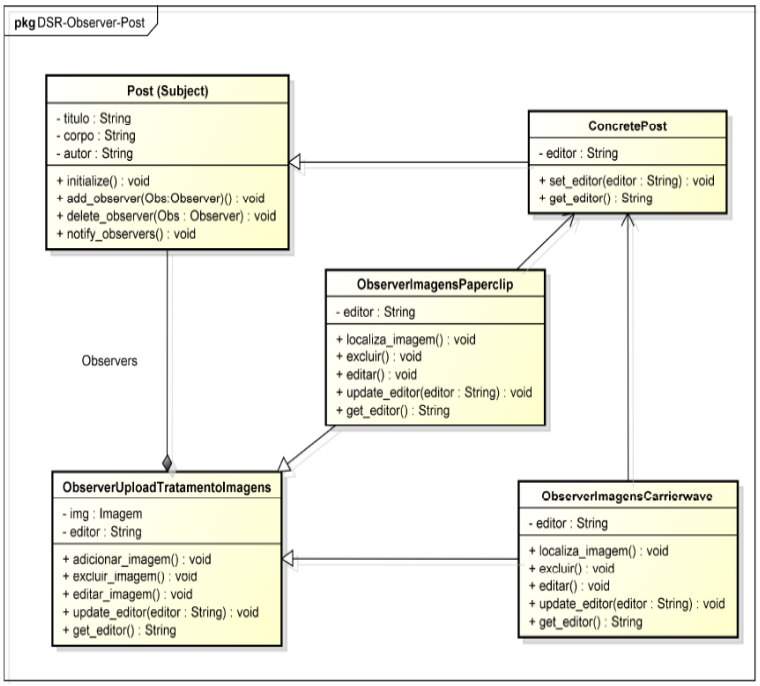

Figura 5.3: Projeto de artefatos do padrão Observer para a *feature* Post

Na Figura 5.3, o *subject* é definido como sendo a classe `Post`, a qual possui uma referência à classe abstrata dos *observers* de `ObserverUploadTratamentoImagens`. A classe `ConcretePost` armazena os dados que podem ser acessados diretamente pelos *observers*. A variabilidade está na quantidade de *observers* que podem ser anexados ou desanexados da interface abstrata de `ObserverUploadTratamentoImagens`.

Projeto de Artefatos: Decorator

Como visto na Seção 4.2.2, as variabilidades podem ser implementadas como *decorators* utilizando o padrão *Decorator*. A estrutura para as *features* `Usuário` e `Post` podem ser vistas nas Figuras 5.4 e 5.5, respectivamente.

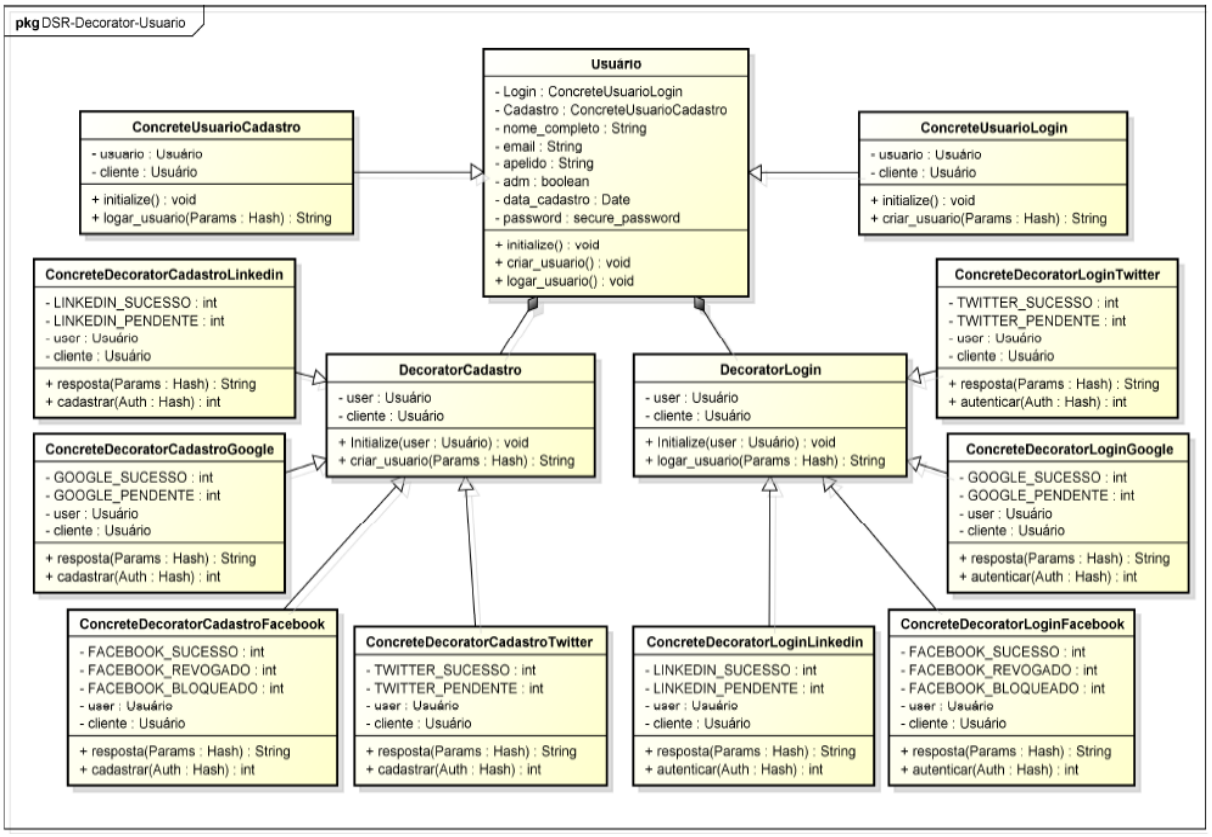

Figura 5.4: Projeto de artefatos do padrão Decorator para a *feature* Usuário

Observando a Figura 5.4, é possível ver que a classe `Usuário` é o componente que possui uma interface para os *decorators* adiconarem funcionalidades. As classes `ConcreteUsuarioCadastro` e `ConcreteUsuarioLogin` são componentes nos quais os *decorators* são adicionados. As classes `DecoratorCadastro` e `DecoratorLogin` possuem referências à classe `Usuário` e define uma interface que se comunica com a interface de `Usuário`, sendo que os *concrete decoratos* são as classes que efetivamente adicionam funcionalidades. A variabilidade está em anexar e desanexar os *decorators*.

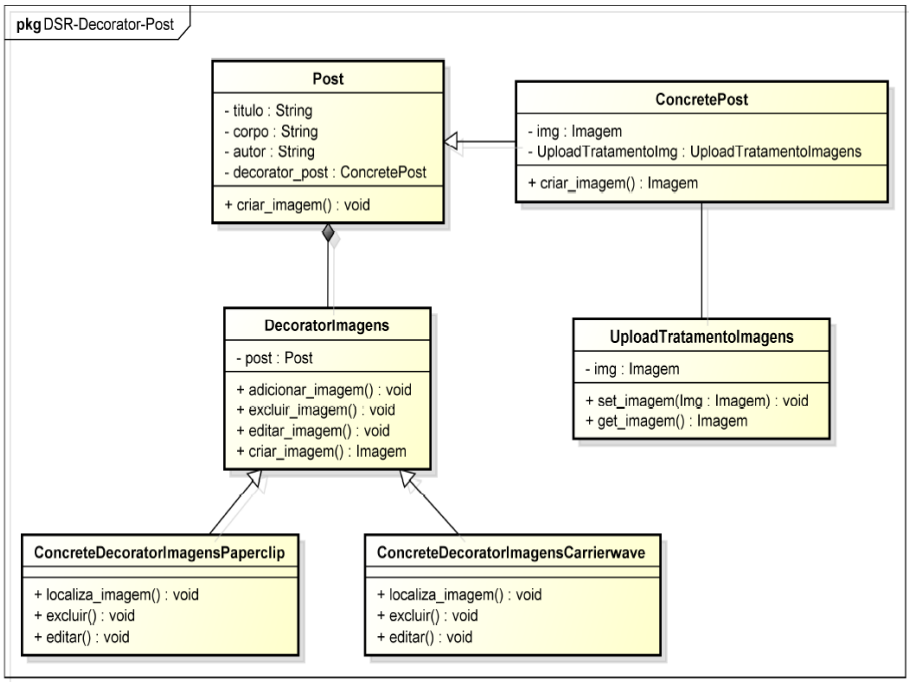

Figura 5.5: Projeto de artefatos do padrão Decorator para a *feature* Post

A Figura 5.5 mostra a classe `Post` como sendo um componente que possui uma interface para os *decorators* adiconarem funcionalidades. A classe `ConcretePost` possui relação com a classe `UploadTratamentoImagens`, na qual serão adicionadas as funcionalidades. A classe `DecoratorImagens` possui referência à classe `Post` e define uma interface que se comunica com a interface de `Post`, sendo que os *concrete decoratos* são as classes que efetivamente adicionam funcionalidades. A variabilidade está em anexar e desanexar os *decorators*.

Projeto de Artefatos: Template Method

Como visto na Seção 4.2.3, as variabilidades podem ser implementadas como *template methods* utilizando o padrão *Template Method*. A estrutura para as *features* `Usuário` e `Post` podem ser vistas nas Figuras 5.6 e 5.7, respectivamente.

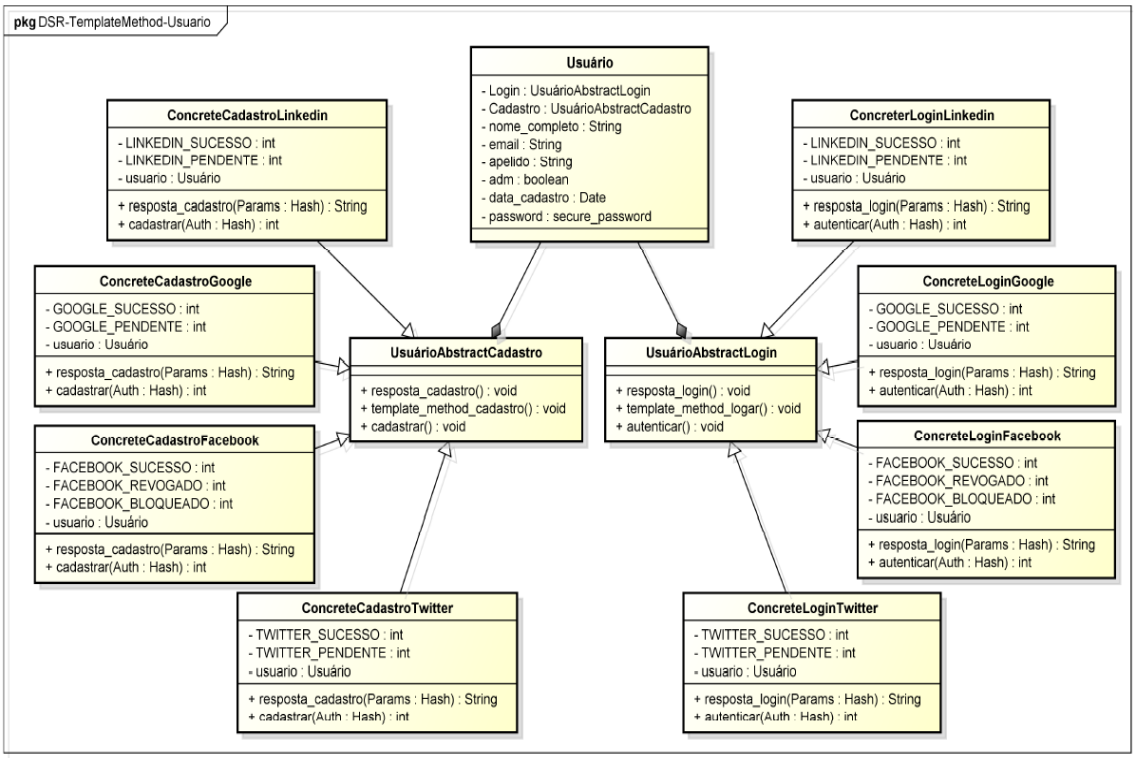

Figura 5.6: Projeto de artefatos do padrão Template Method para a *feature* Usuário

Na Figura 5.6 observa-se que a classe `Usuário` possui referências às classes abstratas `UsuarioAbstractCadastro` e `UsuarioAbstractLogin`, nas quais são definidos as estruturas dos métodos sobrescritos nas subclasses. A variabilidade está na sobrescrita dos métodos de diferentes maneiras, de forma que cada classe implementa um comportamento diferente que será selecionado quando necessário.

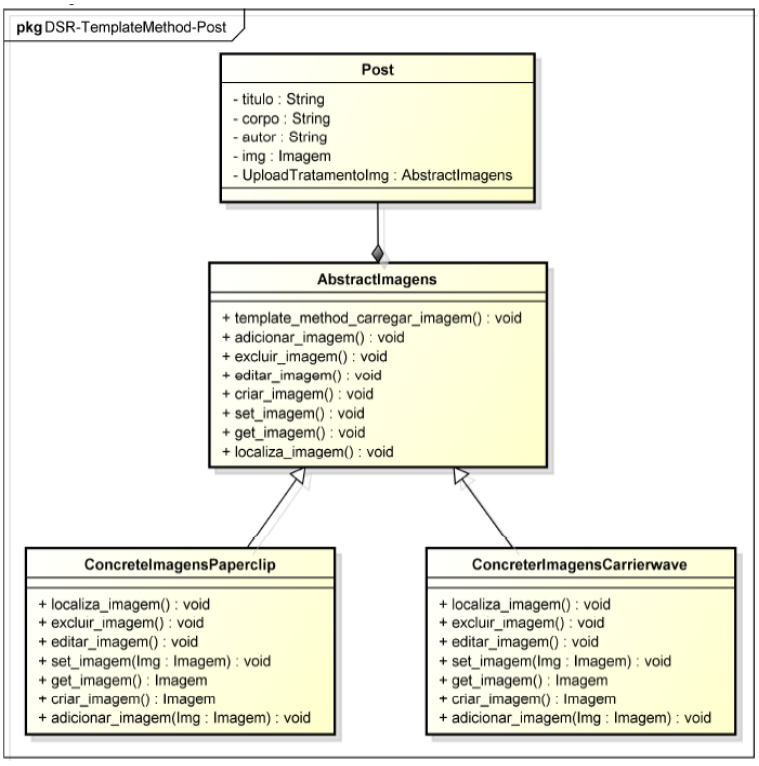

Figura 5.7: Projeto de artefatos do padrão Template Method para a *feature* Post

A Figura 5.7 mostra que a classe `Post` possui referência à classe abstrata `AbstractImagens`, na qual é definido as estruturas dos métodos sobrescritos nas subclasses. A variabilidade está na sobrescrita dos métodos de diferentes maneiras, de forma que cada classe implementa um comportamento diferente que será selecionado quando necessário.

Projeto de Artefatos: Strategy

Como visto na Seção 4.2.4, as variabilidades podem ser implementadas como *strategies* utilizando o padrão *Strategy*. A estrutura para as *features* `Usuário` e `Post` podem ser vistas nas Figuras 5.8 e 5.9, respectivamente.

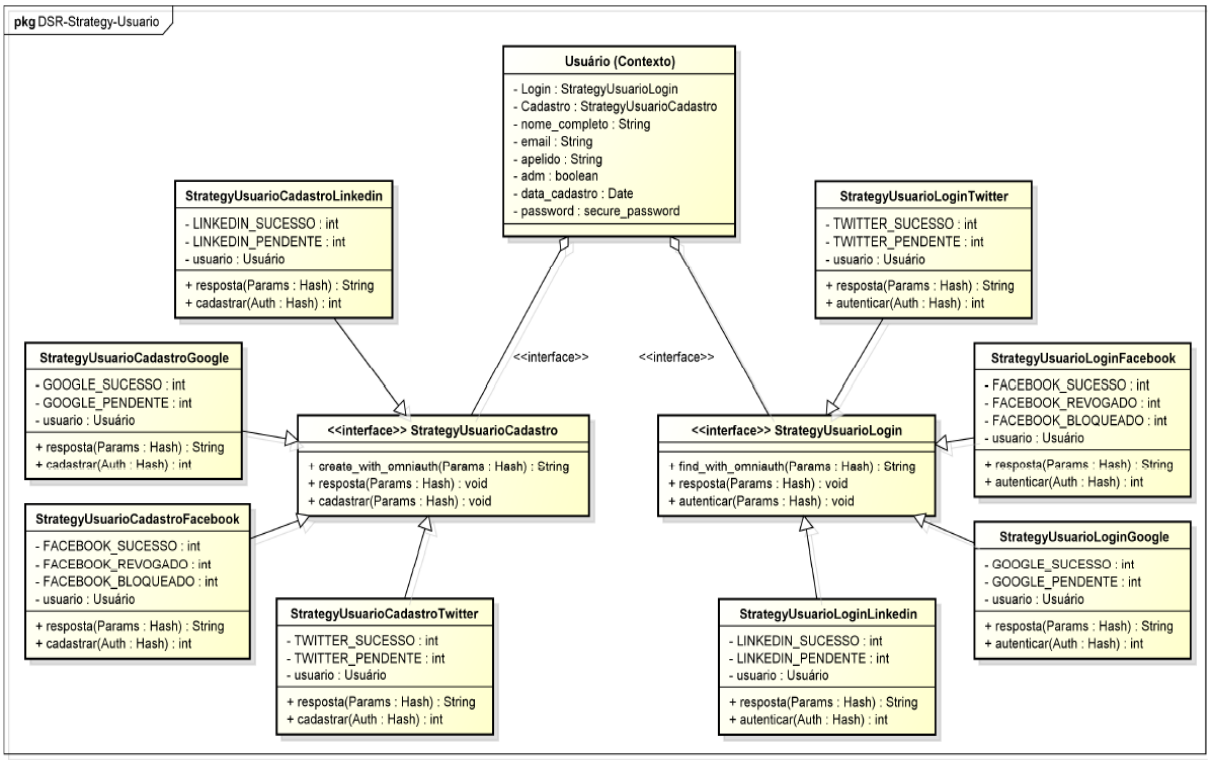

Figura 5.8: Projeto de artefatos do padrão Strategy para a *feature* Usuário

Observa-se na Figura 5.8 que o contexto é a classe `Usuário`, a qual possui referências às interfaces `StrategyUsuarioCadastro` e `StrategyUsuarioLogin`. Estas interfaces são implementadas pelas respectivas *strategies*, fornecendo os comportamentos necessários. A variabilidade é alcançada ao implementar ou eliminar *strategies*.

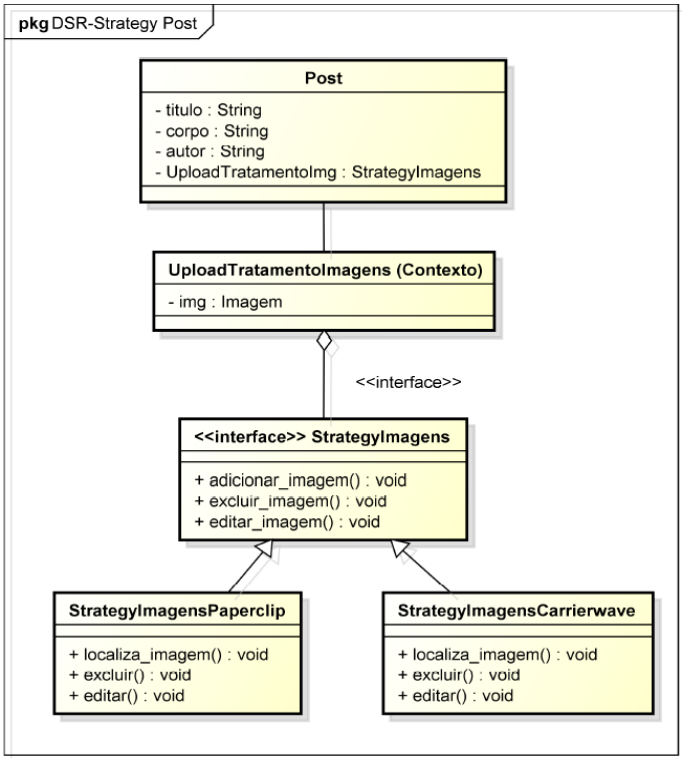

Figura 5.9: Projeto de artefatos do padrão Strategy para a *feature* Post

A Figura 5.9 mostra que o contexto é a classe `UploadTratamentoImagens`, a qual está relacionada à classe `Post`, possuindo referência à interface `StrategyImagens`. Esta interface é implementada pelas respectivas *strategies*, fornecendo os comportamentos necessários. A variabilidade é alcançada ao implementar ou eliminar *strategies*.

5.1.2 Processo de Implementação

O processo de implementação está relacionado à etapa de Desenvolvimento do DSR. Como estratégia de implementação, o cenário foi todo implementado inicialmente sem inserir algum padrão de projeto, de forma a construir o ambiente interno dos artefatos.

Cada *feature* foi implementada como um módulo (ou componente) independente. Após o desenvolvimento desta base, cada padrão de projeto foi inserido individualmente na base através de refatorações de código. Como as refatorações são baseadas em ciclos, ao fim de cada ciclo foi possível obter informações de implementação, de forma que isto se adequou à metodologia proposta pelo DSR, na qual é realizado o desenvolvimento baseado em ciclos e geração de conhecimento/informações através do desenvolvimento.

A definição de refatoração pode ser dividida em três pontos principais. O primeiro refere-se à melhora do projeto existente, porém não se pode confundir melhora com correções de erros. Quanto menor forem os passos para realizar as mudanças, melhor, pois menos erros ocorrerão e será mais fácil gerenciar as mudanças. Ao realizar as mudanças, deve-se evitar de deixar que funcionalidades que estavam funcionando antes parem de funcionar depois das alterações [Fowler 1999].

Existem várias técnicas de refatoração que podem ser aplicadas a fim de obter uma melhoria ou mudança desejada no código. Quando um código é refatorado seguindo estas técnicas, resulta em um código que possui manutenção facilitada. Na sequência, serão apresentadas algumas destas técnicas, as quais foram utilizadas neste trabalho [Fowler 1999]:

- Renomear Método: refere-se a modificar o nome de um método com objetivo único de fazer com que o novo nome represente melhor a funcionalidade implementada no método;
- Extrair Método: é utilizada quando é necessário dividir um método que possui mais de uma responsabilidade;
- Mover Método: é utilizado quando um método usa mais informações de outra classe do que de sua própria. Refere-se a uma mudança de contexto do método;
- Mover Campo: é utilizado quando um atributo é mais utilizado em outra classe do que na sua própria. Realizar esta mudança de contexto do atributo garante que ele fique protegido de mudanças externas;
- Extrair Classe: é utilizado quando uma classe possui mais de uma responsabilidade, sendo necessário dividi-la para reduzir sua complexidade.

O uso de refatorações justifica o projeto de artefatos realizados anteriormente, o qual colaborou na definição de quais locais as mudanças poderiam ser realizadas e as variabilidades inseridas, em relação às *features* definidas no cenário. As refatorações também justificam a forma de coleta de dados utilizada, pois informações são geradas ao fim de cada ciclo e podem ser utilizadas para ajudar a interpretar questões mais abrangentes.

5.1.3 Elaboração do Modelo de Coleta de Dados das Implementações Baseado no GQM

Para que as implementações dos mecanismos com base no modelo de *features* pudessem ser avaliadas, é utilizada a metodologia do GQM como base para guiar a coleta e análise dos dados das implementações.

As questões de pesquisa para este trabalho são baseadas na avaliação dos padrões de projeto realizada no Capítulo 4, na teoria de gerenciamento de variabilidades e de *features* do paradigma de LPS, e em características de `Ruby`.

As métricas definidas no modelo são dados que são coletados das implementações do modelo de *features*, de forma a responderem às questões de pesquisa, fornecendo informações e contexto para a avaliação de cada padrão de projeto implementado em `Ruby on Rails`.

O GQM é aplicado neste contexto com o objetivo de estruturar as questões de pesquisa e ajudar a definir as métricas/dados respectivos a cada questão, os quais são coletados das implementações.

Baseando-se na estrutura proposta pela abordagem do GQM, foram elaborados o escopo e objetivos de coleta de dados, as questões e suas respectivas métricas. O escopo e objetivos foram baseados nos objetivos deste trabalho. As questões foram elaboradas tendo como base os pontos de análise definidos na Seção 1.2.2 e nos dados obtidos da análise dos padrões de projeto (ver Seção 4.3 para o sumário dos dados obtidos desta análise). As métricas desenvolvidas foram baseadas na presença dos tipos de *features*, dos tipos de variabilidade (finalidade e efeito), das unidades de código, do tipo de escopo de variabilidade e do tempo de vinculação.

Este modelo baseado no GQM assume um formato semelhante a um questionário, e pode ser visualizado na Tabela 5.1.

Observando a Tabela 5.1, o 'X' presente nas questões refere-se aos padrões de projeto que são avaliados. Como cada um deles é implementado e avaliado individualmente, na avaliação o 'X' assume o nome do padrão de projeto que está sendo avaliado em determinado momento. Os rótulos *Q1, Q2, Q3* e *Q4* apontam para as questões de pesquisa, e os rótulos *M1.1, M1.2, M2.1, ..., Mi.j* apontam para as métricas que são respectivas às questões de pesquisa.

O nível conceitual foi definido como sendo o objetivo de avaliar a viabilidade técnica de implementar variabilidades em `Ruby`, através da especificação em código dos mecanismos de

Tabela 5.1: Tabela do modelo de coleta de dados das implementações elaborado com base no GQM

Objetivo	
Propósito	Avaliar Viabilidade Técnica
Assunto	Especificação em Código
Objeto (processo)	Mecanismo de Padrões de Projeto para Implementação de Variabilidades
Ponto de vista	Programador Interessado em Desenvolver Variabilidades de Software em `Ruby`
Questão 1)	Q1: Quais as características de `Ruby` foram utilizadas para implementar o padrão de projeto 'X'?
Métricas	M1.1: Número de características que foram utilizadas M1.2: Nome das características utilizadas M1.3: Por que estas características foram utilizadas
Questão 2)	Q2: Como o padrão de projeto 'X' foi implementado em `Ruby`?
Métricas	M2.1: Verificar se foram necessárias adaptações técnicas M2.2: Número de adaptações técnicas utilizadas (somente se foi necessário utilizar adaptações técnicas) M2.3: Descrever as adaptações técnicas utilizadas e por quê foram utilizadas (somente se foi necessário utilizar adaptações técnicas) M2.4: Verificar se houveram problemas ao utilizar o mecanismo e se os problemas impossibilitaram a implementação M2.5: Verificar se foi necessário utilizar ferramentas externas ao `Ruby` e, se sim, por que foram utilizadas:
Questão 3)	Q3: `Ruby` apresenta outras características que podem ser utilizadas para implementar o padrão de projeto 'X'?
Métricas	M3.1: Número de características que podem ser utilizadas M3.2: Nome das características que podem ser utilizadas M3.3: Por que estas características podem ser utilizadas
Questão 4)	Q4: O padrão de projeto 'X' implementado em `Ruby` atendeu aos elementos de variabilidade conforme a teoria (variabilidade quanto à finalidade, unidades de código, variabilidade quanto ao efeito, escopo de variação, tempo de vinculação, features)?
Métricas	M4.1: Variabilidade quanto à finalidade M4.2: Unidades de código M4.3: Variabilidade quanto ao efeito M4.4: Escopo de variação M4.5: Tempo de vinculação M4.6: Feature

variabilidade.

No nível operacional foram definidas as questões a serem respondidas, abrangendo questões como quais características de `Ruby` foram ou não foram utilizadas nas implementações, como os mecanismos foram implementados em `Ruby` e se os mecanismos atenderam aos elementos de variabilidade. Estas questões visam obter dados para dos pontos de análise definidos na Subseção 1.2.2.

No nível quantitativo foram definidas as métricas que se deseja medir e as informações que se deseja coletar, associadas às respectivas questões de pesquisa. Sobre as métricas, grande parte delas são qualitativas e possuem um conjunto de respostas pontuais.

O processo para coleta e avaliação dos dados se baseia em uma abordagem mista de métricas quantitativas e qualitativas, sendo que ambos os tipos de dados foram coletados.

5.1.4 Resultados e Discussão

Nesta seção são apresentados os resultados da aplicação do modelo baseado no GQM nas implementações em `Ruby` dos padrões de projeto *Template Method*, *Decorator*, *Strategy* e *Observer*.

Ao implementar o cenário de blogs com os padrões de projeto, realizando os ciclos de refatorações juntamente com os ciclos do DSR, algumas informações acerca das implementações puderam ser obtidos, os quais foram analisadas traçando um paralelo com as características teóricas dos padrões de projeto. O resultado desta comparação referencia diretamente aos pontos de análise identificados inicialmente na pesquisa.

Resultados Padrão Observer

Tabela 5.2: Tabela dos resultados para o padrão Observer

Padrão Observer	
Questão 1)	
Métricas	M1.1: **1** M1.2: **Orientação a objetos** M1.3: **O Observer é um padrão orientado a objeto, facilitando o uso desta característica**
Questão 2)	
Métricas	M2.1: **Não** M2.2: **Não foram necessárias adaptações técnicas** M2.3: **Não foram necessárias adaptações técnicas** M2.4: **Não houveram problemas** M2.5: **Não foram necessárias ferramentas externas**
Questão 3)	
Métricas	M3.1: **2** M3.2: **Mixins, Gems** M3.3: **Tanto os Mixins quanto as Gems podem ser utilizados de forma combinada com a orientação a objetos para implementar os observers de uma forma que eles sejam módulos independentes**
Questão 4)	
Métricas	M4.1: **Positiva, Negativa** M4.2: **Classes** M4.3: **Variabilidade de Lógica** M4.4: **Escopo Aberto** M4.5: **Execução** M4.6: **Obrigatório, Opcional**

O padrão *Observer* utiliza conceitos da orientação a objetos, e portanto é natural implementá-lo seguindo esta característica. É importante relembrar que `Ruby` suporta outros paradigmas de programação também.

Como é possível perceber analisando a Tabela 5.2, os elementos de variabilidade permaneceram inalterados em relação aos elementos identificados na Tabela 4.2 da Subseção 4.2.1. Estes dados podem ser observados em código no Quadro 5.1.

Observando o código do Quadro 5.1, prova-se que a variabilidade quanto a sua finalidade que é possível de ser obtida com este padrão em `Ruby` é positiva e negativa, pois aos adicionar ou remover *observers* ocorre uma adição ou remoção de funcionalidades, respectivamente.

As unidades de código que o padrão utiliza para fornecer a variabilidade são as classes, pois

Quadro 5.1 Exemplo de variabilidades com o padrão Observer em Ruby - Feature Usuário

Classe Usuario - Subject

```
class Usuario < ActiveRecord::Base
    has_secure_password
    has_many :posts
    has_many :comentarios
    attr_writer :atualiza_provider

    ... #validações de variáveis

    def initialize
        @observers = []
    end

    def add_observer(observer)
        @observers « observer
    end

    def delete_observer(observer)
        @observers.delete(observer)
    end

    def notify_observers
        @observers.each do |observer|
            observer.update(self)
        end
    end

    def self.find_or_create_with_omniauth(auth)
        @atualiza_provider = auth.provider
        notify_observers
    end
end
```

Classe SubjectUsuario - ConcreteSubject

```
class SubjectUsuario
    attr_accessor :provider

    #o atributo @provider é o estado de interesse para os observers
    #o comando attr_accessor define métodos do tipo 'get' e 'set'
    #para a variável @provider
end
```

Classe ObserverRedesSociais - Observer

```
class ObserverRedesSociais
    #O seguinte método define qual rede social será utilizada no login
    def definir_provider
        if @provider == "facebook"
            ObserverFacebook.resposta(params)
        elsif @provider == "twitter"
            ObserverTwitter.resposta(params)
        elsif @provider == "google+"
            ObserverGoogle.resposta(params)
        elsif @provider == "linkedin"
            ObserverLinkedin.resposta(params)
        end
    end

    def provider=(provider)
        @provider = provider
        definir_provider
    end

    def provider
        @provider
    end
end
```

Classe ObserverFacebook - ConcreteObserver

```
#O exemplo dado é em relação ao login por Facebook; de forma semelhante é realizado
#o login por Google, Twitter e LinkedIn, mudando as verificações dos parâmetros de
#sucesso e bloqueio
class ObserverFacebook
    FACEBOOK_SUCESSO = 200
    FACEBOOK_REVOGADO = 403
    FACEBOOK_BLOQUEADO = 408
    attr_writer :provider

    def self.resposta(params)
        ... #definição do corpo do método
    end

    def self.autenticar(auth)
        ... #definição do corpo do método
    end
end
```

cada variante é inserida em uma classe distinta.

A variabilidade quanto ao efeito é a variabilidade de lógica, pois o que muda entre os *observers* é a lógica de seus métodos.

O escopo de variabilidade é aberto pois é possível adicionar ou remover quantas variantes forem necessárias no objeto que as necessita.

O tempo de vinculação é de execução, isto é, os *observers* são inseridos ou removidos durante a execução da aplicação.

Os tipos de *features* possível de se utilizar são as obrigatórias e opcionais, porque o comportamento padrão das *features* obrigatórias pode ser inserido nos *subjects* e o comportamento opcional pode ser inserido nos *concrete observers*.

Caso fossem utilizados `Mixins` ou `Gems`, seria possível implementar os *observers* como módulos (interpretados como sendo as *features*). Porém, utilizar estas características tornaria possível os tipos de variabilidade opcional e alternativo, pois adicionar ou remover módulos significa adicionar ou remover código de programação.

Resultados Padrão Decorator

Tabela 5.3: Tabela dos resultados para o padrão Decorator

Padrão Decorator	
Questão 1)	
Métricas	M1.1: **1** M1.2: **Orientação a objetos** M1.3: **O padrão Decorator é baseado em delegação, o qual é um conceito da orientação a objetos**
Questão 2)	
Métricas	M2.1: **Não** M2.2: **Não foram necessárias adaptações técnicas** M2.3: **Não foram necessárias adaptações técnicas** M2.4: **Não houveram problemas** M2.5: **Não foram necessárias ferramentas externas**
Questão 3)	
Métricas	M3.1: **1A** M3.2: **Mixins** M3.3: **Os Mixins podem ser utilizadas para fornecer implementações concretas de um decorator em forma de módulos independentes**
Questão 4)	
Métricas	M4.1: **Positiva** M4.2: **Classes, Métodos** M4.3: **Variabilidade de Lógica, Variabilidade de Fluxo** M4.4: **Aberto** M4.5: **Execução** M4.6: **Opcional, Obrigatório**

O padrão *Decorator* é um padrão baseado em delegação, e portanto utilizar características da orientação a objetos para implementá-lo é o caminho mais natural.

Na Tabela 5.3 observa-se que os elementos de variabilidade continuam os mesmos em relação aos identificados na teoria, como pode ser conferido se comparado com a Tabela 4.3 da Subseção 4.2.2. Estes dados podem ser observados em código no Quadro 5.2.

No código do Quadro 5.2, observa-se que a variabilidade quanto a sua finalidade que é possível de ser obtida com este padrão em `Ruby` é positiva, pois aos adicionar *decorators* ocorre uma adição funcionalidades, não havendo possibilidade de remover estas funcionalidades.

As unidades de código que o padrão utiliza para fornecer a variabilidade são as classes e os métodos, pois cada variante é inserida em uma classe distinta, e outras classes são utilizadas

Quadro 5.2 Exemplo de variabilidades com o padrão Decorator em Ruby - Feature Usuário

Classe Usuario - Component)

```
class Usuario < ActiveRecord::Base
    has_secure_password
    has_many :posts
    has_many :comentarios
    attr_accessor :tipo_operacao

    ... #validações de variáveis

    def initialize (tipo_op)
        @tipo_operacao = tipo_op
    end

    def criar_usuario (params) #esqueleto de método
    end

    def logar_usuario (params) #esqueleto de método
    end
end
```

Classe ConcreteUsuarioLogin - ConcreteComponent

```
class ConcreteUsuarioLogin < Usuario
    attr_reader :usuario
    attr_accessor :cliente

    def initialize (user)
        @usuario = user
    end

    def logar_usuario (params)
        @cliente = Authorization.find_by_provider_and_uid(params.provider, params.uid)
    end
end
```

Classe ConcreteUsuarioCadastro - ConcreteComponent

```
class ConcreteUsuarioCadastro < Usuario
    attr_reader :usuario
    attr_accessor :cliente

    def initialize (user)
        @usuario = user
    end
```

```
    def criar_usuario (params)
        @cliente = Authorization.create_by_provider_and_uid(params.provider, params.uid)
    end
end

#O exemplo dado é em relação ao login por Facebook; de forma semelhante é realizado
#o login por Google, Twitter e LinkedIn, mudando as verificações dos parâmetros de
#sucesso e bloqueio
```

Classe DecoratorLogin

```
class DecoratorLogin
    attr_reader :usuario #referencia à classe Usuario
    attr_accessor :cliente #referencia à classe Usuario

    def initialize (user)
        @usuario = user
    end

    def self.logar_usuario(params, cliente) #esqueleto de método
    end
end
```

Classe DecoratorFacebook - ConcreteDecorator

```
class DecoratorFacebook < Decorator
    FACEBOOK_SUCESSO = 200
    FACEBOOK_REVOGADO = 403
    FACEBOOK_BLOQUEADO = 408
    attr_reader :usuario
    attr_accessor :cliente

    def initialize (user)
        @usuario = user
    end

    def self.resposta_facebook(params, cliente)
        ... #definição do corpo do método
    end

    def self.autenticar_facebook(params, cliente)
        ... #definição do corpo do método
    end
end
```

para realizar a comunicação entre o código variável e o código padrão, sendo que estas classes são interfaces que definem os esqueletos dos métodos. Os *concrete decorators* sobrescrevem estes métodos, podendo eles possuir implementações distintas e ordem de chamadas de funções diferentes.

A variabilidade quanto ao efeito é a variabilidade de fluxo e de lógica. Como as classes `Usuário` e `Decorator` fornecem interfaces para ser implementadas por subclasses, pode-se obter variabilidade de fluxo ao modificar a ordem de chamadas de métodos. Outra variação existente entre os *decorators* é a lógica de seus métodos, sendo que cada *decorator* pode implementar os métodos de acordo com suas necessidades, fornecendo assim a variabilidade de lógica.

O escopo de variabilidade é aberto pois é possível adicionar quantas *decorators* forem necessárias no objeto que os necessita.

O tempo de vinculação é de execução pois os *decorators* são inseridos ou removidos durante a execução da aplicação.

Os tipos de *features* possível de se utilizar são as obrigatórias e opcionais, porque o comportamento padrão das *features* obrigatórias pode ser inserido nos *components* e o comportamento opcional pode ser inserido nos *concrete decorators*.

Utilizar os `Mixins` do `Ruby` torna possível que as implementações concretas de *decorators* sejam realizadas como módulos, porém é necessário desenvolver código que realize a ligação entre o componente feito com os `Mixins` e a interface.

Resultados Padrão Template Method

Tabela 5.4: Tabela dos resultados para o padrão Template Method

Padrão Template Method	
Questão 1)	
Métricas	M1.1: **1** M1.2: **Orientação a objetos** M1.3: **Porque é um padrão puramente baseado em herança**
Questão 2)	
Métricas	M2.1: **Não** M2.2: **Não foram necessárias adaptações técnicas** M2.3: **Não foram necessárias adaptações técnicas** M2.4: **Não houveram problemas** M2.5: **Não foram necessárias ferramentas externas**
Questão 3)	
Métricas	M3.1: **0** M3.2: **Não há outras características que possam ser utilizadas** M3.3: **Não há outras características que possam ser utilizadas**
Questão 4)	
Métricas	M4.1: **Positiva, Opcional, Alternativa, Funcional** M4.2: **Classes, Métodos** M4.3: **Variabilidade de Lógica, Variabilidade de Fluxo** M4.4: **Seleção** M4.5: **Execução** M4.6: **Obrigatória, Opcional, Alternativa**

O padrão *Template Method* é um padrão baseado em heranças simples. Por este motivo, adequa-se bem ao `Ruby`, pois a linguagem suporta apenas heranças simples.

Caso fosse necessário utilizar heranças múltiplas, seria necessário utilizar de uma adaptação técnica: utilizar os `Mixins` de `Ruby` para gerar módulos que extendessem as funcionalidades das classes. Além disso, não há formas alternativas de realizar a implementação deste mecanismo.

Como pode ser observado na Tabela 5.4, os elementos de variabilidade não mudam em relação à teoria, comparando com a Tabela 4.4 da Subseção 4.2.3. Estes dados podem ser vistos no código apresentado no Quadro 5.3.

No código do Quadro 5.3 observa-se que a variabilidade quanto a sua finalidade que é possível de ser obtida com o padrão *Template Method* `Ruby` é positiva, alternativa e funcional. É positiva pois é possível adicionar funcionalidades através das variantes, alternativa porque o

Quadro 5.3 Exemplo de variabilidades com o padrão Template Method em Ruby - Feature Usuário

Classe Usuário - Define as variáveis do modelo e mantém referências às classes abstratas

```
class Usuario < ActiveRecord::Base
    has_secure_password
    has_many :posts
    has_many :comentarios
    attr_accessor :abstract_login, :abstract_cadastro

    ... #verificações de variáveis
end
```

Classe UsuarioAbstractCadastro - AbstractClass

```
class UsuarioAbstractCadastro
    def self.resposta_cadastro(r) #esqueleto de método
    end

    def self.cadastrar(params) #esqueleto de método
    end

    #esqueleto de método
    def self.template_method_cadastro(params)
        r = cadastrar(params)
        resposta_cadastro(r)
    end
end
```

Classe UsuarioAbstractLogin - AbstractClass

```
class UsuarioAbstractLogin
    def self.resposta_login(r) #esqueleto de método
    end

    def self.cadastrar(params) #esqueleto de método
    end

    #esqueleto de método
    def self.template_method_login(params)
        r = autenticar(params)
        resposta_login(r)
    end
end
```

```
#O exemplo dado é em relação ao login e cadastro por Facebook; de forma semelhante
#é realizado o login e cadastro por Google, Twitter e LinkedIn, mudando as verificações
#dos parâmetros de sucesso e bloqueio
Classe ConcreteLoginFacebook - ConcreteClass
class ConcreteLoginFacebook < UsuarioAbstractLogin
    FACEBOOK_SUCESSO = 200
    FACEBOOK_REVOGADO = 403
    FACEBOOK_BLOQUEADO = 408
    attr_reader :usuario

    def initialize (user)
        @usuario = user
    end

    def self.resposta_login(params)
        ... #definição do corpo do método
    end

    def self.autenticar(params)
        ... #definição do corpo do método
    end
end

Classe ConcreteCadastroFacebook - ConcreteClass
class ConcreteCadastroFacebook < UsuarioAbstractCadastro
    FACEBOOK_SUCESSO = 200
    FACEBOOK_REVOGADO = 403
    FACEBOOK_BLOQUEADO = 408
    attr_reader :usuario

    def initialize (user)
        @usuario = user
    end

    def self.resposta_cadastro(params)
        ... #definição do corpo do método
    end

    def self.cadastrar(params)
        ... #definição do corpo do método
    end
end
```

código da *feature* inserida é substituído e funcional porque a funcionalidade pode mudar.

As unidades de código que o padrão utiliza para fornecer a variabilidade são as classes e os métodos, pois são utilizadas classes abstratas para definir o esqueleto dos métodos, e cada subclasse implementa os métodos da forma que necessita. Os método podem possuir chamadas a outros métodos, sendo possível existir variações na ordem de realização destas chamadas. Cada subclasse implementada é interpretada como uma variante.

A variabilidade quanto ao efeito é a variabilidade de fluxo e de lógica. Como as classes `UsuárioAbstractCadastro` e `UsuarioAbstractLogin` fornecem interfaces para ser implementadas pelas subclasses, pode-se obter variabilidade de fluxo ao modificar a ordem de chamadas de métodos. Outra variação existente entre os *template methods* é a lógica de seus métodos, sendo que cada *template method* pode implementar os métodos de acordo com suas necessidades, fornecendo assim a variabilidade de lógica.

O escopo de variabilidade é seleção pois apenas um *template method* pode ser executado por vez.

O tempo de vinculação é de execução pois os *template methods* são inseridos ou removidos durante a execução da aplicação.

Os tipos de *features* possíveis de se utilizar são as obrigatórias, opcionais e alternativas. O comportamento padrão das *features* obrigatórias pode ser inserido entre as chamadas de métodos nos esqueletos de métodos definidos nas classes `UsuárioAbstractCadastro` e `UsuarioAbstractLogin`. É possível obter *features* opcionais ou alternativas, sendo que isto depende da forma que a seleção de *features* for construída.

Resultados Padrão Strategy

Tabela 5.5: Tabela dos resultados para o padrão Strategy

Padrão Strategy	
Questão 1)	
Métricas	M1.1: **1** M1.2: **Orientação a objetos** M1.3: **É um padrão de projeto que usa delegação**
Questão 2)	
Métricas	M2.1: **Não** M2.2: **Não foram necessárias adaptações técnicas** M2.3: **Não foram necessárias adaptações técnicas** M2.4: **Não houveram problemas** M2.5: **Não foram necessárias ferramentas externas**
Questão 3)	
Métricas	M3.1: **2** M3.2: **Mixins e Gems** M3.3: **É possível utilizar Mixins e Gems para implementar as strategies como módulos**
Questão 4)	
Métricas	M4.1: **Positiva, Negativa** M4.2: **Classes, Métodos** M4.3: **Variabilidade de Lógica, Variabilidade de Fluxo** M4.4: **Seleção** M4.5: **Execução** M4.6: **Opcional, Alternativa**

O padrão *Strategy* é baseado em delegação. Desta forma, ele fornece uma interface que é implementada pelos seus clientes.

Caso seja utilizado `Mixins` ou `Gems` como alternativa de implementação, será necessário adicionar código para realizar a ligação entre o `Mixin` ou `Gem` e a interface.

Como pode ser observado na Tabela 5.5, não há modificação em relação aos elementos de variabilidade presentes na Tabela 4.5 da Subseção 4.2.4. No código apresentado no Quadro 5.4 é possível observar estes dados.

O Quadro 5.4 mostra em seu código que a variabilidade quanto a sua finalidade que é possível de ser obtida com o padrão *Strategy* em `Ruby` é positiva e negativa, porque as variantes quando são instanciadas apenas adicionam ou removem funcionalidades.

As unidades de código que o padrão utiliza para fornecer a variabilidade são os métodos

Quadro 5.4 Exemplo de variabilidades com o padrão Strategy em Ruby - Feature Usuário

Classe Usuário - Context

```
class Usuario < ActiveRecord::Base
    has_secure_password
    has_many :posts
    has_many :comentarios
    attr_accessor :forma_login, :forma_cadastro

    ... #verificações de variáveis
end
```

Classe StrategyUsuarioCadastro - Strategy

```
class StrategyUsuarioCadastro
    def self.create_with_omniauth(params)
        if params[:provider] == "facebook"
            StrategyUsuarioFacebook.resposta(params)
        elsif params[:provider] == "twitter"
            StrategyUsuarioTwitter.resposta(params)
        elsif params[:provider] == "google+"
            StrategyUsuarioGoogle.resposta(params)
        elsif params[:provider] == "linkedin"
            StrategyUsuarioLinkedin.resposta(params)
        end
    end

    def resposta (params) #esqueleto de método
    end

    def cadastrar (params) #esqueleto de método
    end
end
```

Classe StrategyUsuarioLogin - Strategy

```
class StrategyUsuarioLogin
    def self.find_with_omniauth(params)
        if params[:provider] == "facebook"
            StrategyUsuarioFacebook.resposta(params)
        elsif params[:provider] == "twitter"
            StrategyUsuarioTwitter.resposta(params)
        elsif params[:provider] == "google+"
            StrategyUsuarioGoogle.resposta(params)
        elsif params[:provider] == "linkedin"
            StrategyUsuarioLinkedin.resposta(params)
        end
    end
```

```
    def resposta (params) #esqueleto de método
    end

    def autenticar (params) #esqueleto de método
    end
end

#O exemplo dado é em relação ao login e cadastro por Facebook; de forma semelhante
#é realizado o login e cadastro por Google, Twitter e LinkedIn, mudando as verificações
#dos parâmetros de sucesso e bloqueio
```

Classe StrategyUsuarioLoginFacebook - ConcreteStrategy

```
class StrategyUsuarioLoginFacebook < StrategyUsuarioLogin
    FACEBOOK_SUCESSO = 200
    FACEBOOK_REVOGADO = 403
    FACEBOOK_BLOQUEADO = 408
    attr_reader :usuario

    def self.resposta(params)
        ... #definição do corpo do método
    end

    def self.autenticar(params)
        ... #definição do corpo do método
    end
end
```

Classe StrategyUsuarioCadastroFacebook - ConcreteStrategy

```
class StrategyUsuarioCadastroFacebook < StrategyUsuarioCadastro
    FACEBOOK_SUCESSO = 200
    FACEBOOK_REVOGADO = 403
    FACEBOOK_BLOQUEADO = 408
    attr_reader :usuario

    def self.resposta(params)
        ... #definição do corpo do método
    end

    def self.cadastrar(params)
        ... #definição do corpo do método
    end
end
```

e as classes, pois são utilizadas classes abstratas para definir o esqueleto dos métodos, e cada subclasse implementa os métodos da forma que necessita, sendo que os método podem possuir chamadas a outros métodos e estas chamadas e podem variar em sua ordem de realizar as chamadas. Variabilidade de classes porque cada *strategie* é implementada em uma classe distinta.

A variabilidade quanto ao efeito é a variabilidade de fluxo e de lógica. As classes abstratas `StrategyUsuarioCadastro` e `StrategyUsuarioLogin` fornecem esqueletos de métodos que são implementados pelas *strategies*, pode-se obter variabilidade de fluxo ao modificar a ordem de chamadas de métodos, caso exista algum método que realize várias chamadas de métodos. Outra variação existente entre as *strategies* é a lógica de seus métodos, sendo que cada *strategie* pode implementar os métodos de acordo com suas necessidades, fornecendo assim a variabilidade de lógica.

O escopo de variabilidade é seleção pois apenas uma *strategie* pode ser executado por vez.

O tempo de vinculação é de execução pois os *template methods* são inseridos ou removidos durante a execução da aplicação.

É possível utilizar *features* opcionais e alternativas, dependendo da forma que a seleção das *features* for implementada.

Sumário de Comparação dos Elementos de Variabilidade das Implementações

A Tabela 5.6 foi elaborada de forma a sumarizar os elementos de variabilidade identificados em cada padrão de projeto implementado em `Ruby`.

Tabela 5.6: Tabela dos Elementos de Variabilidade Presentes nas Implementações dos Padrões de Projeto

		Observer	Decorator	Template Method	Strategy
Tipo de Variabilidade (Finalidade)	Positivo	X	X	X	X
	Negativo	X			X
	Opcional			X	
	Alternativo			X	
	Funcional			X	
Unidades de Código	Variáveis				
	Métodos		X	X	X
	Classes	X	X	X	X
Tipo de Variabilidade (Efeito)	Atributo				
	Lógica	X	X	X	X
	Fluxo		X	X	X
	Interface				
Escopo	Binário				
	Seleção			X	X
	Aberto	X	X		X
Tipo de Feature	Obrigatório	X		X	
	Opcional	X	X	X	X
	Alternativo			X	X
Tempo de Vinculação	Carregamento				
	Ligação				
	Compilação				
	Execução	X	X	X	X
	Pós-Execução				

Caso não sejam utilizadas as alternativas de implementação presentes no `Ruby`, não há alteração nos elementos de variabilidade presentes na teoria e na prática, tal como pode ser observado na Tabela 5.6 que não teve modificação em relação à Tabela 4.6.

Capítulo 6

Conclusão

Neste trabalho foi estudada a viabilidade técnica de implementar variabilidades utilizando os padrões de projeto *Observer*, *Template Method*, *Strategy* e *Decorator* em `Ruby`.

Este estudo teve como escopo a área de engenharia de domínio presente no paradigma de linha de produtos de *software*, focando na implementação de componentes de *software* que fossem reusáveis ou que possuíssem variabilidades.

O problema foi definir as características destes padrões de projetos em relação à implementação de variabilidades e verificar a validade destas características na linguagem `Ruby`. A solução consistiu em identificar pontos de análise que permitissem obter informações sobre a forma de gerenciamento de variabilidades destes padrões e, utilizando de um cenário no domínio de *blogs* como base de requisitos, avaliar se estas informações eram válidas e se repetiam quando os referidos padrões eram implementados na linguagem `Ruby`.

6.1 Contribuições

A principal contribuição deste trabalho foi em relação à avaliação dos padrões de projeto *Observer*, *Template Method*, *Strategy* e *Decorator*, demonstrando que uma linguagem de programação pode impactar a implementação e o gerenciamento de variabilidades destes padrões e como isso ocorre.

Observando os resultados da análise dos padrões de projeto e das implementações, um ponto que chamou a atenção na avaliação realizada no Capítulo 5 foi que as características da linguagem `Ruby` influenciam na implementação de variabilidades utilizando os padrões de projeto *Decorator*, *Strategy* e *Observer* em `Ruby`. O padrão de projeto *Template Method* não sofre

influências.

As cacterísticas de Ruby influenciam nas implementações no sentido de criar módulos utilizando os Mixins ou Gems pertencentes à linguagem, de forma que estes módulos são as *features* que se deseja implementar.

A utilização da linguagem Ruby afeta positivamente as implementações porque fornece mais opções para implementar os padrões de projeto. Caso estas opções sejam utilizadas, alguns dos padrões sofrem alteração nos seus tipos de elementos de variabilidades, tal como é o caso dos padrões *Observer*, *Decorator* e *Strategy*. Uma vez que Ruby fornece alternativas para implementar variabilidades com padrões de projeto, ele afeta positivamente a implementação e uso das variabilidades. Isto basicamente significa que a linguagem facilita a implementação.

Considerando estes resultados, conclui-se que é viável tecnicamente implementar variabilidades utilizando padrões de projeto na linguagem Ruby.

Outra contribuição foi o uso da metodologia do DSR para gerar os artefatos de código necessários para implementar variabilidades, obtendo um exemplo de uso desta metodologia no contexto de LPS e gerenciamento de variabilidades. O uso da metodologia do GQM para estruturar o processo de coleta e análise de dados também pode ser utilizado como exemplo, sendo outra contribuição deste trabalho.

6.2 Considerações Finais

A utilização de Ruby on Rails, no contexto deste trabalho, se mostrou fácil de trabalhar e com boa produtividade. Contudo, esta observação é passível de influência de experiências pessoais.

Considerando que é possível implementar variabilidades com padrões de projeto em Ruby, que suas características impactam positivamente a implementação de variabilidades com padrões de projeto e que o seu *framework* Rails é caracterizado principalmente por sua alta produtividade em pouco tempo de desenvolvimento, é possível conjecturar algumas vantagens em utilizar Ruby no desenvolvimento de uma LPS.

Neste contexto, utilizar Ruby pode trazer benefícios para o desenvolvimento de uma linha de produto de *software* porque pode colaborar para que os artefatos base, referindo-se às *features* e variabilidades, sejam desenvolvidos mais rapidamente.

Porém, utilizar `Ruby` pode apresentar desvantagens também. Uma delas é em relação à escalabilidade de aplicações. Quando uma aplicação fica muito grande, pode ocasionar problemas no processo de manutenção e desenvolvimento do código. Porém, esta é uma observação empírica realizada no contexto deste trabalho, necessitando de uma pesquisa mais profunda para realizar validações em relação a esta questão.

O uso dos padrões de projeto *Observer*, *Template Method*, *Strategy* e *Decorator* para implementar variabilidades se mostrou de boa escalabilidade, podendo adicionar várias *features* e variabilidades sem alterar significativamente a manutenção e desenvolvimento de código.

Os padrões estudados neste trabalho não contemplaram alguns dos tipos presentes nos elementos de variabilidade estudados. É necessário estudar em trabalhos futuros outros padrões que possam contemplar estes outros tipos de elementos de variabilidade, para que seja possível mapear da forma mais completa possível a utilização de padrões de projeto para implementar variabilidades.

6.3 Trabalhos Futuros

Tal como definido em [Anastasopoulos e Gacek 2001], existem vários critérios de validação existentes para validar um mecanismo de implementação de variabilidades. Neste trabalho, apenas os critérios de escopo, flexibilidade e suporte da linguagem de programação foram avaliados para as implementações dos padrões de projeto em `Ruby`. Existe uma série de outros critérios, envolvendo não somente código, que podem ser estudados e avaliados relacionado os padrões de projeto *Template Method*, *Decorator*, *Strategy* e *Observer* com `Ruby`. Isto mostra-se como sendo uma sequencia lógica a ser explorada em trabalhos futuros.

A implementação de variabilidades utilizando uma combinação de diferentes padrões de projeto pode ser explorada também. Esta abordagem poderá ser utilizada tanto como forma de amenizar as desvantagens e/ou aprimorar as vantagens presentes nos padrões de projeto, e também forma de testar novas formas de gerenciar variabilidades tendo por base o mecanismo de padrões de projeto.

No contexto da engenharia de aplicação, é possível explorar formas de geração e construção de produtos utilizando a linguagem `Ruby` e seu *framework* `Rails`. Esta sugestão de trabalhos futuros e a sugestão anterior podem ser exploradas tanto para o mecanismo de padrões de projeto

quanto para os outros vários mecanismos de implementação de variabilidades existentes.

Outra direção que pode ser explorada é a das linguagens de domínio específico, do inglês *Domain-Specific Languages* (DSLs). Elas podem ser desenvolvidas para cada padrão de projeto a fim de representar de uma forma mais adequada a solução proposta por padrão de projeto em específico, estabelecendo uma espécie de *framework* para este padrão, tendo como base de implementação para a DSL a linguagem `Ruby`.

Referências Bibliográficas

[Amin, Mahmood e Oxley 2011]AMIN, F. e; MAHMOOD, A. K.; OXLEY, A. An analysis of object oriented variability implementation mechanisms. *ACM SIGSOFT Software Engineering Notes*, v. 36, n. 1, p. 1–4, Janeiro 2011.

[Anastasopoulos e Gacek 2001]ANASTASOPOULOS, M.; GACEK, C. Implementing product line variabilities. In: ACM. *ACM SIGSOFT Software Engineering Notes*. Toronto, Ontario, Canada, 2001. v. 26, n. 3, p. 109–117.

[Apel et al. 2013]APEL, S. et al. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Alemanha: Springer-Verlag, 2013.

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

[Basili, Caldiera e Rombach 1994]BASILI, V. R.; CALDIERA, G.; ROMBACH, H. D. The goal question metric approach. *Encyclopedia of software engineering*, Wiley Online Library, v. 2, 1994.

[Blogger 2015]BLOGGER. *Sobre o Blogger*. 2015. Consultado na INTERNET: https://www.blogger.com/about, agosto/2015.

[Böckle, Pohl e Linden 2005]BÖCKLE, G.; POHL, K.; LINDEN, F. Van der. *Software product line engineering: Foundations, Principles and Techniques*. Heidelberg, Alemanha: Springer-Verlag, 2005.

[Bosch, Deelstra e Sinnema 2009]BOSCH, J.; DEELSTRA, S.; SINNEMA, M. Variability assessment in software product families. *Information and Software Technology*, Butterworth-Heinemann, Newton, MA, EUA, v. 51, n. 1, p. 195–218, Janeiro 2009.

[Bosch, Gurp e Svahnberg 2000]BOSCH, J.; GURP, J. v.; SVAHNBERG, M. Managing variability in software product lines. Groningen, Holanda, 2000.

[Bosch, Svahnberg e Gurp 2005]BOSCH, J.; SVAHNBERG, M.; GURP, J. V. A taxonomy of variability realization techniques. *Software: Practice and Experience*, Wiley Online Library, v. 35, n. 8, p. 705–754, 2005.

[Caelum 2016]CAELUM. *TDD - Test-Driven Development*. 2016. Consultado na INTERNET: http://tdd.caelum.com.br/, Janeiro/2016.

[Chang, Her e Kim 2005]CHANG, S. H.; HER, J. S.; KIM, S. D. A theoretical foundation of variability in component-based development. *Information and Software Technology*, Seul, Coréia do Sul, v. 47, n. 10, p. 663–673, Novembro 2005.

[Clements e Northrop 2001]CLEMENTS, P. C.; NORTHROP, L. M. *Software Product Lines: Patterns and practice*. Boston, MA, EUA: Addison Wesley Longman Publishing Co., 2001.

[Creswell 2013]CRESWELL, J. W. *Research design: Qualitative, quantitative, and mixed methods approaches*. [S.l.]: Sage publications, 2013.

[Flanagan e Matsumoto 2008]FLANAGAN, D.; MATSUMOTO, Y. *The Ruby Programming Language*. 1. ed. Sebastopol, EUA: O'Reilly Media, Inc., 2008.

[Fowler 1999]FOWLER, M. *Refactoring: Improving the Design of Existing Code*. [S.l.: s.n.], 1999.

[Fritsch, Lehn e Strohm 2002]FRITSCH, C.; LEHN, A.; STROHM, T. Evaluating variability implementation mechanisms. In: *Proceedings of International Workshop on Product Line Engineering (PLEES)*. [S.l.: s.n.], 2002. p. 59–64.

[Gamma et al. 1995]GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. [S.l.]: Addison Wesley, 1995.

[Günther 2009]GÜNTHER, S. Engineering domain-specific languages with ruby. *H.-K. Arndt and H. Krcmar, editors*, 2009.

[Günther e Sunkle 2009-A]GÜNTHER, S.; SUNKLE, S. *Enabling feature-oriented programming in Ruby*. Magdeburg, Alemanha, 2009–A.

[Günther e Sunkle 2009-B]GÜNTHER, S.; SUNKLE, S. Feature-oriented programming with ruby. In: ACM. *Proceedings of the First International Workshop on Feature-Oriented Software Development*. Denver, Colorado, EUA, 2009–B. p. 11–18.

[Günther e Sunkle 2010]GÜNTHER, S.; SUNKLE, S. Dynamically adaptable software product lines using ruby metaprogramming. In: ACM. *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*. Eindhoven, Holanda, 2010. p. 80–87.

[Günther e Sunkle 2012]GÜNTHER, S.; SUNKLE, S. rbfeatures: Feature-oriented programming with ruby. *Science of Computer Programming*, Elsevier, Magdeburg, Alemanha, v. 77, n. 3, p. 152–173, Fevereiro 2012.

[Gurp e Savolainen 2006]GURP, J. V.; SAVOLAINEN, J. Service grid variability realization. In: IEEE. *Software Product Line Conference, 2006 10th International*. [S.l.], 2006. p. 85–94.

[Hansson, Ruby e Thomas 2013]HANSSON, D. H.; RUBY, S.; THOMAS, D. *Agile Web Development With Rails 4*. EUA: The Pragmatic Bookshelf, 2013.

[Jesus 2013]JESUS, G. S. d. Variabilidade em interfaces de usuário em linhas de produto de software baseadas na web um estudo exploratório. 2013.

[Kang, Park e Sugumaran 2009]KANG, K. C.; PARK, S.; SUGUMARAN, V. *Applied Software Product Line Engineering*. Boston, MA, EUA: Auerbach Publications, 2009.

[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.

[Rommes, Schmid e Linden 2007]ROMMES, E.; SCHMID, K.; LINDEN, F. J. Van der. *Software product lines in action: The Best Industrial Practice in Product Line Engineering*. Heidelberg, Alemanha: Springer-Verlag, 2007.

[Ruby-Lang 2015]RUBY-LANG. *Sobre o Ruby*. 2015. Consultado na INTERNET: https://www.ruby-lang.org/pt/about/, agosto/2015.

[Ruby-On-Rails 2015]RUBY-ON-RAILS. *Desenvolvimento Web sem Dor*. 2015. Consultado na INTERNET: https://www.rubyonrails.com.br/, agosto/2015.

[Schmid e Verlage 2002]SCHMID, K.; VERLAGE, M. The economic impact of product line adoption and evolution. *IEEE software*, IEEE Computer Society, v. 19, n. 4, p. 50–57, Julho 2002.

[Sharp 2000]SHARP, D. C. Containing and facilitating change via object oriented tailoring techniques. In: PROCEEDINGS OF THE FIRST SOFTWARE PRODUCT LINE CONFERENCE. *Proceedings of The First Software Product Line Conference*. Denver, Colorado, EUA, 2000.

[Sommerville 2008]SOMMERVILLE, I. *Engenharia de Software*. 8. ed. [S.l.]: Addison Wesley Brasil, 2008.

[Tiobe 2016]TIOBE. *TIOBE Index for January 2016*. 2016. Consultado na INTERNET: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, Janeiro/2016.

[Venable 2006]VENABLE, J. R. The role of theory and theorising in design science research. In: CITESEER. *Proceedings of the 1st International Conference on Design Science in Information Systems and Technology (DESRIST 2006)*. [S.l.], 2006. p. 1–18.

[Wordpress 2015]WORDPRESS. *Crie um site ou blog gratuito*. 2015. Consultado na INTERNET: https://br.wordpress.com/, agosto/2015.

[Worren, Moore e Elliott 2002]WORREN, N. A.; MOORE, K.; ELLIOTT, R. When theories become tools: Toward a framework for pragmatic validity. *Human Relations*, Sage Publications, v. 55, n. 10, p. 1227–1250, 2002.