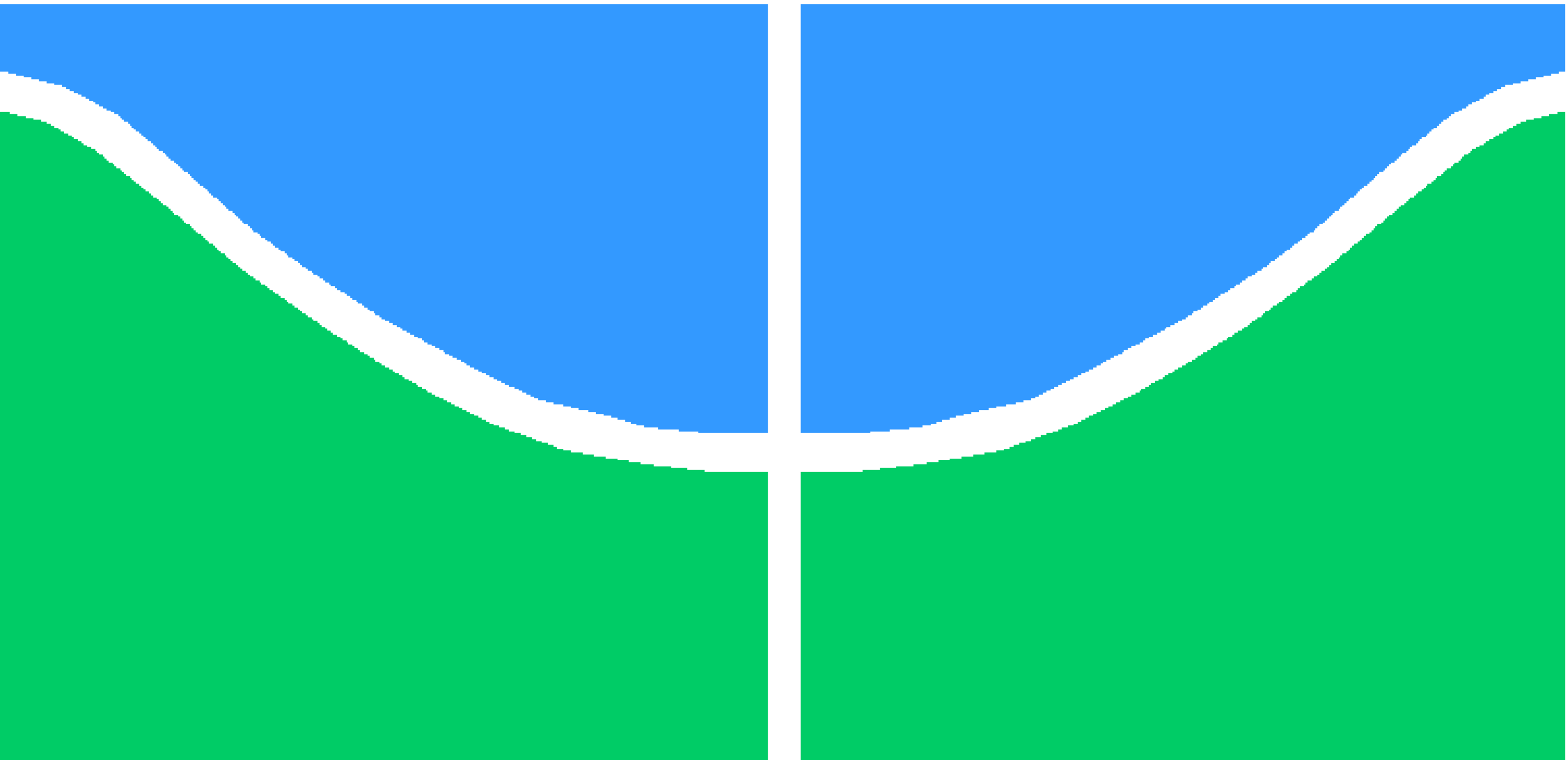

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Engenharia de Software

Framework para Geração de Teste Unitário

Autor: Tomáz Felipe Rodrigues Martins e Thaiane Ferreira Braga

Orientador: Profª Dra. Milene Serrano

Coorientador: Prof. Dr. Maurício Serrano

Brasília, DF

2016

Tomáz Felipe Rodrigues Martins e Thaiane Ferreira Braga

Framework para Geração de Teste Unitário

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

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Profª Dra. Milene Serrano

Coorientador: Prof. Dr. Maurício Serrano

Brasília, DF

2016

Tomáz Felipe Rodrigues Martins e Thaiane Ferreira Braga

Framework para Geração de Teste Unitário/ Tomáz Felipe Rodrigues Martins e Thaiane Ferreira Braga. – Brasília, DF, 2016-

108 p. : il. (algumas color.) ; 30 cm.

Orientador: Profª Dra. Milene Serrano

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2016.

1. Testes Unitários. 2. *Framework*. I. Profª Dra. Milene Serrano. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. *Framework* para Geração de Teste Unitário

CDU 02:141:005.6

Tomáz Felipe Rodrigues Martins e Thaiane Ferreira Braga

Framework para Geração de Teste Unitário

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

Trabalho aprovado. Brasília, DF, 04 de julho de 2016:

Profª Dra. Milene Serrano
Orientador

Prof. Dr. Maurício Serrano
Coorientador

Dr. Fabrício Braz
Convidado 1

Msc. Elaine Venson
Convidado 2

Brasília, DF
2016

Este trabalho é dedicado aos que persistem.
Àqueles que veem no erro um aliado na jornada do aprendizado.
Àqueles que sabem que o sucesso é o final de uma estrada feita de tropeços, quedas e esforço...

Agradecimentos

Eu, Tomáz Martins, agradeço aos meus pais, José Martins e Elody Martins, e irmãos, Hugo Martins e Anaísis Martins, que me apoiaram não apenas nessa etapa da vida, mas em todas as anteriores. Foram guias na construção de meu caráter. Agradeço à minha namorada, Thaiane Braga, por ser uma companheira excepcional, na graduação e na vida. Agradeço ao amigo Leandro Moraes e todos os demais amigos que sempre estiveram dispostos a ajudar-me, dando-me força e palavras de motivação. Agradeço ao meu primo, Márcio Carneiro. Sempre que solicitado, estava pronto a ajudar. Agradeço a todos os colegas de graduação que compartilharam os mesmos sonhos e anseios, especialmente Maxwell Almeida, Fabiana Ofugi, Ruenys Rosa, Arthur Jahn e Paulo Tada. Agradeço também ao meu eterno professor, Fabrício Augusto Gomes, por ter sido um Professor. Um Mestre. Seus conselhos ecoam em mim desde o ensino médio até hoje.

Eu, Thaiane Braga, agradeço à minha família pelo apoio incondicional. Agradeço, em especial, à minha mãe, Leni Braga, por sempre me ajudar a ver o lado bom de cada situação. Agradeço aos meus amigos do ensino médio, em especial à Carmem Luiza, Valkiria Lucena e Rafael Araújo, que torceram por mim desde o início e perdoaram minhas ausências durante os períodos letivos nesses últimos anos. Agradeço aos meus colegas de faculdade. Foi uma honra estudar com vocês. Agradeço também aos meus grandes amigos, Fabiana Ofugi, Ruenys Rosa e Maxwell Almeida, por todos os bons momentos vividos. Agradeço à minha irmã e melhor amiga, Kamila Braga, por todos os conselhos, abraços e ensinamentos que me guiou até aqui. Por fim, meu mais amoroso obrigado ao meu namorado, Tomáz Martins, por alegrar meus dias e me acompanhar em cada maratona de estudo durante a graduação.

Agradecemos aos nossos orientadores Milene Serrano e Maurício Serrano por todo apoio, não só durante esse trabalho, mas durante nossa graduação, e ao professor Hilmer Neri, por sempre acreditar em nós. São exemplos de dedicação. Temos um carinho especial por vocês. Agradecemos ao professor Paulo Meirelles, por todas as oportunidades e incentivo. É um exemplo de profissional apaixonado e zeloso. Agradecemos aos professores Edson Alves e Fabrício Braz, pelos ensinamentos valiosos na área de programação. Agradecemos aos professores George Marsicano e Elaine Venson, pelo carinho e conselhos. Agradecemos a todos os professores, que de alguma forma, impactaram nossa visão profissional e pessoal.

"Teremos coisas bonitas pra contar.
E até lá, vamos viver.
Temos muito ainda por fazer.
Não olha pra trás,
Apenas começamos.
O Mundo começa agora,
Apenas começamos."
(Legião Urbana)

Resumo

A produção de testes tem se tornado uma atividade importante no processo de desenvolvimento de software. Não apenas o mercado vem exigindo maior qualidade, mas o usuário comum vem demandando redução de falhas nos serviços e produtos de software. Além da qualidade relativa ao software, deve-se considerar os custos relacionados à manutenção. A busca e o reparo de *bugs* custa caro à produção de software. Além disso, muitos desenvolvedores não se sentem motivados em produzir testes para seus códigos, o que agrava o problema. Diante dessa problemática, o presente trabalho relata sobre o desenvolvimento de um *framework* capaz de dar suporte ao desenvolvedor na tarefa de produzir testes unitários. Buscou-se, com isso, a produção semiautomatizada de testes unitários relativos a métodos comuns às aplicações *web*: criar, recuperar, atualizar e apagar. Esse *framework* tem como alvo inicial aplicações em *Grails*. Licenciado sob a GPLv3, o *framework* possui *status* de software livre e foi implementado usando a linguagem de programação C++.

Palavras-chaves: Testes Unitários. *Framework*. Semiautomatização. Verificação e Validação.

Abstract

Test production has become an important activity in software development process. Not only the market is demanding higher quality, but the average user is demanding reduction of gaps in services and software products. Beyond the relative quality software, the costs related to maintenance should be considered. The search and the bug fix is expensive to software production. In addition, many developers are not motivated to produce tests for their code, which makes the problem worse. Faced with this problem, this project sought the implementation of a framework capable of supporting the developer in the task of producing unit tests. With this, semi-automated tests will be produced for common methods to web applications: create, read, update and delete. This framework was to initial target applications in Grails. Licensed under GPLv3, this *framework* has status of free software. It has been implemented using the C ++ programming language.

Key-words: Unit Tests. Framework. Semi-automation. Verification and Validation.

Lista de ilustrações

Figura 1 – Diagrama representativo de testes de caixa-preta 30
Figura 2 – Diferença entre formas de reutilização de software 33
Figura 3 – Comparativo entre *framework* horizontal e vertical 39
Figura 4 – Fluxo de atividades do trabalho de conclusão de curso 45
Figura 5 – Fluxo de atividades da implementação da solução de software 45
Figura 6 – Representação dos módulos do Scarefault 52
Figura 7 – Representação expandida dos módulos do Scarefault 52
Figura 8 – Diagrama de domínio do Scarefault 54
Figura 9 – Diagrama de classe completo do Scarefault 55
Figura 10 – Diagrama de classes apresentando o *builder pattern* no *framework* . . . 56
Figura 11 – Diagrama de classes apresentando o *factory pattern* no *framework* . . 58
Figura 12 – Diagrama de classes exemplificando a extensibilidade do *factory pattern* no *framework* 63
Figura 13 – Diagrama de sequência da instanciação de um `Parser` e os efeitos na `CollectorFactory` 66
Figura 14 – Diagrama de sequência das chamadas de funções por trás da `void collect_data(char *, ...)` 69
Figura 15 – Diagrama de classes exemplificando a extensibilidade do *builder pattern* no *framework* 71
Figura 16 – Diagrama de sequência das chamadas de funções por trás da `void generate_testfile()` 74
Figura 17 – Resultado da execução do *script* de teste de identificação 85
Figura 18 – Resultado da execução dos teses gerados para a `FeedbackController` . 90

Lista de tabelas

Tabela 1 – Histórias da *feature* 01 47
Tabela 2 – Histórias da *feature* 02 47
Tabela 3 – Histórias da *feature* 03 48
Tabela 4 – Histórias da *feature* 04 48
Tabela 5 – Cronograma para o trabalho de conclusão de curso 1 48
Tabela 6 – Cronograma para o trabalho de conclusão de curso 2 48
Tabela 7 – Descrição das letras identificadoras e o respectivo tipo de dado a ser coletado 68
Tabela 8 – Marcações de comentários para uso do Scarefault 76

Lista de abreviaturas e siglas

API	*Application Programming Interface*
BPMN	*Business Process Model and Notation*
CRUD	*Create, Read, Update and Delete*
GPL	*General Public License*
JVM	*Java Virtual Machine*
MVC	*Model-View-Controller*
ORM	*Object Relational Mapping*
TCC	Trabalho de Conclusão de Curso
TI	Tecnologia da Informação
VM	*Virtual Machine*

Sumário

1	**INTRODUÇÃO**	**25**
1.1	**Contextualização**	**25**
1.2	**Questão de Pesquisa**	**26**
1.3	**Justificativa**	**26**
1.4	**Objetivos**	**27**
1.4.1	Objetivo Geral	27
1.4.2	Objetivos Específicos	27
1.5	**Organização do Documento**	**27**
2	**REFERENCIAL TEÓRICO**	**29**
2.1	**Testes de Software**	**29**
2.1.1	Testes de Caixa-preta	30
2.1.1.1	Técnicas de Derivação de Casos de Teste de Caixa-preta	30
2.1.2	Testes de Caixa-branca	31
2.1.2.1	Técnicas de Derivação de Casos de Teste de Caixa-branca	31
2.2	**Reutilização de Software**	**32**
2.2.1	*Frameworks*, Biliotecas de Classe e *Design Patterns*	33
2.2.2	Bibliotecas de Classe	34
2.2.3	*Design Patterns*	35
2.2.3.1	Princípios de *Design* da Orientação a Objetos	35
2.2.3.2	Definição de *Design Pattern*	35
2.2.3.3	Classificação dos *Design Patterns*	36
2.2.3.3.1	Padrões Criacionais	36
2.2.3.3.2	Padrões Estruturais	37
2.2.3.3.3	Padrões Comportamentais	37
2.2.4	*Frameworks*	37
2.2.4.1	Vantagens e Desvantagens de *Frameworks*	38
2.2.4.2	Categorias de *Frameworks*	38
2.2.4.2.1	Classificação por Modo de Uso	38
2.2.4.2.2	Classificação por Conhecimento Embutido	39
2.3	**Geração e Automação de Testes**	**40**
2.3.1	Técnicas de Geração e Automação de Teste	40
2.3.1.1	*Record/Playback*	40
2.3.1.2	*Script programming*	40
2.3.1.3	*Data-driven*	41

2.3.1.4 *Keyword-driven* . . . 41
2.4 Resumo do Capítulo . . . 41

3 METODOLOGIA . . . 43
3.1 Planejamento da Metodologia Aplicada . . . 43
3.1.1 Fluxo de Atividades Geral . . . 44
3.1.2 Detalhes da Execução do Desenvolvimento . . . 44
3.1.2.1 FE01 Identificação dos elementos da linguagem de programação Grails . . . 46
3.1.2.2 FE 02 Análise dos métodos da camada de domínio (*model*) . . . 46
3.1.2.3 FE 03 Análise dos métodos da Camada de Controle (*controller*) . . . 46
3.1.2.4 FE 04 Desenvolvimento do Framework . . . 46
3.1.3 Cronograma . . . 47
3.2 Resumo do Capítulo . . . 49

4 SCAREFAULT . . . 51
4.1 Scarefault: Visão Geral . . . 51
4.2 A Arquitetura do Scarefault . . . 51
4.2.1 Visão de Pacotes . . . 51
4.2.2 Visão de Classes . . . 53
4.2.2.1 *Builder Pattern* . . . 56
4.2.2.2 *Factory Pattern* . . . 57
4.3 A Utilização do Scarefault . . . 59
4.3.1 Adaptação para uma Nova Linguagem . . . 59
4.3.1.1 Usando o Ambiente de Desenvolvimento . . . 59
4.3.1.2 Explorando a Extensibilidade do Scarefault . . . 60
4.3.1.2.1 Adição de um *parser* para a nova linguagem . . . 60
4.3.1.2.2 Criação do agente de coleta de dados . . . 62
4.3.1.2.3 Criação do agente de construção de casos de teste . . . 70
4.3.2 Geração de Testes Unitários . . . 75
4.4 Resumo do Capítulo . . . 78

5 RESULTADOS . . . 79
5.1 Identificação de Regras Gramaticais da Linguagem Alvo Inicial . . . 79
5.2 Geração de testes unitários que cubram o CRUD . . . 85
5.3 Inclusão de Outras Linguagens para Geração de Testes Unitários e Extensibilidade do *Framework* . . . 90
5.4 Resumo do Capítulo . . . 91

6 CONSIDERAÇÕES FINAIS . . . 93

Referências . . . 95

APÊNDICE A – SUPORTE TECNOLÓGICO . . . 99

A.1 Geração de Teste Unitário . . . 99
A.1.1 Flexc++ . . . 99
A.1.2 Bisonc++ . . . 99
A.2 Engenharia de Software . . . 100
A.2.1 Gerência de Projetos . . . 100
A.2.1.1 Trello . . . 100
A.2.1.2 Bizagi Modeler . . . 100
A.2.2 ZenHub . . . 100
A.2.3 Desenvolvimento de Software . . . 100
A.2.3.1 Vim . . . 101
A.2.3.2 Ubuntu . . . 101
A.2.4 Gerência de Configuração de Software e Requisitos de Software . . . 101
A.2.4.1 Git . . . 101
A.2.4.2 GitHub . . . 101
A.2.4.3 Vagrant . . . 102
A.2.5 Pesquisa . . . 102
A.2.5.1 LaTeX . . . 102
A.2.5.2 TexMaker . . . 102
A.2.5.3 Zotero . . . 102

APÊNDICE B – RESUMO DO ESTUDO DAS FERRAMENTAS FLEXC++ E BISONC++ . . . 103

B.1 Flexc++ . . . 103
B.1.1 Comparativo entre Flex e Flexc++ . . . 103
B.1.2 Como Executar o Flexc++ . . . 103
B.1.3 A Estrutura do Arquivo de Entrada . . . 103
B.1.3.1 Nome do Arquivo . . . 104
B.1.3.2 Corpo do Arquivo . . . 104
B.1.3.2.1 Seção de Definições . . . 104
B.1.3.2.2 Seção de Regras . . . 104
B.1.4 Compilando com o Flexc++ . . . 105
B.2 Bisonc++ . . . 105
B.2.1 Pequeno Glossário . . . 105
B.2.2 Como Executar o Bisonc++ . . . 106
B.2.3 A Estrutura do Arquivo de Entrada . . . 106
B.2.3.1 Corpo do Arquivo . . . 106
B.2.3.1.1 Seção de Diretivas . . . 106

B.2.3.1.2 Seção de Regras Gramaticas 107
B.2.4 Compilando o Bisonc++ 108

1 Introdução

Neste capítulo, serão descritos o contexto no qual se insere o trabalho, a questão de pesquisa a ser explorada, a justificativa para propor um estudo no tema, e os objetivos.

1.1 Contextualização

A produção de software é uma atividade que, devido à sua natureza abstrata, vem acompanhada da eventual inserção de defeitos no código (TRODO, 2009). A partir da década de 1990, os usuários de software passaram a exigir maior atenção por parte das empresas desenvolvedoras em relação à redução de falhas de seus produtos e serviços (SOMMERVILLE, 2007). Além disso, a demanda por software no mercado está crescendo (PHILIPSON, 2004) e, por conseguinte, a exigência de maior qualidade no resultado final do processo de desenvolvimento aumenta (BARBOSA et al., 2009).

Nesse contexto, a identificação de defeitos que casualmente estejam presentes no código é uma preocupação incessante no processo de produção de software. Para essa finalidade, surge a prática de testes do produto. Em essência, essa etapa visa à execução do software alvo com dados de teste, de forma a verificar se os resultados observados correspondem à expectativa. Isso permite ao desenvolvedor demonstrar aos seus clientes que o software está de acordo com as suas especificações, além de viabilizar ao programador a busca e descoberta de defeitos no software (SOMMERVILLE, 2007).

A prática de testes de software tornou-se parte importante do processo de desenvolvimento (BARBOSA et al., 2009). Diversas categorias de testes, com diferentes intenções de análise sobre o produto, foram surgindo, a saber: testes de unidade, testes de integração e testes de aceitação. Testes de integração são aqueles que permitem a observação de que os componentes que formam o software funcionam corretamente em conjunto. Testes de aceitação são testes cuja finalidade é garantir que as especificações do software foram implementadas, de forma a corresponder às necessidades do usuário final. Em essência, é um teste de validação do software (SOMMERVILLE, 2007). Os testes de unidade têm como proposta a garantia de que uma determinada parte (unidade) do código esteja respondendo como o esperado (SOMMERVILLE, 2007).

Outra questão que surge na produção de software, e que é um fator importante a ser considerado, é o custo que *bugs* geram no desenvolvimento e na fase de manutenção. Estimativas apontam que os gastos com a correção de *bugs* chegam a mais de 59,5 bilhões de dólares anuais nos Estados Unidos (JANTTI, 2008). No Brasil, gasta-se 70% do tempo de desenvolvimento corringindo-se erros (JANONES, 2010). Esse cenário aumenta o preço

do produto final.

A produção de testes afeta positivamente o desenvolvimento de software não apenas no nível estratégico. Segundo Burke e Coyner (2003), há diversas razões para que se escreva testes unitários, dentre elas:

- Testes reduzem defeitos em funcionalidades novas e já existentes;
- Testes auxiliam na documentação do código;
- Testes permitem refatoração com maior qualidade;
- Testes reduzem o receio de alterar o código;
- Testes defendem o código contra alterações indesejáveis de outros programadores.

1.2 Questão de Pesquisa

O intuito deste trabalho de conclusão de curso é auxiliar os desenvolvedores de software em relação à seguinte questão: há como propiciar ao desenvolvedor um suporte que forneça a geração de testes unitários, de forma semiautomatizada, e que permita adaptações, com o intuito de ajustar-se ao código-fonte?

1.3 Justificativa

Testes são cruciais no desenvolvimento de software, como evidenciado na seção anterior. Contudo, de maneira geral, desenvolvedores não escrevem testes para os seus programas (BURKE; COYNER, 2003). As razões são variadas, argumentando que não sabem escrever testes ou que não têm tempo para fazê-los (BURKE; COYNER, 2003). Tendo em vista este cenário, onde o mercado de software torna-se cada vez mais exigente com relação à qualidade dos produtos e serviços, bem como os altos custos relacionados à falta de empenho e previdência sobre a qualidade dos sistemas produzidos, é contraditório observar que a prática de fazer testes não seja comum, ou mesmo prioritária por parte dos programadores.

No artigo de Burke e Coyner (2003), intitulado *Top 12 Reasons to Write Unit Tests*, os autores revelam que algumas das desculpas mais frequentes que já ouviram em suas carreiras para os programadores não fazerem testes de unidade são:

- *"Eu não sei escrever testes."*
- *"Escrever testes é muito difícil."*
- *"Não tenho tempo suficiente para fazer testes."*

- *"Testes não são o meu trabalho."*

Essa lista evidencia um fato já conhecido no desenvolvimento de software: a produção de testes é uma atividade onerosa (BARBOSA et al., 2009). Esse cenário é intricado, pois há evidências dos benefícios da produção de testes e, no entanto, há um distanciamento dos desenvolvedores em relação aos testes unitários. Considerando esse panorama, a elaboração de um suporte capaz de apoiar o programador na tarefa de gerar os testes unitários, reduzindo o esforço e os custos associados, será de uma valia considerável, tanto no âmbito técnico quanto estratégico no processo de desenvolvimento de software.

1.4 Objetivos

Esse trabalho visou alcançar os objetivos, Geral e Específicos, apresentados a seguir.

1.4.1 Objetivo Geral

Desenvolver um *framework* capaz de dar suporte aos engenheiros de software na geração de testes unitários. Essa geração se dá de forma semiautomatizada, ou seja, é preciso que o desenvolvedor forneça alguns parâmetros de forma a possibilitar o funcionamento correto do *framework*.

1.4.2 Objetivos Específicos

Os seguintes itens são considerados importantes, devido à sua relevância para o entendimento de teste de software e aplicação dos conhecimentos sobre engenharia de software e, portanto, fazem parte dos objetivos específicos desse trabalho. São eles:

1. Identificar regras gramáticais da linguagem alvo inicial.
2. Gerar testes unitários por meio do *framework* que cubram os métodos criar, recuperar, atualizar e apagar entidades de negócio.
3. Permitir inclusão de outras linguagens para geração de testes unitários e extensibilidade do *framework*.

1.5 Organização do Documento

Este documento está dividido da seguinte forma:

Capítulo 2 - Referencial Teórico: apresenta conceitos, modelos e abordagens associados ao tema foco desse trabalho;

Capítulo 3 - Metodologia: especifica a metodologia utilizada para pesquisa e desenvolvimento deste trabalho;

Capítulo 4 - Scarefault: caracteriza o *framework* desenvolvido, Scarefault, evidenciando seu funcionamento e arquitetura;

Capítulo 5 - Resultados: descreve os resultados, considerando, principalmente os objetivos específicos desse projeto.

Capítulo 6 - Considerações Finais: apresenta a conclusão.

2 Referencial Teórico

Neste capítulo, são apresentadas as bases teóricas para o desenvolvimento do *framework*. O capítulo está organizado em seções. Na seção 2.1, aborda-se uma temática mais específica da qualidade de software, os testes. Traz a motivação para fazê-los, bem como sobre os níveis de testes e as vantagens e estratégias de automatizá-los. Na seção 2.3, discuti-se sobre *frameworks*, sua definição, motivação, vantagens e desvantagens e classificação. Na seção 2.4, explana-se sobre a geração de testes e técnicas de implementá-la.

2.1 Testes de Software

Software é uma atividade que requer um processo com o envolvimento de diversas atividades e diferentes pessoas. Essa característica acaba por favorecer - mesmo não sendo desejado - a inserção de defeitos no produto final (TRODO, 2009). Além disso, há o risco de se produzir algo que não foi solicitado, devido ao mau entendimento dos requisitos(BARBOSA et al., 2009). Esse conjunto de fatores fez surgir uma maior preocupação em relação à qualidade de software e, inclusive nesse contexto, é que há o advento da engenharia de software, com o objetivo de produzir software com qualidade (BUENO; CAMPELO, 2013).

Testes de software representam uma atividade que compõe o processo de verificação e validação de software. É uma das técnicas mais utilizadas no âmbito de garantia de confiabilidade de software (BARBOSA et al., 2009). Em linhas gerais, é um trabalho de abordagem dinâmica sobre o código-fonte. Isso significa que há a exigência do funcionamento do software para que possa ser realizada essa atividade (BARBOSA et al., 2009). Essa análise dinâmica abre a possiblidade de identificação de defeitos no código-fonte, bem como a prevenção de inserções de novos defeitos. A produção de testes de software fornece, ainda, impulso para atividades de refatoração, manutenção e medição do software (BARBOSA et al., 2009).

Os testes podem ser classificados em **níveis**. Essa divisão refere-se ao nível de abstração dos testes. Quanto mais próximo do código, menos abstrato o nível do teste (SOMMERVILLE, 2007). Os testes de software são divididos da seguinte forma, em ordem descrescente de abstração: testes de aceitação, testes de sistema, teste de integração e testes de unidade (SOMMERVILLE, 2007). O *framework* proposto auxiliará na criação de testes de unidade, cuja finalidade é verificar o funcionamento de partes específicas e do código-fonte. Essa análise possibilita a verificação do código, de forma a identificar defeitos e mitigá-los (SOMMERVILLE, 2007).

Os testes podem também ser classificados como: testes de caixa-preta e testes de caixa-branca (BARBOSA et al., 2009).

2.1.1 Testes de Caixa-preta

Testes de caixa-preta, também conhecidos como testes baseados em especificações. Esses testes visam à avaliação do produto em relação aos requisitos funcionais e não funcionais. São caracterizados por realizarem análises a partir de documentação e do software em funcionamento, sem necessidade de averiguação do código-fonte (BARBOSA et al., 2009).

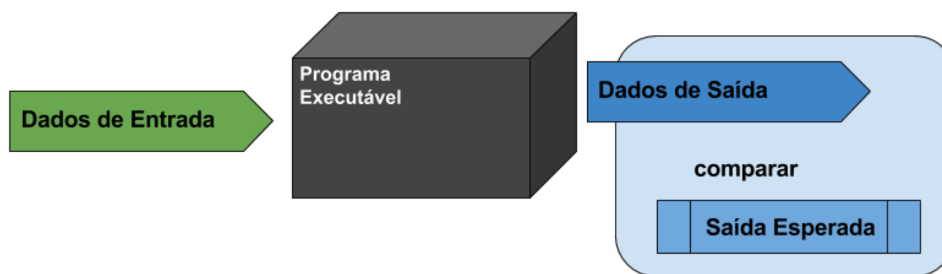

Figura 1 – Diagrama representativo de testes de caixa-preta

A Figura 1 representa um esquemático do funcionamento de um teste de caixa-preta: utiliza-se de dados de entrada no software, e observa-se a correspondência, ou não, com as expectativas.

Ao fazer uso de testes de caixa-preta, os testes sob o ponto de vista do usuário são desenvolvidos, o que auxilia na observação de falhas nas especificações. Além disso, permite que os casos de teste possam ser produzidos, assim que as especificações estiverem prontas (SOFTWARE TESTING FUNDAMENTALS, 2010). Essa abordagem de testes também possui pontos negativos, como a pequena quantidade de entradas que podem ser inseridas para testar o produto. Sem especificações claras, produzir bons casos de testes de caixa-preta torna-se uma tarefa difícil (SOFTWARE TESTING FUNDAMENTALS, 2010).

A seguir são demonstradas algumas técnicas para a derivação de testes de caixa-preta.

2.1.1.1 Técnicas de Derivação de Casos de Teste de Caixa-preta

Testes de caixa-preta não são o foco deste trabalho. Dessa forma, apenas um *overview* das técnicas para essa abordagem de testes será considerado. Para a concepção de casos de testes de caixa-preta, há algumas técnicas utilizadas como guias. São elas: **classes de equivalência**, **análise de valores limite** e **grafo de causa e efeito**.

Classes de equivalência: visa a identificação de agrupamentos de casos de testes que cubram diferentes classes de erros. Essa estratégia permite a redução do número de testes que serão produzidos, diminuindo os custos. A técnica de classes de equivalência divide as entradas do domínio do software em classes. Essas classes devem ser derivadas a partir de um valor significativo para o domínio (WILLIAMS, 2006).

Análise de valores limite: um ponto crítico, e já observado por especialistas, é que grande parte dos defeitos no código é inserido nos limites de uma classe de equivalência. Isso permite a conclusão de que o principal foco no caso de teste deve ser sobre os valores limites. Entende-se como **limite** um valor que esteja imediatamente próximo ao valor de entrada significativo, que divide o domínio em classes de equivalência. Sendo assim, num contexto onde o valor significativo, para separar duas classes de equivalência, seja 50, possíveis valores limites seriam 49 e 51 (WILLIAMS, 2006).

Grafo de causa e efeito: é uma representação visual das relações entre as entradas e saídas. Permite a observação lógico-combinatória das especificações do programa. É uma representação semelhante a circuitos lógicos. Sua principal vantagem é a visualização que se dá, permitindo identificar casos de teste muito facilmente. No entanto, quando a rede fica muito grande e complexa, passa a ser considerado o uso de outras técnicas (BARBOSA et al., 2009).

2.1.2 Testes de Caixa-branca

Testes de caixa-branca, também referenciados como testes baseados em programa, almejam a avaliação do produto por meio da execução do código-fonte. Isso possibilita o exercício do software em busca de defeitos para serem corrigidos. Para que ocorra a dinâmica de testes de caixa-branca, os casos de testes que englobem cenários significativos são selecionados (BARBOSA et al., 2009).

Algumas vantagens de testes de caixa-branca podem ser listadas: pode-se dar início ao desenvolvimento de testes desde o começo da codificação, na medida em que a interação com interfaces não é necessária. Além disso, é um tipo de teste capaz de cobrir mais caminhos, pois tem um alcance mais profundo no software. Com relação aos pontos negativos, cita-se a necessidade de programadores para efetuar os testes de caixa-branca e não garante que o software esteja de acordo com a especificação (BARBOSA et al., 2009).

A seguir são demonstradas algumas técnicas para a derivação de testes de caixa-branca.

2.1.2.1 Técnicas de Derivação de Casos de Teste de Caixa-branca

Os testes de caixa-branca são utilizados objetivando que todos os caminhos independentes possam ser percorridos e testados. Algumas técnicas para derivação de casos de teste são utilizadas, com o intuito de garantir que todos os caminhos tenham sido percorridos, pelo menos um vez, e assegurar uma quantidade de teste exclusivamente necessária. São elas: **Cobertura de Comandos**, **Cobertura de Decisão**, **Cobertura de Condição** e **Cobertura de Decisão-condição** (MYERS; BADGETT; SANDLER,).

Cobertura de comandos: também conhecida como cobertura de linha ou cobertura de segmento. Tem por propósito a execução de todos os comandos do código, ao menos uma vez. Caracteriza-se por cobrir unicamente as linhas com condição verdadeira, e por permitir a observação dos comandos que, por algum motivo, não estão sendo executados. É considerada uma estratégia incompleta, pois não permite o teste de condições falsas, muito menos a execução de testes de *loops* de forma a ter certeza de sua condição de término (MYERS; BADGETT; SANDLER,).

Cobetura de decisão: semelhante à técnica de cobertura de comando, entretanto, desenhada para que tanto as condições verdadeiras quanto as falsas sejam cobertas pelos casos de teste. Da mesma forma que a cobertura de comando, pode deixar de identificar defeitos, pois cobrir uma linha de código não garante que o comando esteja livre de defeitos. Também é chamada de cobertura de ramificações. (MYERS; BADGETT; SANDLER,).

Cobertura de condição: define-se por ser uma estratégia mais sensível aos possíveis caminhos no fluxo de controle. Busca cobrir todos os possíveis valores significativos relativos a uma condição no fluxo (MYERS; BADGETT; SANDLER,).

Cobertura de decisão-condição: é uma estratégia que pretende unir as vantagens da cobertura de decisão e de condição (MYERS; BADGETT; SANDLER,).

Utilizando-se de um grafo de fluxo de controle, fica mais fácil a visualização e identificação dos caminhos a serem testados (COPELAND, 2003).

2.2 Reutilização de Software

O conceito de reutilização é antigo dentro do domínio da engenharia. Foi utilizado há bastante tempo por culturas antigas, como os egípios (*design* das pirâmides) e os romanos (arcos na construção civil). A engenharia civil utiliza-se do conceito de reutilização de soluções e de componentes (SUTCLIFFE, 2002). O uso do desenvolvimento baseado em componentes também está presente no universo de software, entretanto, num cenário menos consolidado (SUTCLIFFE, 2002).

Reutilização é o uso de conceitos, ou objetos, previamente adquiridos, mas em um novo contexto (SUTCLIFFE, 2002). Tendo isso em vista, entende-se que para alcançar a reutilização há a necessidade de entender o problema e a solução. A partir disso, encapsular em diferentes níveis de abstração, para conseguir extrair conceitos passíveis de serem duplicados e/ou adaptados (SUTCLIFFE, 2002).

Portanto, pode-se entender como reutilização de software o desenvolvimento de software a partir de abstrações, sendo estas conceituais ou componentizadas, que já foram utilizadas com sucesso em outros contextos (SUTCLIFFE, 2002).

2.2.1 *Frameworks*, Biliotecas de Classe e *Design Patterns*

Durante o desenvolvimento de software, a reutilização de soluções é uma prática bastante comum. Há diversas formas de colocar em prática a reutilização, como por exemplo o uso de ***frameworks***, **bibliotecas** e ***design patterns***. Essas formas de reutilização de software, em alguns momentos, podem ser semelhantes, no entanto, são técnicas distintas (BARRETO JUNIOR, 2006).

Os três modos de reutilização de software possuem pontos parecidos, como refletir abstrações, generalizando um domínio de problemas e demonstrando uma solução (*frameworks* e *design pattern*), ou mesmo configurando-se como um conjunto de classes capaz de dar suporte ao desenvolvedor por meio de soluções já pensadas e implementadas (*frameworks* e bibliotecas de classes e/ou funções).

Da mesma forma, há também pontos que diferenciam esses tipos de reutilização de software.

A Figura 2 esquematiza algumas diferenças, bem como conceitos desses modos de reutilização de software. Como demonstrado na Figura 2, biliotecas de classes possuem componentes, classes e/ou funções, já prontos, da mesma forma que *frameworks*. Entretanto, não há conexão entre eles. São independentes. Em *frameworks*, as classes que o compõe possuem dependências já encravadas por seu projetista, de forma que o modelo de colaboração entre elas já está embutido. Enquanto que ao se utilizar de uma biblioteca, o desenvolvedor deve criar, por si só, esse modelo de colaboração, caso necessário (BARRETO JUNIOR, 2006).

Frameworks, em contra ponto às bibliotecas de classe, são responsáveis por fazer chamadas ao código da aplicação a qual está vinculado. A esse tipo de relação, dá-se o nome de *Hollywood Principle* (*"don't call us, we'll call you"*) (SAUVÉ, 2006).

Outra diferença entre *frameworks*, bibliotecas e *design patterns* é que, no primeiro caso, é prevista a justaposição de conhecimento de domínio, são mais especilizados. Enquanto que, nos demais casos, não é previsto isso (SAUVÉ, 2006).

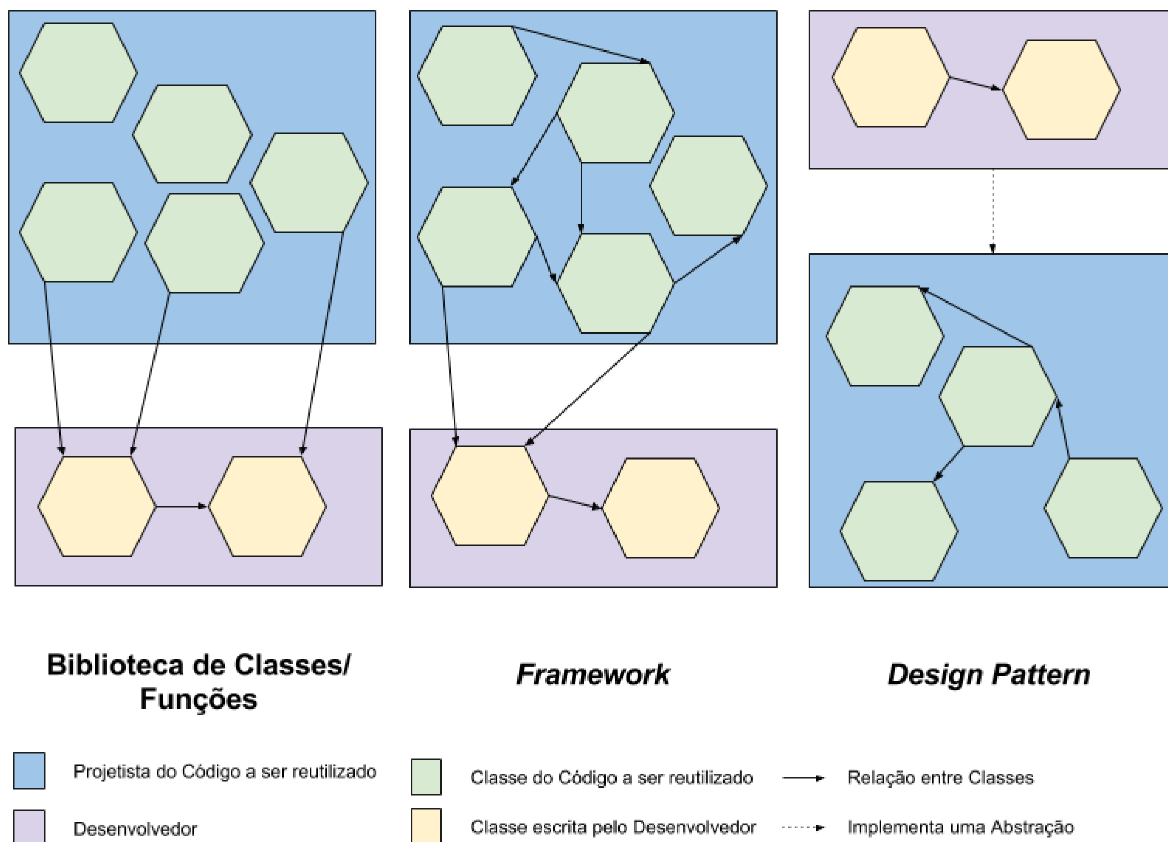

Figura 2 – Diferença entre formas de reutilização de software

A Figura 2 também evidencia algumas diferenças entre os *frameworks* e *design patterns*. *Design patterns* são elementos de software mais abstratos que *frameworks*. Enquanto que *frameworks*, concentram a sua característica de reusabilidade a nível de código, os *design patterns* fazem-no exclusivamente a nível de abstração. *Design pattern* é uma ideia, um conceito capaz de abstrair uma solução padrão para um problema já conhecido. *Frameworks* são abstrações de uma solução para um domínio de problema, materializadas a nível de código (SAUVÉ, 2006). Outra diferença a ser citada é que *design pattern* são elementos de software menores que *frameworks*. Normalmente, há uma relação de parte e todo, onde o todo é o *framework* e as partes são os *design patterns*, e essa relação nunca é inversa (SAUVÉ, 2006).

2.2.2 Bibliotecas de Classe

API, *Application Programming Interface*, é um conjunto particular de regras e especificações que permite a comunicação entre software (SIMSEK, 2004). API's normalmente são vistas como as especificações de como utilizar uma biblioteca de classes, afim de realizar uma tarefa, por meio da interação com a biblioteca (SIMSEK, 2004).

Bibliotecas de Classes ou funções, podem ser definidas como sendo a implemen-

tação de regras de uma **API**. Elas disponibilizam um conjunto de funções pré definidas, encarregadas de suprir necessidades comuns de programadores em um dado contexto (SIMSEK, 2004). As Bibliotecas são auto-suficientes e abstraem um conjunto de funções comuns a um conceito, como por exemplo, bibliotecas de *strings*, funções matemáticas, manipulação de imagens, algoritmos de ordenação e manipulação de estruturas de dados. Essas características concedem ao programador a facilidade de escrever códigos menores e modularizados, garantindo maior legibilidade e manutenibilidade do código (SIMSEK, 2004).

As Bibliotecas podem ser classificadas como **bibliotecas estáticas** ou **bibliotecas compartilhadas**. A principal diferença entre essas categorias é em relação à compilação.

Bibliotecas estáticas: esse tipo de biblioteca é conectada ao programa que faz uso dela durante a compilação, sem a necessidade de recompilar o aquivo da biblioteca. A principal razão para o uso desse tipo de biblioteca é a redução de tempo de compilação (SIMSEK, 2004).

Bibliotecas compartilhadas: é um tipo de biblioteca que, a partir da ativação do programa, é inserida, o que exige a sua recompilação. O programa não necessariamente inclui código da biblioteca, mas referências às funções contidas nela (SIMSEK, 2004).

2.2.3 *Design Patterns*

O conceito de *Design patterns* está relacionado ao de arquitetura de software. Por esse motivo, é importante entender o que é arquitetura de software e qual sua importância no desenvolvimento de aplicações. Há muitas formas de definir arquitetura de software. Em uma discussão de alto nível, pode-se dizer que arquitetura de software são os formatos e as estruturas de uma aplicação de software. Já em uma visão de mais baixo nível, pode-se afirmar que arquitetura de software diz respeito aos módulos da aplicação e suas interconexões, como se relacionam (MARTIN, 2000). Um cuidado que projetistas procuram ter com a arquitetura de suas aplicações de sotware é que essas devem ser manuteníveis, livres de falhas críticas, confiáveis e extensíveis (KLEIN; WEISS, 2009).

2.2.3.1 Princípios de *Design* da Orientação a Objetos

A partir da necessidade de produzir arquiteturas de software consideradas de qualidade, surgiram alguns princípios considerados essenciais para o *design* de projetos orientados a objetos: *open closed principle*, *liskov substitution principle*, *single responsability principle*, *dependency inversion principle* e *interface segregation principle* (MARTIN, 2000).

***Open closed principle*:** um módulo deve ser aberto para extensões, no entanto, fechado para modificações.

***Liskov substitution principle*:** subclasses devem ser substituíveis por suas classes base.

***Single responsability principle*:** uma classe deve ter apenas uma razão para existir. Cada classe deve lidar apenas com uma responsabilidade.

***Dependency inversion principle*:** deve-se depender de abstrações e não de classes concretas.

***Interface segregation principle*:** é melhor possuir interfaces específicas para os clientes de uma classe, ao invés de uma interface genérica.

2.2.3.2 Definição de *Design Pattern*

Com o tempo, algumas estruturas utilizadas para solucionar determinados problemas, e que respeitavam os princípios descritos anteriormente, foram sendo repetidas. Essas estruturas são conhecidas como *design patterns*. *Design patterns*, de uma maneira simplista, são definidos, segundo Martin (2000), como *"a well worn and known good solution to a common problem"*[1]. Segundo Gamma et al. (1994), *design pattern* é uma solução simples e elegante para um problema específico. São formados por quatro elementos: **nome**, **problema associado**, **a solução** e as **consequências**.

Nome: normalmente é fomado por uma ou duas palavras capazes de invocar tanto o problema a ser tratado e sua solução, bem como as consenquências relativas ao *desgin pattern*. Nomear um *design pattern*, de maneira correta, é importante, na medida em que permite a comunicação fácil entre os desenvolvedores. Permite a referência a um arcabouço de conhecimento e o entendimento entre as partes da conversa de forma simples (MARTIN, 2000).

Problema: é a descrição de quando se deve aplicar o *design pattern*. Portanto, em qual circunstância deve-se utilizá-lo, explicando o problema e o contexto. É possível ainda listar condições que devem ser observadas para que faça sentido a aplicação do *design pattern* (MARTIN, 2000).

Solução: descreve os elementos que devem ser inseridos no *design*, apontando suas responsabilidades, estruturas e relacionamentos. É importante ressaltar que a solução não deve descrever em detalhes a implementação, e sim a abstração, o conceito por trás da solução. Isso se deve ao fato de que a solução pode ser aplicada a diferentes situações, variando inclusive a linguagem de programação utilizada (MARTIN, 2000).

[1] **Tradução:** uma solução bem conhecida e bastante utilizada para problemas comuns.

Consequência: são os resultados da aplicação do *design pattern* ao projeto. Isso envolve a descrição dos benefícios e e os custos do uso do padrão. Demonstra os impactos à flexibilidade, extensibilidade do sistema e tantas outras possíveis características. Tudo deve ser listado para melhorar o entendimento do padrão (MARTIN, 2000).

2.2.3.3 Classificação dos *Design Patterns*

A classificação dos *design patterns* vem da necessidade de organizá-los. Além disso, o entendimento do contexto associado a cada uma deles torna-se mais evidente a partir da classificação (GAMMA et al., 1994). Os *design patterns* podem ser classificados entre: **padrões criacionais**, **padrões estruturais** e **padrões comportamentais**:

2.2.3.3.1 Padrões Criacionais

Esse tipo de *design pattern* objetiva a abstração do processo de instantiação de objetos, permitindo o desenvolvimento de um sistema independente de criação, composição e representação (GAMMA et al., 1994).

Esse tipo de padrão viabiliza que o próprio programa decida quais os objetos devem ser criados, para cada caso que surja, adequando-se à situação. Evita determinados problemas que aparecem no método tradicional de criação, por meio de um controle maior dessa ação. Os padrões criacionais possuem duas características marcantes: encapsular o conhecimento sobre as classes concretas utilizadas pelo sistema, e esconder como essas instâncias de classes concretas são criadas e combinadas (GAMMA et al., 1994).

2.2.3.3.2 Padrões Estruturais

Concentra-se em como classes e objetos são compostos para que possam formar e suportar grandes estruturas. É comum o uso de **herança** para compor implementações utilizando esses padrões (GAMMA et al., 1994).

Descrevem formas de compor objetos, afim de adicionar flexibilidade à composição dos objetos, por meio da capacidade de alterá-la em tempo de execução (GAMMA et al., 1994).

2.2.3.3.3 Padrões Comportamentais

Os padrões comportamentais tem por intuito lidar com a comunicação entre objetos e as classes, e qual o comportamento obtido por meio dessa comunicação. Esse padrões fazem uso de complexos fluxos de controle efetuados em tempo de execução. Permite que o desenvolvedor não perca o foco de como interconectar os objetos, na medida em que

os *design patterns* concentram-se no fluxo de controle a eles associado (GAMMA et al., 1994).

Muitos dos padrões comportamentais utilizam **composições**, ao invés de herança. Em muitos casos, é possível observar a cooperação entre objetos para o desenvolvimento de uma atividade que, sozinhos, seriam incapazes de fazê-lo (GAMMA et al., 1994).

2.2.4 *Frameworks*

Frameworks podem ser definidos como uma forma de reutilização de software. *Frameworks* são um conjunto de objetos e classes, abstratas e/ou concretas, que constitui uma arquitetura especializada a solucionar uma família de problemas (BARRETO JUNIOR, 2006). Esse tipo de solução para a reutilização de software é, em essência, uma aplicação não plenamente concluída, mas instanciável. Permite adaptações no código para um funcionamento específico, visando à solução de um problema dentro do domínio englobado pelo *framework* (BARRETO JUNIOR, 2006).

Dentre as características de um *framework*, podem ser destacadas (SAUVÉ, 2006):

- Provê solução para problemas com um mesmo domínio;
- Possui arquitetura baseada em classes e interfaces que abstraem o domínio;
- É flexível e extensível.

2.2.4.1 Vantagens e Desvantagens de *Frameworks*

As vantagens que podem ser citadas para o **uso** de *frameworks* são (BARRETO JUNIOR, 2006) (SAUVÉ, 2006).

- Redução de custos e tempo, na medida em que desenvolvedores passam a agregar valor, ao invés de "reinventar a roda". Além disso, há redução em termos de manutenção de código, pois o *framework* é uma peça confiável;
- Há um aumento da consistência entre as aplicações e compatibilidade, caso o *framework* seja usado;
- O conhecimento dos especialistas no domínio relativo ao *framework* fica empacotado, dessa forma, não há o risco de perda de conhecimento em uma eventual saída de profissionais. A isso dá-se o nome de *Strategic Asset Building*, um patrimônio estratégico da empresa.

As desvantagens em um *framework* concentram-se no período de **produção** dele. Algumas dessas desvantagens são (BARRETO JUNIOR, 2006) (SAUVÉ, 2006):

- A construção de *framewoks* exige planejamento e esforço. São elementos complexos;
- Os benefícios advindos de um *framework* são sentidos a longo prazo;
- O uso de *frameworks* em empresas exige modificação nos processos de desenvolvimento, o que custa dinheiro e tempo.

2.2.4.2 Categorias de *Frameworks*

As formas comumente utilizadas para,se classificar *frameworks* são pelo modo como eles foram projetados para serem utilizados e pelo tipo de conhecimento encapsulado por eles.

2.2.4.2.1 Classificação por Modo de Uso

Essa classificação é utilizada com o intuito de evidenciar a forma como o *framework* é utilizado.

***Inheritance-focused*:** conhecido também como *white-box* ou *architeture-driven*. Esse tipo de *framework* apresenta ao desenvolvedor a possibilidade de estender, ou alterar, funcionalidades por meio do recurso de **herança**. Assim, pode-se utilizar sub-classes e sobreescrita de métodos (ou funções-membro) para aplicar as alterações necessárias (SAUVÉ, 2006).

***Composition-focused*:** chamada de *black-box* ou *data-driven*. Nesse tipo de *framework*, não é possível a alteração, apenas o uso das funcionalidades já fornecidas. Como não há acesso ao código do *framework*, utiliza-se de interfaces, as quais são providas pelo *framework* (SAUVÉ, 2006).

Híbridos: a maioria dos *frameworks* é híbrida, contendo uma grande porção de seu código como *inheritance-focused*, e algumas funcionalidades já pré-definidas (*composition-focused*).

2.2.4.2.2 Classificação por Conhecimento Embutido

Essa classificação é usada para demonstrar como o conhecimento de domínio está encapsulado, e de que forma o *framework* pode auxiliar o desenvolvedor.

Framework de Horizontal: é conhecido também como *framework* de aplicação. O conhecimento embutido é passível de ser utilizado em uma grande variedade de aplicações, pois o seu foco é na generalização. Isso permite que o *framework* possa ser utilizado em uma grande gama de aplicações, mas a parte do problema que ele resolve é menor (BARRETO JUNIOR, 2006).

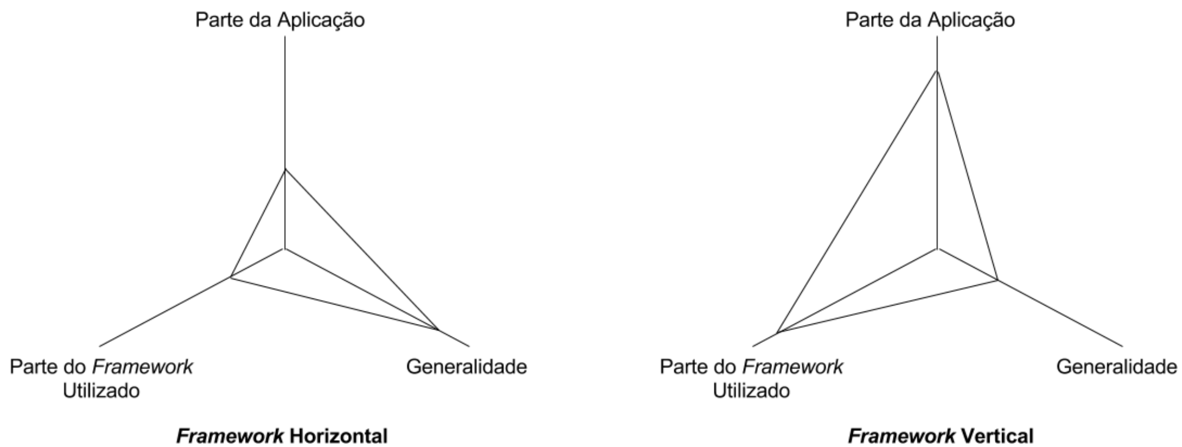

Figura 3 – Comparativo entre *framework* horizontal e vertical

Framework de Vertical: também conhecido como *framework* de domínio. O conhecimento encapsulado nesse tipo de *framework* é mais específico a um domínio particular, o que confere ao *framework* uma capacidade de dar maior suporte e, portanto, maior participação no desenvolvimento da aplicação. Porém, não permite a reutilização em uma diversidade de aplicações, na medida em que o domínio é específico. (BARRETO JUNIOR, 2006).

2.3 Geração e Automação de Testes

A produção de testes de software é uma atividade que vem ganhando foco no desenvolvimento de aplicações. A qualidade do produto, ou serviço, resultante de projetos de software é um fator considerado essencial (BARBOSA et al., 2009). Testes de unidade são uma categoria de testes de software, cujo objetivo é identificar os defeitos no código-fonte, o mais cedo possível. Isso implica que o desenvolvimento desse tipo de teste ocorra em paralelo com o desenvolvimento do software. No entanto, é comum que haja a defasagem entre a produção de testes e de código-fonte do software (FANTINATO et al., 2004). Essa falta de sincronia ocorre por diversos fatores, dentre eles a falta de tempo e de qualificação técnica dos envolvidos (FANTINATO et al., 2004).

Nesse contexto, o uso de ferramentas e outras abordagens de automação de testes de software vêm se tornando medidas bem vistas para amenizar o problema (FANTINATO et al., 2004). Em essência, a automação de testes de software é a prática de criar, normalmente via *scripts*, meios do computador executar a tarefa de testar o código. Tendo isso em vista, as atividades de teste consideradas para automação são a geração de testes e execução (FANTINATO et al., 2004).

2.3.1 Técnicas de Geração e Automação de Teste

Há diversas técnicas para automação da geração de testes. As principais técnicas são: ***Record/Playback***, ***script programming***, ***data-driven*** e ***keyword-driven*** (FANTINATO et al., 2004).

2.3.1.1 *Record/Playback*

Em essência, a técnica de *Record/Playback* consiste em repetir, de forma automática, um teste executado sobre a interface gráfica. As ações produzidas são gravadas pelo usuário e transcritas para um *script*. A partir desse *script*, pode-se repetir as ações do usuário sempre que necessário. Assim, testa-se a aplicação utilizando-se das ações anteriormente bem sucedidas. Caso alguma das ações tenha insucesso, significa que algo foi quebrado. Os *scripts* armazenam não apenas o procedimento de teste, mas também os dados utilizados pelo usuário para a execução deles (KENT, 2007).

Como vantagens relativas ao *Record/Playback*, pode-se citar a sua praticidade. No entanto, a quantidade de desvantagens é alta. Ao lidar-se com um número grande de casos de teste, essa técnica mostra dificuldades relacionadas ao custo e à complexidade de manutenção. Além disso, traz uma inerente sensibilidade a mudanças do software. O fato de ser específico a uma determinada interface também inviabiliza a reutilização dos testes (FANTINATO et al., 2004).

2.3.1.2 *Script programming*

Essa técnica é um melhoramento da técnica de *Record/Playback*. Possui a mesma dinâmica de gravar as ações executadas por um usuário para, a partir de um *script*, repetí-las. Diferencia-se pelo fato de alterar os *scripts* originais de teste, inserindo novos comportamentos ao teste. Dessa forma, pode-se manter uma cópia do *script* original e fazer diversas cópias que contemplem ações não executadas (KENT, 2007).

Traz taxas maiores de manutenabilidade e reutilização que a *Record/Playback*, como vantagens. Entretanto, ainda possui a problemática de gerar grandes quantidades de *scripts* (FANTINATO et al., 2004).

2.3.1.3 *Data-driven*

Também conhecida como técnica orientada a dados. Nessa abordagem, que é um aprimoramento da técnica de *script programming*, dá-se foco maior aos dados de teste utilizados em cada *script*. Faz-se a extração desses dados dos *scripts*, mantendo-os armazenados em outro arquivo. Assim, apenas os procedimentos - lógica de execução - e as ações de testes escritos nos *scripts* permanecem. Dessa maneira, os *scripts* devem aces-

sar os arquivos contendo os dados de teste para que haja a execução correta dos testes (KENT, 2007).

Isto posto, observam-se algumas vantagens do uso dessa técnica: maior abstração dos testes, permitindo que haja o trabalho tanto do projetista de testes quanto do implementador, sem que haja preocupação de um nível de abstração com o outro. Enquanto o projetista visa a seleção dos melhores dados para o teste, o implementador prepara os *scripts*. A facilidade em generalizar os *scripts* e alterar os dados de teste pode ser vista como outra vantagem dessa técnica. Permite maior reutilização e manutenibilidade dos *scripts* (FANTINATO et al., 2004).

2.3.1.4 *Keyword-driven*

Por meio dessa técnica, pretende-se realizar a extração da lógica (os procedimentos), de teste dos *scripts*, mantendo-os armazenados em outro arquivo. A partir do arquivo de procedimentos, os *scripts* retiram a lógica para efetuar as ações determinadas (KENT, 2007).

Essa abordagem facilita a adição, a remoção ou a alteração dos passos de execução de algum teste (FANTINATO et al., 2004).

2.4 Resumo do Capítulo

O capítulo abordou temas relevantes para o aprofundamento do conhecimento relativo ao presente trabalho de conclusão de curso. Foram tratados os níveis e classificação dos testes de software, citando técnicas de derivação de casos de testes. Abordou-se também sobre a reutilização de software, abrangendo algumas das maneiras mais comuns de alcançar isso, como o uso de bibliotecas de classes, *design patterns* e *frameworks*. Por fim, há uma seção sobre a Geração de Testes, que observa contemporaneidade do assunto, na medida em que muitas ferramentas buscam a automação da produção de testes.

3 Metodologia

Este capítulo aborda a metodologia e procedimentos utilizados durante o desenvolvimento do trabalho. O capítulo está organizado em seções. Na seção 3.1 é explanado sobre a metodologia *Scrum*. Em seguida, na seção 3.2, é apresentada a metodologia aplicada para o desenvolvimento do software, bem como relatado o fluxo de atividades e a execução dessas atividades conforme o cronograma.

3.1 Planejamento da Metodologia Aplicada

Para o desenvolvimento do trabalho de conclusão de curso, utilizou-se uma adaptação de metodologia para o desenvolvimento do *framework*, embasada em práticas do *Scrum* (SCRUM GUIDES, 2014) e do eXtreme Programming (WELLS, 2009).

O *Scrum* é uma metodologia ágil voltada para o desenvolvimento de software. É um *framework* simples que visa dividir o trabalho em pequenos ciclos de esforço, focados na entrega contínua de resultados para o cliente (SCRUM GUIDES, 2014). O *Scrum* nomeia esses pequenos ciclos como *sprints*. As *sprints* devem conter a mesma quantidade de tempo, onde o desenvolvimento deverá ser executado. O *Scrum* também prevê eventos. Um evento é uma reunião de caráter informal entre os membros do time. Cada evento possui seus objetivos, mas de forma geral, servem para fazer o alinhamento da equipe. Os principais eventos são o *Sprint Planning Meeting*, *Daily Scrum*, *Sprint Review Meeting* e o *Sprint Retrospective*. O *Sprint Planning Meeting* é uma reunião para que a equipe planeje o próximo ciclo de desenvolvimento. O *Daily Scrum* é uma breve reunião, com tempo médio de quinze minutos, que deve ser feita em pé e todos os dias, num horário fixo. Nesta reunião a equipe diz o que cada um fez e o que pretende fazer até o próximo encontro. O *Sprint Review Meeting* e o *Sprint Retrospective* normalmente são feitos no mesmo dia e um seguido do outro. São reuniões feitas com a finalização da *Sprint*. Nestas reuniões discute-se sobre como foi o andamento do último ciclo de desenvolvimento e o que pode ser melhorado (SCRUM GUIDES, 2014).

Outra prática importante no *Scrum* é a forma como se deve lidar com os requisitos do projeto. No *Scrum* os requisitos são representados por cartões, conhecidos como histórias de usuário. É importante ressaltar que não há uma relação de um para um entre as histórias de usuário e os requisitos, na medida em que um requisito pode derivar mais de uma história de usuário. Essas histórias são definidas e pontuadas, conforme o esforço necessário para a sua implementação. Durante o *Sprint Planning Meeting* há a criação do da lista de histórias de usuário que deverão ser implementadas naquele ciclo de desenvolvimento. Neste processo, a equipe deve também definir os responsáveis por cada história

e qual o esforço necessário para alcançar o sucesso em cada uma delas (SCRUM GUIDES, 2014).

Para o desenvolvimento deste trabalho, as *Sprints* tiveram uma duração de quatorze dias e fez-se uso de algumas práticas como *Planning Poker*, Padronização do Código, Programação em Pares (WELLS, 2009), estimativas relativas, *timebox*, *backlog*, definição de pronto e quadro de tarefas (SCRUM GUIDES, 2014).

3.1.1 Fluxo de Atividades Geral

Utilizando as metodologias definidas para a execução do trabalho de conclusão de curso 01 e 02, foi estabelecido um fluxo de atividades que possibilitou o alcance dos objetivos do projeto.

Inicialmente, foram realizados estudos sobre a temática escolhida. Em seguida, foram determinados o escopo do projeto e a proposta inicial,, resumindo os objetivos, justificativa, cronograma preliminar, entre outros aspectos introdutórios que guiaram as atividades seguintes. O estudo e a definição da metodologia utilizada, bem como o do suporte tecnológico, ocorreram em paralelo com a pesquisa e a documentação do referencial teórico base para o desenvolvimento do *framework* proposto. Após essas definições, foi elaborada a primeira versão detalhada da proposta do software a ser implementado. Definiu-se também uma prova de conceito, de forma a auxiliar o refinamento da proposta. Em seguida, ocorreu a implementação da prova de conceito utilizando os ciclos de pesquisa-ação e *sprints*. Com os resultados, houve a finalização da monografia com as atualizações que se fizeram necessárias. A finalização do trabalho de conclusão de curso 01 ocorreu após a primeira apresentação para a banca avaliadora.

O trabalho de conclusão de curso 02 iniciou-se com as correções e refinamentos solicitados na apresentação. Em seguida, deu-se início ao desenvolvimento do *framework* proposto utilizando pesquisa-ação e adaptação do Scrum. A cada término de ciclo, foi efetuada uma análise dos resultados obtidos, verificando se os objetivos estabelecidos no início do ciclo foram satisfatoriamente atendidos. Por fim, incrementou-se à monografia com o relato do que foi executado, descrição do *framework* e seus resultados. A apresentação do trabalho de conclusão de curso 02 representará o fim do projeto.

A Figura 4 representa o fluxo de atividades utilizado no trabalho de conclusão de curso, como descrito anteriormente.

A Figura 5 demonstra a adaptação do Scrum e o uso de pesquisa-ação no desenvolvimento do *framework*.

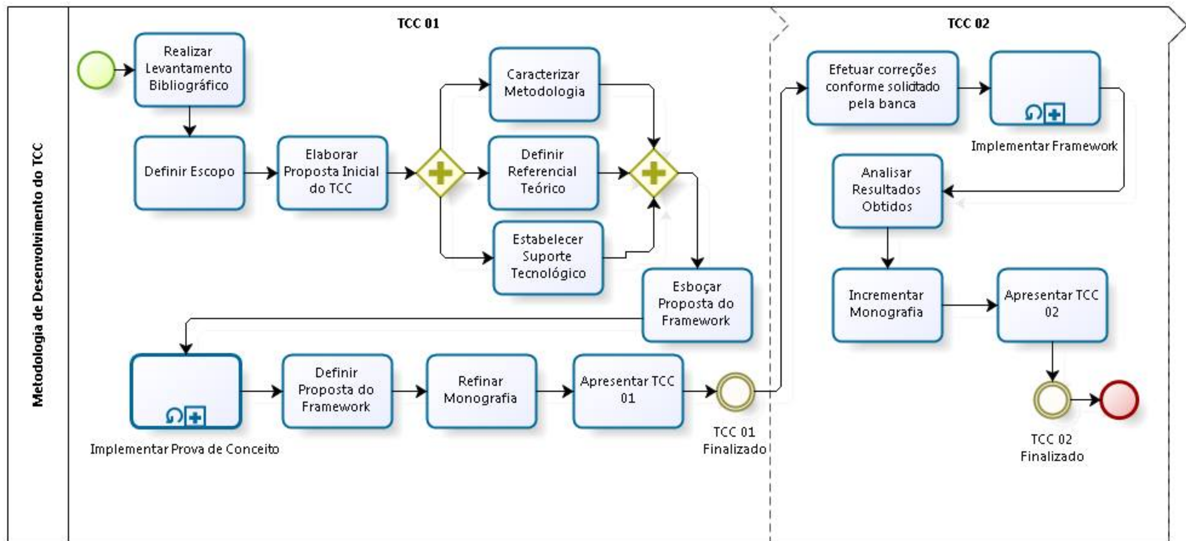

Figura 4 – Fluxo de atividades do trabalho de conclusão de curso

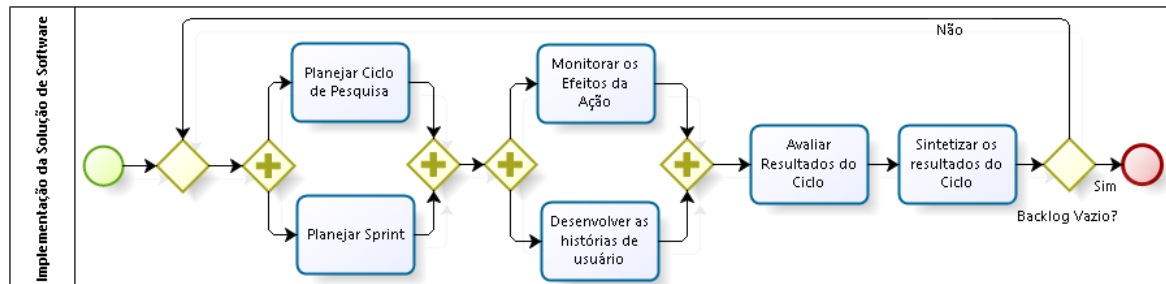

Figura 5 – Fluxo de atividades da implementação da solução de software

3.1.2 Detalhes da Execução do Desenvolvimento

Inicialmente, definiu-se os requisitos iniciais do projeto. Para esse fim, foi especificado um épico que atende o objetivo geral do projeto: Geração de Testes Unitários a partir de marcações em comentários no próprio código fonte. A partir desse épico, foram derivados as seguintes *features*:

Identificação dos elementos da linguagem de programação Grails: permitir que o gerador de testes seja capaz de identificar os elementos da linguagem de programação Grails, garantindo análises futuras;

Análise dos métodos da camada de domínio (*model*): permitir que o gerador faça uma análise dos métodos presentes nas classes de domínio, produzindo casos de teste específicos para cada método desta camada;

Análise dos métodos da Camada de Controle (*controller*): permitir que o gerador faça uma análise dos métodos presentes nas classes controladoras, produzindo casos de teste específicos para cada método desta camada;

Desenvolvimento do Framework: permitir que o gerador criado possa ser extensível e adaptável para diferentes linguagens de programação.

O detalhamento de cada *feature* ocorreu conforme execução do projeto e maior entendimento do desenvolvimento de cada funcionalidade.

3.1.2.1 FE01 Identificação dos elementos da linguagem de programação Grails

Esse funcionalidade é responsável pela criação da gramática capaz de identificar a linguagem de programação Grails. Para isso, foram identificadas 10 histórias de usuários que estão listadas na Tabela 1. Essa *feature* foi a primeira detalhada e implementada. No desenvolvimento, houve dificuldade de construir uma gramática com o mínimo de conflitos possível, capaz de identificar toda a linguagem Grails e, por esse motivo, a implementação dessa funcionalidade foi concluída na *sprint* 3.

3.1.2.2 FE 02 Análise dos métodos da camada de domínio (*model*)

Essa funcionalidade engloba a análise e a criação dos testes de modelo. Embora a quantidade de histórias de usuários desse *feature* seja reduzida, a dificuldade técnica associada foi desafiadora. Objetivou-se, nesse momento, concluir um gerador de testes para *model* que fosse capaz de gerar casos de testes para métodos da camada de modelo. Esse funcionalidade foi implementada nas *sprints* 4 e 5 e as histórias relacionadas estão listadas na Tabela 2.

3.1.2.3 FE 03 Análise dos métodos da Camada de Controle (*controller*)

Funcionalidade similar à *feature* 02, cujo objetivo pode ser resumido na geração de testes. Entretanto, a *feature* 03 focou em gerar testes para a camada de controle. Os métodos abordados foram os responsáveis pela criação, leitura, atualização e exclusão de uma classe. A implementação dessa funcionalidade ocorreu em paralelo com a *feature* 02, ou seja, nas *sprints* 4 e 5. Segue, na Tabela 3, as histórias dessa funcionalidade.

3.1.2.4 FE 04 Desenvolvimento do Framework

A *feature* 04 visou tornar o gerador de testes em um *framework*, tornando-o extensível e adaptável para diferentes linguagens de programação. Para isso, foi necessário identificar os pontos do software que são genéricos para qualquer linguagem e os pontos que são adaptáveis, conhecidos como *hotspots*. Para auxiliar nesse processo, utilizou-se diagramas de classe e de sequência. Em seguida, executou-se uma refatoração aplicando os padrões de projeto identificados como candidatos e, por fim, documentou-se o uso do *framework*. Essa funcionalidade foi implementada na *sprint* 6, e suas histórias estão detalhadas na Tabela 4.

Tabela 1 – Histórias da *feature* 01

ID	Descrição	Pt.	Status
US01	Eu, como desenvolvedor, quero identificar os elementos da declaração de variáveis, para análises futuras desses elementos.	3	Feito
US02	Eu, como desenvolvedor, quero identificar os elementos de declarações iniciais, para análises futuras desses elementos.	3	Feito
US03	Eu, como desenvolvedor, quero identificar os elementos de atribuição de variáveis, para análises futuras desses elementos.	3	Feito
US04	Eu, como desenvolvedor, quero identificar os elementos de declaração de classes, para análises futuras desses elementos.	3	Feito
US05	Eu, como desenvolvedor, quero identificar os elementos de declaração de métodos, para análises futuras desses elementos.	3	Feito
US06	Eu, como desenvolvedor, quero identificar os elementos de declaração de estruturas condicionais, para análises futuras desses elementos.	3	Feito
US07	Eu, como desenvolvedor, quero identificar os elementos de declaração estruturas de repetição, para análises futuras desses elementos.	3	Feito
US08	Eu, como desenvolvedor, quero identificar os elementos de comentários, para análises futuras desses elementos.	3	Feito
US09	Eu, como desenvolvedor, quero identificar elementos de expressões da linguagem grails, para análises futuras desses elementos.	5	Feito
US10	Eu, como desenvolvedor, quero identificar elementos que forma uma strings, para análises futuras desses elementos.	3	Feito

3.1.3 Cronograma

Os cronogramas a seguir evidenciam a execução, visão temporal, das atividades referentes ao trabalho de conclusão de curso como um todo, incluindo as atividades referentes ao TCC_01 e ao TCC_2.

Tabela 2 – Histórias da *feature* 02

ID	Descrição	Pt.	Status
US11	Eu, como desenvolvedor, quero analisar métodos de validação de classes de domínio, para gerar casos de testes.	8	Feito
US12	Eu, como desenvolvedor, quero analisar métodos de instanciação de classes de domínio, para gerar casos de testes.	8	Feito
US13	Eu, como desenvolvedor, quero analisar métodos de relacionamento de classes de domínio, para gerar casos de testes	8	Feito

Tabela 3 – Histórias da *feature* 03

ID	Descrição	Pt.	Status
US14	Eu, como desenvolvedor, quero analisar métodos de criação de classes controladoras, para gerar casos de testes.	8	Feito
US15	Eu, como desenvolvedor, quero analisar métodos de exclusão de classes controladoras, para gerar casos de testes.	8	Feito
US16	Eu, como desenvolvedor, quero analisar métodos de alteração de classes controladoras, para gerar casos de testes.	8	Feito
US17	Eu, como desenvolvedor, quero analisar métodos de visualização de classes controladoras, para gerar casos de testes.	8	Feito

Tabela 4 – Histórias da *feature* 04

ID	Descrição	Pt.	Status
US18	Eu, como desenvolvedor, quero identificar os pontos de generalização e hotspots, para criação do framework.	5	Feito
US19	Eu, como desenvolvedor, quero implementar padrão de projeto Builder, para facilitar a criação de arquivos de testes em outras linguagens.	8	Feito
US20	Eu, como desenvolvedor, quero implementar padrão de projeto Factory, para facilitar a criação de arquivos de coleta de dados em outras linguagens.	8	Feito
US21	Eu, como desenvolvedor, quero refatorar o framework desenvolvido, para facilitar na adaptação e uso em outras linguagens.	5	Feito
US22	Eu, como desenvolvedor, quero documentar o framework, para facilitar o seu uso.	5	Feito

Tabela 5 – Cronograma para o trabalho de conclusão de curso 1

Atividades	Agosto	Setembro	Outubro	Novembro	Dezembro
Realizar Levantamento Bibliográfico	X				
Definir Escopo	X				
Elaborar Proposta Inicial do TCC	X	X			
Caracterizar Metodologia		X			
Definir Referencial Teórico		X			
Estabelecer Suporte Tecnológico		X			
Esboçar Proposta		X	X		
Implementar Prova de Conceito			X		
Analisar Resultados da Prova de Conceito			X		
Definir Proposta do Projeto				X	
Refinar Monografia				X	
Apresentar TCC 01				X	X

Tabela 6 – Cronograma para o trabalho de conclusão de curso 2

Atividades	Fevereiro	Março	Abril	Maio	Junho
Efetuar Correções conforme solicitado pela banca	X				
Implementar Incremento de Software		X	X	X	
Testar Incremento de Software		X	X	X	X
Analisar Resultados Obtidos			X	X	X
Incrementar Monografia					X
Apresentar TCC 2					X

3.2 Resumo do Capítulo

Este capítulo abordou sobre a metodologia *Scrum* e alguns de seus pontos básicos. Foi definida também uma adaptação da metodologia de *Scrum*. Em seguida, detalhou-se o fluxo de atividades do projeto, e apresentou-se a execução das atividades no cronograma.

4 Scarefault

Neste capítulo, será abordado o Scarefault, o produto final desse trabalho. Este capítulo está dividido em seções. A primeira seção aborda uma visão geral do Scarefault. A segunda seção faz uma análise da arquitetura do *framework* e suas dependências. Na seção seguinte, explana-se a respeito da utilização do Scarefault.

4.1 Scarefault: Visão Geral

O Scarefault é um *framework* que tem a intenção de gerar testes unitários de forma semiautomatizada. Para isso, ele necessita da intervenção do desenvolvedor que o utiliza por meio de breves especificações nos comentários de documentação. A partir dessas especificações, o Scarefault coleta informações necessárias para a produção dos casos de teste e gera um arquivo contendo os testes solicitados. É importante ressaltar que, para esse trabalho, os testes gerados são apenas testes caixa preta.

O *framework* propõe-se em permitir a acoplagem de diferentes linguagens de programação. Para isso, há a necessidade da construção de um *parser* que identifique a sintaxe da linguagem alvo por meio de duas ferramentas: Flexc++ e Bisonc++. Essas duas ferramentas são comuns em contextos de produção de compiladores. Com o *parser* implementando, o desenvolvedor pode implementar as especificidades relacionadas aos testes da linguagem alvo, tendo como base o código oferecido pelo Scarefault.

O Scarefault foi escrito em C++, e está licenciado dentro das condições da GPLv3. Todo o código fonte desse framework encontra-se em: <https://github.com/Scarefault/scarefault>.

4.2 A Arquitetura do Scarefault

Essa seção evidencia os elementos da arquitetura do Scarefault. Como ele foi desenhado, quais conceitos foram utilizados, e como ele funciona. Em um primeiro momento, são exploradas visões estáticas da arquitetura. A visão dinâmica da arquitetura é evidenciada pela seção 4.3.

4.2.1 Visão de Pacotes

Partindo-se de uma visão macroscópica, o Scarefault pode ser dividido em dois módulos básicos: o *identifier* e o *generator*. A Figura 6 ilustra esses módulos básicos.

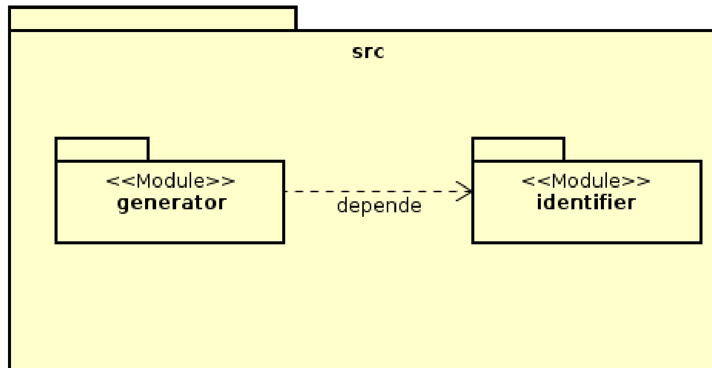

Figura 6 – Representação dos módulos do Scarefault

***identifier*:** é responsável pela identificação das regras que regem a gramática da linguagem de programação alvo, bem como pela coleta dos dados necessários para a produção dos casos de teste.

***generator*:** responsável pela geração dos casos de teste. A partir dos dados coletados das especificações do desenvolvedor e do próprio código fonte, esse módulo é capaz de gerar casos de teste.

Expandindo-se os módulos, podem ser observados os pacotes associados a cada um deles, e o relacionamento entre esses pacotes, como mostra a Figura 7.

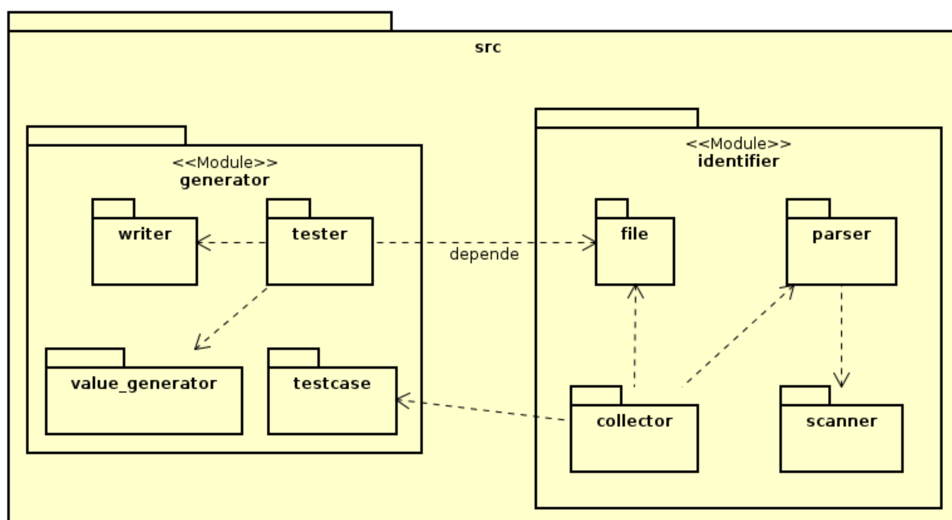

Figura 7 – Representação expandida dos módulos do Scarefault

Como indicado na Figura 7, os pacotes associados ao módulo *identifier* são:

***scanner*:** responsável pela análise léxica do arquivo contendo o código fonte em avaliação. As classes contidas nesse pacote são geradas pelo Flexc++ durante a compilação.

***parser*:** responsável pela análise sintática do arquivo contendo o código em avaliação. As ações que ele executa quando determinada regra gramatical é encontrada são, essencialmente, ações de coleta de dados. As classes associadas a esse pacote são geradas pelo Bisonc++ durante a compilação.

***collector*:** responsável pela coleta de dados necessários para a criação de testes unitários.

***file*:** responsável por guardar os dados coletados do código fonte. Dados esses, tais como as dependências (*libraries* importadas), métodos e classe do arquivo em análise.

Os pacotes associados ao módulo *generator*, como mostra a Figura 7, são:

***tester*:** responsável pela construção dos arquivos de teste. A partir dele é que os testes ganham forma, conforme a linguagem de programação.

***testecase*:** responsável por guardar os dados relacionados a cada caso de teste especificado pelo desenvolvedor, a partir dos comentários no código fonte.

***value_generator*:** responsável por gerar valores randômicos para o desenvolvimento dos testes.

***writer*:** responsável por escrever as informações do arquivo de teste, enviadas pelo *tester*. Para isso, ele cria o arquivo e escreve todas as informações a ele solicitadas.

4.2.2 Visão de Classes

As Figuras 6 e 7 mostram a visão de pacotes do Scarefault e como eles se relacionam, bem como a responsabilidade que cabe a cada um deles. Aprofundando-se mais, chega-se a uma visão das entidades associadas a cada pacote. A visão de quais são essas entidades e como elas se relacionam pode ser verificada pelo Modelo de Domínio, mostrado na Figura 8.

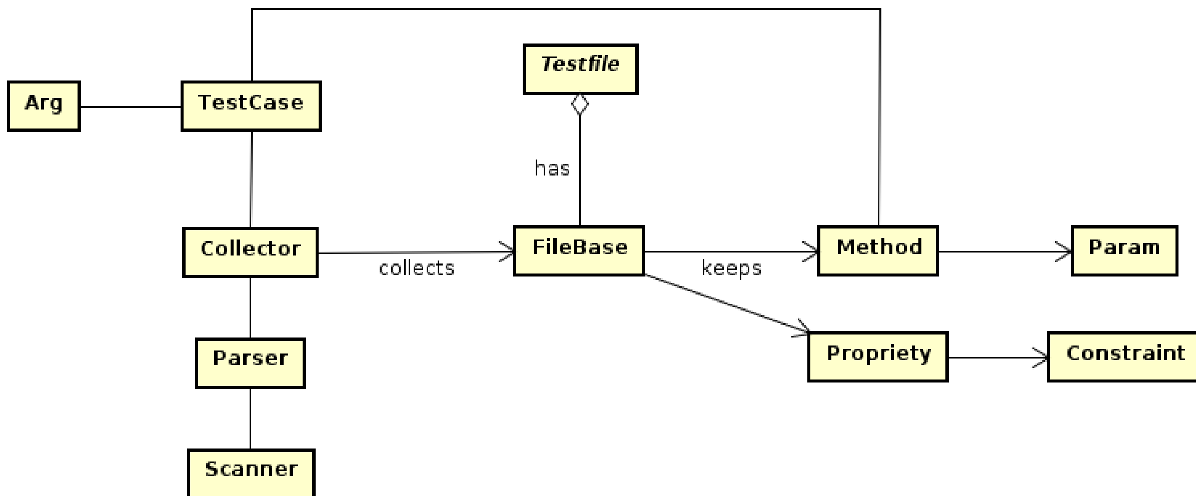

Figura 8 – Diagrama de domínio do Scarefault

O modelo de domínio na Figura 8 demonstra a relação entre as principais entidades que o *framework* trabalha. O *Collector* recolhe as informações essencias para os testes a partir do *Parser* que, por sua vez, recebe um *stream* de dados a partir do *Scanner*. Como dados importantes para a produção de testes pelo Scarefault, tem-se métodos (*Method*) e seus parâmetros (*Param*) e, para arquiteturas voltadas ao MVC, tem-se propriedades (*Propriety*) e suas restrições (*Constraint*). Esses dados estão intimamente ligados ao arquivo em análise, portanto, são armazenados na entidade arquivo (*FileBase*). Há também a entidade *TestCase* que tem papel fundamental na produção dos testes, pois carrega as informações dispostas pelo desenvolvedor sobre os testes a serem criados. Uma parte importante dos casos de teste são os argumentos (*Arg*) que devem ser passados ao método em avaliação durante o teste.

Partindo-se do modelo de domínio exposto na Figura 8, pode-se visualizar o diagrama de classe, mostrado na Figura 9.

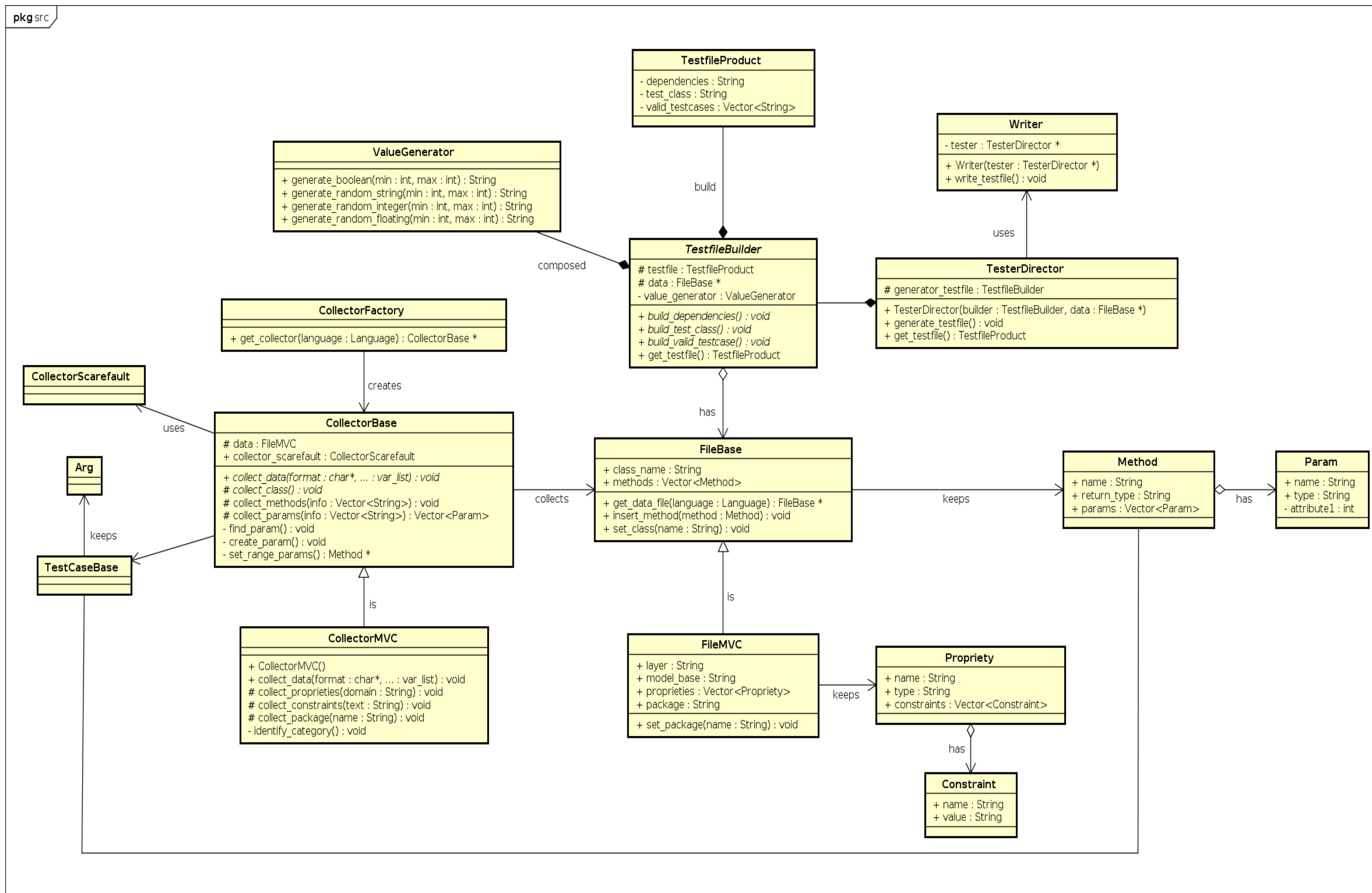

Figura 9 – Diagrama de classe completo do Scarefault

O diagrama de classes permite obter uma visualização mais apurada dos relacionamentos entre as classes contidas no *framework*. As principais entidades do *framework*, apresentadas na Figura 8, estão presentes e muitas delas passaram a ter um conjunto auxiliar de classes, de forma a permitir a sua adaptação ao contexto em que será inserida.

Por meio da Figura 9, pode-se identificar o uso de alguns *design patterns* de criação, com o intuito de permitir a extensibilidade do *framework* e a sua facilidade de uso. Para esse fim, foram usados os conceitos do ***builder pattern*** e ***factory pattern***.

4.2.2.1 *Builder Pattern*

O uso do *builder pattern* procurou colaborar com a criação de objetos complexos. Seu intuito é a separação do processo de construção da representação em si. Desse modo, é possível que o mesmo processo de construção sirva para diferentes representações (GAMMA et al., 1994).

No desenvolvimento do *framework*, observou-se essa necessidade na criação de objetos `Testfile`. Essa entidade é diferente para cada linguagem associada a ele. Não é possível manter uma mesma representação desse objeto, no entanto, o processo de criação de um `Testfile` é igual em todos os casos: escreve-se as dependências da classe de teste, o cabeçalho da classe de teste, as ações iniciais para os testes (*setup*), os casos de teste, as ações para depois de cada teste (*teardown*) e a finalização do arquivo.

Nesse sentido, viu-se a solução proposta pelo *builder pattern* como saída para o problema dentro do contexto do desenvolvimento do Scarefault. A Figura 10 mostra como a solução foi desenhada. É importante ressaltar que construiu-se uma adaptação do padrão, e não a sua forma pura.

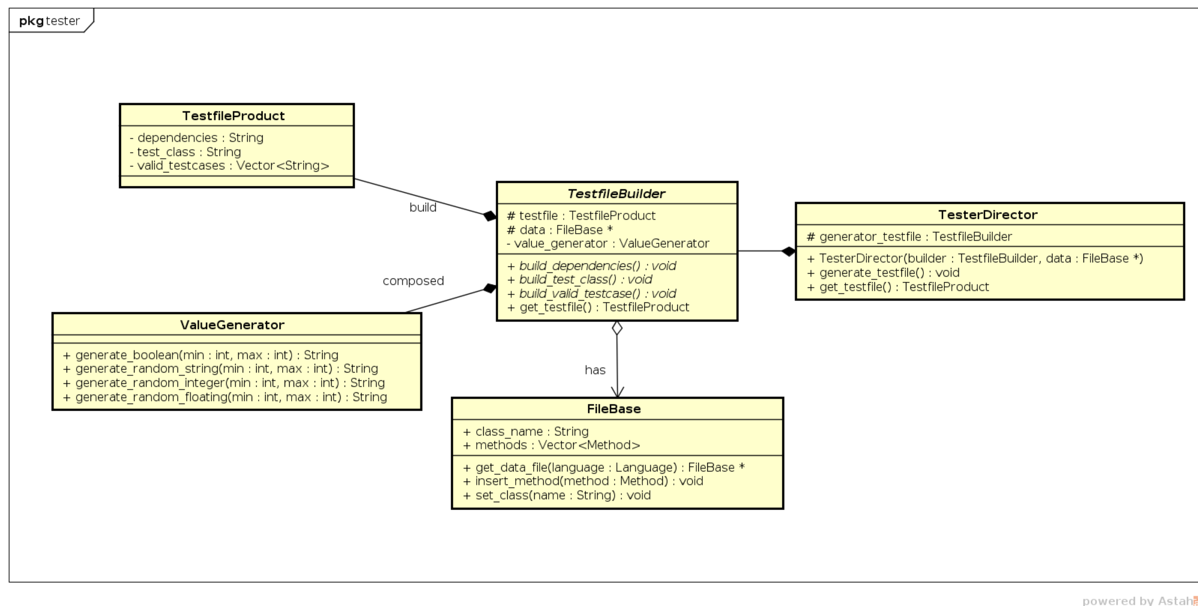

Figura 10 – Diagrama de classes apresentando o *builder pattern* no *framework*

O objeto a ser construído é da classe `TestfileProduct`. Há diversas representações que esse objeto pode tomar, conforme a linguagem. No entanto, o processo de construção é o mesmo. Os passos desse processo são definidos pela classe `TestfileBuilder`. Essa classe é uma classe `virtual` (abstrata). Ela define uma interface que deve ser implementada pelas suas classes filhas, afim de construir o objeto `TestfileProduct`. Dessa forma, garante-se que qualquer classe com a responsabilidade de construir um `TestfileProduct` terá funções-membro que criem os elementos essenciais do produto. Com a garantia que os passos de construção estão implementados, pode-se fazer o processo de criação. Essa atividade é responsabilidade do `TestfileDirector`. O método `generate_testfile()` possui a definição do processo de construção, a partir dos passos estipulados pelo `TestfileBuilder`.

4.2.2.2 *Factory Pattern*

Factory pattern é uma solução pensada com o propósito de deixar a instanciação de objetos de uma mesma família mais flexível. Através desse padrão, é possível instanciar um objeto sem expor a lógica dessa ação, bem como ser capaz de referenciar-se a novos objetos por uma interface comum (GAMMA et al., 1994).

No construção do Scarefault , deparou-se com a necessidade de flexibilizar a instanciação de objetos de coleta de dados. Isso fornece maior liberdade ao desenvolvedor usuário do *framework* para criar suas próprias classes de coleta, que estarão relacionadas a uma determinada linguagem de programação e suas peculiaridades.

Por esse motivo, foi pensado em uma adaptação do *Factory pattern* como solução. A Figura 11 mostra a solução desenhada para o *framework*.

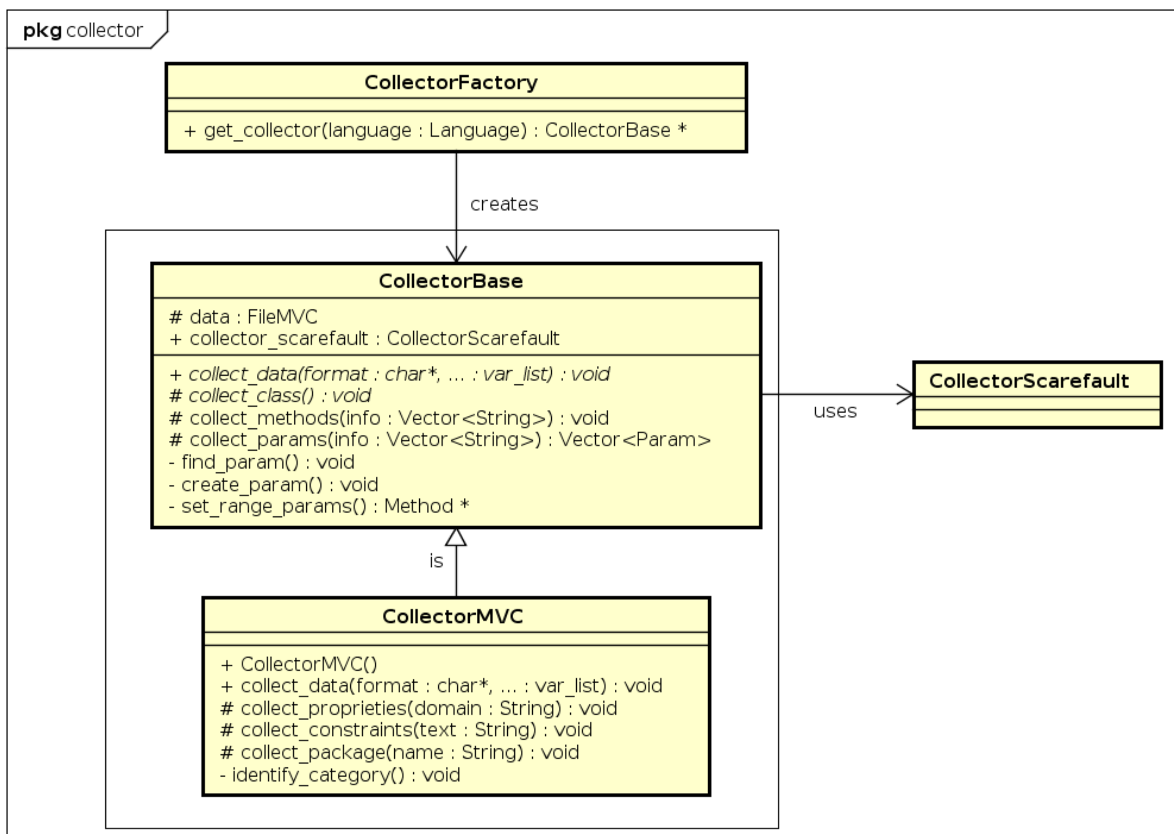

Figura 11 – Diagrama de classes apresentando o *factory pattern* no *framework*

A solução de *Factory pattern* desenvolvida utiliza-se de uma classe `virtual` denominada `CollectorBase`. Optou-se por uma classe desse tipo, ao invés de uma `virtual` pura (*interface*), pois imaginou-se em já deixar implementadas determinadas funções-membro auxiliares, preparadas para serem usadas pelo usuário. Essas funções-membro, embora já implementadas, são passíveis de sobreescrita, o que dá a possibilidade do desenvolvedor decidir criar as sua própria forma de coletar determinados elementos ou utilizar-se de uma já disponível.

A única função-membro obrigatória de implementação para as classes derivadas da `CollectorBase` é a `collect_data(char *, ...)`. Além disso, já há uma outra classe de coleta preparada para linguagens que se utilizem do padrão de projeto MVC: a `CollectorMVC`. Ela também estende a `CollectorBase`, mas traz outras funções-membro auxiliares.

A classe `CollectorFactory` é reponsável por esconder a lógica de instanciação da família Collector. Por meio da `get_collector()` um objeto `CollectorFactory` retorna uma instância do Collector solicitado na classe cliente.

4.3 A Utilização do Scarefault

A seção 4.2 apresentou a visão estática da arquitetura do Scarefault. Nesta seção, pretende-se mostrar a visão dinâmica, por meio de explicações a partir do uso do *framework*. A utilização do Scarefault pode ser dividida em dois objetivos: a adaptação do *framework* para atender uma nova linguagem e a geração de testes unitários.

4.3.1 Adaptação para uma Nova Linguagem

A adaptação do Scarefault para novas linguagens exige o uso de um ambiente específico de desenvolvimento, bem como o uso de janelas de extensibilidade do *framework*, chamadas de *hotspots*.

4.3.1.1 Usando o Ambiente de Desenvolvimento

Para auxiliar na customização do *framework*, criou-se um ambiente virtual de forma a garantir a instalação das dependências necessárias do Scarefault. Para levantar esse ambiente, é necessário que o usuário instale as versões mais recentes das seguintes ferramentas:

Git: ferramenta de controle de versão. Versão 1.9.1;

VirtualBox: ferramenta de virtualização de máquinas virtuais. Versão 5.0.22;

Vagrant: ferramenta de criação e configuração de ambientes virtuais de desenvolvimento. Versão 1.8.4.

Com essas ferramentas instaladas, o usuário deve efetuar o clone do projeto e executar o comando responsável por criar o ambiente virtual com as configurações pre-definidas. Nesse momento, será instalado e configurado o Flexc++, Bisonc++ e demais dependências. Por fim, deve-se acessar essa máquina virtual, e então iniciar a customização no código do Scarefault. O Código 4.1 apresenta as linhas de comando referentes aos passos citados:

Código 4.1 – Levantamento do ambiente de desenvolvimento

```
1 $ git clone git@github.com:Scarefault/scarefault.git
2 $ cd scarefault
3 $ vagrant up
4
5 # Aguarde instalação e configuração das dependências
6
7 $ vagrant ssh
```

O Scarefault contempla a linguagem de programação Grails até o momento. Para facilitar a compilação do Scarefault, criou-se um Makefile responsável por fazer o *build*. Isso se fez necessário porque a compilação dos arquivos-fonte exige um linha de comando extensa. Além disso, há algumas alterações fundamentais a serem feitas em determinados arquivos para que se faça a comunicação entre o *scanner* e o *parser*.

4.3.1.2 Explorando a Extensibilidade do Scarefault

O Scarefault por si só não é uma aplicação. Ele precisa ter determinados pontos estendidos para que possa ser executado. Essencialmente, o Scarefault permite a acoplagem de novas linguagens. Para isso, há alguns pontos de extensão para que isso ocorra. São eles: criação do agente de coleta de dados e do agente de construção de casos de teste. No entanto, para que se faça uso do Scarefault, é necessária a adição de um *parser* para a nova linguagem.

4.3.1.2.1 Adição de um *parser* para a nova linguagem

O *parser* é desenvolvido utilizando-se o Flexc++, versão 1.08, e o Bisonc++, na versão 4.05. Para uso dessas ferramentas, recomenda-se a utilização da documentação oficial, disponível em <https://fbb-git.github.io/flexcpp/> e <https://fbb-git.github.io/bisoncpp/>, respectivamente. O Apêndice B é um guia básico para o uso dessas ferramentas.

Após a gramática da nova linguagem ter sido criada e compilada utilizando as ferramentas Flexc++ e o Bisonc++, é gerado o *parser* e *scanner* em C++. Para prosseguimento do uso do *framework* com a nova gramática, há a necessidade de uma série de alterações nos arquivos de código-fonte, tanto do *parser* quanto do *scanner* gerados. Algumas delas já são previstas no uso normal do Flexc++ e do Bisonc++ para a comunicação das duas ferramentas. A seguir, são apresentadas as alterações que objetivam o uso da gramática criada no *framework*. Salienta-se a importância de replicar essas alterações no *parser* e *scanner* gerados para a nova linguagem. As alterações com o objetivo de gerar a comunicação entre as ferramentas são descritas no Capítulo de Resultados.

O Código 4.2 apresenta as alterações exigidas para o bom funcionamento do Scarefault, efetuadas no arquivo `Parser.h`.

Código 4.2 – Alterações no arquivo `Parser.h` para uso do scarefault

```
1 class Parser: public ParserBase
2 {
3    Scanner d_scanner;
4       Collector::CollectorFactory * factory;
5       Collector::CollectorBase * collector;
```

```
6       LogSystem::Log log;
7
8     public:
9         explicit Parser(Collector::Language,
10                              std::istream &in = std::cin,
11                              std::ostream &out = std::cout);
12 [...]
```

No Código 4.2, nas linhas 4 e 5, observa-se a declaração de alguns objetos que serão usados pelo *parser*. A linha 9 mostra a adição de um construtor para a classe, cuja a diferença para o construtor gerado pelo Bisonc++ é o acréscimo de um parâmetro do tipo `Collector::Language`.

O Código 4.3 apresenta as alterações exigidas no arquivo `Parser.ih`.

Código 4.3 – Alterações no `Parser.ih` para uso do Scarefault

```
1 #include "Parser.h"
2
3 Parser::Parser(Collector::Language language,std::istream &in, std
    ::ostream &out)
4 {
5       d_scanner.switchStreams( in );
6       d_scanner.setSval(&d_val__);
7       factory = new Collector::CollectorFactory();
8       collector = factory->get_collector(language);
9 }
10 [...]
```

No Código 4.3, a linha 3 inicia a implementação do construtor declarado no arquivo `Parser.h`. A linha 5 altera o *stream* de dados que o *parser* deve utilizar (por *default*, o *stream* é a entrada padrão). Isso é necessário, na medida em que o *stream* que se deseja ler não é a entrada padrão, mas sim o arquivo do código fonte, que será passado como argumento para o construtor. A linha 6 é um dos passos para implementar a comunicação entre o *scanner* e o *parser*. A linha 7 instancia um objeto `CollectorFactory` que é usado na linha 8 para instanciar, conforme a linguagem de programação passada, um coletor de dados apropriado.

Com as alterações efetuadas, pode-se usar o *parser* implementado. O Código 4.4 demonstra o *parser* em uso.

Código 4.4 – *Parser* em uso

```
1 [...]
2
3 std::ifstream target;
```

```
4 target.open( argv[ SOURCE_FILE_NAME ], std::fstream::in );
5
6 [...]
7
8 std::istream& stream = target;
9
10 Parser parser( GRAILS, stream ) ;
11 parser.parse();
12
13 target.close();
14
15 [...]
```

No Código 4.4, a linha 4 instancia um objeto `std::ifstream` para abrir o arquivo passado como argumento para a aplicação, representado como `argv[SOURCE_FILE_NAME]`. O arquivo foi aberto apenas para leitura. A linha 8 converte o objeto `target` para o tipo `std::istream` que pode ser tratado pelo *parser*. A linha 10 instancia um objeto do tipo `Parser` passando como argumentos a linguagem a ser usada e o *stream* aberto para o arquivo. A linha 11 chama a função que faz o parser sobre o *stream* passado como argumento, ou seja, o código fonte.

4.3.1.2.2 Criação do agente de coleta de dados

O Scarefault é um *framework* que visa a geração de testes unitários, possibilitando o uso em diferentes linguagens de programação. Para que isso seja feito, é necessário que haja a coleta dos dados a serem explorados para a criação dos testes unitários. Devido a essa necessidade, criou-se a classe `CollectorBase`. Essa classe, conforme exposto no Código 4.2, é usada pelo *parser*. Sua instanciação é feito segundo o Código 4.3. A seleção do collector ideal é realizada por meio de um objeto `CollectorFactory`. Para explicar o que ocorre a partir desse ponto, é preciso retomar a Figura 11, que traz o diagrama de classes representando o *Factory Pattern*, desenhado para a selecionar o *collector* em tempo de execução.

Para exemplificar o uso desse ponto de extensão, foi utilizado o Grails, com o intuito de demonstrar o uso do Scarefault. A Figura 12 traz o diagrama de classes para o *Factory Pattern* estendido.

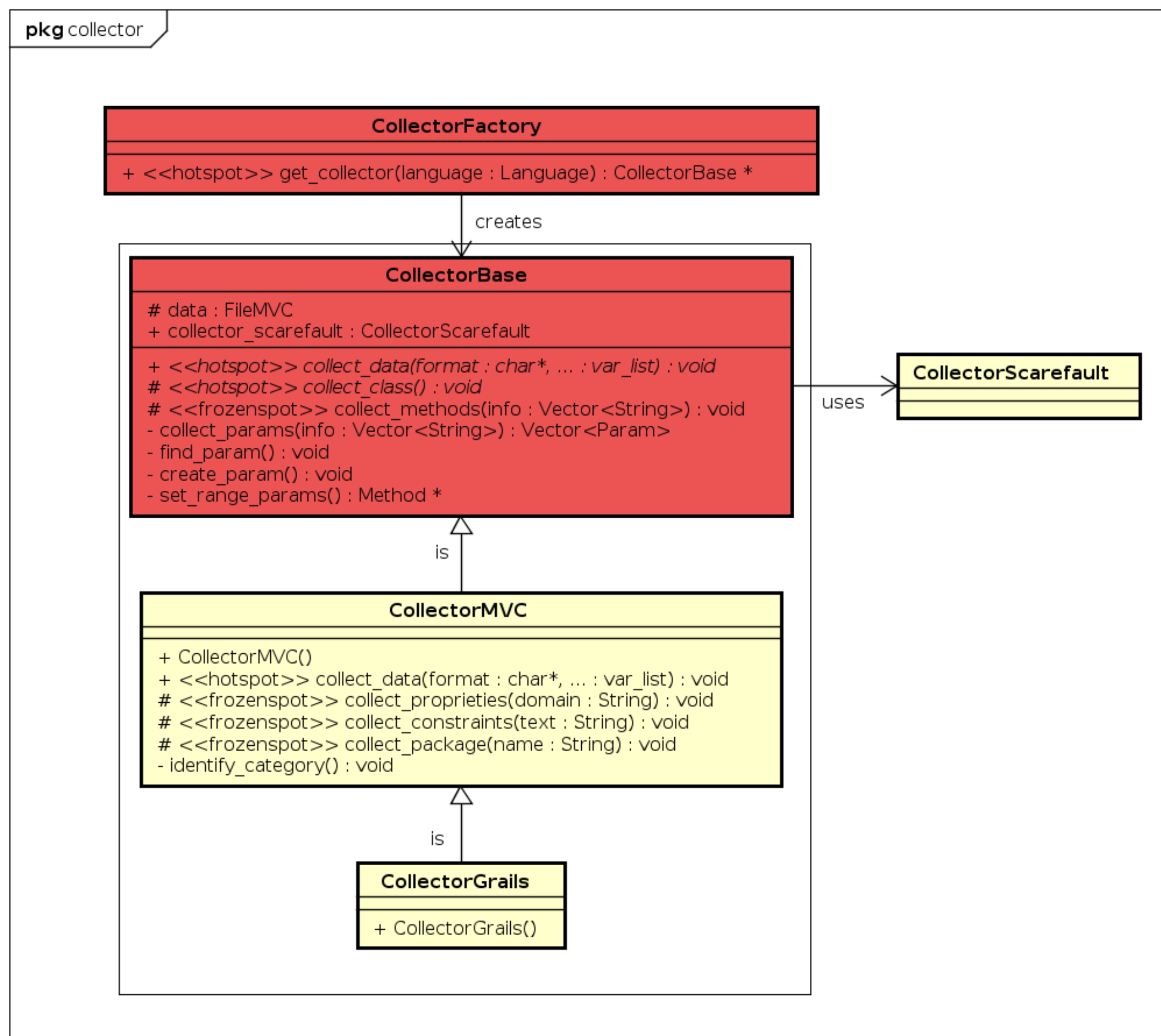

Figura 12 – Diagrama de classes exemplificando a extensibilidade do *factory pattern* no *framework*

Na Figura 12 observa-se que a classe `CollectorBase`, em vermelho, é o ponto de extensão. Como já descrito na subseção 4.2.2.2, há também uma classe, a `CollectorMVC`, já disponível para se estender, caso a linguagem alvo esteja acoplada arquiteturalmente ao padrão de projeto MVC. Dessa forma, cria-se uma classe derivada da `CollectorMVC`, a `CollectorGrails`.

Devido a essa herança, o *framework* disponibiliza, à nova classe derivada, alguns benefícios. Pela Figura 12, observa-se que há algumas funções marcadas como *hotspots* e outras como *frozenspots*. Cabe a definição desses termos:

***Frozenspots*:** são partes fixas do *framework*. São as funções já implementadas e que devem percamencer inalteradas. São comuns para qualquer linguagem alvo e auxiliam na adaptação do *framework* para novas linguagens.

***Hotspots*:** são as partes flexíveis do *framework*. São nesses pontos que há a extensão das funções do *framework* por adição de código, especificando a funcionalidade.

Assim sendo, a `CollectorGrails` possui a mesma interface que a sua classe mãe. O *framework* abre como *hotspot* a função membro `void collect_data(char *, ...)`, que é a única função membro com escopo público. Qualquer classe que derive da `CollectorBase` deve implementar essa função. Não é diferente para a `CollectorGrails`. No entanto, ela é derivada da `CollectorMVC`, que já implementa essa função, de forma que a `CollectorGrails` já está isenta de fazê-lo, mas não proibida. Caso o desenvolvedor queira criar sua própria versão da `void collect_data(char *, ...)` ele pode criá-la. Para isso, ele conta com algumas funções auxiliares já implementadas (*frozenspots*), como a `void collect_methods (std::vector<std::string>)`.

Com a `CollectorGrails` já implementada, pode-se utilizar outro *hotspot* do contexto da família dos *collectors*: a `CollectorFactory`. A Figura 12 apresenta essa classe também como ponto de extensão, pois fazendo ajustes no método `CollectorBase * get_collector(Language)` estende-se sua capacidade de seleção de coletores de dados. Um ajuste, nesse ponto, influencia em qual coletor será escolhido dentro do *parser*.

O Código 4.5 apresenta a alteração feita para estender a `CollectorBase * get_collector (Language)`.

Código 4.5 – Implementação e extensão da `get_collector(Language)`

```
1  CollectorBase * CollectorFactory::get_collector( Collector::
      Language language )
2   {
3     CollectorBase * instance;
4
5     switch( language )
6     {
7       case GRAILS:
8         instance = new CollectorGrails();
9         break;
10    }
11
12    return instance;
13  }
```

No Código 4.5, a linha 1 mostra o cabeçalho da função. Nesse ponto, é importante ressaltar o uso como parâmetro da enumeração `Language`. Ela é definida no `namespace Collector` para auxiliar o desenvolvedor. No caso exemplificado, foi adicionado à enumeração a linguagem Grails. Na linha 5, usa-se um `switch` para, a partir do parâmetro `language`, selecionar qual a linguagem. A linha 8 faz a instanciação de um objeto do tipo

`CollectorGrails` caso a linguagem seja orientada pela linguagem base do Grails. Para cada novo coletor adicionado, um novo caso deve ser criado no `switch`. Na linha 12, retorna-se a instância.

A Figura 13 traz um diagrama de sequência para melhorar visualização dos efeitos no Scarefault quando da instanciação de um objeto `Parser`. Essa visão é interessante para observar o comportamento esperado da `CollectorFactory`.

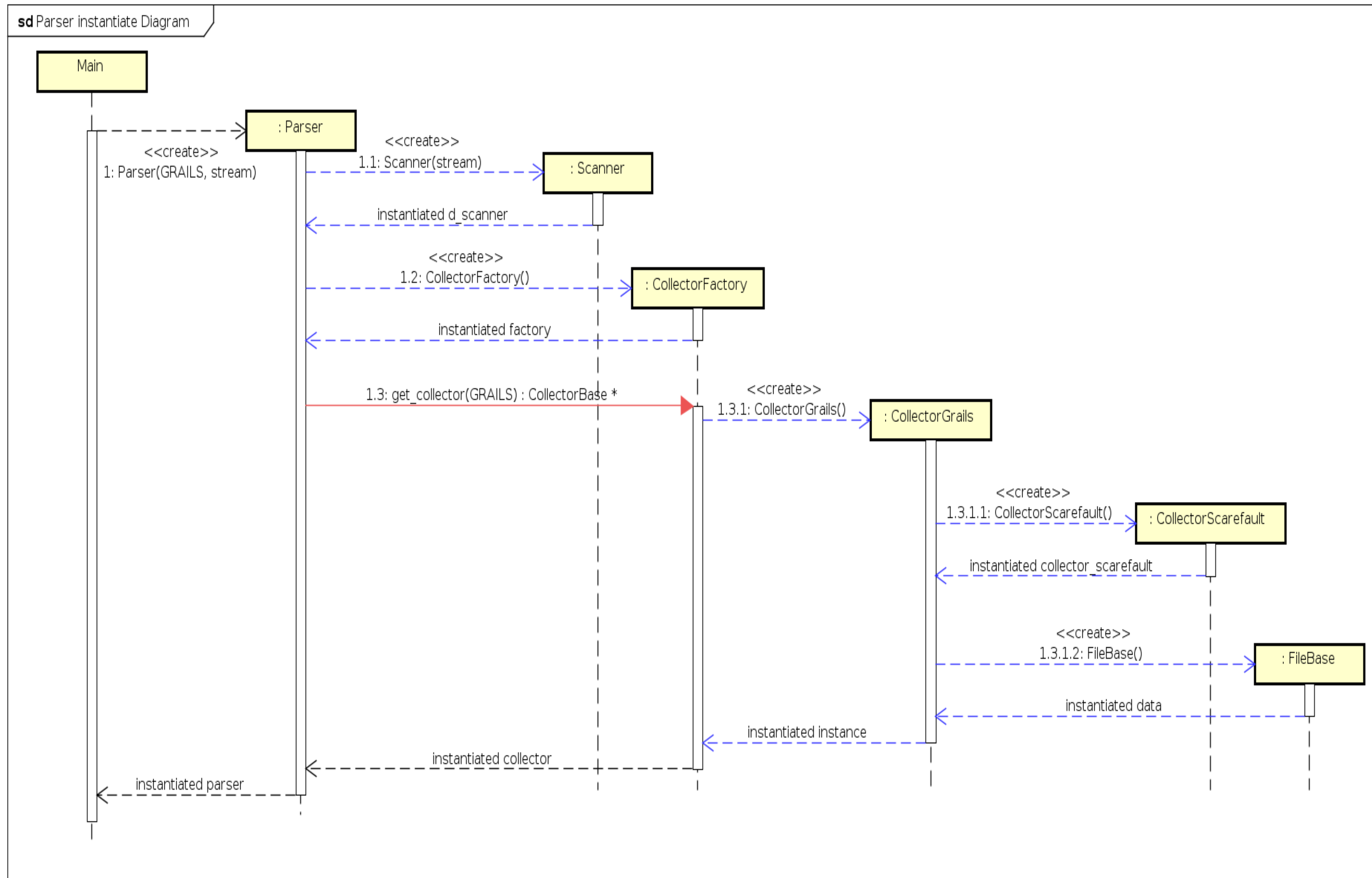

Figura 13 – Diagrama de sequência da instanciação de um `Parser` e os efeitos na `CollectorFactory`

A Figura 13 mostra os objetos sendo instanciados após a instância de um objeto `Parser` ser solicitada. A `CollectorFactory`, recebendo como argumento a valor Grails, faz a instanciação de um objeto `CollectorGrails` que, por sua vez, traz consigo outros dois objetos importantes em sua formação: uma instância de um `CollectorScarefault`, que será responsável por extrair informações dos comentátios de documentação, e um `FileBase`, que mantém as informações coletadas a partir do código fonte.

Nesse momento, o `Parser` possui um agente de coleta de dados que atuará durante o processamento do arquivo com o código fonte a ser testado. Devido à instanciação no construtor do `Parser`, é possível usá-lo no arquivo que contém as regras gramaticais da linguagem alvo, nesse exemplo, a linguagem que orienta a programação na plataforma Grails. No Código 4.6, é possível observar como ocorre esse processo.

Código 4.6 – Agente coletor em uso no *parser*

```
1 [...]
2 untyped_method_stmt:
3   DEF IDENTIFIER '(' param_list ')' {
4     std::string identifier_token( $2 );
5     std::string params_token( $4 );
6
7     collector->collect_data( "mp", identifier_token.c_str(),
    params_token.c_str() );
8   }
9 ;
10 [...]
```

No Código 4.6, a linha 2 apresenta a declaração de um novo símbolo não terminal da gramática, que é definido pela regra gramatical ditada na linha 3. Como pode ser observado, a regra ilustrada no exemplo, é a regra que rege a definição de métodos sem um tipo de retorno. As linhas 4 e 5 atribuem a *strings* os valores identificados na segunda e na quarta posições da regra, identificadas pelos *tokens* `INDETIFIER` e `param_list`.

A linha 7 apresenta o uso do agente coletor na definição da gramática. O operador `->` é usado pois o objeto `collector` é um ponteiro ao agente coletor. A sintaxe de C++ exige que a referência a membros pertencentes a um objeto, a partir de um ponteiro, seja feita por meio desse operador. A função `void collect_data(char * format, ...)` recebe como parâmetro uma *string* em *c-style* e uma quantidade ilimitada de argumentos do tipo *string* em *c-style*. O primeiro parâmetro, `format`, espera uma *string* que indique os tipos de dados que estão sendo coletados. A Tabela 7 apresenta uma referência ao tipo de dado e o código a ser usado no parâmetro `format`. No caso exemplificado, o argumento é `"mp"`, indicando a passagem do nome do método e a lista de parâmetros desse método. Em seguida, são listados os dados a serem coletados. Isso é feito passando os itens declarados

nas linhas 4 e 5.

Tabela 7 – Descrição das letras identificadoras e o respectivo tipo de dado a ser coletado

Letra Identificadora	**Tipo de Dado**
P	Pacote
c	Classe
m	Método
p	Lista de Parâmetros

É importante ressaltar que, embora já haja uma versão disponível da `collect_data (char *, ...)` para as classes derivadas da `CollectorMVC`, pode-se sobreescrevê-la. A Tabela 7 é relativa à versão já disponibilizada pelo *framework*. No caso, preferiu-se utilizar a já disponível.

Ao utilizar a função `void collect_data(char *, ...)`, uma série de chamadas a outras funções é feita. A Figura 14 apresenta a dinâmica desse processo.

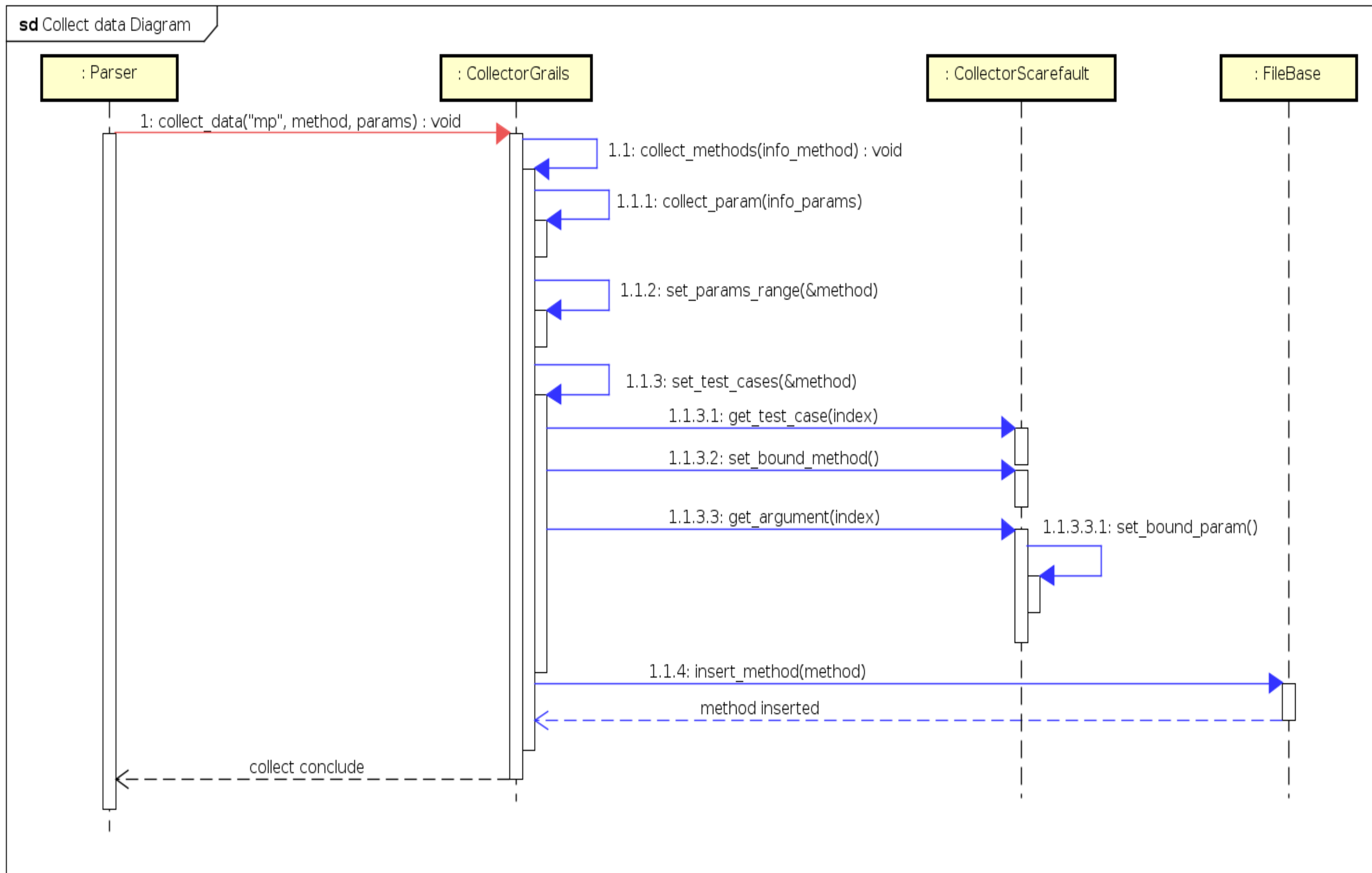

Figura 14 – Diagrama de sequência das chamadas de funções por trás da `void collect_data(char *, ...)`

Nesse diagrama, houve a diferenciação da ação tomada por um *hotspot*, em vermelho, e um *frozenspot*, em azul. É possível verificar que o `void collect_data(char *, ...)` realiza a coleta dos dados em etapas. Inicialmente, é coletado as informações do método, sendo elas o nome do método e nome e tipo da lista de parâmetros. Em seguida, são definidos parâmetros de teste conforme o intervalo solicitado no código-fonte, caso haja. São criados, então, os casos de testes baseado nas informações coletadas, relaciona-os com o método identificado. Por fim, os casos de testes são inseridos no `FileMVC`, e ocorre a conclusão da coleta dos dados desse método.

Com isso, a primeira parte para o uso do Scarefault está finalizada, ou seja, o *framework* está preparado para coleta de dados. A segunda etapa é responsável pela produção dos casos de teste.

4.3.1.2.3 Criação do agente de construção de casos de teste

Com os dados essenciais para os testes já coletados por um `CollectorBase` e armazenados em um `FileBase`, faz-se necessária a criação de um agente responsável pela produção do arquivo de teste. Para isso, o *framework* disponibiliza como ponto de extensão a classe `TestfileBuilder`. Como já explicitado na Figura 10, ela faz parte de um conjunto de classes que seguem o *Builder Pattern*. Esse padrão permite separar a criação de objetos complexos em dois elementos: a sua representação e o processo de criação. Arquivos de teste são elementos, que têm estruturas diferentes, conforme a linguagem de programação, mas o processo de construção é semelhante.

Retomando a Figura 10, na qual é apresentada a estrutura do *Builder Pattern* no projeto, para o caso de exemplo com o Grails o diagrama de classes passa a ser o exposto na Figura 15.

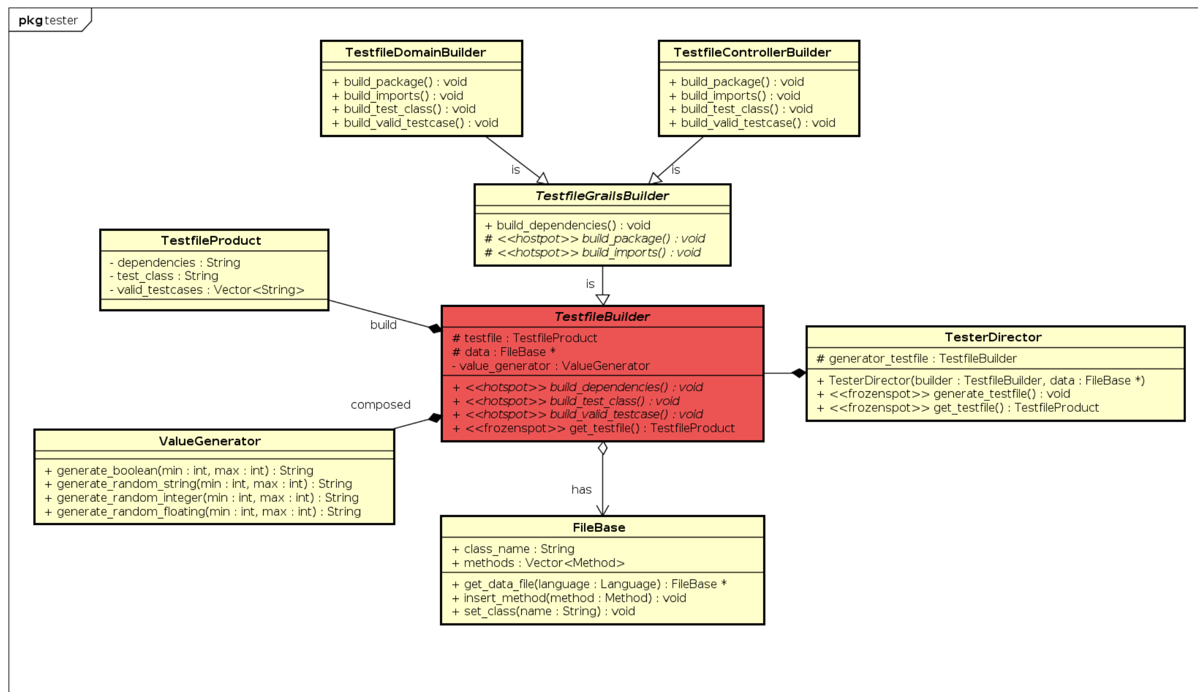

Figura 15 – Diagrama de classes exemplificando a extensibilidade do *builder pattern* no *framework*

Na Figura 15, destaca-se em vermelho o ponto de extensão: a classe TestfileBuilder. Essa classe é responsável pela construção do objeto `testfile`, pertencente à classe `TestfileProduct`. Essa classe possui três *hotspots* para adaptação, as funções `void build_dependencies()`, `void build_class()` e `void build_validtestcase()`. Qualquer classe derivada a partir dessa classe é obrigada a implementar sua própria versão dessas funções. Isso permite a extensão do *framework* e garante que o desenvolvedor seguirá as regras para a construção do `TestfileProduct` pelo processo comum. No caso de exemplo com o Grails, criou-se a classe `TestfileGrailsBuilder`. Com base nessa classe, surgem outras duas: `TestfileControllerBuilder` e `TestfileDomainBuilder`. A classe `TestfileGrailsBuilder` implementa a `void build_dependencies()`, mas passa a responsabilidade para as suas classes derivadas de implementar outras duas funções: a `void build_package()` e a `void build_imports()`. Assim, as classes `TestfileControllerBuilder` e `TestfileDomainBuilder` são responsáveis por implementar as quatro funções exigidas no processo construção de arquivos de teste para Grails. Essas duas classes exigem, pois testes de controller configuram-se de forma distinta de testes feitos para *models*. Assim, optou-se por realizar duas classes diferentes para esses dois tipos de arquivos.

Tendo o *Builder Pattern* extendido pode-se usar o*framework*. O Código 4.7 apresenta um exemplo de `main` para o uso do Scarefault.

Código 4.7 – Agente testador sendo usado na função `main`

```
1 #define ACCEPTABLE_QTD_ARGS 3
```

```
2 #define SOURCE_FILE_NAME 2
3 #define OPTION 1
4 #define TYPE_TEST 3
5 [...]
6 if( !strcmp( argv[ OPTION ], "generate" ) )
7 {
8   if( !strcmp( argv[ TYPE_TEST ], "controller" ) )
9   {
10     TesterDirector tester( new TestfileControllerBuilder(
      collector_ptr->get_data() ) );
11     tester.generate_testfile();
12
13     Writer writer( &tester, argv[ SOURCE_FILE_NAME ] );
14     writer.write_testfile();
15   } else if( !strcmp( argv[ TYPE_TEST ], "domain" ) )
16   {
17     TesterDirector tester( new TestfileDomainBuilder(
      collector_ptr->get_data() ) );
18     tester.generate_testfile();
19
20     Writer writer( &tester, argv[ SOURCE_FILE_NAME ] );
21     writer.write_testfile();
22   }
23 }
24 [...]
```

No Código 4.7, a linha 6 faz a comparação da opção passada pela linha de comando, sendo que a palavra *generate* entra no bloco de instruções. As linhas 8 e 15 fazem a comparação da *string* passada na linha de comando: caso seja igual a *controller*, executa um trecho de código específico. Se for *domain*, executa outro trecho de código. As linhas 10 e 17 explicitam a instanciação de um objeto da classe `TestfileDirector`. Esse objeto exige no seu construtor a passagem de um objeto do tipo `TestfileBuilder`, que por sua vez exige um `FileBase` para trabalhar os dados. Essa é a forma de trabalho do *Builder Pattern*: o `TesterDirector` mantém o processo de construção do arquivo de teste, requisitando apenas o tipo de representação que deve ser construída. Isso é revelado por meio do tipo de *builder* passado como argumento. As linhas 11 e 18 chamam a função que constrói o arquivo de teste.

As linhas 13 e 20 instanciam um objeto do tipo `Writer`. Ele é responsável por escrever o arquivo de teste gerado pelo `TesterDirector`. Para isso, ele recebe o objeto `TesterDirector` e o nome do arquivo fonte.

Para melhor visualizar a dinâmica do funcionamento por trás do Código 4.7, a Figura 16 traz um diagrama de sequência que reflete esse funcionamento.

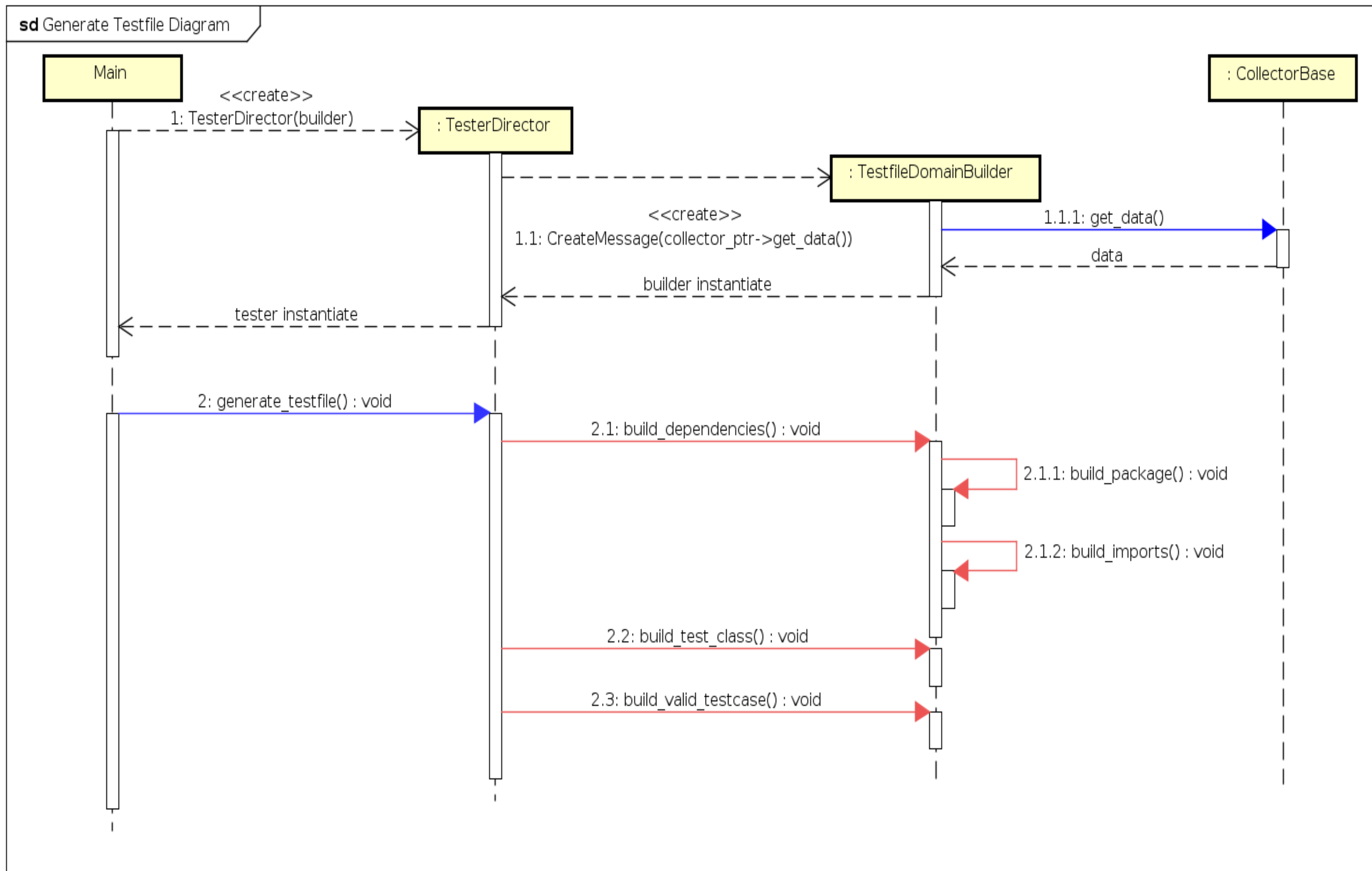

Figura 16 – Diagrama de sequência das chamadas de funções por trás da `void generate_testfile()`

O diagrama na Figura 16 mostra a instanciação de um objeto do tipo TesterDirector Para o exemplo, foi selecionado o caso em que a representação para o `testfile` é de uma *domain*. Assim sendo, é passado como argumento para o construtor do `testerDirector` uma nova instância de `TesterDomainBuilder`, que por sua vez exige no seu construtor um objeto do tipo `FileBase`. Este é passado através da chamada da função `get_data()` do `CollectorBase`. Assim, o `TesterDirector` é instanciado.

Na segunda parte do diagrama de sequência, observa-se a chamada da função `void generate_testfile()`. A cor azul indica que ela é um *frozenspot*. Ela carrega o processo de construção de um arquivo de teste. Para isso, ele faz a chamada dos passos de construção, implementados pelo `TestfileDomainBuilder`. Vale lembrar que todas as funções chamadas após a `void generate_testfile()` são hotspots, em vermelho. O diagrama da Figura 16 é interessante, pois evidencia bem o *hollywood principle*: o desenvolvedor não faz a chamada ao *framework*. É o *framework* que chama as funções criadas pelo desenvolvedor.

4.3.2 Geração de Testes Unitários

Além da utilização do Scarefault como *framework*, pode-se usá-lo como aplicação, após às devidas adaptações por parte dos desenvolvedores. Supondo que o Scarefault já tenha sido adaptado para Grails, o alvo inicial desse projeto, segue um exemplo de como utilizar o Scarefault como uma aplicação.

A geração de testes unitários feita pelo Scarefault é dada em duas etapas: na primeira é solicitada a criação de casos de teste ao Scarefault. Isso quer dizer que o Scarefault vai gerar, a partir de determinados parâmetros expostos pelo usuário, casos de teste. Na segunda etapa, é requisitada a geração dos testes definidos como casos de teste, completados com informações adicionais pelo usuário.

Para a demonstração, foi criado uma função `main`. Por meio dessa função, é possível a chamada do Scarefault e seus benefícios por linha de comando. Segue o Código 4.8 que traz um *template* de chamada do Scarefault.

Código 4.8 – Template para chamada do Scarefault pela linha de comando

```
1 $ ./scarefault <opção> <caminho ao arquivo a ser testado> [
    categoria do padrão MVC]
```

O argumento referenciado como `<opção>` indica qual a opção que o usuário escolheu. Essa `<opção>` pode variar entre `create` e `generate`.

create: opção que solicita ao Scarefault a criação de casos de teste. Isso é efetuado através da leitura de alguns dados diponibilizados pelo usuário dentro dos comentários de documentação dos métodos.

generate: opção que requisita ao Scarefault a geração dos testes, baseando nos casos de teste criados anteriormente. Os casos de teste criados pela opção `create` devem ser completados com o valor esperado pelo teste. O usuário também pode acrescentar casos de testes, caso sejam necessários, seguindo, seguindo o padrão proposto pelo Scarefault.

O argumento referenciado como `<caminho ao arquivo a ser testado>` deve ser preenchido com o caminho que leva até o arquivo com o código fonte a ser testado. O argumento referenciado como `[categoria do padrão MVC]` deve ser preenchido ou com `controller` ou com `domain`.

Considerando-se o que foi dito, toma-se o arquivo domain.groovy como exemplo. Ele contém o Código 4.9.

Código 4.9 – Código do arquivo domain.groovy

```
1 package math
2
3 class Math {
4   def sumTwoNumbers(int parcel1, int parcel2) {
5     def total = parcel1 + parcel2
6     return total
7   }
8 }
```

O usuário do Scarefault deve adicionar ao Código 4.9 comentários de documentação contendo algumas marcações próprias do Scarefault. Essas marcações são referenciadas pela Tabela 8.

Tabela 8 – Marcações de comentários para uso do Scarefault

Marcação	Descrição
@param	Indica para o Scarefault o nome do parâmetro
@range	Indica para o Scarefault os valores limites relativos ao parâmetro
@test	Indica para o Scarefault os argumentos que devem ser passados ao método durante o teste. Evidência um caso de teste
@expect	Indica para o Scarefault o valor esperado no caso de teste

Assim, com a adição do comentário de documentação, o Código 4.9 passa a ficar como o Código 4.10.

Código 4.10 – Código do arquivo domain.groovy com as marcações do Scarefault

```
1 package math
2
3 class Math {
```

```
4    /**
5      @param parcel1 @range 0..100
6      @param parcel2 @range 0..100
7    */
8    def sumTwoNumbers(int parcel1, int parcel2) {
9      def total = parcel1 + parcel2
10     return total
11   }
12 }
```

O usuário então solicita ao Scarefault a criação dos casos de teste. Isso se dá pela linha de comando: `./scareafault create domain.groovy`. Assim, o código passa a ficar como o Código 4.11.

Código 4.11 – Código do arquivo domain.groovy com os casos de teste

```
1  package math
2
3  class Math {
4    /**
5      @param parcel1 @range 0..100
6      @param parcel2 @range 0..100
7      @test (0, 50) @expect INSERT_HERE
8      @test (1, 50) @expect INSERT_HERE
9      @test (99, 50) @expect INSERT_HERE
10     @test (100, 50) @expect INSERT_HERE
11     @test (50, 0) @expect INSERT_HERE
12     @test (0, 1) @expect INSERT_HERE
13     @test (0, 99) @expect INSERT_HERE
14     @test (0, 100) @expect INSERT_HERE
15   */
16   def sumTwoNumbers(int parcel1, int parcel2) {
17     def total = parcel1 + parcel2
18     return total
19   }
20 }
```

O usuário então deve inserir as expectativas dele para cada caso de teste e, em seguida, executar a linha de comando: `./scareafault generate domain.groovy domain`. Dessa forma, o Scarefault gera os testes especificados em um outro arquivo, chamado de `domainTests.groovy` no mesmo diretório do `domain.groovy`.

4.4 Resumo do Capítulo

Este capítulo detalho o produto final desse trabalho: o Scarefault. Abordou-se incialmente uma visão geral, seguindo da análise da arquitetura e, por fim, a utilização do *framework*.

5 Resultados

Este capítulo apresenta os resultados obtidos. Eles serão apresentados, sendo correlacionando aos objetivos específicos, indicados na Seção 1.4.2. Para cada objetivo, foi gerada uma seção neste capítulo.

5.1 Identificação de Regras Gramaticais da Linguagem Alvo Inicial

A linguagem escolhida como alvo inicial do *framework* desenvolvido neste trabalho foi a linguagem que orienta à programação na plataforma Grails. *Grails* é um *web framework* para plataforma Java (GRAILS, 2015). Para melhor entender a sintaxe da linguagem de programação alvo, partiu-se da documentação oficial do Groovy, com o intuito de aprofundar o conhecimento sobre as regras gramaticais do alvo inicial do *framework*.

Para a identificação das regras gramaticias, foram selecionadas duas ferramentas, bastante utilizadas em contextos de desenvolvimento de compiladores e *parsers*: Flexc++ e Bisonc++. Ambas as ferramentas, Flexc++ e Bisonc++, foram desenvolvidas por Frank B. Brokken, gerente de segurança em TI e conferencista na *University of Groningen*, Holanda.

A primeira ferramenta foi utilizada para definir *scanners*, cujo propósito é a identificação de padrões léxicos em um arquivo de texto. Esses padrões foram definidos a partir de *regex*, na primeira seção do arquivo fonte do Flexc++. Na próxima seção, definiu-se o que acontece quando um dos padrões é identificado. No Código 12, demonstra-se um fragmento do arquivo produzido para a execução do Flexc++.

Código 5.1 – Fragmento do código fonte para o Flexc++

```
1 TRUE     true
2 FALSE    false
3 BOOLEAN  {TRUE}|{FALSE}
4
5 %x oneline_string
6 %x multiline_string
7 %x multiline_comment
8
9
10 %%
11
12 "class" {
13   *d_val = matched();
```

```
14   return Parser::CLASS;
15 }
```

Observa-se no Código 5.1, entre as linhas 12 e 14, um exemplo de como o Flexc++ é usado: define-se um padrão léxico a ser encontrado no texto linha (12) e a ação a ser executada para aquele padrão. No caso exemplificado, o padrão a ser encontrado é a palavra class. Assim, quando o Flexc++ identifica essa palavra, ele armazena o resutado da comparação entre a palavra em análise e o padrão definido (linha 13). Caso haja correspondência o Flexc++ retorna um *token* definido no arquivo gerado pelo Bisonc++ (linha 14).

Um outro recurso do Flexc++ utilizado para a identificação das regras gramaticais do Grails foi a definição de *mini scanners*. É um recurso fornecido pelo Flexc++ que permite a definição de *scanners* com regras mais específicas dentro do *scanner* maior. Isso foi útil para definir os padrões de identificação de *strings* e comentários do alvo inicial deste trabalho.

O Código 5.2 apresenta o *mini scanner* definido para identificação de *strings*.

Código 5.2 – Implementação do mini scanner de strings

```
1 // Condição inicial para o mini scanner
2 "\'\'\'" |
3 "\"\"\"" {
4   more();
5   begin( StartCondition__::multiline_string );
6 }
7
8
9 // Mini scanner e suas regras
10 <multiline_string> {
11   "\'\'\'"  |
12   "\"\"\"" {
13     begin( StartCondition__::INITIAL );
14     *d_val = matched();
15     return Parser::STRING;
16   }
17
18   .|\n {
19     more();
20   }
21 }
```

O uso de *mini scanners* é semelhante ao *scanner* comum. É importante ressal-

tar que há dois tipos de *mini scanners*: os exclusivos e o inclusivos. Um *mini scanner* inclusivo permite a inclusão de novos padrões ao identificador a partir do momento em que este é chamado. Já o *exclusivo* cessa as atividades das regras definidas pelo *scanner* e faz valer apenas as definidas por ele. Para este trabalho, foram definidos apenas *mini scanners* exclusivos. Além disso, *mini scanners* possuem duas funções a mais: a *StartCondition___::[nome do mini scanner]*, que inicia o *mini scanner*, e a *StartCondition___::INITIAL* que finaliza seu uso. Quando o *scanner* encontra no texto em análise o padrão que dá início a um *mini scanner*, ele interrompe o seu processo e inicia o do *mini scanner*. Assim, apenas as regras do *mini scanner* passam a ser válidas. No momento que o *miniscanner* encontra o padrão de finalização ele retorna processo do *scanner*.

Em conjunto com o Flexc++, foi utilizado o Bisonc++. Como já apresentada, a primeira ferramenta identifica padrões léxicos, enquanto que a segunda ferramenta é responsável pela definição das regras sintáticas da gramática.

Da mesma forma que o Flexc++, o arquivo a ser compilado pelo Bisonc++ é divido em duas seções. A primeira traz algumas diretivas que fornecem infomações ao Bisonc++ sobre configurações e opções que serão adicionadas ao código fonte gerado pela ferramenta. A segunda parte desse mesmo arquivo traz as regras que definem a gramática. O Código 5.3 descreve um fragmento do arquivo com a implementação das regras gramaticais do Grails.

Código 5.3 – Fragmento do código fonte para o Bisonc++

```
1  // Seção de diretivas
2  %scanner                          ../scanner/Scanner.h
3  %scanner-token-function           d_scanner.lex()
4  %baseclass-preinclude             ParserPreinclude.h
5
6  %stype std::string
7  %start startrule
8
9  %include spec/tokens.y
10
11 %%
12
13
14 // Definição de uma das regras gramaticais do Grails
15 class_definition:
16   CLASS IDENTIFIER {
17     std::string identifier_token( $2 );
18     collector->collect_data( "c", identifier_token.c_str() );
19   }
```

```
20 | ABSTRACT class_definition
21 | class_definition class_complements
22 ;
```

Como dito anteriormente, na primeira parte do arquivo, foram utilizadas algumas diretivas do Bisonc++ que forneceram opções úteis para o desenvolvimento do *parser* e, consequentemente, da identificação das regras gramaticais. Das diretivas apresentadas, entre as linhas 2 e 9, as que merecem menção são a da linha 2, que indica o *scanner* que foi utilizado; a 3, que indica de onde serão lidas as palavras identificadas pelo *scanner*, e a linha 9, que inclui outro arquivo contendo os *tokens* utilizados na construção da gramática.

Entre as linhas 15 e 21 do Código 5.3, pode-se observar uma das regras construídas neste trabalho. A linha 15 indica o nome da regra a ser construída, o que gera a adição de um novo símbolo não terminal, chamado `class_definition`. Essa nova regra possui três definições possíveis, listadas nas linhas 16, 20 e 21. É importante observar que as definições de regras no Bisonc++ são recursivas. Dessa forma, estabeleceu-se uma definição base para a `class_definition` que foi reutilizada nas outras duas possibilidades. As linhas 17 e 18 são as ações que foram implementandas para serem executadas quando essa regra for identificada. No caso, a linha 17 captura o *token* e o atribui a uma *string*, enquanto que a linha 18 faz a coleta da palavra encontrada na segunda posição da regra. A coleta de dados foi explicada na seção anterior, sobre o Scarefault.

Flexc++ e Bisonc++, quando executados, geram duas classes escritas em C++. É necessário fazer a comunicação entre o analisador léxico, produzido pelo Flexc++, com o analisador sintático gerado pelo Bisonc++. Para isso, foi necessária a criação de um ponteiro dentro do analisador léxico que apontasse para um atributo especial no analisador sintático. Assim, a comunicação entre ambos os módulos foi possível.

No Código 5.4, apresentam-se as alterações feitas no arquivo `Scanner.h`. Na linha 11, foi adicionado um atributo, do tipo ponteiro, com a finalidade de receber a localização de um atributo especial do *parser*. Para atribuir valores a esse atributo, foi definida uma função membro, linha 7, e sua implementação, linha 14.

Código 5.4 – Alterações no Scanner.h para comunicação entre analisadores léxico e sintático

```
1 #include "../parser/Parserbase.h"
2
3 class Scanner: public ScannerBase
4 {
5   public:
6   [...]
7   void setSval(Parser::STYPE__* d_val__);
8
```

```
9     private:
10    [...]
11    Parser::STYPE__* d_val;
12  };
13
14  inline void Scanner::setSval(Parser::STYPE__* d_val__)
15  {
16    d_val = d_val__;
17  }
```

No Código 5.5, são obsevadas as alterações feitas no arquivo `Parser.h` para que houvesse a comunicação com o `Scanner.h`. Na linha 7, declarou-se um objeto do tipo `Collector::CollectorBase *`, cuja função é explicada mais adiante. Na linha 8, é declarado o objeto responsável pelas mensagens de *log*, enquanto que na linha 11, declarou-se um novo construtor para o Parser.

Código 5.5 – Alterações no Parser.h para comunicação entre analisadores léxico e sintático

```
1  #include "../scanner/Scanner.h"
2
3  class Parser: public ParserBase
4  {
5    Scanner d_scanner;
6    Collector::CollectorFactory * factory;
7    Collector::CollectorBase * collector;
8    LogSystem::Log log;
9
10   public:
11     explicit Parser(Collector::Language,
12                           std::istream &in = std::cin,
13                           std::ostream &out = std::cout);
14   [...]
15 };
```

No Código 5.6, encontra-se a implementação do construtor. Permite-se, assim, ao objeto `d_scanner` conhecer o endereço de `d_val__`, garantindo a comunicação entre os analisadores léxico e sintático. Além disso, é nesse construtor que se instanciou o coletor de dados.

Código 5.6 – Alterações no Parser.ih para comunicação entre analisadores léxico e sintático

```
1  #include "Parser.h"
2
3  Parser::Parser(Collector::Language language,
```

```
4                std::istream &in, std::ostream &out)
5 {
6    d_scanner.switchStreams( in );
7      d_scanner.setSval(&d_val__);
8      factory = new Collector::CollectorFactory();
9    collector = factory->get_collector(language);
10 }
11
12 [...]
```

Tendo as regras gramaticais analisadas e construídas, fazendo-se uso do Flexc++ e do Bisonc++, foram feitos diversos testes manuais de identificação da linguagem base do *Grails*, como uso do *parser* e do *scanner* gerados. Esses testes deram-se por meio da execução do *parser* sobre um arquivo fonte escrito em Grails. Quando a execução finalizava, observava-se algum erro de sintaxe era encontrado. Para tornar esse tipo de teste de identificação das regras gramaticais melhores, foi implementado um *script* que faz a comparação entre o arquivo de *log* gerado e um arquivo contendo as mensagens esperadas após a execução do *parser* utilizando três códigos fonte.

Mesmo sendo capaz de identificar grande porção das regras gramaticais da linguagem base Grails, o *parser* ainda não é capaz de identificar todas as regras definidas para essa linguagem. Algumas especificidades, como alguns operadores, exemplo o elvis, e cadeias de expressões contidas em parênteses. Ao adicionar essas regras, alguns conflitos do tipo `reduce/reduce` eram reportados, o que inviabilizava o prosseguimento da implementação. Foram efetuadas diversas pesquisas sobre como resolver esse tipo de conflito numa gramática, mas não foi obtido sucesso. Optou-se por dar continuidade nos trabalhos, mesmo sem cobrir por completo a gramática.

A Figura 17 apresenta o resultado da execução do *script* de teste de identificação das regras gramaticais definidas para o Grails.

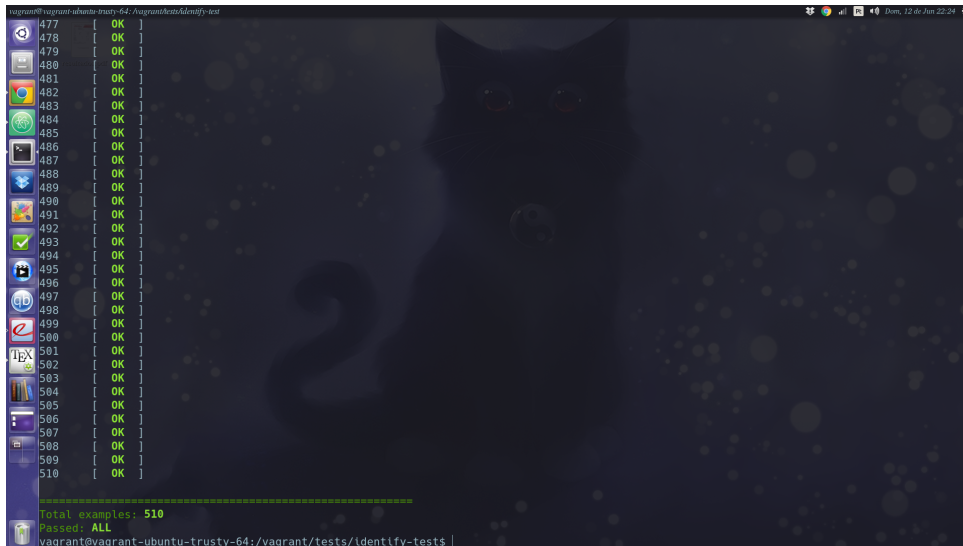

Figura 17 – Resultado da execução do *script* de teste de identificação

Como evidenciado na Figura 17 o *parser* passou em uma bateria de mais de 500 tentativas de identificação de regras em diversas combinações.

5.2 Geração de testes unitários que cubram o CRUD

O segundo objetivo específico deste trabalho é Gerar testes unitários por meio do framework que cubram os métodos criar, recuperar, atualizar e apagar entidades de negócio. Esta seção demonstra que o Scarefault atende esse objetivo.

Utilizou-se os resultados do primeiro objetivo específico como insumo para o segundo objetivo específico. Dessa forma, tendo uma gramática e um *parser* disponíveis para o Grails foram feitas as adaptações necessárias no *framework* para que fosse possível gerar testes nesse contexto. Assim, foi possível fazer a geração de testes unitários que cobrissem métodos relacionados ao CRUD. Os métodos de `controller` foram escolhidos para essa demonstração, pois evidenciam melhor os métodos do CRUD dentro do Grails.

Uma `controller` típica do Grails possui os seguintes métodos básicos de CRUD: `create()`, `save()`, `show()`, `edit()`, `update()` e `delete()`. Esses métodos representam o CRUD dentro da aplicação. Foi desenvolvida uma aplicação simples em Grails para a demonstração. É um pequeno projeto chamado de *guestbook*. A entidade alvo para esse teste do *framework* foi a `Feedback`. A `FeedbackController` possui os mesmos métodos citados anteriormente. Assim, foi executada a linha de comando: `./scarefault generate FeedbackController.groovy controller`. Com isso, foram gerados os testes unitários. O Código 5.7 mostra um fragmento importante do arquivo de teste gerado.

Código 5.7 – Fragmento do arquivo de teste gerado para a `FeedbackController`

```
1 package guestbook
```

```
2
3 import org.junit.*
4 import grails.test.mixin.*
5
6 @TestFor(FeedbackController)
7 @Mock(Feedback)
8 class FeedbackControllerTests {
9
10   def populateValidParams(params) {
11     assert params != null
12
13   params["title"] = "w2Z1AWVpZ8F3"
14   params["feedback"] = "
      aodijmweidjMN3TxXcXIdrAxxgp1ZsVR0rBm6K8JR3UsM0stcSFINRHL870"
15   }
16
17 void testIndex() {
18     controller.index()
19     assert "/feedback/list" == response.redirectedUrl
20   }
21
22   void testList() {
23     def model = controller.list()
24
25     assert model.feedbackInstanceList.size() == 0
26     assert model.feedbackInstanceTotal == 0
27   }
28
29   void testCreate() {
30     def model = controller.create()
31
32     assert model.feedbackInstance != null
33   }
34
35   void testSave() {
36     controller.save()
37
38     assert model.feedbackInstance != null
39     assert view == '/feedback/create'
40
41     response.reset()
42
```

```
43      populateValidParams(params)
44      controller.save()
45
46      assert response.redirectedUrl == '/feedback/show/1'
47      assert controller.flash.message != null
48      assert Feedback.count() == 1
49    }
50
51    void testShow() {
52      controller.show()
53      assert flash.message != null
54      assert response.redirectedUrl == '/feedback/list'
55
56      populateValidParams(params)
57      def feedback = new Feedback(params)
58
59      assert feedback.save() != null
60
61      params.id = feedback.id
62
63      def model = controller.show()
64
65      assert model.feedbackInstance == feedback
66    }
67
68    void testEdit() {
69      controller.edit()
70
71      assert flash.message != null
72      assert response.redirectedUrl == '/feedback/list'
73
74      populateValidParams(params)
75      def feedback = new Feedback(params)
76
77      assert feedback.save() != null
78
79      params.id = feedback.id
80
81      def model = controller.edit()
82
83      assert model.feedbackInstance == feedback
84    }
```

```
85
86   void testUpdate() {
87     controller.update()
88
89     assert flash.message != null
90     assert response.redirectedUrl == '/feedback/list'
91
92     response.reset()
93
94     populateValidParams(params)
95     def feedback = new Feedback(params)
96
97     assert feedback.save() != null
98
99   }
100
101  void testDelete() {
102    controller.delete()
103    assert flash.message != null
104    assert response.redirectedUrl == '/feedback/list'
105
106    response.reset()
107
108    populateValidParams(params)
109    def feedback = new Feedback(params)
110
111    assert feedback.save() != null
112    assert Feedback.count() == 1
113
114    params.id = feedback.id
115
116    controller.delete()
117
118    assert feedback.count() == 0
119    assert feedback.get(feedback.id) == null
120    assert response.redirectedUrl == '/feedback/list'
121  }
122 }
```

Nas linhas 13 e 14, os parâmetros são populados. Esses parâmetros serão recebidos pelo objeto do tipo `Feedback`, durante o processo de testes. Observa-se que os argumentos passados para os parâmetros são inteiramente **randômicos**. Esses valores são produzidos

pela classe auxiliar do *framework* chamada de `ValueGenerator`. Ela faz a análise das restrições do objeto, buscando essa restrições na classe de domínio `Feedback`, gera um valor aleatório respeitando essas restrições e insere como argumento para os parâmetros. O Código 5.8 apresenta a entidade `Feedback` e as suas restrições *constraints*.

Código 5.8 – Arquivo de `Feedback`

```
1  package guestbook
2
3  class Feedback {
4    String title
5    String feedback
6    Date dateCreated
7    Date lastUpdated
8
9    User user
10   static hasMany=[ comments:Comment ]
11
12   static constraints = {
13     title( blank: false, nullable: false, size: 3..80 )
14     feedback( blank: false, nullable: false, size: 3..500 )
15     user( nullable: false )
16   }
17 }
```

É importante ressaltar que nesse trabalho não foi possível a geração de valores para todos os tipos de restrições. Mas, para uma visão preliminar, as contribuições foram focadas nas restrições que limitavam o argumento por meio de intervalos de valores. Para versão preliminar, com base nas intenções desse trabalho de conclusão de curso, tem-se que os valores gerados são para testes de valores válidos. Tratar valores inválidos e demais restrições serão considerados como trabalhos futuros.

Com o arquivo de teste gerado foram executados os testes, be, como verificado o resultado de sua execução. Isso é apresentado na Figura 18.

Figura 18 – Resultado da execução dos teses gerados para a `FeedbackController`

5.3 Inclusão de Outras Linguagens para Geração de Testes Unitários e Extensibilidade do *Framework*

O terceiro objetivo específico deste trabalho é Permitir inclusão de outras linguagens para geração de testes unitários. Esta seção demonstra que o Scarefault atende esse objetivo.

Como *framework*, o Scarefault deve permitir ao desenvolvedor que o utiliza fazer adaptações por meio de extensões de seu código. O Scarefault provê essas janelas de extensibilidade, como já demonstrado no decorrer do Capítulo 4. Esse capítulo evidencia as partes extensíveis e os pontos de adaptação do *framework* . Para a inclusão de novas linguagens de programação para a geração de testes unitários, foi demonstrado também, no decorrer do Capítulo 4, utilizando como exemplo o Grails para inclusão de uma nova linguagem.

Uma recapitulação breve para resumir como o Scarefault atende a esse objetivo:

- Ao definir a gramática e um *parser* para uma linguagem de programação, utilizando-se do Flexc++ e do Bisonc++, pode-se utilizar os coletores de dados e os contrutores de arquivos de teste do Scarefault, com as devidas adaptações, para que haja a inclusão da nova linguagem.

- O *framework* possui diferentes pontos de extensão, como a classe `CollectorBase` e a `TestfileBuilder`. A partir dessas classes, pode-se derivar outras com o objetivo

de gerar testes para arquivos de código fonte de uma determinada linguagem.

- O uso de *design pattern*, como o *Builder Pattern* e o *Factory Pattern*, apontam a capacidade de extensão do *framework* . Por meio deles, é possível criar *hotspots* e *frozenspots*. Pelos *hotspots*, o desenvolvedor pode estender ou adaptar o Scarefault às suas necessidades, desde que respeite as necessidades do *framework* . Isso é bem evidenciado no uso do *Builder Pattern*. Ele abre espaço para que o desenvolvedor adapte a construção de `testfiles` à linguagem incluída, mas mantem rígido o processo de construção.

Tendo isso em vista, pode-se concluir que o Scarefault, em seu estado atual, atende a esse objetivo específico.

5.4 Resumo do Capítulo

O capítulo apresentou os resultados alcançados com a finalização do trabalho. Esses resultados estão vinculados aos objetivos específicos, o que permite melhor avaliar e validar esses objetivos. A identificação da linguagem alvo, geração de testes para CRUD e permitir a extensibilidade do *framework* e adição de novas linguagens de programação foram os objetivos específicos validados através deste capítulo.

6 Considerações Finais

O trabalho aponta a necessidade e as exigências do mercado de software sobre a qualidade dos produtos e serviços na área. Demonstra os problemas e alguns dos motivos da falta da produção de testes unitários, uma técnica fundamental para evitar *bugs* e dificuldades no desenvolvimento de software.

O *framework* desenvolvido nesse trabalho foi proposto com o objetivo de facilitar a geração de testes unitários para diversas linguagens de programação. A questão que buscou-se responder foi: *há como propiciar ao desenvolvedor um suporte que forneça a geração de testes unitários, de forma semiautomatizada, e que permita adaptações, com o intuito de ajustar-se ao código-fonte?* Para resolver esse questionamento, foi planeado um sistema extensível capaz de gerar testes unitários com uma intervenção breve do desenvolvedor. No decorrer do projeto fez-se uso do planejamento inicial para cumprir com as tarefas propostas, como consta o Capítulo 3. No decorrer do projeto algumas dificuldades surgiram, em especial a respeito da identificação da linguagem alvo por meio do Flexc++ e do Bisonc++. Isso teve impactos no planejamento inicial, mas que foram contornados. O processo de desenvolvimento foi alinhado com análises breves do que deveria ser alterado na produção do *framework* para alcançar um resultado satisfatório. Essa análise se dava nas conversas entre os dois autores, que identificaram pontos de melhoria no decorrer do trabalho. A finalização do esforços no desenvolvimento do *framework* culminaram nos resultados expostos no Capítulo 5. Por meio da análise dos resultados atingidos para cada objetivo específico, é possível concluir que o projeto, de modo geral, obteve sucesso.

O desenvolvimento desse projeto permitiu aplicar diversos conceitos e conhecimentos adquiridos durante o curso de Engenharia de Software. Entre eles, destacam-se a Arquitetura de Software, Requisitos de Software, Gerência e Configuração de Software e Testes de Software. No que diz respeito aos conceitos relativos à Arquitetura de Software, estão presentes o uso de padrões de projeto (*design patterns*), e conceitos de orientação a objetos, como herança e polimorfismo. O entendimento desses conceitos permitiu o avanço e o desenvolvimento da arquitetura do Scarefault, a ponto de produzi-lo para que tivesse extensibilidade suficiente para que pudesse ser classificado como *framework*.

Os conceitos aprendidos sobre requisitos de software tiveram papel importante no início do projeto, na medida em que por meio deles foram estabelecidas metas para os ciclos de desenvolvimento. A Gerência e Configuração de Software permitiu a automatização de diversos elementos que exigiam tempo durante a produção, como a compilação do código e o levantamento do ambiente de desenvolvimento.

Os autores observam que ainda há melhorias a serem feitas no Scarefault. Algumas

sugestões para trabalhos futuros são:

Desenvolver um interpretador próprio: a seleção do Flexc++ e do Bisonc++ para auxílio desse trabalho se tornou correta. Isso porque o foco do trabalho foi o desenvolvimento do *framework* voltado para geração de testes. Os autores decidiram por usar ferramentas que já forneceriam meios de interpretar a linguagem alvo para, assim, não terem que produzir esse interpretador. No entanto, o uso dessas dependências prova-se inviável no futuro, pois engessa o *framework* no que diz respeito a interpretação de linguagens. Exige um esforço significativo, o qual deve ser evitada para que o Scarefault possa evoluir mais. Assim, é visto com bons olhos o desenvolvimento de um interpretador próprio que seja flexível e extensível, como parte do *framework*.

Desenvolver *hotspots* para geração de testes inválidos: atualmente, o Scarefault cobre testes com valores válidos. É interessante que haja também a possibilidade de gerar testes para valores inválidos.

Produzir outras estratégias para geração de testes: o Scarefault gera testes de caixa preta. Para isso, utiliza-se da estratégia de valores limites. A adição da possiblidade do uso de outras técnicas de derivação de testes caixa preta seria bem vinda. Além disso, seria bom que houvesse a possibilidade da geração de testes de caixa branca.

Referências

AABY, A. A. *Compiler Construction using Flex and Bison*. [s.n.], 2004. Disponível em: <http://foja.dcs.fmph.uniba.sk/kompilatory/docs/compiler.pdf>. Citado na página 99.

BARBOSA, E. F. et al. *Introdução ao Teste de Software*. [S.l.], 2009. 49 p. Disponível em: <http://www.duguay.com.br/uploads/arquivos/apostilaUSP_Teste_de_Software.pdf>. Citado 6 vezes nas páginas 25, 27, 29, 30, 31 e 40.

BARRETO JUNIOR, C. G. *Agregando Frameworks de Infra-Estrutura em uma Arquitetura Baseada em Componentes: Um Estudo de Caso no Ambiente AulaNet*. Tese (Dissertação de Mestrado) — Pontifícia Univeridade Católica do Rio de Janeiro, Rio de Janeiro, mar. 2006. Disponível em: <http://www2.dbd.puc-rio.br/pergamum/tesesabertas/0410823_06_cap_02.pdf>. Citado 5 vezes nas páginas 33, 34, 37, 38 e 39.

BIZAGI. *Bizagi Modeler*. 2015. Disponível em: <http://www.bizagi.com/en/bpm-suite/bpm-products/modeler>. Citado na página 100.

BRACHET, P. *TexMaker The universal LaTeX editor*. 2014. Disponível em: <http://www.xm1math.net/texmaker/>. Citado na página 102.

BROKKEN, F. B. *Bisonc++ V 4.12.03*. 2015. Disponível em: <https://fbb-git.github.io/bisoncpp/>. Citado na página 99.

BROKKEN, F. B. *Flexc++ V 2.03.03*. 2015. Disponível em: <http://fbb-git.github.io/flexcpp/>. Citado na página 99.

BUENO, C. d. F. d. S.; CAMPELO, G. B. *Qualidade de Software*. Recife, PE, 2013. 28 p. Disponível em: <http://sistemas.riopomba.ifsudestemg.edu.br/dcc/materiais/1022789570_Qualidade%20de%20Software.pdf>. Citado na página 29.

BURKE, E. M.; COYNER, B. M. *Top 12 Reasons to Write Unit Tests*. 2003. Disponível em: <http://www.onjava.com/pub/a/onjava/2003/04/02/javaxpckbk.html>. Citado na página 26.

CANONICAL. *Comunidade Ubuntu Brasil*. 2010. Disponível em: <http://ubuntu-br.org/>. Citado na página 101.

CHACON, S.; STRAUB, B. *Pro Git*. 2ª. ed. Apress, 2015. ISBN 978-1-4842-0077-3. Disponível em: <https://progit2.s3.amazonaws.com/en/2015-10-10-9ced5/progit-en.870.pdf>. Citado na página 101.

COPELAND, L. *A Practitioner's Guide to Software Test Design. Norwood, MA, USA: Artech House*. [S.l.]: Inc, 2003. Citado na página 32.

FANTINATO, M. et al. AutoTest: Um Framework Reutilizável para a Automação de Teste Funcional de Software. In: . Campinas, São Paulo: [s.n.], 2004. p. 15. Disponível em: <http://www.lbd.dcc.ufmg.br/colecoes/sbqs/2004/025.pdf>. Citado 2 vezes nas páginas 40 e 41.

GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994. Disponível em: <http://www.uml.org.cn/c++/pdf/DesignPatterns.pdf>. Citado 4 vezes nas páginas 36, 37, 56 e 57.

GITLAB INC. *GitLab*. 2015. Disponível em: <https://about.gitlab.com/>. Citado na página 101.

GRAILS. *Grails*. 2015. Disponível em: <https://grails.org/>. Citado na página 79.

HASHICORP. *Vagrant*. 2015. Disponível em: <https://www.vagrantup.com/>. Citado na página 102.

JANONES, R. d. S. *Qualidade de Software: Uma questão de eficiência*. 2010. Disponível em: <http://www.devmedia.com.br/qualidade-de-software-uma-questao-de-eficiencia/17803>. Citado na página 25.

JANTTI, M. *Difficulties in Managing Software Problems and Defects*. Tese (Dissertaç ão de Doutorado) — University of Kuopio, Kuopio, Finlândia, jan. 2008. Disponível em: <http://epublications.uef.fi/pub/urn_isbn_978-951-27-0109-4/urn_isbn_978-951-27-0109-4.pdf>. Citado na página 25.

KENT, J. Test automation: From record/playback to frameworks. *Proceedings of EuroSTAR*, 2007. Citado 2 vezes nas páginas 40 e 41.

KLEIN, J.; WEISS, D. What is Architecture? In: *Beautiful Architecture*. [S.l.]: O'Reilly Media, 2009. Citado na página 35.

LATEX. *LaTeX – A document preparation system*. 2015. Disponível em: <https://latex-project.org/>. Citado na página 102.

MARTIN, R. C. Design Principles and Design Patterns. *Object Mentor*, p. 34, 2000. Disponível em: <http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf>. Citado 2 vezes nas páginas 35 e 36.

MOOLENAAR, B. *Vim The Editor*. 2015. Disponível em: <http://www.vim.org/>. Citado na página 101.

MYERS, G. J.; BADGETT, T.; SANDLER, C. *The Art of Software Testing*. 3. ed. [S.l.: s.n.]. Citado 2 vezes nas páginas 31 e 32.

PHILIPSON, G. *A Short History of Software*. 2004. Disponível em: <http://www.thecorememory.com/SHOS.pdf>. Citado na página 25.

ROY ROSENZWEIG CENTER FOR HISTORY AND NEW MEDIA. *Zotero*. 2015. Disponível em: <https://www.zotero.org/>. Citado na página 102.

SAUVÉ, J. P. *O que é um framework?* 2006. Disponível em: <http://www.dsc.ufcg.edu.br/~jacques/cursos/map/html/frame/oque.htm>. Citado 4 vezes nas páginas 34, 37, 38 e 39.

SCRUM GUIDES. *The Scrum Guides*. 2014. Disponível em: <http://www.scrumguides.org/scrum-guide.html>. Citado 2 vezes nas páginas 43 e 44.

SIMSEK, B. *Libraries*. 2004. Disponível em: <http://www.enderunix.org/simsek/articles/libraries.pdf>. Citado 2 vezes nas páginas 34 e 35.

SOFTWARE TESTING FUNDAMENTALS. *Black Box Testing*. 2010. Disponível em: <http://softwaretestingfundamentals.com/black-box-testing/>. Citado na página 30.

SOMMERVILLE, I. *Engenharia de Software*. 8ª. ed. São Paulo: Pearson Addison-Wesley, 2007. Citado 2 vezes nas páginas 25 e 29.

SUTCLIFFE, A. *The Domain Theory: Patterns for Knowledge and Software Reuse*. Londres: Lawrence Erlbaum Associates, 2002. Citado na página 32.

TRELLO. *Trello*. 2015. Disponível em: <https://trello.com/>. Citado na página 100.

TRODO, L. D. *Uso de Métricas nos Testes de Software*. Tese (TCC) — Universidade Federal do Rio Grande do Sul, Porto Alegre, nov. 2009. Disponível em: <http://www.lume.ufrgs.br/bitstream/handle/10183/18574/000730980.pdf?...1>. Citado 2 vezes nas páginas 25 e 29.

WELLS, D. *The Rules of Extreme Programming*. 2009. Disponível em: <http://www.extremeprogramming.org/rules.html>. Citado 2 vezes nas páginas 43 e 44.

WILLIAMS, L. *Testing Overview and Black-Box Testing Techniques*. 2006. Disponível em: <http://agile.csc.ncsu.edu/SEMaterials/BlackBox.pdf>. Citado 2 vezes nas páginas 30 e 31.

ZenHub. Disponível em: <https://www.zenhub.com/support>. Citado na página 100.

APÊNDICE A – Suporte Tecnológico

É apresentado a seguir um compilado das principais ferramentas que foram utilizadas no desenvolvimento do *framework* para geração de testes unitários. Com a finalidade de melhorar a organização, os recursos tecnológicos foram divididos em categorias: Geração de Teste Unitário e Engenharia de Software. A primeira refere-se aos suportes que foram utilizados na construção do *framework*; enquanto a segunda categoria apresenta suportes que auxiliaram o gerenciamento e desenvolvimento do software e documentação relacionada.

A.1 Geração de Teste Unitário

Os recursos tecnológicos identificados como capazes de auxiliar no desenvolvimento do *framework* estão apresentados a seguir:

A.1.1 Flexc++

Flexc++ é um analisador léxico escrito por Frank B. Brokken, gerente de segurança em TI e conferencista na *University of Groningen*, Holanda(BROKKEN, 2015b). O desenvolvimento desse analisador começou em 2008 e é mantido até hoje. Atualmente, está na versão 2.03.03, sendo mantido por: Frank B. Brokken, Jean-Paul van Oosten e Richard Berendsen (BROKKEN, 2015b). No entanto, a versão utilizada foi a 1.05.

A.1.2 Bisonc++

Bisonc++ é um gerador de parsers capaz de converter uma gramática de contexto livre em uma classe *parser* em C++ (BROKKEN, 2015a). Bisonc++ foi escrito e é mantido por Frank B. Brokken, da *University of Groningen*, Holanda. Bisonc++, diferentemente do Bison, fornece uma classe em C++ após ser compilado para que possa ser usada. Bison fornece um código em C (BROKKEN, 2015a).

Ambas as ferramentas mencionadas, Flexc++ e Bisonc++, são muito utilizadas no contexto de compiladores (AABY, 2004) e foram úteis na dinâmica desse projeto, pois conferiram suporte ao reconhecimento léxico e à interpretação de linguagens. Para a geração de testes, é necessário o reconhecimento de palavras-chave capazes de revelar informações sobre os métodos a serem testados. Também forneceram suporte na interpretação dos arquivos de código fonte, bem como no desenvolvimento de um *parser*, cujos produtos finais foram os testes gerados.

A.2 Engenharia de Software

As ferramentas a seguir foram utilizadas para auxiliar no gerenciamento e na implementação, bem como na documentação associada. Buscou-se aplicar os conhecimentos adquiridos no decorrer do curso de engenharia de software para apoiar o processo, tanto em tecnologia, quanto em práticas de desenvolvimento de software.

A.2.1 Gerência de Projetos

No que tange à área de gerência de projetos, as seguintes ferramentas serão utilizadas:

A.2.1.1 Trello

Serviço *online* que permite o gerenciamento de projetos e tarefas por meio de quadros e listas. Trello possui uma interface simples que possibilita anexar arquivos, compartilhamento de conteúdo por equipe, adicionar responsável em tarefas, entre outras funcionalidades. Embora haja recursos pagos, não há necessidade de pagamento para utilizar essa ferramenta (TRELLO, 2015).

O Trello centralizou as informações gerenciais do projeto como as tarefas a serem executadas, descrição e prazos.

A.2.1.2 Bizagi Modeler

Ferramenta gratuita para modelagem de processos que respeita a especificação BPMN (*Business Process Modeling Notation*). Bizagi fornece suporte de ajuda para usuário com guias e tutoriais, além de permitir a edição compartilhada de processos entre suas funcionalidades (BIZAGI, 2015). Foi utilizada a versão 2.9.

O Bizagi Modeler permitiu a modelagem de um processo que guiou o trabalho de conclusão de curso. O modelo desse processo possibilitou a visualização e o entendimento do processo seguido.

A.2.2 ZenHub

Extensão de navegador capaz de adicionar recursos de gerenciamento na interface do GitHub (ZenHub,). Com isso, foi possível centralizar o gerenciamento do projeto no repositório do código.

A.2.3 Desenvolvimento de Software

Em relação ao desenvolvimento de software, os seguintes suportes foram selecionados:

A.2.3.1 Vim

Vim é um editor de texto configurável, construído para permitir a edição de texto de forma eficiente. Essa ferramenta foi criada com base no editor Vi, distribuído com a maioria dos sistemas UNIX (MOOLENAAR, 2015). Licenciado pela *charityware*, Vim é distribuído gratuitamente.

Esse editor de texto foi utilizado para a edição dos arquivos de código fonte.

A.2.3.2 Ubuntu

Distribuição linux, livre e de código aberto. Patrocinado pela Canonical Ltda., Ubuntu utiliza kernel Linux baseado em Debian e, sendo licenciado pela *General License Public* (GPL) (CANONICAL, 2010). A versão utilizada foi a 14.04.

O Ubuntu foi utilizado como sistema operacional padrão para o desenvolvimento do *framework*.

A.2.4 Gerência de Configuração de Software e Requisitos de Software

Para dar suporte ao desenvolvimento, automatizando o máximo possível determinadas tarefas, as seguintes ferramentas foram escolhidas para a gerência de configuração de software.

A.2.4.1 Git

Git é um sistema de versão distribuído de código aberto e gratuito. Criado em 2005, Git foi desenvolvido visando manter velocidade e eficiência para pequenos e grandes projetos (CHACON; STRAUB, 2015).

O Git permitiu o versionamento do código fonte, bem como da monografia, proporcionando controle das versões e possíveis resgates de informações.

A.2.4.2 GitHub

GitHub é um serviço de hospedagem de projetos de software que utiliza Git como controle de versão. GitHub fornece, entre suas funcionalidades, o gerenciamento de *issues*, uso de *wiki*, controle de permissão de *branchs*, revisão de código e utilização de repositórios abertos (GITLAB INC., 2015).

O GitHub tornou-se um local central, onde diversas informações sobre o projeto serão encontradas, inclusive o próprio código. É nesse centro de informações do projeto que estão relacionados os requisitos de software, e sua rastreabilidade com as histórias de usuário.

A.2.4.3 Vagrant

Vagrant é uma ferramenta capaz de gerenciar a criação de máquinas virtuais para ambientes de desenvolvimento. Sua utilização proporciona a automação da instalação e configuração de ferramentas, além de facilitar a padronização dos ambientes de desenvolvimento utilizados por uma equipe de software (HASHICORP, 2015). Inicialmente criado por Mitchell Hashimoto em janeiro de 2010, Vagrant é um software aberto, mantido pela empresa HashiCorp (HASHICORP, 2015).

A utilização do Vagrant permitiu a padronização do ambiente de desenvolvimento utilizado na implementação do *framework* proposto. Também simplificou a replicação desse ambiente, caso seja necessário.

A.2.5 Pesquisa

Para dar suporte ao andamento da pesquisa e produção da monografia, as seguintes ferramentas foram usadas:

A.2.5.1 LaTeX

Criado inicialmente em 1985 por Leslie Lamport, o LaTeX é um sistema de preparação de documentos para composição tipográfica que tem como principal objetivo reduzir o esforço dos autores na formatação de textos técnicos ou científicos. Baseado na linguagem TeX, criada por Donald E. Knuth, LaTeX está disponível como software livre e é mantido pela *LaTeX3 Project* (LATEX, 2015).

A.2.5.2 TexMaker

Editor de texto multi-plataforma para LaTex. Software livre licenciado pela General Public License (GPL), o TexMaker inclui suporte *unicode*, verificação ortográfica, função de auto-completar, visualizador de arquivo em PDF, além do editor para escrita de arquivos LaTex (BRACHET, 2014). Foi utilizada a versão 4.1.

A.2.5.3 Zotero

Zotero é um software livre (GPLv3), criado e mantido pela *George Mason University*, que tem por finalidade a gerência de bibliografias e materiais relacionados à pesquisa. Entre suas principais funcionalidades estão a geração de relatório de referências e a exportação de bibliografias já formatadas (ROY ROSENZWEIG CENTER FOR HISTORY AND NEW MEDIA, 2015). A versão utilizada foi a 4.0.28.

APÊNDICE B – Resumo do Estudo das Ferramentas Flexc++ e Bisonc++

B.1 Flexc++

Flexc++ é um analisador léxico escrito por Frank B. Brokken, gerente de segurança em TI e conferencista na *University of Groningen*, Holanda. O desenvolvimento desse analisador começou em 2008 e é mantido até hoje. Atualmente está na versão 2.03.03, é mantido por: Frank B. Brokken, Jean-Paul van Oosten e Richard Berendsen. No entanto, por hora, será utilizado a versão 1.05.

B.1.1 Comparativo entre Flex e Flexc++

Basicamente, a diferença é que Flexc++ tem por intuito gerar código em C++ para que possa ser usado juntamente com programas escritos em C++. Enquanto isso, Flex gera código escrito em C e Flex++ dificilmente dar suporte a todo o pontencial que C++ oferece hoje.

B.1.2 Como Executar o Flexc++

Sua execução se dá pelo comando:

Código B.1 – Comando para execução do flexc++

```
1 flexc++ [opções] rules-file
```

Onde:

rules-file: é o arquivo que contém as regras.

opções: são opções que o Flexc++ suporta. O intuito delas é customizar algum elemento do *Scanner* ou alguns detalhes, mas que provavelmente não serão usadas aqui.

B.1.3 A Estrutura do Arquivo de Entrada

A seguir, algumas informações sobre a estrutura do Arquivo de entrada para o Flexc++.

B.1.3.1 Nome do Arquivo

O arquivo pode ser chamado de qualquer forma, no entanto, sua extensão deve ser do tipo .l.

B.1.3.2 Corpo do Arquivo

O arquivo deve possuir duas seções: **seção de definições** e **seção de regras**. Essas seções devem estar estruturadas da seguinte forma:

Código B.2 – Corpo do arquivo do flexc++

```
1 seção de definições
2 %%
3 seção de regras
```

Onde a linha 2 define o fim e o início entre as seções.

B.1.3.2.1 Seção de Definições

A seção de definições é onde deve-se definir, por meio de expressões regulares, o que será avaliado e como será identificado pelo analisador léxico. A seção deve seguir o padrão sendo o nome do identificador seguido da *regex*. Segue o exemplo de como ficaria:

Código B.3 – Seção de definições

```
1
2 DIGIT       [0-9]
3 INTEGER     {DIGIT}
4 REAL        -?{INTEGER}("."{INTEGER})?
5
6 CHARACTER   [a-zA-Z]
7 LITERAL     \"[a-zA-Z0-9][a-zA-Z0-9].*\"
8 %%
```

B.1.3.2.2 Seção de Regras

A seção de regras é onde deve-se explicitar o que deve ser feito pelo analisador após a identificação dos padrões no texto. A seção deve seguir o seguinte padrão: *padrão ação*. Ficaria conforme exemplo abaixo:

Código B.4 – Seção de regras

```
1 %%
2 LITERAL     { cout << "Hello World!"; }
```

B.1.4 Compilando com o Flexc++

Para executar a compilação utilizando o Flexc++, supondo possuir um arquivo de entrada flexcpp.l, deve-se:

1. Compilar o arquivo de entrada com o Flexc++

Código B.5 – Comando para compilar flexc++

```
1 flexc++ flexcpp.l
```

2. Com isso, serão gerados alguns arquivos. Deve-se alterar a extensão do arquivo lex.ccpara lex.cpp

Código B.6 – Comando para alterar extensão dos arquivos

```
1 mv lex.cc lex.cpp
```

3. Agora, supondo que o arquivo que utilizará o analisador léxico chama-se analizer.cpp. Assim, para compilar o programa deve-se:

Código B.7 – Compilar o programa

```
1 g++ -o [nome_arquivo_saida] analizer.cpp lex.cpp
```

Pronto! Agora você possui um programa analisador léxico chamado analizer. Para executar use:

Código B.8 – Executando o programa

```
1 ./analizer
```

B.2 Bisonc++

Bisonc++ é um gerador de parsers capaz de converter uma gramática de context livre em uma classe paser em C++. Bisonc++ foi escrito e é mantido por Frank B. Brokken, da *University of Groningen*, Holanda. Bisonc++, diferentemente do Bison, fornece uma classe em C++ após ser compilado para que possa ser usada. Bison fornece um código em C.

B.2.1 Pequeno Glossário

Símbolo Terminal: elemento que por si só já define algo. Já possui significado em si mesmo na gramática.

Símbolo Não-terminal: elemento que é formado para ter significado completo depende dos símbolos terminais.

B.2.2 Como Executar o Bisonc++

Sua execução se dá pelo comando:

Código B.9 – Comando para executar bisonc++

```
1 bisonc++ [opções] file
```

Onde:

file: é o arquivo que contém as regras e ações.

opções: são opções que o Bisonc++ suporta. O intuito delas é customizar algum elemento do *Parser* ou alguns detalhes, mas que provavelmente não serão usadas aqui.

B.2.3 A Estrutura do Arquivo de Entrada

A seguir, algumas informações sobre a estrutura do Arquivo de entrada para o Bisonc++.

B.2.3.1 Corpo do Arquivo

É formado por duas seções: **seção de diretivas** e **seção de regras gramaticais**. Essas seções devem estar estruturadas da seguinte forma:

Código B.10 – Corpo do arquivo do bisonc++

```
1 seção de diretivas
2 %%
3 seção de regras gramaticais
```

Onde a linha 2 define o fim e o início entre as seções.

B.2.3.1.1 Seção de Diretivas

Na seção de diretivas, são especificadas algumas opções relativas ao *parser*. Além disso, é nessa seção que se deve definir o nome de todos os *tokens*, exceto aqueles com o nome formado por um único caractere. A inclusão de bibliotecas externas também é feita nessa seção.

Algumas diretivas importantes são:

start: designa algum símbolo não-terminal para se a regra inicial.

token: é utilizado para definir símbolos terminais. Sua sintaxe segue os termos abaixo:

Código B.11 – Sintaxe

```
1 token [<type>] terminal token
```

A opção *<type>* é usada quando se deseja especificar o tipo que será declarado aquele *token*. Alguns nomes não devem ser usados como *token*: ABORT, ACCEPT, ERROR, clearin, debug, error, setDebug.

namespace: define um *namespace* para a classe *Parser*.

include: é usada para alterar o *pathname* enquanto a gramática é processada.

B.2.3.1.2 Seção de Regras Gramaticas

De forma geral, uma regra gramatical é definida da seguinte forma:

Código B.12 – Definição de regra gramatical

```
1 result:
2   components
3 ;
```

onde *result* é o símbolo não-terminal que a regra descreve. Por sua vez, *components* são vários símbolos terminais e não-terminais que são postos em conjunto, formando a regra. Um exemplo:

Código B.13 – Exemplo de regra gramatical

```
1 expression:
2   expression '+' expression
3 ;
```

Pode-se também determinar ações para cada símbolo, terminal ou não-terminal, por meio de código em C++:

Código B.14 – Exemplo de regra gramatical utilizando C++

```
1 expression:
2   expression '+' expression {
3     C++ statements
4   }
5 ;
```

Pode-se criar múltiplas regras para o mesmo*result*:

Código B.15 – Exemplo de regra gramatical utilizando múltiplas regras

```
1 result:
2   rule1-components {
3     C++ statements
4   }
5 |
```

```
6   rule2-components {
7     C++ statements
8   }
9 ;
```

Há também a opção de deixar um componente vazio. Nesse caso, *result* pode corresponder a uma string vazia.

Regras Recursivas: é quando o *result* não-terminal aparece também do lado direto de um componente desse mesmo *result*. Praticamente, todas as regras no Bisonc++ precisam ser recursivas. Há também como utilizar a recursão pelo lado esquerdo. Inclusive é a forma correta. Outra forma possível é fazer a recursividade indireta. Essa pode ocorrer da seguinte forma:

Código B.16 – Exemplo de regra gramatical utilizando recursividade

```
1 expr:
2   primary '+' primary
3 |
4   primary
5 ;
6 
7 primary:
8   constant
9 |
10   '(' expr ')'
11 ;
```

B.2.4 Compilando o Bisonc++

Para executar a compilação utilizando o Bisonc++, supondo possuir um arquivo de entrada bisoncpp.y, deve-se compilar o arquivo de entrada com o Bisonc++. Para isso, utiliza-se o seguinte comando:

Código B.17 – Comando para compilação do bisonc++

```
1 bisonc++ bisoncpp.y
```