
Compreensão de Mudanças Estruturais no Código Fonte usando Análise Dinâmica e Estática

Janio Rosa da Silva



UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Uberlândia
2015

Janio Rosa da Silva

**Compreensão de Mudanças Estruturais no
Código Fonte usando Análise Dinâmica e
Estática**

Dissertação de mestrado apresentada ao Programa de Pós-graduação da Faculdade de Computação da Universidade Federal de Uberlândia como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Ciência da Computação

Orientador: Marcelo de Almeida Maia

Uberlândia
2015

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Os abaixo assinados, por meio deste, certificam que leram e recomendam para a Faculdade de Computação a aceitação da dissertação intitulada **“Compreensão de Mudanças Estruturais no Código Fonte usando Análise Dinâmica e Estática”** por **Jânio Rosa da Silva** como parte dos requisitos exigidos para a obtenção do título de **Mestre em Ciência da Computação**.

Uberlândia, 17 de Dezembro de 2015

Orientador: _____

Prof. Dr. Marcelo de Almeida Maia
Universidade Federal de Uberlândia

Banca Examinadora:

Prof. Dr. Autran Macêdo
Universidade Federal de Uberlândia

Prof. Dr. Ricardo Terra Nunes Bueno Villela
Universidade Federal de Lavras

*Este trabalho é dedicado à minha noiva e a meus
pais que sempre me compreenderam nos momentos em que estive ausente em suas vidas
para me dedicar aos estudos.*

Agradecimentos

Agradeço ao meu orientador, Marcelo Maia, sua capacidade de motivar e de contribuir para o desenvolvimento deste trabalho e seu profissionalismo, que sempre demonstrou para o bom andamento desta pesquisa. Eu quero agradecer especialmente à minha noiva, Luciana, seu apoio nos momentos de pressão e suas dicas.

*“Só podemos ver um pouco do futuro, mas o suficiente para perceber que há muito a
fazer.”*
(Alan Turing)

Resumo

A compreensão de sistemas é fundamental para a atividade de manutenção. Durante a manutenção e evolução dos sistemas, mudanças contínuas podem degradar o projeto modular do sistema, aumentando sua complexidade. Consequentemente, as empresas gastam muito tempo e recursos financeiros na compreensão e manutenção dos sistemas. Portanto, entender como o sistema evolui é uma etapa importante para o bom planejamento, desenvolvimento e gerenciamento de mudanças. Os desenvolvedores geralmente precisam entender rapidamente mudanças recentes antes de implementar novas mudanças. Mesmo que já existam abordagens para a compreensão, elas ainda são limitadas para detectar componentes diferentes com tarefas similares, para localizar tarefas no código fonte e para medir o impacto de uma determinada mudança nas funcionalidades do sistema. Neste trabalho, é proposto um mecanismo para localizar impactos causados por uma mudança no projeto e quais mudanças estruturais ocorreram em um sistema de uma versão para outra. Dada uma funcionalidade específica, o objetivo é localizar mudanças estruturais e mudanças de relacionamento entre componentes entre duas versões. Cada mudança estrutural detectada na primeira etapa é checada no código fonte para ambas as versões. Depois, as mudanças são classificadas em cinco padrões: i) movimentação de classe; ii) movimentação de método; iii) adição de método; iv) remoção de método; e v) mudança no modificador de acesso (os três últimos que representam mudanças de interface nas classes). A abordagem proposta é avaliada com três sistemas de código aberto com o objetivo de validar a metodologia: *jFreeChart*, *Tomcat* e *JHotDraw*. Os resultados revelam mudanças estruturais como movimentação de método, movimentação de classe e mudança de relacionamento de pacotes. Além disso, também é feito um levantamento de mudanças estruturais que afetam múltiplas funcionalidades, o que é chamado de avaliação de impacto de mudanças; análises sobre mudanças de relacionamento de pacotes no *jFreeChart* validado em termos de precisão e recall. Os resultados mostram que movimentações em métodos, em média, aparecem em 28,4% dos casos. Existem classes que afetam muitas funcionalidades, logo o desenvolvedor terá noção de quais funcionalidades serão afetadas com tais mudanças. As mudanças no *jFreeChart* mostram que a abordagem detecta os

padrões de alterações em pacotes a uma precisão de 100% e revocação de 83%. Ao detectar uma mudança considerável na relação de pacotes entre o *jFreeChart* versões 0.7.0 e 0.9.5, foi constatado que os novos pacotes possuem um menor acoplamento, conforme a métrica *effeient coupling*, podendo indicar uma melhor modularização. Portanto detectar estas mudanças nos pacotes pode ajudar o desenvolvedor a explicar a coesão, acoplamento e a modularidade do sistema em termo de pacotes. Um dos resultados importantes da abordagem, a avaliação de impacto de mudanças, permite ao desenvolvedor avaliar o passado do sistema, e prever o impacto e abrangência de mudanças futuras em classes e funcionalidades.

Palavras-chave: Análise Dinâmica. Análise Estática. Mudanças Estruturais. Evolução do Software. Padrões de Mudanças.

Abstract

Software system comprehension is a key maintenance activity. During the software maintenance and evolution, continuous changes may degrade the modular design overtime, thus, increasing its complexity. Consequently, companies spend a lot of time and resources trying to understand and implement changes on software. Therefore, understanding how system changes evolve is an important step towards future development planing and management. Developers usually need to rapidly understand recent changes before implementing a new feature. Despite of several approaches to improve software comprehension, they are still limited to different components with similar roles, to locate features in the source code and to measure the impact of an specific change in other features. In this work, we present an approach centered on dynamic and static analysis to reduce program comprehension effort. More specifically, we propose a mechanism to locate what structural changes have occurred in a program from one version to another. Given one specific functionality, we locate structural changes and component relationship changes between two versions. Each structural change previously detected in the first step, is then verified by a next step of static analysis to confirm if the method in the trace really exists in only one version or both versions. The candidate changes are classified in five patterns by parsing the source code of both analyzed version: i) Move Class, ii) Move Method, iii) Add Method, iv) Remove Method, and v) Access Modifier Change (where they represent Class Interface Change). We evaluated our approach with three open source-software systems: jFreeChart, Tomcat, and JHotDraw. Our results reveals structural changes such as, move method, move class, and package relationship changes. In this study, we further investigate the impact of structural changes over multiple functionalities. We also evaluated the package relationship change found in jFreeChart using precision and recall. The results show that the pattern *Move Method* dominates, in average, appearing in 28,4% of the changes. Also, there are changes in classes that affect many functionalities. Also the results show that in *jFreeChart* there were changes in packages detected with a precision of 100% and a recall of 83%. After the approach detected many changes between versions 0.7.0 and 0.9.5 of *jFreeChart*, further analysis showed that the new package structure has

less coupling measures, according to the Efferent Coupling metric. That can mean the package structure has a better modular structure. Then detecting those changes in the package structure can be valuable to the developer evaluate the cohesion, coupling and package modular structure. One of the results presented by this approach, the impact analysis, allow the developer, by evaluating the past of the system, foresee the impact and coverage of future changes that will be made in the system.

Keywords: Dynamic Analysis. Static Analysis. Structural Changes. Software Evolution. Change Patterns.

Lista de ilustrações

Figura 1 – Resumo dos bugs que foram introduzidos na adição de métodos no Tomcat 7. Números de problemas reportados no Bugzilla #52607 and #52952.	23
Figura 2 – Métodos adicionados na versão 7 do Tomcat.	23
Figura 3 – Exemplo de rastros de execução.	32
Figura 4 – Exemplo de aplicação da técnica de compressão de rastros.	33
Figura 5 – Exemplo de alinhamento de rastros de execução.	34
Figura 6 – O processo de detecção de mudanças combinando análise dinâmica e estática.	36
Figura 7 – Exemplo de alinhamento de rastros de execução.	39
Figura 8 – Exemplo de movimentação de classe entre as versões 7.2 e 7.3 para a funcionalidade <i>Draw Line Dashes</i> , do <i>JHotDraw</i>	40
Figura 9 – Exemplo de movimentação de método (<i>Move Method</i>).	41
Figura 10 – Exemplo de adição de método, <i>Add Method</i> . <i>jFreeChart</i> - alinhamento entre as versões 0.7.0 e 0.9.5 para <i>Time Series Charts-Time Series 2</i>	42
Figura 11 – Exemplo de remoção de método (<i>Remove Method</i>). <i>JHotDraw</i> - alinhamento entre as versões 7.2 e 7.3 para a funcionalidade <i>Save As</i>	42
Figura 12 – Mudança de modificador de acesso (<i>Access Modifier Change</i>).	43
Figura 13 – Mudança de modificador de acesso. <i>jFreeChart</i> - alinhamento entre as versões 0.7.0 e 0.9.5 para a funcionalidade <i>Combined Charts - Combined and Overlaid Dynamic Chart</i>	43
Figura 14 – Cobertura relativa de mudanças de interface de classe entre duas versões do sistema.	51
Figura 15 – Número de funcionalidades afetadas por mudanças estruturais em classes.	53
Figura 16 – As 10 classes alteradas que mais impactam funcionalidades.	54

Lista de tabelas

Tabela 1	– Exemplos de cenários de execução.	31
Tabela 2	– Sistemas avaliados (Métricas de tamanho).	47
Tabela 3	– Pares de sistemas avaliados nos experimentos para aplicar o alinhamento.	47
Tabela 4	– Número de cenários usados por cada sistema.	48
Tabela 5	– Número de segmentos alinhados e desalinhados para cada par de versões do sistema.	49
Tabela 6	– Número de mudanças classificadas como <i>Move Class</i> de uma versão para outra.	50
Tabela 7	– Pacotes adicionados e removidos entre <i>jFreeChart 0.7.0/0.9.5</i>	55

Sumário

1	INTRODUÇÃO	21
1.1	Problema e Motivação	21
1.2	Objetivos e Desafios da Pesquisa	25
1.3	Hipótese	25
1.4	Contribuições	26
1.5	Organização da Dissertação	28
2	FUNDAMENTAÇÃO TEÓRICA	29
2.1	<i>Feature Location</i>	29
2.2	Análise Estática	30
2.3	Análise Dinâmica	30
2.3.1	Coleta de Rastros de Execução	30
2.3.2	Compressão de Rastros de Execução	32
2.3.3	Sumarização de rastros de execução.	32
2.3.4	Alinhamento de Rastros de Execução	33
2.4	Conclusão	34
3	PROPOSTA	35
3.1	Fase I — Extração e Preprocessamento de Rastros de Execução	35
3.1.1	Extração de Rastros de Execução	36
3.1.2	Redução dos rastros de execução	37
3.1.3	Separação por cenários de execução	37
3.2	Fase II — Detecção de Mudanças Estruturais de Código	38
3.2.1	Alinhamento de Rastros de Execução	38
3.2.2	Extração de Mudanças Estruturais	39
3.3	Considerações Finais	43
4	AVALIAÇÃO	45

4.1	Perguntas de pesquisa	45
4.2	Experimentos	46
4.2.1	Sistemas analisados	46
4.2.2	Fase 1 - Extração e Preprocessamento dos Rastros	47
4.2.3	Fase 2 - Detecção de Mudanças Estruturais	49
4.3	Resultados	50
4.3.1	RQ1 - Há algum padrão de mudança mais frequente?	50
4.3.2	RQ2 - As funcionalidades são impactadas pelas mudanças estruturais detectadas?	52
4.3.3	RQ3 - Quão efetivo é o uso de alinhamento de rastros para detectar mudanças estruturais em pacotes?	54
4.3.4	Discussão	57
4.3.5	Limitações	59
5	TRABALHOS RELACIONADOS	61
5.1	Análise Estática do Código Fonte	61
5.2	Análise de Histórico de Versões	64
5.3	Análise Dinâmica	67
6	CONCLUSÃO	69
6.1	Principais Contribuições	69
6.2	Trabalhos Futuros	71
	REFERÊNCIAS	73

Introdução

Primeiramente, neste capítulo introduz-se o problema central investigado neste trabalho (Seção 1.1). Em seguida, apresentam-se os objetivos, desafios de pesquisa e hipótese de trabalho (Seções 1.2 e 1.3). Na Seção 1.4 apresentam-se as contribuições e para concluir este capítulo apresenta-se brevemente o conteúdo de cada capítulo deste trabalho (Seção 1.5).

1.1 Problema e Motivação

As empresas ou fábricas de software estão quase sempre sob pressão constante no mercado e essa pressão acaba chegando até a equipe de desenvolvimento, o que pode comprometer a qualidade dos sistemas de software (AUSTIN, 2001). Além dos desenvolvedores poderem introduzir *bugs* no código fonte (VAHABZADEH; FARD; MESBAH, 2015), o projeto do sistema pode-se degradar com o tempo (OIZUMI et al., 2014; EICK et al., 2001; TUFANO et al., 2015). Muitas decisões feitas sob pressão podem impactar negativamente na qualidade do software. Até bons desenvolvedores podem escrever códigos ruins e introduzir anomalias e erros (LAVALLEE; ROBILLARD, 2015; GUIMARÃES; GARCIA; CAI, 2015; SILVA et al., 2015). Além disso, sabe-se que muito do esforço para modificar o código fonte é empregado na tarefa de compreensão. Um desenvolvedor, ao realizar uma mudança, gasta boa parte do tempo nesse processo compreendendo o código fonte. Muitas vezes esse esforço de compreensão é diário e ocorre muitas vezes no mesmo dia (SANCHEZ; ROBBES; GONZALEZ, 2015). O esforço tende a aumentar quando não há documentação disponível e atualizada. Normalmente a implementação de tarefas complexas requerem da equipe de desenvolvimento, a compreensão do estado atual do sistema, em termos de *design*, do relacionamento entre os artefatos e a estrutura atual do sistema (WILLIAMS; CARVER, 2010).

A compreensão de mudanças anteriores no código fonte, portanto, é essencial para o desenvolvedor durante a tarefa de manutenção e a fase de evolução do sistema. Durante a implementação de novos requisitos, o desenvolvedor procura no código fonte onde ele

deve aplicar mudanças e embora isso pareça trivial, é uma tarefa difícil (TAO et al., 2012). Além disso, a prática de entender o código é feita, na maioria das vezes, concentrada nas funcionalidades que o desenvolvedor precisa modificar (TAO et al., 2012). No entanto, geralmente os desenvolvedores não estudam as mudanças no código fonte do ponto de vista comportamental da execução dos métodos daquelas funcionalidades em que estão interessados. Atualmente, existem muitos estudos na literatura que visam a melhorar a compreensão de programas (SARTIPI; DEZHKAM, 2007; DUCASSE; POLLET, 2009) e abordagens sobre a evolução de sistemas (HU et al., 2015; JIANG et al., 2015; EICK et al., 2001). Há diferentes fontes de informações que podem ser utilizadas para auxiliar na compreensão de programas como o próprio código fonte (XING; STROULLIA, 2005; HASHIMOTO; MORI; IZUMIDA, 2015), históricos de versão (WONG et al., 2011; SILVA et al., 2015; KIM et al., 2007; CODOBAN et al., 2015), e os rastros de execução (ARISHOLM; BRIAND; FOYEN, 2004; SILVA et al., 2011).

Nesta dissertação é proposta uma técnica para direcionar o desenvolvedor na avaliação de mudanças aplicadas a cada versão lançada do sistema, facilitando a análise de impacto de mudanças e a interação entre os pacotes evidenciadas pela técnica. Por exemplo, se um pacote começa a se desviar do seu projeto estrutural inicial, o desenvolvedor tem condições de perceber isso prematuramente e avaliar se é preciso aplicar correções. A Figura 1 ilustra um exemplo de motivação observada nos comentários de desenvolvedores sobre duas adições de métodos (veja Figura 2) que ocorreram na versão 7 do sistema Tomcat¹ da Apache. O método `validateApplication` foi adicionado como estático na classe `ExtensionValidator` e o método `initInternal` foi adicionado na classe `StandardServer`.

No entanto, a adição destes métodos introduziu erros no código, onde foram reportados no sistema de *issue tracker* Bugzilla (*issues* de número #52607 e #52952). Conforme o desenvolvedor que abriu a *issue* #52607, quando é especificado o diretório dos arquivos de extensão `jar`, é disparada a mensagem de `ClassNotFoundException`. Já o problema reportado na *issue* #52952, conforme a descrição do desenvolvedor deveria haver uma validação para cada instância ao invés de um por contexto.

Nas Figuras 1 e 2, a adição dos métodos, foi detectada pela técnica proposta neste trabalho em experimento realizado entre as versões 6 e 7 do Tomcat. Esta adição foi detectada no cenário de inicialização com bootstrap, o que permite ligar a mudança com uma funcionalidade específica.

Um outro exemplo de mudança estrutural detectada pela técnica proposta mostra o impacto que uma mudança aplicada em uma funcionalidade pode ter sobre outras, da seguinte maneira: o método `org.apache.catalina.connector.Request.recycle` foi movido para a classe `org.apache.coyote.http11.Http11Processor` no Tomcat 7. Esta mudança foi detectada na funcionalidade “Verificar Status do Servidor”. No entanto, ela

¹ <http://tomcat.apache.org/>

Bug: 52607

Status: RESOLVED FIXED

Revision: 1242101

Version: 7.0.25

Descrição Resumida: A classe `ExtensionValidator` precisa estar ciente das classes nos carregadores compartilhados e comuns.

Descrição: Como um resultado do email[1], eu estou reportando um bug.

...

- Contra: Durante `ExtensionValidator.validateApplication`, pode parecer que uma grande quantidade de arquivos são digitalizados sem necessidade.

Bug: 52952

Status: NEW

Revision: 1242101

Version: 7.0.26

Reported: 2012-03-20 15:22

Descrição Resumida: Melhorar o uso da `ExtensionValidator` para cenários embutidos.

Descrição: A chamada a seguir foi adicionada em r1242101. Em `StandardServer#initInternal()`:

```
> + ExtensionValidator.addSystemResource(f);
```

O método acima é um método estático e não deveria verificar por duplicatas.

Não escala bem, como por exemplo, se existe várias instâncias do Tomcat ...

Eu diria que implementar `ExtensionValidator` com métodos estáticos é uma falha aqui. O único lugar onde é chamado é

`StandardContext#startInternal()` > que chama `ExtensionValidator.validateApplication()`.

Figura 1 – Resumo dos bugs que foram introduzidos na adição de métodos no Tomcat 7. Números de problemas reportados no Bugzilla #52607 and #52952.

```
-----
java.org.apache.catalina.util.ExtensionValidator.java.static.validateApplication
java.org.apache.catalina.core.StandardServer.protected.initInternal
-----
```

Figura 2 – Métodos adicionados na versão 7 do Tomcat.

também afetou as funcionalidades “Acessar o Manager” e “Inicializar Tomcat com Bootstrap”. Portanto, para garantir o bom funcionamento do sistema após a mudança o desenvolvedor deveria aplicar testes nas três funcionalidades impactadas por esta alteração. Por isso é importante o uso dos cenários - a abordagem permite avaliar quais cenários foram afetados por tal mudança.

Embora existam várias técnicas para reduzir o esforço de compreensão de programas por comparar versões diferentes, não existe uma abordagem amplamente adotada pelos engenheiros e desenvolvedores (TAO et al., 2012). A manutenção de software ainda possui um alto custo e as empresas gastam muito tempo tentando entender e implementar mudanças nos sistemas (SEACORD; PLAKOSH; LEWIS, 2003; DIT; REVELLE;

POSHYVANYK, 2013; MEYER, 2000). Mesmo as abordagens úteis existentes baseadas no código fonte (que detectam mudanças estruturais) e em históricos de versões (que detectam mudanças lógicas), carecem de informações comportamentais e não capturam satisfatoriamente mudanças associadas a funcionalidades específicas. Além disso, a compreensão do código fonte é sujeita ao viés do desenvolvedor, por exemplo, um desenvolvedor pode querer implementar uma mudança na classe A e outro pode querer implementar na classe B ou um desenvolvedor pode procurar por informações relativas a uma mudança em um local e outro desenvolvedor procura em local diferente (PIORKOWSKI et al., 2015). Uma informação objetiva ajuda a amenizar a subjetividade na interpretação do código fonte por diferentes desenvolvedores, por exemplo, ao mostrar que uma mudança afeta determinadas classes, evita que desenvolvedores procurem em outras classes que podem acabar não sendo afetadas por tal mudança. Por exemplo, UMLDiff é uma técnica capaz de encontrar renomeação de classes, movimentação, adição e remoções de métodos (XING; STROULIA, 2005). Para o *jFreeChart*, no entanto, das 26.028 mudanças detectadas, 17.018 representam adições e 5.082 remoções avaliando-se o sistema inteiro. Assim, pode-se observar que esse tipo de comparação estrutural não é capaz de comparar entidades diferentes, mas com papéis similares no sistema. Quando o objetivo é compreender o comportamento do sistema – a interação entre os objetos e o comportamento dos componentes – o uso de análise dinâmica pode ser uma ferramenta útil para tal problema. Uma abordagem dinâmica é usada no UMLDiff onde o sistema é avaliado por inteiro. O uso desta ferramenta traz alguns benefícios ao usuário, mas o real benefício para o desenvolvedor quando ele quer fazer uma modificação específica é pequeno. Porque ele precisa buscar mudanças que ocorreram no sistema inteiro ao invés de mudanças localizadas. Então, o desenvolvedor não saberá, por exemplo, quais classes serão impactadas caso ele efetue uma determinada mudança no sistema. Além de o resultado do UMLDiff conter muita informação para avaliar, as mudanças não possuem relação com cenários de uso e nem uma avaliação de impacto de mudanças específicas. A abordagem proposta neste trabalho, por outro lado, mostra mudanças ocorridas por cenários. Assim se o desenvolvedor quiser, ele pode até exibir todas as mudanças como ocorre no UMLDiff. Mas poderá, também, exibir mudanças ocorridas em um cenário específico. Ou ainda, quais os cenários que podem ser afetados por uma determinada mudança. Com esta informação, dos cenários que serão afetados, o desenvolvedor poderá julgar melhor sobre a mudança, efetuar-la em menor tempo, com mais segurança, uma vez que ele terá uma informação mais específica e mais objetiva. Os casos de uso, histórias de usuário, solicitações de mudança e descrições de mudanças em alto nível podem ser usados para montar um cenário de execução ou para eleger um cenário de execução e assim o desenvolvedor poderá trabalhar com mudanças específicas daquele cenário e seus impactos.

1.2 Objetivos e Desafios da Pesquisa

As abordagens dinâmicas propostas até o momento não tratam da localização de mudanças estruturais, mas abordam, sobretudo, problemas de *bugs* e novas implementações. As abordagens dinâmicas existentes, buscam ajudar o desenvolvedor a localizar funcionalidades para corrigir erros, implementar novas funcionalidades (CORNELISSEN et al., 2009; SILVA et al., 2011; DIT; REVELLE; POSHYVANYK, 2013) e recuperar diagramas UML (BRIAND; LABICHE; LEDUC, 2006). Para tratar essa lacuna—a localização de mudanças estruturais—este trabalho propõe uma abordagem centrada na análise dinâmica e estática, que investiga meios para reduzir o esforço do desenvolvedor na tarefa de avaliação e compreensão de mudanças no código fonte. Mais especificamente, este trabalho propõe um mecanismo para localizar quais mudanças estruturais ocorreram no programa entre uma versão e outra do sistema associadas a funcionalidades específicas.

Por representar uma necessidade comum entre os desenvolvedores, o objetivo deste trabalho é auxiliar na compreensão das mudanças, visto que é possível estudá-las, no sistema, com base nas funcionalidades. Por exemplo, saber quais classes são afetadas por mudanças em uma funcionalidade é uma tarefa de compreensão de mudanças. O engenheiro de software e a equipe de desenvolvedores precisam ter uma compreensão rápida do código em que querem trabalhar. Por exemplo, ao implementar um requisito é necessário saber quais outros lugares do sistema podem ser afetados. Fazer isso com uma funcionalidade por vez, dá ao desenvolvedor a oportunidade de ter um ganho, reduzindo o tempo de compreensão, pois ele não precisará empregar uma técnica que avalia o código inteiro ou as técnicas de *feature location*, técnicas de localização de mudanças que associam trechos do código fonte a uma funcionalidade (Seção 2.1). Ao invés de aplicar uma técnica de localização de funcionalidades, ele pode usar somente a abordagem proposta, pois ela já relaciona as mudanças estruturais com funcionalidades específicas. Assim é possível saber em qual funcionalidade ocorreram as mudanças e mais ainda, em qual parte do código fonte. De forma mais simplificada, o desenvolvedor pode empregar a técnica proposta neste trabalho que permite avaliar uma funcionalidade por vez.

1.3 Hipótese

Este trabalho busca categorizar mudanças estruturais em cinco categorias, sendo elas *Move Class*, *Move Method*, *Add Method*, *Remove Method* e *Access Modifier Change*. Para classificar essas mudanças, os métodos são extraídos de desalinhamentos de rastros de execução. Por hipótese, esta abordagem supõe que, métodos com granularidade maior que um são mais importantes para detectar as mudanças estruturais do que aqueles com granularidade um. Métodos com baixa granularidade são, geralmente, métodos privados ou métodos *getters* e *setters* e que, portanto, não são usados em outras classes e não

afetam o sistema estruturalmente. Conhecer os padrões de mudança é útil ao desenvolvedor pois o ajudará a compreender as mudanças estruturais do sistema. Além disso, pode existir um padrão que seja mais frequente (por exemplo, o *Remove Method* pode ser mais frequente entre duas versões de um sistema), e detectá-los pode auxiliar a equipe de desenvolvimento, que permite, por exemplo, preparar testes para tais mudanças. Além disso, é possível avaliar o impacto das mudanças em relação às funcionalidades. A abordagem proposta permite obter a informação de quantas e quais são as funcionalidades afetadas por uma mudança estrutural, pois uma mudança no código pode afetar várias partes que o usam. Esse resultado é obtido pela separação das classes que pertencem a um cenário. Por exemplo, na coleta de rastros uma classe pode aparecer em rastros de três funcionalidades. Sabendo-se disso, essas partes que causam muito impacto podem ser classes que devem ser tratadas com mais cuidado, pela tendência de causar mudanças em outras partes do código. Por exemplo, classes genéricas do sistema ou até mesmo classes com alto acoplamento de interesses (que são usadas por funcionalidades semelhantes). Neste trabalho, estuda-se também, a relação entre artefatos do sistema e especialmente, o relacionamento entre os pacotes. Assim, três perguntas de pesquisa são definidas:

- ❑ RQ #1: Quais os padrões de mudanças mais frequente?
- ❑ RQ #2: As funcionalidades são impactadas pelas mudanças estruturais detectadas?
- ❑ RQ #3: Quão efetivo é o uso de alinhamento de rastros para detectar mudanças estruturais em pacotes?

O Capítulo 4 conduz estudos empíricos para responder às perguntas de pesquisa apresentadas acima.

1.4 Contribuições

Esse trabalho contribui com uma abordagem híbrida baseada em análise dinâmica e estática para detectar e localizar mudanças estruturais no código fonte.

As contribuições dessa abordagem foram elencadas em cinco partes principais:

Localização de mudanças estruturais orientada a features - - A proposta de uma técnica para auxiliar na compreensão de mudanças estruturais em um sistema de software: O problema de localização de funcionalidades no código fonte é um problema difícil e amplamente debatido na literatura (DIT; REVELLE; POSHYVANYK, 2013; BECK et al., 2015; DIT et al., 2013; DIT; MORITZ; POSHYVANYK, 2012; DIT et al., 2011; REVELLE; DIT; POSHYVANYK, 2010; SAVAGE; REVELLE; POSHYVANYK, 2010; REVELLE; POSHYVANYK, 2009; LIU et al., 2007; POSHYVANYK et al., 2007). A

abordagem proposta neste trabalho traz uma contribuição para a localização de mudanças estruturais por funcionalidade. Para uma funcionalidade específica do sistema, as mudanças efetuadas entre duas versões do sistema são detectadas, trazendo como resultado, quais métodos, classes e pacotes foram modificados.

Categorização automática das mudanças estruturais - A detecção de cinco padrões de mudanças para auxiliar o desenvolvedor na compreensão de uma feature específica. Através da investigação do histórico dos sistemas, pôde-se observar que algumas mudanças estruturais podem provocar falhas futuras. Assim, conhecendo-se as mudanças estruturais e suas categorias, é possível prevenir falhas futuras ou até mesmo saber onde elas acontecerão, para que sejam corrigidas mais rapidamente: A metodologia proposta detecta mudanças estruturais e classificam-nas em cinco padrões de mudanças:

- ❑ **Move Class**—classes que foram movidas de pacotes.
- ❑ **Move Method**—métodos movidos para classes em pacote diferentes.
- ❑ **Add Method**—implementação de um novo método no sistema.
- ❑ **Remove Method**—métodos removidos do código fonte.
- ❑ **Access Modifier Change**—mudança na visibilidade de métodos.

Análise de impacto de mudanças estruturais sobre as features: A técnica permite o desenvolvedor prever o impacto que futuras mudanças provocarão em outras classes ou outras *features*, bem como preparar casos de teste para *features* que dificilmente seriam testadas sem a aplicação da abordagem. Mais especificamente, quando uma mudança afeta estruturas de classes e pacotes, o desenvolvedor consegue avaliar quantas e quais são as funcionalidades afetadas por tal mudança. Dessa maneira, estas informações podem orientar o desenvolvedor na identificação dos testes de unidade que estão associados às funcionalidades impactadas para detectar possíveis erros adicionados após a mudança.

Detecção de mudança de comportamento entre pacotes: Esta abordagem detecta a mudança de comportamento entre os artefatos do sistema, em especial a mudança de relacionamento entre pacotes do sistema. Isso pode indicar uma possível mudança arquitetural no sistema, uma vez que as relações entre os componentes podem ter sido modificadas.

Por fim, uma série estudos empíricos em três projetos Java de código aberto. Primeiro, uma análise sobre o impacto de mudanças estruturais aplicadas em diferentes versões dos sistemas. Por último, uma investigação sobre as mudanças no relacionamento entre pacotes, que fornece uma informação mais alto nível acerca de como se relacionam os componentes do sistema que estão distribuídos nos pacotes.

1.5 Organização da Dissertação

Este trabalho apresenta uma abordagem para extração e compreensão de mudanças usando análise dinâmica e estática, sendo organizado da seguinte maneira:

O Capítulo 2 explica a fundamentação teórica desse trabalho de dissertação de mestrado, onde os principais conceitos fundamentais para a compreensão da dissertação são mostrados. Essa pesquisa faz uso de conceitos já definidos na literatura. Esses conceitos são exibidos com detalhes e exemplos. Os conceitos fundamentais são: *Feature Location*, Análise Dinâmica, como são definidos os Cenários ou Funcionalidades, a Coleta de Rastros de Execução o Alinhamento dos Rastros de Execução e a Análise Estática.

O Capítulo 3 detalha a abordagem, definido em duas fases principais: i) Fase 1—extração de rastros de execução e passos de pré-processamento, ii) Fase 2—detecção, classificação e listagem das mudanças. Na fase 1, os sistemas são instrumentados e os cenários são planejados. Em seguida, os rastros de execução são coletados, para versões diferentes do mesmo sistemas. No último passo, os rastros são reduzidos e separados por funcionalidade. Na fase 2 é aplicada uma técnica de alinhamento em pares de rastros (um rastro pertence a cada versão do sistema) para detectar mudanças e a aplicação da análise estática para organizar as mudanças estruturais e classificá-las em cinco padrões.

O Capítulo 4 mostra uma avaliação da abordagem proposta com seis experimentos realizados em três sistemas, sendo três experimentos com pares de versões do *Tomcat*, dois experimentos com pares de versão do *jFreeChart* e um experimento com um par de versões do *JHotDraw*. São avaliados critérios para validar a abordagem: se as mudanças detectadas foram, de fato, efetuadas; quais cenários são afetados por determinadas mudanças; e se as mudanças afetam as relações existentes entre pacotes.

O Capítulo 5 apresenta os trabalhos relacionados com esta abordagem separados por tópicos de interesse: Análise Estática do Código Fonte, Análise de Versões e Análise Dinâmica.

Por fim, o Capítulo 6 conclui o trabalho proposto, mostrando as contribuições, limitações e possíveis trabalhos futuros.

Fundamentação Teórica

A Seção 2.1 descreve o conceito de *feature location*. A Análise Dinâmica é abordada na Seção 2.3, mostrando os conceitos essenciais para compreensão da técnica utilizada neste trabalho. Em seguida, a Seção 2.2 apresenta conceitos básicos relacionados a análise estática. Por último, a Seção 2.4 conclui o capítulo.

2.1 *Feature Location*

Um sistema possui características de funcionamento, por exemplo: gravar um arquivo, mudar a cor de um texto, localizar texto. Essas características são implementadas por meio do código fonte do sistema. Porém é difícil identificar, examinando o código fonte, qual parte é responsável por cada uma dessas características. A localização de conceitos ou de características é uma tarefa que demanda esforço, onde partes do código fonte são associadas a uma determinada funcionalidade (DIT; REVELLE; POSHYVANYK, 2013; BECK et al., 2015; DIT et al., 2013). A característica do sistema pode ser abstraída para vários outros conceitos, como falhas, casos de testes, *commits* e cenários de execução (DIT; REVELLE; POSHYVANYK, 2013). Neste trabalho foi adotado, por hipótese, que um cenário representa uma funcionalidade do sistema. Um cenário consiste em um conjunto de passos e, quando se aplica, um conjunto de dados para executar uma funcionalidade ou para usar uma característica.

Existem técnicas em *feature location* que se baseiam em diferentes fontes de informação como, análise estática (dependências estruturais do código fonte) (XIE; POSHYVANYK; MARCUS, 2005), histórica (histórico de mudanças do sistema) (CHEN et al., 2001), textual (se baseia nos textos do código fonte) (ZHAO et al., 2004) e dinâmica (coleta de rastros de execução) (DIT; REVELLE; POSHYVANYK, 2013). Nesta dissertação, a técnica proposta utiliza informação dinâmica para localizar uma funcionalidade no sistema e informação estática para detectar mudanças estruturais que ocorreram em uma determinada funcionalidade de uma versão para outra.

2.2 Análise Estática

Parte da técnica proposta neste trabalho se baseia na informação estática do sistema. A análise estática geralmente é aplicada para extrair dependências estruturais do código fonte. Tipicamente, a árvore sintática abstrata—abstract syntax tree (AST)—é gerada para representar a estrutura do código fonte.

Neste trabalho a AST é utilizada para localizar um método. Por exemplo, suponha que seja necessário localizar a classe que o método `A.m1` esteja implementado. Como o rastro de execução representa informação dinâmica do código, não necessariamente o método `m1` está implementado na classe `A`. Assim, é necessário percorrer as superclasses de `A` para certificar qual classe `m1` se encontra. Como a técnica proposta tem como objetivo detectar mudanças estruturais que ocorreram em um sistema de uma versão para outra, é preciso verificar se o método foi removido do sistema, movido ou simplesmente alterado (modificador, mensagem, tipo de retorno).

2.3 Análise Dinâmica

A análise dinâmica consiste nas técnicas de avaliação do sistema do ponto de vista comportamental. A avaliação dinâmica usa dados que são provenientes da execução do sistema.

2.3.1 Coleta de Rastros de Execução

Para se fazer a coleta de rastros de execução, é preciso instrumentar o código fonte do sistema e instrumentá-lo. Existem maneiras diferentes de instrumentar o código e neste trabalho é utilizada uma ferramenta baseada em AspectJ devido sua facilidade de uso e integração com as ferramentas de desenvolvimento. A coleta de rastros com a ferramenta é feita a partir da execução de um cenário definido previamente.

Cenário de Execução: Basicamente, um cenário de execução consiste em um conjunto de passos a serem seguidos durante a execução de uma funcionalidade (ou tarefa) específica do sistema. A Tabela 1 mostra quinze exemplos de cenários de execução, cinco para cada sistema (JFreeChart¹, Tomcat² e JHotDraw³, respectivamente).

Para executar a ferramenta, o desenvolvedor deve informar o início e o fim de cada passo definido no cenário de execução. Enquanto os passos do cenário é executado, a ferramenta armazena as chamadas de métodos no arquivo de rastros. Além disso, para cada *thread* disparada durante a execução do cenário, um arquivo é gerado. Por exemplo, a Tabela 1 mostra cenários executados neste trabalho para avaliar a técnica. Tais cenários

¹ Biblioteca para desenhar gráficos, www.jfree.org/jfreechart/

² Servidor Web, tomcat.apache.org/

³ Ferramenta de desenhos, www.jhotdraw.org/

Tabela 1 – Exemplos de cenários de execução.

Sistema	Cenários
JFreeChart	Desenhar Gráfico em Barras Horizontais
	Desenhar Gráfico de Barras Empilhadas
	Desenhar Gráfico em Barras Vertical
	Desenhar Gráfico de Gantt
	Desenhar Gráfico Pizza
Tomcat	Acessar o <i>Manager</i>
	Fazer <i>Deploy</i> de Aplicacao
	Inicializar Tomcat Bootstrap
	Listar Aplicações
	Verificar Status do Servidor
JHotDraw	Desenhar Elipse
	Desenhar Polígono
	Desenhar Retângulo
	Digitar Texto em Caixa de Texto
	Novo Arquivo

possuem somente um passo para executar suas respectivas funcionalidades. No entanto, dependendo do sistema, um cenário pode consistir de um conjunto de passos para executar uma funcionalidade. Na avaliação da técnica proposta neste trabalho, vários cenários de execução foram definidos para realizar a coleta dos rastros de execução. No caso do *Tomcat* e *jFreeChart* foram usados os casos de teste dos respectivos sistemas. Por outro lado, para o *JHotDraw* foram usadas as tarefas definidas pelo sistema (Tabela 1).

Como cada rastro de execução é coletado por cenários, é possível, então, saber que aquela parte do código fonte pertence a uma determinada funcionalidade. Rastros de execução, de fato, são uma informação importante para o estudo de *feature location* ((DIT; REVELLE; POSHYVANYK, 2013)).

A Figura 3 mostra um fragmento de arquivo de rastros de execução. Cada linha do arquivo contém a uma chamada de método com o nome qualificado, o nível da chamada na pilha de execução, o seu *timestamp* e o conteúdo dos parâmetros passados durante sua chamada. Por exemplo, na linha um mostra o primeiro método que foi chamado na execução do cenário que o rastro representa. Pode-se observar também que o método `public.actionPerformed` da classe `com.jrefinery.chart.demo.JFreeChartDemo` não possui passagem de parâmetros e está no nível 1 da pilha de execução. Como outro exemplo, o método na linha dois `private.displayCombinedAndOverlaidDynamicXYChart` está no nível 2 da pilha de execução.

O volume de informações gerado pelos rastros de execução é enorme porque eles possuem muitas informações redundantes como chamadas recursivas e laços. Portanto, os rastros devem ser preprocessados para reduzir de tamanho tornando viável o uso da informação dinâmica.

```

com.jrefinery.chart.demo.JFreeChartDemo,public.actionPerformed,1,1,1444950121178,[]
com.jrefinery.chart.demo.JFreeChartDemo,private.displayCombinedAndOverlaidDynamicXYChart,2,1,...
com.jrefinery.chart.HorizontalNumberAxis,protected.<init>,3,8512,1444950121179,[X]
com.jrefinery.chart.HorizontalNumberAxis,private.createStandardTickUnits,4,8512,1444950121179...
com.jrefinery.chart.TickUnits,public.<init>,5,8513,1444950121179,[null]
com.jrefinery.chart.TickUnits,public.add,6,8513,1444950121179,[]
com.jrefinery.chart.HorizontalNumberAxis,public.setTickMarksVisible,3,8512,1444950121191,[true]
com.jrefinery.chart.HorizontalNumberAxis,public.setAutoRangeIncludesZero,3,8512,1444950121191...
com.jrefinery.chart.VerticalNumberAxis,protected.<init>,4,8567,1444950121191,[Y]
com.jrefinery.chart.VerticalNumberAxis,private.createStandardTickUnits,5,8567,1444950121191,...
com.jrefinery.chart.TickUnits,public.<init>,6,8568,1444950121191,[null]
com.jrefinery.chart.TickUnits,public.add,7,8568,1444950121191,[]
com.jrefinery.chart.VerticalNumberAxis,public.setAutoRangeIncludesZero,4,8567,1444950121205,...
com.jrefinery.chart.CombinedPlot,public.<init>,3,8783,1444950121231,[]
com.jrefinery.chart.HorizontalNumberAxis,public.setPlot,4,8512,1444950121231,[]
com.jrefinery.chart.CombinedPlot,public.<init>,3,8783,1444950121231,[1]
com.jrefinery.chart.CombinedPlot,public.setInsets,4,8783,1444950121231,[]
com.jrefinery.chart.ChartFactory,public.static.createCombinableXYChart,4,-1,1444950121231,[]
com.jrefinery.chart.XYPlot,public.<init>,5,8785,1444950121231,[]
com.jrefinery.chart.HorizontalNumberAxis,public.setPlot,6,8512,1444950121231,[]

```

Figura 3 – Exemplo de rastros de execução.

2.3.2 Compressão de Rastros de Execução

O algoritmo de compressão reduz os tamanhos volumosos dos rastros por remover informações redundantes que não são relevantes para este estudo. Assim, basta haver um desalinhamento para que uma possível mudança seja detectada. Portanto, se um método for executado muitas vezes no laço, não é necessário armazenar cada chamada, bastaria uma chamada para que o método fosse detectado. Logo, usa-se a compressão dos rastros para eliminar repetições de laços e chamadas recursivas. Após essa etapa, os rastros são reduzidos e a quantidade de informações fica muito menor para posterior avaliação. Em resumo, o algoritmo de compressão busca remover chamadas recursivas e laços. A Figura 4 mostra um exemplo da técnica de compressão. Na Figura 4a contém o rastro sem nenhum préprocessamento (crú) e a Figura 4b mostra o rastro comprimido.

2.3.3 Sumarização de rastros de execução.

O objetivo do algoritmo de sumarização, proposto (SILVA et al., 2011), é podar chamadas irrelevantes para reduzir o volume de informação que é muito grande. Os métodos são podados por meio de um parâmetro de *threshold* na granularidade do método em relação à sua árvore de execução. Esse parâmetro leva em consideração a profundidade do método na árvore de execução, contando da folha para a raiz. Assim, um método que é folha na árvore de execução—o último método que foi executado—terá granularidade um. Por hipótese, assume-se que métodos com granularidade maior são mais relevantes para a execução de uma determinada funcionalidade. Para esse trabalho, foi considerado o *threshold* de um. Com isso, podam-se métodos privados e métodos *get* e *set* que não possuem muita relevância para este trabalho, pois geralmente são os últimos ou únicos métodos a serem executados por uma determinada chamada. Os métodos privados não são

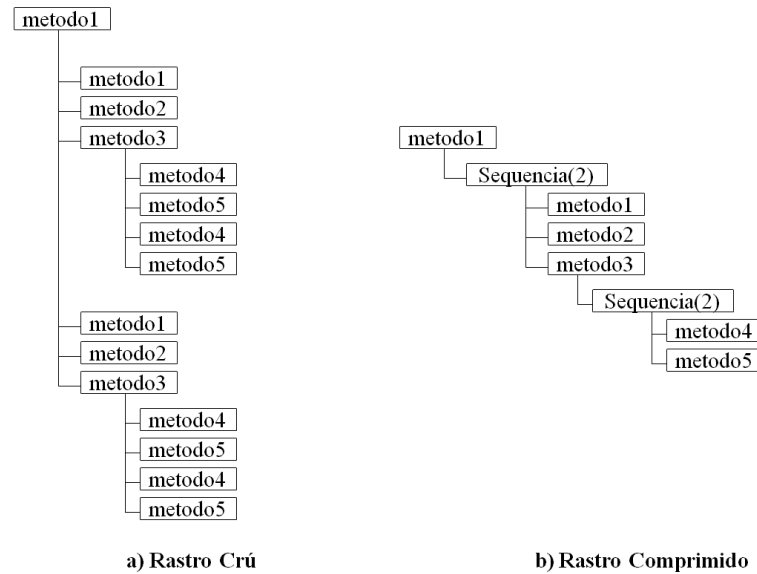


Figura 4 – Exemplo de aplicação da técnica de compressão de rastros.

muito relevantes porque não mudam o sistema de forma comportamental, por exemplo: um dos objetivos deste trabalho é detectar mudanças de relacionamentos entre sistemas. Se, em uma versão do sistema, duas classes se comunicam e depois deixam de se comunicar. Um método privado não influenciaria no relacionamento dessas classes, uma vez que uma classe não acessa o método privado de outra classe. Os métodos *get* e *set* geralmente são usados para atualizar atributos (ou propriedades) das classes, não influenciando nas mudanças estruturais. Dessa forma, a técnica de sumarização não considera esses métodos muito relevantes e portanto, são descartados pelo algoritmo com parâmetro um. Como exemplo, a Figura 4 mostra um rastro de execução representado em uma árvore.

2.3.4 Alinhamento de Rastros de Execução

Needleman and Wunsch propuseram um algoritmo de programação dinâmica para o problema de alinhamento de sequências (NEEDLEMAN; WUNSCH, 1970). O algoritmo foi adaptado para o problema de alinhamento de rastros de execução por (PAIXÃO, 2009). A adaptação consiste em sequências que contêm ruídos, característica abundante nos rastros. Por exemplo, as duas sequências “AxxxxxB” e “AxxxB” indicam que foram geradas a partir do mesmo código fonte. O que causou a diferença no número de *x* pode ter sido por uma chamada recursiva ou laço que executou o método *x* cinco vezes na primeira sequência e três vezes na segunda. Este é o principal objetivo de aplicar a técnica de compressão porque estes ruídos tendem a ser eliminados/reduzidos.

Assim, depois de realizar a coleta dos rastros de execução (Seção 2.3.1) e preprocessá-los, é feito o alinhamento dos rastros. Em (SILVA et al., 2011; SILVA et al., 2010), o alinhamento foi feito sobre pares de rastros que representam cenários diferentes, mas na mesma versão do sistema. Neste trabalho, usa-se rastros que representam cenários simi-

lares/iguais, mas em versões diferentes de um sistema. Semelhante ao trabalho proposto por (SILVA et al., 2011; SILVA et al., 2010), o alinhamento é feito por funcionalidade. Assim, espera-se que a mesma parte do código fonte seja executado e o alinhamento seja quase total. Em vários experimentos realizados neste trabalho, os rastros de execução se alinham na maioria dos casos. No entanto, existem alguns desalinhamentos. Estes desalinhamentos são os alvos de estudo porque eles podem revelar mudanças importantes que ocorreram no sistema. A Figura 5 mostra um exemplo de alinhamento de rastros de execução. Pode-se observar após o alinhamento dos dois rastros, dois blocos foram alinhados (retângulos verdes), a chamada de método **b** no Rastro 1 desalinhou, ou seja, não existe esta chamada entre os blocos alinhados. Assim, houve um deslocamento na sequência do Rastro 2. De maneira semelhante, ocorreu com a chamada de método **u**. Por outro lado, as demais chamadas de métodos ocorreram no Rastro 2, por esta razão, foi necessário ter um deslocamento da sequência no Rastro 1.

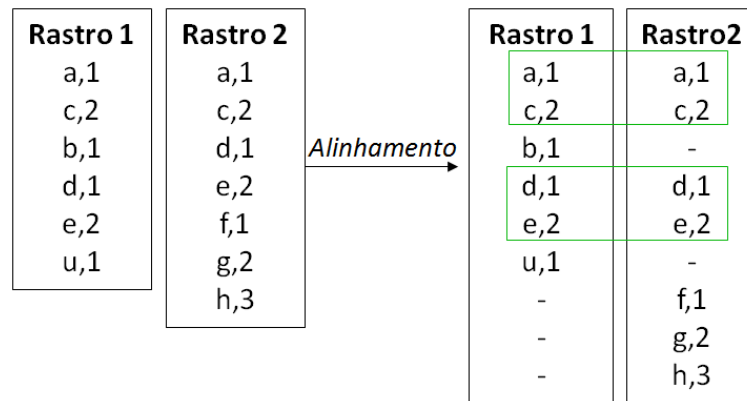


Figura 5 – Exemplo de alinhamento de rastros de execução.

2.4 Conclusão

Neste capítulo foi apresentado conceitos importantes necessários para a compreensão da técnica proposta deste trabalho. Em resumo, *Feature Location* consiste na localização de funcionalidades no código fonte (Seção 2.1). Como a técnica proposta neste trabalho faz combinação de análise dinâmica e estática, na Seção 2.2 foi apresentado os conceitos básicos de análise estática. Além disso, na Seção 2.3 foi apresentado conceitos chaves de análise dinâmica como coleta de rastros de execução e aplicação de técnicas de compressão e sumarização de rastros. Por fim, foi descrito detalhadamente a técnica de alinhamento de rastros de execução na Seção 2.3.4.

Extração de Mudanças Estruturais

Neste capítulo é apresentada a abordagem proposta para detectar mudanças estruturais por meio do alinhamento de rastros de execução. A abordagem combina análise estática e dinâmica para reduzir o esforço de compreensão de mudanças estruturais de sistemas de software. Como a localização de características em sistemas é considerada uma tarefa bastante complexa (DIT; REVELLE; POSHYVANYK, 2013), as abordagens centradas na detecção de mudanças que ocorreram em uma funcionalidade específica podem auxiliar desenvolvedores na compreensão e modificação de programas. Pois seria possível identificar quais mudanças ocorreram especificamente em uma determinada funcionalidade. Consequentemente, o tempo para entender mudanças aplicadas no sistema é reduzido porque o desenvolvedor pode analisar mudanças por funcionalidades ao invés de tentar localizá-las no código fonte. Este trabalho direciona o desenvolvedor a encontrar mudanças chaves para uma funcionalidade particular do sistema. A ideia é detectar mudanças estruturais de uma versão para outra a partir da perspectiva de diferentes características do sistema. Além disso, uma vez que as mudanças são detectadas, o desenvolvedor consegue localizar quais funcionalidades foram impactadas após modificações aplicadas no sistema. Esta abordagem consiste em duas fases: Na Seção 3.1 é apresentada a primeira, onde é feita a extração e pré-processamento dos rastros de execução (remoção ruídos como laços e chamadas recursivas). Por fim, na Seção 3.2 as mudanças são detectadas e categorizadas em cinco padrões.

3.1 Fase I — Extração e Pré-processamento de Rastros de Execução

Nesta seção, apresenta-se a fase para capturar e reduzir os rastros de execução dos sistemas. A Figura 6 mostra uma visão geral da abordagem proposta. Os primeiros passos definidos são: (i) definir os cenários, (ii) instrumentar o código fonte e (iii) capturar os rastros de execução. Em seguida, a funcionalidade definida no cenário é executada no

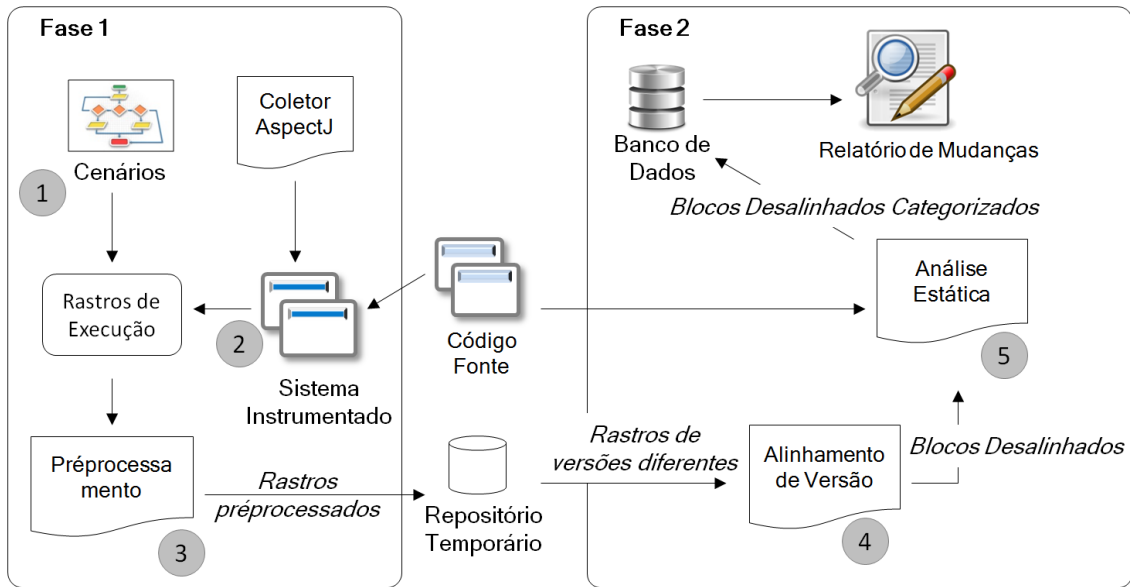


Figura 6 – O processo de detecção de mudanças combinando análise dinâmica e estática.

sistema para que os métodos chamados sejam armazenados. Após a etapa de coleta dos rastros, os rastros de execução são préprocessados devido aos seus enormes volumes de informação (ex: chamadas recursivas e laços). Técnicas de compressão e de sumarização de rastros são aplicadas para reduzi-los de forma que possam ser interpretados por seres humanos de maneira viável. O algoritmo de compressão é responsável por eliminar laços, enquanto que o algoritmo de sumarização reduz os rastros por meio da remoção de chamadas, que representam detalhes de implementação.

Por fim, com a remoção de informações redundantes e de pouca relevância, os rastros de execução são separados por cenários para reduzir ainda mais o tamanho de cada rastro e facilitar as análises. Assim, eles podem ser armazenados em arquivos diferentes conforme o cenário a que pertencem.

3.1.1 Extração de Rastros de Execução

Para que seja feita a coleta dos rastros de execução é necessário preparar o sistema. O sistema é instrumentado com a ferramenta de instrumentação baseada em AspectJ¹ (definido na Seção 2.3.1). Após ser instrumentado, é adicionada a biblioteca de coleta de rastros de execução e o sistema é recompilado. Em seguida, o sistema é executado para a funcionalidade desejada. Uma das contribuições dessa abordagem é justamente permitir ao desenvolvedor coletar rastros para funcionalidades específicas. Assim, o sistema inteiro não precisa ser usado e, como os rastros podem ser alinhados a qualquer momento para detectar mudanças, eles podem ser armazenados e guardados após a compressão. A ferramenta de coleta de rastros acompanha a execução da tarefa enquanto se usa o sistema. Durante a execução do cenário, a ferramenta permite que o desenvolvedor ou o analista

¹ Ferramenta usada para instrumentar o código fonte: <https://eclipse.org/aspectj/>

indique o início do cenário. Quando a tarefa usada no sistema termina, é indicado também o final do cenário, marcando o início e fim da execução de cada tarefa. Esse processo se repete quantas vezes forem necessárias até que as funcionalidades desejadas tenham sido coletadas. Durante a execução da tarefa do sistema, um arquivo de rastro é gerado para cada *thread* executada. Cada linha do arquivo de rastro corresponde a um método executado. A linha é composta pelo *full qualified method name*, o nome do pacote e a classe aos quais o método pertence. O rastro captura, também, o nível da chamada na pilha de execução e o seu *timestamp*. O *timestamp* é usado posteriormente para separar os métodos executados por tarefa, extraíndo os métodos cujo *timestamp* seja equivalente ao início e fim de cada cenário executado.

3.1.2 Redução dos rastros de execução

Depois de terminar a coleta de rastros, são aplicadas as técnicas de compressão e sumarização de rastros. Para reduzir o tamanho dos rastros sem perda de informação relevante, neste trabalho é aplicado uma técnica de compressão (HAMOU-LHADJ; LETHBRIDGE, 2002) e em seguida uma técnica de sumarização de rastros (SILVA et al., 2010; SILVA et al., 2011). Com isso os rastros são reduzidos de maneira significativa, sendo, portanto, mais fácil de extrair informações importantes.

O algoritmo de compressão remove informações redundantes—como laços e chamadas recursivas—que não são relevantes para este estudo. Em seguida, a técnica de sumarização é executada para eliminar chamadas de métodos com profundidade pequena na sub-árvore de execução. O cálculo do tamanho de uma sub-árvore de uma chamada de método é feita sobre o número de chamadas que houve dentro da chamada atual. Desta maneira, quanto maior for o número de chamadas internas, maior é a granularidade. A técnica considera chamadas de métodos com granularidades maiores como os métodos mais relevantes para a execução de uma determinada funcionalidade. Para sumarizar é necessário informar um filtro, onde quanto menor é a granularidade informada, menor será o número de chamadas que serão removidas. Em (SILVA et al., 2011), o processo de poda das folhas nas sub-árvores de execução ocorre em duas etapas. Neste trabalho é utilizada a mesma configuração dos autores.

3.1.3 Separação por cenários de execução

Como os rastros são coletados por cenários de execução, cada cenário tem seu início e término indicados durante a execução da tarefa. Assim, após a coleta e redução dos rastros é feita a separação por cenários de execução. Cada funcionalidade será armazenada em um diretório com seus respectivos rastros. Através do *timestamp* coletado para o método, o início e o fim de cada cenário, é possível separar os métodos de cada funcionalidade e armazená-los separadamente. Essa separação deixa os arquivos de rastros de execução

ainda menores, já que estão separados por cenário. Pode-se, por exemplo, descartar algum cenário irrelevante a critério do desenvolvedor, se ele julgar necessário. Um conjunto de cenários podem ser armazenados separadamente. Além disso, a avaliação das informações fica mais fácil visualmente quando isso é feito para cada cenário.

3.2 Fase II — Detecção de Mudanças Estruturais de Código

Como pode ser visto na Figura 6, a Fase II consiste de dois passos: i) alinhamento de versão que revela possíveis mudanças estruturais para uma funcionalidade específica e ii) análise estática que remove falsos positivos e categoriza em cinco padrões as mudanças que realmente aconteceram no código.

3.2.1 Alinhamento de Rastros de Execução

Após coletar e aplicar os passos de preprocessamento dos rastros de execução é feito o alinhamento dos rastros entre duas versões para detectar mudanças. Portanto, a coleta dos rastros deve ser feita para duas versões do sistema e as mesmas funcionalidades deverão ser executadas para a coleta. O alinhamento é feito por cenários, ou seja, um cenário da versão anterior é alinhado com o mesmo cenário da nova versão do sistema. Assim, a etapa de separação de cenários de execução é importante para o alinhamento. Pode-se fazer o alinhamento dos rastros inteiros, mas, por hipótese neste trabalho, assume-se que os rastros são alinhados por cenário. Por exemplo: o desenvolvedor pode, a cada teste que faça no sistema ou a cada *test case* executado, coletar e armazenar os rastros de execução. Mais tarde, em momento oportuno—depois de ser comunicado sobre uma mudança no sistema—ele pode usar as informações dos rastros extraídos e alinhados para analisar as mudanças e os impactos de tais mudanças. A Figura 7 mostra uma ilustração do alinhamento dos rastros de execução. O algoritmo compara os rastros que representam a execução de uma funcionalidade particular do sistema, *Trace1* (da versão A) e *Trace2* (da versão B). O algoritmo de alinhamento de sequência mostra três resultados diferentes:

- ❑ ***Aligned Sequence***: Significa que o método é o mesmo em ambas as versões, ou seja, o mesmo método executado na versão A, foi também executado na versão B. Assim não há uma mudança na chamada de método entre as sequências.
- ❑ ***Shifted Sequence #1***: Significa que houve uma mudança na sequência de chamadas de métodos na versão B, causando um desalinhamento na sequência 1.
- ❑ ***Shifted Sequence #2***: De forma análoga, significa que houve uma mudança na sequência de métodos na versão A, causando um desalinhamento na sequência 2.

Como o objetivo é usar os segmentos desalinhados para extrair as mudanças no código fonte, os segmentos repetidos causariam um esforço desnecessário para a abordagem, fazendo com que as etapas de compressão e sumarização sejam importantes para eliminar tais segmentos repetidos. Por exemplo, se duas sequências "**AcccccBE**" e "**AccccBD**" forem produzidas, a diferença entre essas sequências sugerirá que o método **c** foi executado cinco vezes em um laço na primeira sequência, enquanto que, na segunda sequência, foi executado quatro vezes. Consequentemente, o número de desalinhamentos seria maior porque o primeiro segmento geraria mais um alinhamento na última letra **c**. Se as repetições fossem eliminadas, os dois segmentos seriam: "**AcBE**" e "**AcBD**", em que o desalinhamento importante para a abordagem é apenas o último, cujas letras **E** e **D** são diferentes e pode ser um desalinhamento com mudanças. Como exemplo, se o método **D** e **E** forem o mesmo e apenas mudaram de lugar, pode ser uma mudança; se o método **D** e **E** forem diferentes, podem ser duas mudanças ou podem ser chamadas diferentes de métodos de uma versão para outra. A avaliação será refinada na etapa da análise estática. A Figura 7 mostra um exemplo com três segmentos desalinhados.

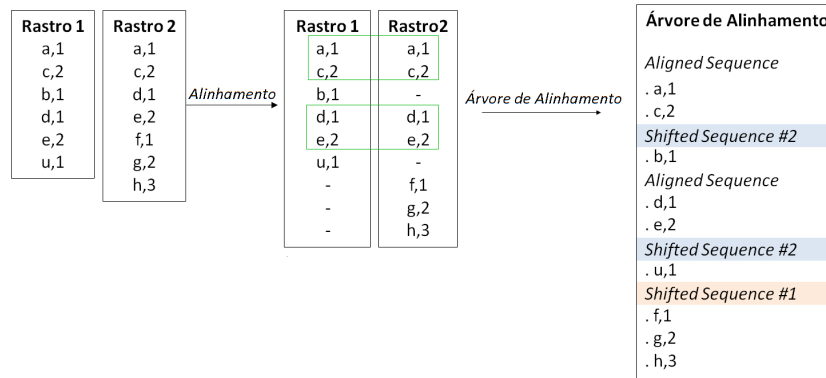


Figura 7 – Exemplo de alinhamento de rastros de execução.

3.2.2 Extração de Mudanças Estruturais

O objetivo do alinhamento de sequência com rastros de execução é recuperar mudanças efetuadas para uma funcionalidade específica, como movimentação de classes e métodos. Porém, após armazenar as informações dos desalinhamentos, da análise dinâmica, é importante eliminar falsos positivos, além de classificar mudanças em padrões. Assim, é realizada a etapa de análise estática, que irá verificar para cada desalinhamento, para cada método, em sua respectiva classe, se eles existem no código fonte da versão anterior e no código fonte da nova versão. Para fazer isso, usa-se um *parser* que verifica a existência dos métodos nas respectivas classes. Uma vez que a classe já está indicada explicitamente pelo caminho do código fonte e pelo nome do pacote, a classe é encontrada diretamente, não exigindo muito esforço computacional. Esse algoritmo é de ordem $O(n*k)$, em termos de ordem de complexidade de tempo de execução, onde **n** é o número

de métodos detectados nos desalinhamentos e k é o número de superclasses da classe avaliada em questão (o método pode não existir na classe atual, mas pode existir em uma de suas superclasses). Como, geralmente, o número de superclasses é limitado (seria incomum e estranho, uma classe herdar todas as outras classes do sistema), assume-se, por hipótese, que k é uma constante. Logo, o algoritmo $O(n*k)$ é $O(n)$, no número de métodos encontrados, pois $k \ll n$.²

Deteção de Padrões de Mudanças: A abordagem propõe a categorização de cinco padrões de mudanças estruturais para classificar os segmentos desalinhados. Depois de alinhar os rastros de execução e fazer a análise estática, os métodos são automaticamente classificados em um ou mais padrões de mudança estrutural. Essa informação também é armazenada no banco de dados, para consulta posterior e apresentação dos resultados. Os cinco padrões de mudanças estruturais são:

- ❑ **Move Class:** Segmentos desalinhados podem revelar que classes moveram de um pacote para outro. A Figura 8 mostra um exemplo de desalinhamento apontado pela abordagem como *Move Class*. A figura consiste em um exemplo de movimentação de classe entre as versões 7.2 e 7.3 para a funcionalidade *Draw Line Dashes*, do *JHotDraw*. Note que o segmento desalinhado apresenta a nova localização para a classe *VerticalGridLayout*. O padrão é investigado na fase de análise estática, em que o código fonte é verificado, para confirmar se a classe foi movida do pacote *org.jhotdraw.draw.action* para o pacote *org.jhotdraw.gui*. Se acontecer de a classe existir em ambos os pacotes, a classe é descartada e não é considerada para esse padrão. Por outro lado, se apenas um dos pacotes contém a classe, ela é classificada no padrão de movimentação de classes.

<i>Shifted Sequence #2</i>
<code>.org.jhotdraw.draw.action.VerticalGridLayout.preferredLayoutSize</code>
<i>Shifted Sequence #1</i>
<code>.org.jhotdraw.gui.VerticalGridLayout.preferredLayoutSize</code>

Figura 8 – Exemplo de movimentação de classe entre as versões 7.2 e 7.3 para a funcionalidade *Draw Line Dashes*, do *JHotDraw*.

- ❑ **Class Interface Change:** Segmentos desalinhados também podem revelar mudanças na interface da classe, como movimentação de método (quando um método vai para outra classe ou outro pacote), remoção ou adição de métodos, assim como mudança no modificador de acesso dos métodos. Essa mudança de interface de uma classe é importante, visto que pode ser uma informação significativa para o desenvolvedor, podendo indicar uma mudança de relacionamento entre componentes, por

² Ferramenta usada como *parser* para a análise estática: <http://javaparser.github.io/javaparser>

exemplo, em que uma classe deixa de usar um método de outra classe, pois ele era público e passou a ser privado. A mudança de interface pode indicar, também, encapsulamento de informações, ou seja, informações que antes estavam disponíveis para a classe, agora podem estar escondidas. A Figura 9 mostra um exemplo de segmento desalinhado. A figura mostra um exemplo de movimentação do método (*Move Method*). Esse exemplo foi extraído do *Tomcat*, no alinhamento entre as versões 6 e 8 para a funcionalidade *Start Tomcat by Bootstrap*. A fase de análise estática classifica esses desalinhamentos em quatro categorias:

<i>Shifted Sequence #2</i>
. org.apache.jk.server.JKMain.start
<i>Shifted Sequence #1</i>
. org.apache.catalina.deploy.NamingResourcesImpl.start

Figura 9 – Exemplo de movimentação de método (*Move Method*).

- ***Move Method***: Segmentos desalinhados podem revelar que um método mudou de uma classe para outra, no mesmo pacote ou em pacotes diferentes, o que é confirmado pela fase de análise estática. A Figura 9 mostra um exemplo de segmento desalinhado que representa a mudança estrutural de movimentação de método. O segmento desalinhado mostra que o método *start* mudou da classe **.jk.server.JKMain* para a classe **.catalina.deploy.NamingResourcesImpl*. O segmento *Shifted Segment #2* mostra que o método *JKMain.start*, é executado na versão 6 do *Tomcat*. Por outro lado, o segmento *Shifted Sequence #1* mostra que o método *NamingResourcesImpl.start* é executado na versão 8 do *Tomcat*. Para confirmar essa mudança da análise dinâmica, as classes *JKMain* e *NamingResourcesImpl* são verificadas na fase de análise estática para ambas as versões do *Tomcat* para verificar a implementação do método *start*. Se o método for implementado apenas na classe *JKMain* da versão 6 do *Tomcat* e for implementado na classe *NamingResourcesImpl* apenas na versão 8 do *Tomcat*, então o segmento desalinhado é classificado como *Move Method*.
- ***Add Method***: Segmentos desalinhados podem revelar a implementação de novos métodos. Métodos que não foram executados na versão anterior do sistema geraram desalinhamento. A Figura 10 mostra um segmento desalinhado. A figura mostra um exemplo de adição de método (*Add Method*) para o *jFreeChart*, no alinhamento entre as versões 0.7.0 e 0.9.5 para a funcionalidade *Time Series Chart - Time Series 2*. Esse segmento foi classificado como *Add Method*, em que o método *translateValueToJava2D* foi adicionado na classe *VerticalNumberAxis* para a nova versão do *jFreeChart 1.0.16*. O código fonte é conferido na fase de análise estática para confirmar a adição do método, veri-

ficando se ele não existia na versão anterior do sistema, ou seja, no *jFreeChart* 0.7.0.

Aligned Sequence
Shifted Sequence #1
. com.jrefinery.chart.axis.VerticalNumberAxis.translateValueToJava2D

Figura 10 – Exemplo de adição de método, *Add Method*. *jFreeChart* - alinhamento entre as versões 0.7.0 e 0.9.5 para *Time Series Charts-Time Series 2*.

- **Remove Method:** De forma análoga, segmentos desalinhados podem indicar a remoção de um método do código fonte. O segmento desalinhado representado na Figura 11 mostra um exemplo de remoção do método *fileSaved* da classe *SaveAsAction* no *JHotDraw* 7.3. Até métodos removidos das superclasses são detectados pela fase de análise estática. Por exemplo, o método *fileSaved*, na versão anterior, quando existia, era implementado na superclasse *SaveAction* da classe *SaveAsAction*, ou seja, método foi removido da classe *SaveAction* na versão 7.3 do *JHotDraw*.

Aligned Sequence
Shifted Sequence #2
. org.jhotdraw.app.action.SaveAsAction.fileSaved

Figura 11 – Exemplo de remoção de método (*Remove Method*). *JHotDraw* - alinhamento entre as versões 7.2 e 7.3 para a funcionalidade *Save As*.

- **Access Modifier Change:** Finalmente, segmentos desalinhados podem revelar que a visibilidade do método foi alterada. A Figura 12 mostra que o modificador de acesso do método *createStandardTickUnits* mudou de *private* para *public*. A figura mostra uma mudança entre as versões 0.7.0 e 0.9.5, do *jFreeChart*, para a funcionalidade *Test Charts - Single Series Bar Chart*. Já, o desalinhamento da Figura 13 mostra que o modificador de acesso do método *autoAdjustRange* mudou de *public* para *privado*. Finalmente, as Figuras 12 e 13 mostram que as classes se moveram do pacote *com.jrefinery.chart* para o pacote *com.jrefinery.chart.axis*. Portanto, o padrão *Access Modifier Change* pode ter intersecção com o padrão de mudança *Move Class*. As Figuras 12 e 13 também mostram uma mudança de modificador de acesso (*Access Modifier Change*) para o *jFreeChart* no alinhamento entre as versões 0.7.0 e 0.9.5.

Um segmento desalinhado não pode ser classificado como *Move Method* e *Remove Method* nem como *Move Method* e *Add Method*. Isso acontece, porque o segmento desalinhado só é classificado como *Remove Method* se ele não existe mais na nova versão

<i>Shifted Sequence #2</i>
. com.jrefinery.chart.NumberAxis. private .createStandardTickUnits
<i>Shifted Sequence #1</i>
. com.jrefinery.chart. axis .NumberAxis. public .createStandardTickUnits

Figura 12 – Mudança de modificador de acesso (*Access Modifier Change*).

<i>Shifted Sequence #2</i>
. com.jrefinery.chart.VerticalNumberAxis. public .autoAdjustRange
<i>Shifted Sequence #1</i>
. com.jrefinery.chart. axis .VerticalNumberAxis. private .autoAdjustRange

Figura 13 – Mudança de modificador de acesso. *jFreeChart* - alinhamento entre as versões 0.7.0 e 0.9.5 para a funcionalidade *Combined Charts - Combined and Overlaid Dynamic Chart*.

do sistema. Analogamente, um segmento desalinhado só é classificado como *Add Method* se a versão anterior do sistema não contém o método. Por exemplo: um alinhamento ou desalinhamento de rastros é composto sempre por um par de segmentos de rastros, um para cada versão do sistema. Se o segmento de rastro de execução, ao ser alinhado, identifica que o método aparece somente na nova versão, então isso significa que trata-se de uma adição de método. Para que fosse classificado como movimentação, o método deveria aparecer nos dois segmentos, mas, neste exemplo, ele só aparece no segmento correspondente à nova versão do sistema. Portanto, o segmento não pode ser classificado ao mesmo tempo como movimentação e adição nem como movimentação e remoção.

3.3 Considerações Finais

O objetivo da proposta é armazenar, categorizar os segmentos desalinhados e permitir futuras consultas e a análise dos resultados sob diferentes perspectivas, tais como:

- ☐ Quais são os padrões de mudanças existentes entre duas versões do sistema?
- ☐ Quais são os padrões de mudanças mais frequentes?
- ☐ Quantas e quais funcionalidades são afetadas por mudanças em uma classe?
- ☐ Quais padrões de mudança são aplicados em uma classe e quais funcionalidades são impactadas por cada padrão de mudança?

Há métodos que não aparecem nos dois segmentos desalinhados. Eles podem aparecer em apenas um dos segmentos, como métodos que foram adicionados ou removidos. Esses métodos podem não estar relacionados com outros do mesmo segmento por nome, pacote ou classe, mas eles ainda possuem significado semântico, uma vez que aparecem no mesmo

segmento para o mesmo cenário do sistema. Isso é muito bom, pois ajuda a identificar, no código, mudanças em métodos e classes para um determinado cenário de execução que dificilmente seriam identificadas estruturalmente. Então isso ajuda a encontrar mudanças estruturais que eventualmente ocorrem juntas. Identificar uma relação entre tais mudanças estruturalmente ou pelo código fonte é difícil pois requer o uso de técnicas de *Feature Location*. Mas como neste trabalho as mudanças são agrupadas por cenário, é possível identificar tais relações, o que enriquece ainda mais o resultado para o desenvolvedor. Pois ele poderá perceber mudanças que ocorreram juntas para determinados cenários e assim fazer mais um tipo de previsões, planejamento e testes para futuras mudanças.

Avaliação da abordagem

Nesta seção é apresentado o estudo de avaliação da proposta, mostrando os experimentos realizados nos sistemas *jFreeChart*, *Tomcat* e *JHotDraw*. São apresentados, também, os parâmetros usados, as etapas seguidas para coleta e preparação dos rastros de execução, separação por funcionalidades, o alinhamento, a análise estática e extração das mudanças estruturais.

4.1 Perguntas de pesquisa

O objetivo desta pesquisa é extrair e categorizar, em padrões, as mudanças estruturais ocorridas no código fonte, que levam em consideração a comparação entre duas versões do sistema. Outra investigação também é feita no impacto das mudanças nas funcionalidades do sistema; mais especificamente, este trabalho busca avaliar quantas funcionalidades são afetadas por determinados padrões de mudanças estruturais. A parte do código que afeta muitas funcionalidades pode indicar os seguintes tipos de classes: classes que precisam ser tratadas de forma mais cuidadosa pois ao alterá-las, elas podem propagar modificações pelo código fonte; classes genéricas herdadas por outras classes espalhadas em funcionalidades do sistema; ou classes com alto acoplamento de conceitos. Por fim, investiga-se a relação entre pacotes do sistema sendo avaliada em termos de precisão e revocação. Para atingir esses objetivos, este estudo de caso busca responder às seguintes perguntas de pesquisa:

- ❑ **RQ #1** Há algum padrão de mudança mais frequente?
- ❑ **RQ #2** As funcionalidades são impactadas pelas mudanças estruturais detectadas?
- ❑ **RQ #3** Quão efetivo é o uso de alinhamento de rastros para detectar mudanças estruturais em pacotes?

Com a primeira pergunta de pesquisa, investiga-se se o sistema normalmente possui mudanças que se encaixam nos padrões de mudanças propostos e quais padrões dominam

as mudanças detectadas. Avaliar quais padrões ocorrem mais frequentemente pode orientar o time de desenvolvimento na tarefa de compreensão do software e entender melhor o caminho que tomam as mudanças. Por exemplo, se os padrões *Move Method* ou *Move Class* dominam as mudanças, o software pode estar passando por uma fase de refatoração ou reestruturação. Além disso, se o padrão *Access Modifier Change* ocorre com mais frequência, as classes podem estar passando por uma modificação de interface, como encapsulamento. Com a segunda pergunta de pesquisa, avalia-se o impacto das mudanças estruturais nas funcionalidades do sistema. Algumas classes podem representar um alto acoplamento, podem ser classes genéricas ou podem espalhar mudanças no código fonte quando forem alteradas. Listar essas informações ao desenvolvedor é importante para ajudá-lo a entender a relevância da classe quando ele a modificar. Por fim, com a terceira pergunta de pesquisa, verifica-se se o quanto as alterações classificadas nos padrões de mudança foram, de fato, efetuadas no código fonte (precisão) e quantas mudanças realmente efetuadas foram categorizadas (revocação).

4.2 Experimentos

Foi realizado um experimento com três sistemas e seis pares de versões para avaliar a metodologia. A abordagem foi avaliada com três sistemas de código aberto: *Tomcat*, *jFreeChart* e *JHotDraw*. Considerando uma funcionalidade específica do sistema, os respectivos rastros de execução foram coletados de duas versões diferentes do sistema. Os experimentos foram realizados em um *notebook* com um processador *Intel i5* de 2.5GHZ e 8 GB de memória RAM, no sistema operacional Windows 8 - 64 bits. Os sistemas usados no experimento foram o *jFreeChart* com dois pares de versões, *Tomcat* com três pares de versões e *JHotDraw* com um par de versões.

4.2.1 Sistemas analisados

O experimento foi feito com sistemas diferentes com o objetivo de evitar viés de validação por uso em algum sistema específico. O *Tomcat* é um servidor *Web* e é executado, geralmente, como um *daemon* em segundo plano no sistema operacional. O *jFreeChart* é uma biblioteca para gerar gráficos e não possui uma interface própria, que geralmente é importada em outros projetos pelos desenvolvedores para gerarem gráficos em suas aplicações próprias. Porém, embora o *jFreeChart* não possua uma interface gráfica, ele possui códigos para demonstração dos gráficos que consegue gerar. Já o *JHotDraw* é um sistema com uma interface gráfica e é usado como biblioteca para desenho. Esses desenhos baseiam-se em formas pré-definidas, como retângulos, linhas, entre outros. A Tabela 2 mostra os sistemas considerados no estudo, incluindo informações básicas sobre eles, como o número de linhas de código (LOC), número de pacotes (NOP) e o número de métodos (NOM).

Tabela 2 – Sistemas avaliados (Métricas de tamanho).

Sistema	Versão	Release	LOC	NOP	NOM
JFreeChart	0.7.0	0.7.0	11,076	8	789
	0.9.5	0.9.5	44,453	24	3,048
	1.0.15	1.0.15	142,195	95	10,711
	1.0.16	1.0.16	134,361	69	10,362
Tomcat	6	6.0.37	173,102	196	15,123
	7	7.0.47	233,238	265	19,287
	8	8.0.1	249,212	268	20,901
JHotDraw	7.2	7.2	72,050	68	6,348
	7.3	7.3	73,414	68	6,494

A Tabela 3 mostra os experimentos realizados no estudo. Cada par de experimento é realizado com dois lançamentos dos sistemas, por exemplo, no par de versões de código de identificação 2, foi avaliado o *jFreeChart* para detectar mudanças da versão *1.0.15* para a versão *1.0.16*.

Tabela 3 – Pares de sistemas avaliados nos experimentos para aplicar o alinhamento.

Identificação	Sistema	Versão 1	Versão 2
1	JFreeChart	0.7.0	0.9.5
2		1.0.15	1.0.16
10	Tomcat	6	7
11		7	8
12		6	8
13	JHotDraw	7.2	7.3

4.2.2 Fase 1 - Extração e Preprocessamento dos Rastros

Para coletar os rastros do sistema é importante definir os cenários de execução, que podem, por exemplo, serem definidos como *test cases*, aproveitando a etapa de testes do sistema antes do lançamento, para a coleta de rastros. Com o objetivo de ampliar a investigação para cada um dos três sistemas foi usado um tipo de cenário diferente para cada sistema. Após a preparação dos cenários, foi feita a coleta dos rastros, que, em seguida são processados para serem alinhados na análise dinâmica. Depois disso, foi feita a análise estática e a classificação das mudanças dentro dos padrões propostos.

4.2.2.1 Definição de Cenários

A definição de cenários consiste em preparar os sistemas e os dados para coletar os rastros de execução. No caso do *Tomcat*, foram usados cenários baseados na aplicação *Tomcat Manager* que é bem documentado e é distribuído no *Tomcat*. Assim foram usados os cenários: (i) Acessar o Manager, (ii) Fazer deploy de aplicacao, (iii) Inicializar tomcat Bootstrap, (iv) Listar aplicacoes e (v) Verificar status do servidor, totalizando cinco

cenários para o Tomcat. Quando há a necessidade de dados de entrada externos, foram usados os mesmos dados para todos os pares de versões avaliados. Para um dos cenários, por exemplo, foi preciso fazer depuração de uma aplicação *Web*. Logo, a mesma aplicação *Web* foi usada para fazer *deploy* em ambos os cenários. Para o *jFreeChart* foram usados *test cases*, sendo um para cada gráfico gerado pela biblioteca, totalizando 35 cenários para o primeiro par de versões do *jFreeChart*, versões 0.7.0 e 0.9.5 e 301 cenários para o segundo par de versões do *jFreeChart* avaliado, versões 1.0.15 e 1.0.16. As classes de demonstração que implementam os casos de teste já possuem os dados de entrada para a geração dos gráficos. Por fim, no *JHotDraw*, foram usados 64 cenários, sendo que cada cenário foi definido como sendo uma opção de desenho disponível no sistema, onde todos os desenhos disponíveis foram usados. Como o desenho é feito pelo usuário com os periféricos, *mouse* e teclado, a entrada de informações destes dispositivos são os dados externos necessários para executar as tarefas. Após a definição dos cenários, passou-se para a coleta dos rastros de execução. A Tabela 4 mostra o número de cenários usados em cada sistema. No par de versões entre o *jFreeChart* 1.0.15 e 1.0.16 há muito mais gráficos disponíveis para desenhar do que no par de versões anterior *jFreeChart* 0.7.0 e 0.9.5. O *Tomcat* tem um número de cenários significantemente menor, visto que ele é um *daemon* e, por isso, possui menos cenários, embora cada cenário requeira mais etapas para executar usando-se o *Tomcat Manager*. Por outro lado, o *jFreeChart* possui um número bem maior de cenários, um para cada gráfico disponível.¹

Tabela 4 – Número de cenários usados por cada sistema.

Sistema	Versão #1	Versão #2	# Cenários
jFreeChart	0.7.0	0.9.5	35
	1.0.15	1.0.16	301
Tomcat	6	7	5
	7	8	5
	6	8	5
JHotDraw	7.2	7.3	64

4.2.2.2 Seleção dos parâmetros

Para esta avaliação foi usado o parâmetro de granularidade um. Uma das etapas de redução dos rastros busca podar métodos que não são muito relevantes para aquela funcionalidade. Os métodos mais irrelevantes são os que estão mais próximos das folhas na árvore de execução, como métodos privados e métodos *getters* e *setters*. Assim métodos com granularidade igual a um foram descartados.

¹ Como o número de cenários é grande, para o *jFreeChart* e *JHotDraw*, eles não foram especificados um a um. Mas estão disponíveis em <http://www.janio.silva.nom.br>

4.2.3 Fase 2 - Detecção de Mudanças Estruturais

Após a coleta e preparação dos rastros, foi feito o alinhamento das versões, fazendo uso da análise dinâmica para detectar mudanças. Em seguida a análise estática foi realizada para complementar a análise dinâmica. Um último filtro foi feito e as mudanças foram classificadas em padrões de mudanças.

4.2.3.1 Alinhamento de Rastros por Cenário

No primeiro passo, o par das versões de cada sistema é alinhado considerando os seus respectivos cenários definidos na Fase I. Assim, os rastros processados organizados por cenário—das duas versões—são usados como entrada para o algoritmo de alinhamento de sequências. Os segmentos alinhados indicam que não houve mudança estrutural no sistema, ou seja, o mesmo método foi executado para ambas as versões do sistema. Pode ter acontecido alguma mudança no código fonte internamente, dentro do método, mas essa mudança não é objeto de estudo nesta abordagem; consequentemente, esses segmentos são descartados. Já os segmentos desalinhados podem representar mudança estrutural no código fonte. Mesmo assim, ainda existem ruídos deixados pelas etapas anteriores. Para limpar esses ruídos e reforçar essa característica de mudança estrutural, uma análise estática é efetuada adiante. A Tabela 5 mostra o número de segmentos alinhados e desalinhados para cada par de versões. Por exemplo, o par de versões entre o *Tomcat 7 e 8* gerou 484 segmentos desalinhados e 308 segmentos alinhados. Durante a etapa de alinhamento, os segmentos desalinhados foram armazenados em um banco de dados relacional.

Tabela 5 – Número de segmentos alinhados e desalinhados para cada par de versões do sistema.

Sistema	Versão #1	Versão #2	# Alinhados	# Desalinhados
JFreeChart	0.7.0	0.9.5	243	392
	1.0.15	1.0.16	5,485	6,368
Tomcat	6	7	690	975
	7	8	308	484
	6	8	628	908
JHotDraw	7.2	7.3	83,391	153,079

Armazenamento no Banco de Dados. Foi criado um banco de dados relacional para armazenar as informações do experimento, o que facilita as etapas posteriores, além de permitir ao desenvolvedor eventuais consultas complementares. Durante a etapa de alinhamento dos cenários é quando ocorre a gravação das informações. Assim pode-se obter as informações cruas, antes de se passar pelo filtro da análise estática se o desenvolvedor desejar fazer uma consulta mais detalhada. Além disso, as informações obtidas após a análise estática também são armazenadas.

4.2.3.2 Extração de Mudanças Estruturais

Nesta etapa os métodos selecionados na etapa anterior foram verificados no código fonte através um *parser*. A informação da existência do método é armazenada para ter certeza se o método foi adicionado ou não. Essa informação é atualizada no banco de dados. Durante esta etapa é feita a categorização das mudanças nos padrões propostos na abordagem.

Filtro de informações. As informações são filtradas pelas cinco categorias classificadas durante a análise estática (inspeção do código) e dinâmica (alinhamento dos rastros). Alguns métodos aparecem nos segmentos desalinhados mas não são classificados nas cinco categorias propostas foram descartados. Após a seleção das mudanças, uma listagem de mudanças categorizadas é apresentada para o usuário.

4.3 Resultados

Nesta seção são apresentados os resultados obtidos com a aplicação da abordagem, respondendo as perguntas de pesquisas propostas.

4.3.1 RQ1 - Há algum padrão de mudança mais frequente?

Para responder esta RQ, os segmentos desalinhados são categorizados como *Move Class*, *Move Method*, *Add Method*, *Remove Method* e *Access Modifier Change*. Os resultados são descritos na Tabela 6. Em resumo, 324 classes foram classificadas como *Move Class*. Como pode ser visto, esse padrão está presente em todos os experimentos. Por exemplo, no *jFreeChart* foram identificadas 48 classes que moveram da versão *0.7.0* para a versão *0.9.5* e houve mais mudanças de classes na comparação entre as versões *1.0.15* e *1.0.16*. Foram detectados também padrões que podem mudar a interface da classe: *Move Method*, *Add Method*, *Remove Method* e *Access Modifier Change*.

Tabela 6 – Número de mudanças classificadas como *Move Class* de uma versão para outra.

Sistema	Versões	# Moved Classes	# de Classes
jFreeChart	0.7.0 e 0.9.5	48	221 e 441
	1.0.15 e 1.0.16	81	1.011 e 1.018
Tomcat	6 e 7	59	1.495 e 2.171
	7 e 8	51	2.171 e 2.334
	6 e 8	85	1.495 e 2.334
JHotDraw	7.2 e 7.3	41	940 e 982

A Figura 14 mostra a porcentagem de padrões de mudanças estruturais detectados entre duas versões dos sistemas (872 classes para todos os experimentos). Como pode

ser observado, os quatro padrões foram identificados em todos os experimentos, com exceção do experimento entre o *jFreeChart 1.0.15* e *1.0.16* no qual não foi detectado o padrão *Remove Method*. O primeiro gráfico mostra a relação de cobertura dos padrões de mudança no nível de classe entre as duas versões, como quantas classes distintas foram afetadas por cada padrão. Pode ser observado que o padrão *Access Modifier Change* afetou poucas classes quando comparado aos outros padrões (em torno de 13% em média), com exceção do *jFreeChart 1.0.15/1.0.16* (24%). Além disso, atividades de manutenção que requerem a implementação de novos métodos, o padrão *Add Method*, estão presentes em todos os experimentos (17% em média). Por outro lado, atividades relacionadas com o padrão *Move Method* foram muito comuns em praticamente todos os experimentos, na média, em 28.4% dos casos.

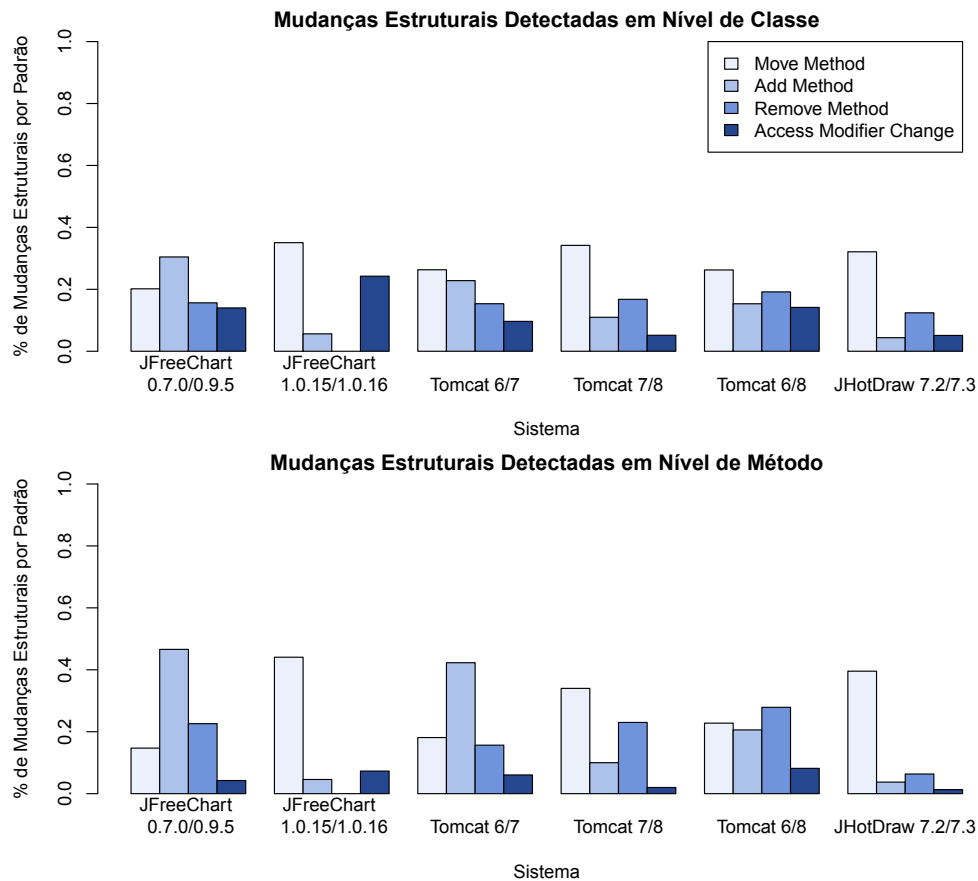


Figura 14 – Cobertura relativa de mudanças de interface de classe entre duas versões do sistema.

O segundo gráfico mostra a cobertura relativa dos padrões de mudança no nível de método, como quantos métodos distintos foram afetados por cada padrão. No experimento entre o *jFreeChart 0.7.0/0.9.5* e o *Tomcat 6/7*, houve uma porcentagem significativa de mudanças categorizadas como o padrão *Add Method*, 47% e 42% respectivamente. Já o resultado obtido para o experimento *jFreeChart 1.0.15/1.0.16* revela poucas adições de métodos para a última versão *1.0.16* (5%); como exemplo, as mudanças do padrão

Move Method representam 44% e 34% para os experimentos *jFreeChart 1.0.15/1.0.16* e *Tomcat 6/7*, respectivamente. Finalmente, o padrão *Move Method* afetou 32% das classes e 39% dos métodos no experimento feito para o *JHotDraw 7.2/7.3*. Algumas mudanças são classificadas em mais de um padrão. O padrão *Remove Method* é afetado, também, por renomeação de métodos, onde cada método renomeado gera uma inclusão (com o novo nome) e uma exclusão (do nome antigo). Movimentações de métodos para classes diferentes também geram uma remoção de método daquela classe. Isso aumenta a quantidade de mudanças no padrão *Remove Method*. Mas essas mudanças precisam ser evidenciadas para a ciência do desenvolvedor, sendo, portanto, importante classificá-las.

4.3.2 RQ2 - As funcionalidades são impactadas pelas mudanças estruturais detectadas?

Para responder a essa pergunta de pesquisa, focou-se nas mudanças estruturais categorizadas para avaliar o impacto das mudanças na última versão do sistema. Uma das contribuições da abordagem proposta é permitir avaliar o impacto das mudanças sem precisar usar recursos de *Feature Location* (Seção 2.1). Portanto, pela abordagem proposta, não é preciso aplicar técnicas para associar partes do código fonte com funcionalidades. Pois essa associação é feita justamente pela coleta dos rastros de execução por cenários. Assim, as sequências de rastros de execução que pertencem a um cenário, indicam quais classes fazem parte daquela funcionalidade, associando o código fonte com os cenários. O objetivo é investigar quantos cenários (funcionalidades do sistema) são impactados quando as mudanças são realizadas nas classes, ou seja, objetiva-se avaliar o impacto das mudanças das classes nas funcionalidades. Para tanto, foram consideradas apenas as mudanças detectadas e classificadas nos cinco padrões propostos: o padrão *Move Class* mais os quatro padrões associados com as mudanças de interface da classe. Pois algumas mudanças são descartadas e não são classificadas em nenhum padrão. As classes selecionadas foram aquelas que afetaram duas ou mais funcionalidades, uma vez que esta investigação busca avaliar o impacto da mudança em muitas funcionalidades. Assim, mudanças que não foram categorizadas nos cinco padrões propostos, foram descartadas. Além disso, as classes alteradas detectadas mas que afetaram somente uma funcionalidade do sistema foram também descartadas. A Figura 15 mostra o número de classes modificadas (com mudanças estruturais) em relação às funcionalidades afetadas para todos os experimentos de alinhamento realizados neste estudo. Um ponto no gráfico indica que Y funcionalidades foram afetadas por X classes (onde o eixo X contém as classes e o eixo Y contém as funcionalidades). Como pode ser visto, há muitas classes usadas por muitas funcionalidades. Por exemplo, no experimento *jFreeChart 1.0.15/1.0.16* 248 de 301 funcionalidades (82%) foram impactadas por uma única classe. No *JHotDraw 7.2/7.3*, uma única classe afetou 51 das 64 funcionalidades (80%). Assim ao olhar para o passado e examinar as

mudanças estruturais que ocorreram no sistema, é possível saber, com antecedência, para futuras mudanças, quais as classes poderão impactar muitas funcionalidades. Isso é uma informação relevante para o desenvolvedor, pois ele poderá preparar casos de testes para classes específicas que poderão ser impactadas, bem como casos de testes para outras funcionalidades, além daquela funcionalidade que será modificada. Portanto se houver um requisito solicitando alterações em uma funcionalidade específica, o desenvolvedor saberá com antecedência informações relevantes, como: quais são as outras funcionalidades que podem ser impactadas e quais são as classes impactadas nessas funcionalidades e nas outras.

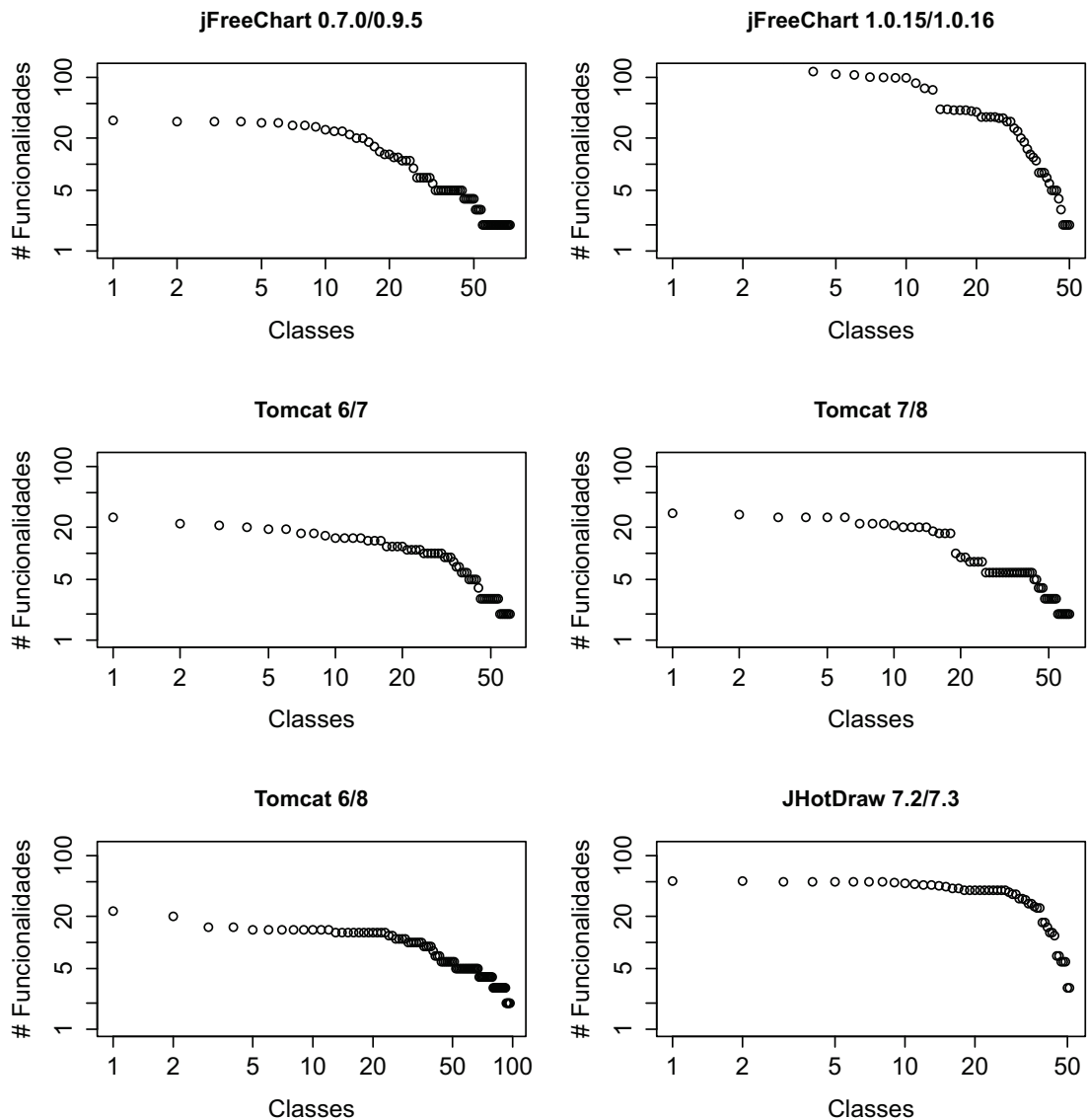


Figura 15 – Número de funcionalidades afetadas por mudanças estruturais em classes.

A Figura 16 mostra o nome das classes por meio de um *dotplot* para os experimentos *jFreeChart 1.0.15/1.0.16*, *Tomcat 7/8* e *JHotDraw 7.2/7.3*. Padrões diferentes foram detectados para a classe *NumberAxis*: *Add Method*, *Move Method*, *Move Class* e *Access*

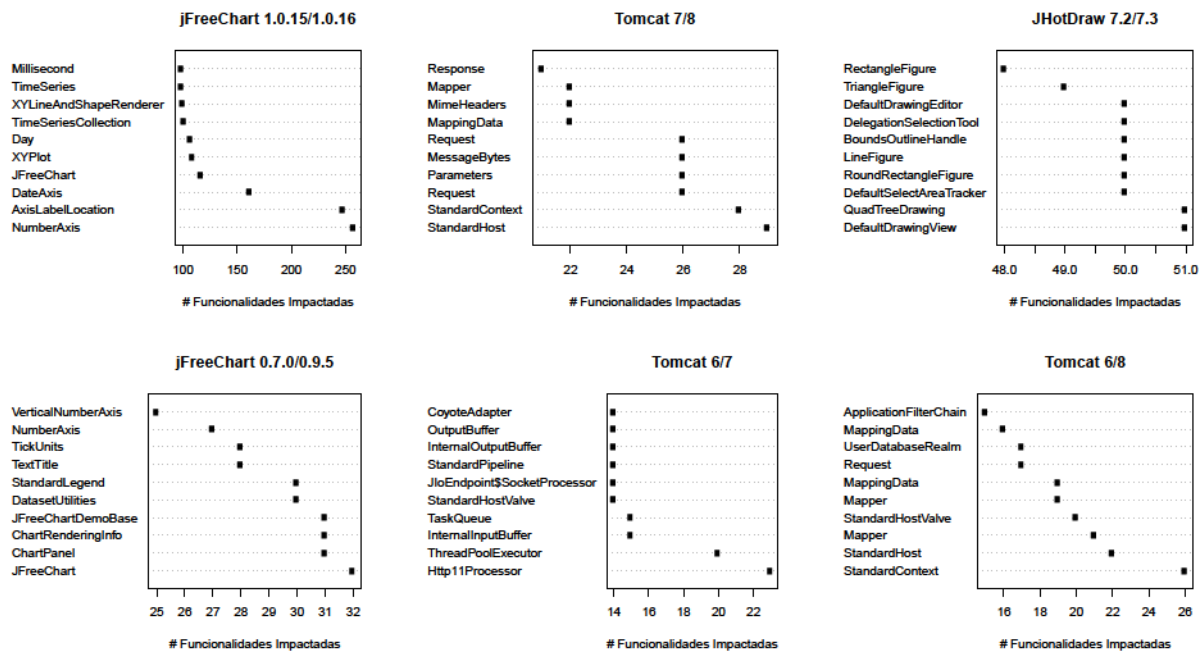


Figura 16 – As 10 classes alteradas que mais impactam funcionalidades.

Modifier Change. Essas mudanças afetaram 258 funcionalidades no experimento entre *jFreeChart 1.0.15/1.0.16*. De forma similar, por um lado, no experimento entre *Tomcat 7/8* foram detectados todos os cinco padrões de mudança para a classe *StandardHost* e essas mudanças impactaram 29 funcionalidades. Por outro lado, as mudanças aplicadas na classe *DefaultDrawingView* foram apenas do padrão *Add Method* porém isso afetou 51 funcionalidades, para o experimento *JHotDraw 7.2/7.3*². Pelo gráfico é possível observar que algumas classes podem afetar muitas funcionalidades no sistema, quando se avalia as mudanças estruturais do sistema no passado. Portanto essas classes precisam ser modificadas com mais cuidado. Assim, se o desenvolvedor receber um requisito que pede modificações em funcionalidades que podem ser impactadas por essas classes, a equipe poderá planejar com mais cautela essa mudança. Assim como definir mais testes que, sem essa avaliação de impacto, dificilmente seriam feitos.

4.3.3 RQ3 - Quão efetivo é o uso de alinhamento de rastros para detectar mudanças estruturais em pacotes?

Para responder esta RQ, é feita uma investigação nas mudanças em pacotes porque eles refletem o nível mais alto da estrutura do sistema e esse tipo de mudança pode representar mudanças na relação entre os componentes do sistema. Para tanto, a avaliação é conduzida em termos de precisão e revocação das mudanças. A precisão indica se a

² A quantidade de mudanças impactadas para todas as classes pode ser obtida em <http://www.janio.silva.nom.br>

Tabela 7 – Pacotes adicionados e removidos entre *jFreeChart 0.7.0/0.9.5*.

Mudanças	Nome do Pacote
Adicionadas	com.jrefinery.data
	com.jrefinery.chart.plot
	com.jrefinery.chart.renderer
	com.jrefinery.chart.demo
	com.jrefinery.chart.axis
	com.jrefinery.chart.tooltips
Removidas	com.jrefinery.chart.entity
	com.jrefinery.chart.data

mudança detectada pela abordagem foi realmente feita no código fonte; então, se a precisão é máxima, isso significa que todas as mudanças encontradas foram de fato feitas no código fonte. A revocação indica quantas das mudanças realmente realizadas no sistema foram detectadas pela abordagem. A revocação máxima, neste caso, indica que todas as mudanças realizadas pelos desenvolvedores no sistema foram detectadas. Para avaliar a revocação, portanto, é preciso saber de todas as mudanças realizadas no sistema, para poder compará-las com aquelas encontradas na abordagem. Se não houver documentação disponível mostrando as mudanças, avaliar a revocação torna-se muito difícil. Devido a essa dificuldade de avaliação da revocação, essa investigação não foi realizada para todos os experimentos, pois muitos deles não possuem documentação disponível que mostre as mudanças realizadas nos pacotes do sistema. Mesmo assim, foi possível avaliar a revocação do *jFreeChart* nos experimentos entre as versões *0.7.0* e *0.9.5*. Neste experimento, foram detectadas a adição de muitos pacotes e a remoção do seguinte pacote: *com.jrefinery.chart.data*. A Tabela 7 mostra os pacotes que foram adicionados e removidos para a versão *0.9.5*, a partir da versão *0.7.0*. Entre essas duas versões, os pacotes foram adicionados pelo time de desenvolvimento e como o pacote *com.jrefinery.chart.data* foi removido e o pacote *com.jrefinery.data* foi adicionado, eles parecem ser o mesmo, quando se observa as classes que ambos contêm, além do nome. Isso pode indicar que o pacote moveu de lugar. Após realizar os experimentos, observou-se que o pacote, de fato, foi movido, pois nos mesmos cenários onde antes aparecia o pacote *com.jrefinery.chart.data*, agora aparece o pacote *com.jrefinery.data*. Assim, pode-se dizer que o pacote é o mesmo por causa da sua alta similaridade, conforme indicado nos experimentos pela quantidade de cenários.

Para avaliar a precisão, as alterações foram manualmente verificadas no código fonte para os dois sistemas. Foi investigado se os pacotes realmente foram adicionados, ao longo das alterações que aconteceram no sistema, da versão *0.7.0* até chegar na versão *0.9.5*. Foram detectadas seis adições de pacotes com a abordagem e todas as seis adições também foram detectadas pela investigação manual. A remoção do pacote também foi confirmada pela investigação manual também confirma sua remoção. Assim, para calcular a precisão,

das sete mudanças detectadas pela abordagem, sete mudanças foram confirmadas manualmente, uma a uma, o que dá uma precisão de 100%. A investigação indica, portanto, que todas as mudanças detectadas pela abordagem foram realmente realizadas.

Para avaliar a revocação foi conduzida uma investigação minuciosa na documentação³. Há uma documentação consistente do desenvolvedor para as mudanças realizadas entre as *versões 0.6.0 e 1.0.13*, a última versão que possui documentação disponível. Dessas versões que possuem documentação disponível, foi possível selecionar com segurança as modificações realizadas para o experimento *jFreeChart 0.7.0/0.9.5*, uma vez que o documento possui ainda modificações anteriores à versão *0.7.0* (desde a *0.6.0*) e as modificações feitas posteriores à última versão realizada no experimento, que é a *0.9.5* (há documentação até a versão *1.0.13*). Assim, foram selecionadas, na documentação, as mudanças realizadas entre as versões do sistema: *0.7.0* e *0.9.5*. Foram encontradas seis modificações: cinco adições de pacotes e uma remoção de pacote. O pacote removido foi justamente o *com.jrefinery.chart.data*, que é documentado como remoção, seguido da adição do pacote *com.jrefinery.data*. Isso confirma a mudança detectada pelo experimento nesses dois pacotes. Os outros pacotes adicionados foram *axis*, *plot*, *renderer* e *combination*. Quase todas as adições foram detectadas, exceto uma: o pacote *combination*. Assim, quatro adições foram detectadas e uma remoção de pacote, o que dá um total de cinco mudanças detectadas de um total de seis mudanças documentadas, gerando uma revocação de 83%.

Isso porque o pacote *com.jrefinery.chart.combination* não foi detectado pela abordagem. Pela experiência empírica dos vários experimentos realizados era esperada uma revocação próxima do total, pois todos os cenários disponíveis foram executados dando uma cobertura completa para os experimentos. Para confirmação desta hipótese, o código fonte foi inspecionado e as modificações realizadas para tentar detectar a razão da abordagem não ter indicado o pacote *com.jrefinery.chart.combination* nas mudanças. Todas as versões foram checadas começando da mais nova, *0.9.5* até a mais antiga *0.7.0*. O pacote *com.jrefinery.chart.combination* não estava disponível na última versão. Isso já deixa a abordagem mais consistente, pois seria mais ameaçador não detectar um pacote que estivesse presente na última versão. No entanto, a documentação intermediária, entre as versões do experimento, indicava a adição do pacote. A investigação continuou para as versões anteriores e detectou-se que o pacote em questão foi realmente adicionado na *versão 0.7.1* e permaneceu até a versão *0.8.1*, quando foi removido. Assim, embora a adição do pacote tenha ocorrido, sua remoção também ocorreu, e essa é a razão pela qual a abordagem não o detectou, uma vez que ele não está presente no código fonte da *versão 0.9.5* e nem no código fonte da primeira *versão 0.7.0* do experimento. Assim, não haveria possibilidade de o pacote ser coletado pelos rastros de execução.

³ <https://github.com/apetresc/JFreeChart>

4.3.4 Discussão

Uma das vantagens do trabalho proposto é permitir ao desenvolvedor, ao olhar para o passado, prever o impacto mudanças, que fará no futuro, nas funcionalidades do sistema. Podendo, assim, se preparar melhor para tais mudanças, como preparar casos de uso para as funcionalidades afetadas. Essa previsão de impacto em funcionalidades não é possível de ser feita com os trabalhos relacionados com este. Além disso, o desenvolvedor pode avaliar mudanças no comportamento estrutural do sistema se prevenindo contra a inserção de erros no sistema (algumas mudanças estruturais podem provocar falhas). Finalmente, a categorização das mudanças permite ao desenvolvedor compreender melhor as modificações feitas no sistema, diminuindo o tempo gasto com: rastreamento do código, leitura da documentação, exigência de conhecimento prévio do sistema. Pois o resultado contém uma informação objetiva acerca das mudanças estruturais ocorridas no passado. Esse acompanhamento das mudanças fica armazenado, de forma que, a cada lançamento de uma nova versão, a equipe de desenvolvimento pode fazer novas comparações. Os segmentos alinhados, dos rastros de execução, representam partes do código em que não houve mudança estrutural, pois o código permaneceu estável e considerou-se o mesmo cenário nas duas versões (o código alinhado pode conter mudanças internas mas não mudanças relevantes para este estudo). Já os segmentos desalinhados indicam mudanças estruturais no código fonte. Além disso, o alinhamento de rastros baseado em funcionalidade possui confiança no apontamento de mudanças, pois espera-se que a mesma parte do código fonte seja executada. Os segmentos desalinhados foram analisados e foi observado que muitos deles representam mudanças estruturais, como mudanças que impactam a relação entre os pacotes, tornando a abordagem relevante para ajudar em um possível estudo de mudanças arquiteturais do sistema, já que a arquitetura do sistema engloba o relacionamento entre seus artefatos.

Os desenvolvedores podem usar a abordagem proposta nesta pesquisa para estudar a mudança de relação estrutural entre as duas versões, sob uma perspectiva de funcionalidades. Particularmente para análise de impacto, a abordagem pode detectar quais funcionalidades são afetadas quando mudanças alteram a estrutura de classes e pacotes do sistema. As informações providas pela abordagem podem auxiliar o desenvolvedor identificar testes de unidade associados às funcionalidades que foram impactadas pela mudança. Por exemplo, se uma mudança que afeta a classe *HostConfig* da funcionalidade Acessar o Manager também afeta a funcionalidade Fazer Deploy de Aplicação e outras classes, da próxima vez que o desenvolvedor for planejar uma mudança na funcionalidade Acessar o Manager, que tenha chances de afetar a classe *HostConfig*, ele irá preparar casos de teste para as mesmas classes afetadas anteriormente, bem como também para a funcionalidade Fazer Deploy de Aplicação.

No caso do *jFreeChart 0.7.0* e *0.9.5* foram detectadas mudanças significativas nas relações entre os pacotes. Os resultados foram avaliados usando precisão e revocação

comparando os resultados e a documentação. Porém, no *jFreeChart 1.0.15* e no *jFreeChart 1.0.16*, a estrutura de pacotes demonstrou ser mais estável, não havendo mudanças profundas. Embora existissem mais mudanças, essas não implicaram em uma profunda mudança de relação entre os componentes estruturais do sistema conforme a categorização de mudanças deste estudo. Do ponto de vista de modularidade, é interessante notar que na versão 0.9.5 do *jFreeChart*, o pacote *com.jrefinery.chart* teve sua métrica de acoplamento de pacote, *Efferent Coupling*, medida em 30, um valor bem menor do que na versão anterior do par comparado, 0.7.0, onde esse valor era de 76. Foram coletadas métricas para o *jFreeChart 0.7.0* e 0.9.5 e o acoplamento de pacote foi comparado.⁴ Mesmo que a nova versão tenha mais linhas de código, 44.453 na versão 0.9.5 contra 11.076 na versão 0.7.0, a medida de acoplamento de pacote diminuiu, indicando uma melhor modularização na estrutura de pacotes para a nova versão. Portanto, as mudanças nos relacionamentos de pacotes, que foram detectadas pela abordagem, tiveram um efeito positivo do ponto de vista de modularidade no sistema.

Durante os experimentos houve um controle de qualidade dos rastros de execução. Quando se trabalha com rastros de execução deve-se tomar cuidado com o procedimento de coleta, as funcionalidades usadas e os dados que servirão de entrada para executar as tarefas, pois ruídos podem poluir os rastros, comprometendo sua qualidade.

Tomcat, *jFreeChart* e *JHotDraw* são implementados na linguagem de programação *Java*, usando o paradigma de programação orientada a objetos. Porém, como a abordagem usa rastros de execução, qualquer sistema feito em linguagem procedural ou outra linguagem que possua seus rastros de execução coletados, pode ser avaliado pela abordagem, desde que possua uma estrutura de pacotes bem definida.

⁴ Para coletar as métricas, foi usada a ferramenta CodePro AnalytiX, disponível em <http://developers.google.com/java-dev-tools/codepro/doc/>

4.3.5 Limitações

Algumas limitações relacionadas com rastros de execução podem ocorrer na abordagem. O custo de coletar e armazenar os rastros de execução, em geral, é alto. Porém, para armazenar os rastros de execução, foi usada uma técnica de compressão e sumarização que diminui o tamanho dos rastros de execução, permitindo que fossem armazenados sem um alto custo de espaço ocupado. Por exemplo, para o *JHotDraw 7.3*, os rastros tiveram uma redução de 97% de espaço (os rastros sem redução ocupavam *40,20MB* e após as fases de compressão e sumarização passaram a ocupar *1,2MB*). Portanto, o time de desenvolvimento é encorajado a armazenar os rastros dos lançamentos anteriores para sempre poderem comparar com lançamentos atuais e estudarem as mudanças sem um esforço humano e computacional muito grande. Portanto, a redução no tamanho das informações ajuda a diminuir esta limitação.

Outra limitação prática está relacionada com a dificuldade de coletar os rastros de execução, pois podem ser necessárias informações, dados e definição de cenários para execução do sistema. Quando o interesse é capturar mais mudanças ao invés de capturar mudanças localizadas por funcionalidades, pode ser necessário executar muitos cenários, de forma que tenham uma boa cobertura do sistema e executar a maior parte do código fonte. Para amenizar esta limitação pode-se coletar os rastros de execução usando *demos*, durante a fase de execução de casos de teste, especialistas e cenários definidos por desenvolvedores. Nos experimentos deste estudo, todos os *test cases* disponíveis e empacotados junto do código do *jFreeChart* foram executados, pois possuem documentação e vem com os dados necessários para a geração dos gráficos, sendo assim, mais confiáveis do que produzir cenários manualmente. Para o *Tomcat* foi usada a aplicação *Tomcat Manager*, distribuída no código fonte e documentada. No caso do *JHotDraw*, cada ferramenta de desenho foi usada como um cenário e estas opções de desenho requerem informações externas obtidas com os periféricos disponíveis no computador.

Uma limitação de localização de mudanças estruturais no código fonte, para essa abordagem, está relacionada com a implementação de novas funcionalidades, que obviamente, estão presentes somente na nova versão. Como a abordagem é orientada a cenários, a nova funcionalidade não possui correspondente na versão anterior. Esta limitação pode ser superada coletando os rastros das novas funcionalidades e, em seguida, alinhando-se estes rastros de execução com uma ou mais funcionalidades semelhantes da versão anterior. O alinhamento de rastros de execução é uma técnica capaz de comparar cenários similares, não apenas os mesmos. No trabalho realizado por Silva et al. (SILVA et al., 2011), os autores alinham funcionalidades diferentes, que possuem similaridades, para obter indício de onde programar quando se quer implementar novas funcionalidades no código fonte. Isso mostra que alinhar cenários semelhantes ainda reflete a localização do código fonte, e suas similaridades e diferenças. Portanto, se uma nova funcionalidade é criada, ainda é possível alinhá-la com uma semelhante na versão anterior e ainda detectar mudanças.

Trabalhos Relacionados

Neste capítulo discute-se trabalhos relacionados com a abordagem apresentada. A discussão é organizada em três sub-seções: Análise Estática de Código Fonte, Análise de Histórico de Versões (como *SVN*, *Git*) e Análise Dinâmica.

5.1 Análise Estática do Código Fonte

Xing e Stroulia propuseram uma técnica para recuperar um conjunto de fotografias do código fonte através de uma análise da evolução do sistema (XING; STROULIA, 2005). Foi proposta a criação da ferramenta UMLDiff para comparar, par a par, versões do controle de código fonte. O UMLDiff reconhece as mudanças de uma versão para outra de forma semi-automática, usando análise estática. O trabalho de Xing e Stroulia, também propõe que, os resultados do UMLDiff sejam analisados para detectar padrões de mudanças de uma única classe, um conjunto de classes e o sistema inteiro. Além disso, extraíram mudanças feitas no código fonte inteiro. Porém não há uma relação entre a classe avaliada e qual funcionalidade ela pertence. Assim, quando o desenvolvedor está avaliando as modificações de uma determinada classe, não é possível saber a qual funcionalidade esta classe pertence. O trabalho de Xing e Stroulia também avalia apenas uma mudanças isoladas em uma classe ou em um conjunto de classes mas sem vinculá-la a um cenário. Tomar um conjunto de classes e dizer que este conjunto pertence a um determinado cenário exigiria um trabalho de *Feature Location*, um trabalho difícil, complexo e ainda pesquisado. Assim isso não é feito neste trabalho citado. Similarmente à abordagem proposta nesta dissertação, o trabalho citado avaliada duas versões de um sistema, em par, comparando uma com a outra, para avaliar a evolução do sistema como um todo, usando análise estática. Por outro lado, a abordagem proposta neste trabalho, busca detectar mudanças localizadas baseando-se em funcionalidades específicas usando análise dinâmica, em conjunto com análise estática. Rastros de execução são coletados de um cenário específico, onde um *test case* pode ser usado, uma funcionalidade específica ou um cenário definido pelo desenvolvedor. Na abordagem do UMLDiff, para aplicar cenários

específicos, uma técnica de *Feature Location* seria necessária para indicar quais as partes do código fonte que estão ligadas a uma funcionalidade, enquanto que, na abordagem proposta, não é necessário aplicar tal técnica, uma vez que a coleta dos rastros de execução já garante isto: a localização de cenários.

Kobori, Matsushita e Inoue propuseram uma abordagem que usa análise estática para estudar os modificadores de acesso em atributos e métodos (KOBORI; MATSUSHITA; INOUE, 2015). Esses modificadores de acesso podem ser: privado, público, protegido ou padrão. O uso apropriado de modificadores de acesso é essencial para um bom encapsulamento de programas em Java (ECKEL, 2005; BOOCH et al., 2007). Foi estudado quando a acessibilidade dos atributos ou métodos é excessiva ou não. E foi definido um processo de análise chamado de *Accessibility Excessiveness*. Foi implementada a ferramenta *ModiChecker* que recebe como entrada um programa Java, analisa e dá como resultado a excessividade de cada modificador de acesso declarado em atributos e métodos. Por exemplo, quando um método público só é usado por outro método da mesma classe, foi considerado que este método público está com excessividade de acesso, porque ele deveria ser privado. Se ele só é usado pela mesma classe onde está declarado, há um mal uso do modificador de acesso. Pois o ideal seria que este método fosse privado. Então, foi definido uma métrica baseando-se nessa diferença, entre a declaração e o uso real dos modificadores de acesso. Essa métrica consiste em uma tabela de pontuação definida pela diferença entre a declaração e o uso do método, por exemplo: *pub-pri*, é uma classificação de um método declarado como público, mas que é usado como método privado. Esta classificação recebe uma nota. A categoria *pri-pri*, para métodos declarados como privados e usados também como privados, terá uma nota menor. E assim sucessivamente. De modo que, um valor maior para a métrica de excessividade de acesso indica um método muito acessível, mas que deveria ser encapsulado e ter uma acessibilidade menor. Um método muito acessível pode causar problemas de manutenção e trazer riscos para o sistema (VIEGA et al., 2000). Assim, é importante saber sobre as alterações que ocorrem nos modificadores de acesso dos métodos, avaliando se estão ficando mais acessíveis ou menos acessíveis. No trabalho proposto nesta dissertação, também é investigado as alterações nos modificadores de acesso dos métodos das classes. Diferente do trabalho proposto por Kabori, enquanto a abordagem proposta neste trabalho, usa análise dinâmica, Kabori usa análise estática. Outra diferença é que a abordagem apresentada por Kabori avalia o código fonte inteiro do sistema usado como entrada na ferramenta desenvolvida. A abordagem proposta nesta dissertação, apenas uma, duas ou mais funcionalidade específicas podem ser avaliadas, conforme for a necessidade do desenvolvedor, sem a necessidade de se verificar o código fonte inteiro. Além disso, este trabalho categoriza outros tipos de mudanças, enquanto que Kabori avalia apenas os modificadores de acesso.

Maqbool e Babri avaliam um algoritmo de clusterização hierárquica para recuperar arquitetura do sistema e a sua modularização (MAQBOOL; BABRI, 2007). Maqbool e

Babri também avaliam várias métricas de similaridade e distância para avaliar os pontos fortes e fracos dos algoritmos de clusterização. Por outro lado, a abordagem proposta nesta dissertação usa alinhamento de rastros de execução, ao invés de uma técnica de clusterização, para avaliar mudanças estruturais e relação entre componentes dos pacotes. O objetivo da abordagem citada é detectar, para o mesmo sistema, arquiteturas diferentes. Então o trabalho foca em descobrir o melhor algoritmo de clusterização usando métricas. Por outro lado, as mudanças detectadas pela abordagem proposta nessa dissertação aponta mudanças estruturais de uma forma mais direta, pois baseia-se em desalinhamentos de rastros de execução ao invés de métricas. Além disso, detecta também a mudança de relação em componentes dos pacotes. Embora ambos os trabalhos propostos lidem com mudanças, o trabalho citado recupera mudanças arquiteturais e o trabalho proposto nessa dissertação tem como resultado mudanças estruturais. Essas mudanças estruturais podem ou não serem, também, arquiteturais. Enquanto o trabalho citado busca um melhor algoritmo de clusterização para mostrar como resultado uma informação bastante alto nível, a arquitetura, o trabalho proposto na dissertação mostra um maior número de mudanças, sendo vantajoso para o desenvolvedor.

Hashimoto, Mori e Izumida criaram um método compreensivo para identificar mudanças no código fonte baseado em análise estática (HASHIMOTO; MORI; IZUMIDA, 2015). Propuseram a criação de um *factbase*, um banco de dados, para armazenar informações da árvore sintática abstrata dos sistemas, uma para cada versão. Depois disso, o desenvolvedor pode obter informações fazendo consultas (*queries*) neste banco de dados, comparando a árvore sintática abstrata de uma versão com a outra do sistema. De maneira semelhante, no trabalho proposto nessa dissertação, o desenvolvedor também pode efetuar consultas no banco de dados para analisar os resultados. Em contraste com o trabalho citado, a abordagem proposta neste trabalho combina análise estática e dinâmica para identificar mudanças estruturais entre duas versões de um sistema. Já o trabalho citado usa apenas análise estática.

Ren desenvolveu a ferramenta *Chiant* que usa a abordagem estática para análise de impacto de mudanças em projetos implementados na linguagem *Java* (REN et al., 2004). *Chiat* avalia duas versões de um sistema e decompõe as diferenças entre elas em um conjunto de mudanças. Como resultado, é apresentado o conjunto de testes afetados. Diferente da abordagem citada, o trabalho proposto nesta dissertação faz uso de análise dinâmica para detectar conjunto de funcionalidades afetadas por classes que foram modificadas estruturalmente.

Kim e David propuseram a ferramenta *LSDiff* que infere diferenças estruturais como regras lógicas a partir de mudanças no código fonte (KIM; NOTKIN, 2009). As mudanças do código fonte são agrupadas, independentemente de sua distribuição, sem levar em conta cenários ou funcionalidades, baseando-se em regras lógicas. Em contraste, no trabalho proposto nessa dissertação, as mudanças são extraídas de segmentos desalinhados

em rastros de execução, ao invés de regras lógicas. Além disso, as mudanças são classificadas em padrões permitindo o desenvolvedor fazer diferentes análises das mudanças que ocorreram no sistema, enquanto a abordagem citada agrupa mudanças baseando-se em regras lógicas. No trabalho citado não há relação das mudanças com funcionalidades. No trabalho proposto nesta dissertação, como as informações foram coletadas pela análise dinâmica em rastros de execução, cada mudança está, necessariamente, associada com um cenário. Então é possível fazer uma avaliação de impacto de uma mudança em outras funcionalidades, sendo bastante vantajoso para o desenvolvedor, em relação ao trabalho proposto, nesse aspecto.

Kpodjedo et al. propuseram MADMatch, uma técnica de casamento de diagramas aproximados de muitos para muitos (KPODJEDO et al., 2013). Para isso, eles utilizaram uma formulação de casamento de grafos tolerantes a erros. Os diagramas são representados como grafos, os custos são dados pelas diferenças possíveis entre dois grafos. O objetivo é encontrar o casamento mais barato. MADMatch é centrada nas dependências estruturais e análise textual (visão estática) para extrair diagramas que serão comparados pela técnica. Em contraste, a técnica proposta neste trabalho combina rastros de execução (visão dinâmica) para detectar mudanças e dependências estruturais (visão estática) para remover ruídos e classificá-las em padrões. Além disso, enquanto MADMatch detecta mudanças que ocorreram entre dois diagramas, a técnica proposta neste trabalho é orientada a cenários que representam a execução de funcionalidades específicas de um sistema. Em adição, além da técnica proposta neste trabalho permitir a análise de mudanças que ocorreram em uma dada funcionalidade do sistema, também permite medir o impacto de cada mudança no sistema (analisar quais são funcionalidades são afetadas por uma dada modificação).

5.2 Análise de Histórico de Versões

Maruyama (MARUYAMA et al., 2012) usou o conceito de cortes (WEISER, 1981) de sistemas para reproduzir mudanças passadas. O corte de um sistema consiste em um conjunto de instruções, ou uma parte do sistema que satisfaz o critério de corte. O critério de corte consiste em um conjunto de valores esperados como resultado de uma determinada parte do sistema. Foi proposto um mecanismo de corte de históricos de operações para eliminar ações não essenciais ignorando a edição de operações para classes e membros da classe que não são interessantes. A abordagem citada faz uma poda de operações não essenciais. Ou seja, partes do código fonte consideradas não essenciais, que não atendem ao critério de corte, são descartados. Semelhantemente, a abordagem proposta descarta segmentos alinhados que não representam mudanças estruturais (modificações que afetam as relações estruturais entre classes e pacotes). Além disso, na abordagem proposta, os segmentos detectados na fase de análise dinâmica que não são essenciais e

não revelam mudanças, os falsos positivos, são descartados automaticamente na fase de análise estática. Finalmente, uma outra diferença é que, neste trabalho, a abordagem proposta detecta mudanças estruturais associadas a funcionalidades específicas, enquanto que, isso não é feito na abordagem citada.

Risi, Scanniello e Tortora, aplicaram técnica de clusterização semântica para automatizar a recuperação de arquitetura baseando-se em LSI, melhorando posteriormente a arquitetura, de forma recursiva, usando um mecanismo de *Fold In* e *Fold Out* (RISI; SCANNIELLO; TORTORA, 2012). LSI consiste em uma técnica de recuperação da informação para calcular semelhanças em textos. Na clusterização semântica, eles usam algoritmos de mineração de dados que levam em consideração a semântica dos textos do código fonte e seus comentários. Para decidir se determinada parte do código-fonte pertence ou não a um cluster, foi usada a técnica LSI, que extrai vocabulários do código fonte e agrupar entidades similares. Para a técnica de *Fold In* e *Fold Out*, foi usado estrutura de pacotes, observando-se as pastas ou diretórios. A técnica de *Fold In* e *Fold Out*, consiste em pegar os clusters descobertos pelas técnicas citadas acima e melhorá-los, criando novos clusters. Para essa melhoria, a técnica avalia quais objetos de um cluster que pertencem a um mesmo diretório, avaliando-se a estrutura de pacotes do sistema. Wong apresentou uma abordagem para detectar e localizar violações de modularidade (WONG et al., 2011). *CLIO* compara como componentes devem mudar juntos de acordo com a estrutura modular. Uma mudança é classificada como violação de modularidade quando dois componentes mudam juntos, mas pertencem a módulos diferentes (que supostamente deveriam evoluir de maneira independente). Em contraste, a abordagem proposta neste trabalho detecta e categoriza mudanças estruturais ao invés da arquitetura do sistema, como é feito na abordagem citada. A critério do desenvolvedor, tendo as mudanças estruturais exibidas como resultado, ele pode procurar e identificar mudanças arquiteturais a critério de seu julgamento. Mas o foco da abordagem proposta é exibir um resultado que contenha mudanças estruturais e não arquiteturais. Já semelhantemente à abordagem citada, no trabalho proposto nesta dissertação a análise da estrutura de pacotes no código é usada e extraída para avaliar mudanças na relação de componentes entre pacotes. A vantagem direta da abordagem proposta sobre as citadas é que um número maior de mudanças serão exibidos para o desenvolvedor. Em ambos os trabalhos, o desenvolvedor pode identificar mudanças: na abordagem citada, pode identificar mudanças arquiteturais se ele tiver a arquitetura original no sistema para comparar. Na abordagem proposta, ele identifica mudanças estruturais. Mas como a abordagem citada exibe uma arquitetura do sistema, o resultado é uma informação mais alto nível do que uma mudança em um método ou classe, e portanto, pode ocultar muitas informações que seriam úteis para o desenvolvedor. E se a quantidade de informações exibidas pela abordagem proposta for muito grande, o desenvolvedor ainda pode separá-las por funcionalidade, tendo assim, um conjunto de informações menor para avaliar e mais baixo nível do que uma abstração

arquitetural.

Hu aplicara a técnica de *Dynamic Topic Models* (DTM) em mensagens de repositório de código fonte para recuperar tópicos relacionados com a evolução do sistema (HU et al., 2015). Os tópicos são investigados manualmente para organizá-los em categorias de mudanças, como mudanças corretivas e adaptativas. Em contraste, a abordagem proposta neste trabalho usa análise dinâmica e estática para, automaticamente, categorizar mudanças em padrões.

Buse and Weimer propuseram a técnica *DeltaDoc* com o objetivo de descrever as condições que disparam a mudança no código e como a mudança impacta o comportamento funcional e o estado do programa (BUSE; WEIMER, 2010). A técnica combina execução simbólica e uma nova abordagem para sumarizar código. *DeltaDoc* mostra efeito de uma mudança em tempo de execução do programa, como as condições em que o comportamento do sistema muda e qual é o novo comportamento. Em contraste com *DeltaDoc*, a abordagem proposta neste trabalho detecta quais são as mudanças estruturais e qual é o impacto delas sobre as funcionalidades do sistema.

Dit et al. propuseram ImpactMiner, uma ferramenta para análise de impacto de mudança (DIT et al., 2014). A abordagem combina diferentes fontes de informações, análise textual estática, rastros de execução e *co-changes* por minerar repositórios. Basicamente, relações de *co-changes* representam entidades que geralmente mudam juntas, por exemplo, sempre que uma método `A.m1` muda, o método `B.m2` também muda. A ferramenta permite desenvolvedores extrair *co-changes* de repositórios em granularidade de métodos. Além disso, permite usar diferentes técnicas (por exemplo, recuperação de informação a partir de textos do código e rastros de execução) ou até mesmo combiná-las. Enquanto a abordagem citada avalia apenas uma versão do sistema, o trabalho proposto nessa dissertação trabalha com duas versões de um sistema. A vantagem do trabalho citado é poder extrair informações de *co-changes* obtidas de uma única versão do sistema, sendo, em alguns casos mais leve do que a abordagem proposta (alguns casos porque a abordagem citada também trabalha com rastros de execução, a fonte de informação que é mais pesada). Porém, o trabalho citado não avalia o impacto de mudanças em funcionalidades. Pois, embora trabalhem com rastros, não usam o conceito de funcionalidades ou cenários. Assim, a abordagem proposta nessa dissertação avalia o impacto que uma mudança pode ter em outras funcionalidades, uma informação relevante que não é obtida no trabalho citado. Além disso, o trabalho citado tem foco em *co-changes*, ou seja, mudanças que ocorrem juntas. Já o trabalho proposto nessa dissertação de mestrado tem foco em mudanças estruturais, e, a partir de mudanças estruturais, é possível saber, também, se outras classes podem ser afetadas por essas mudanças. Ou seja, a abordagem proposta também irá apontar mudanças que podem ocorrer juntas. A vantagem de se obter, também, as mudanças estruturais é que essas mudanças tendem a ter maior impacto no código.

5.3 Análise Dinâmica

Abordagens baseadas em análise dinâmica são usadas para medir o acoplamento dinâmico (ARISHOLM; BRIAND; FOYEN, 2004), implementar novas funcionalidades (SILVA et al., 2011), tratamento de *bug* (ROEHM; NOSOVIC; BRUEGGE, 2015), reconstrução da arquitetura (RIVA; RODRIGUEZ, 2002) e em projetos *SOA* (NAYROLLES; MOHA; VALTCHEV, 2013; PAUW et al., 2005). Um algoritmo foi proposto para remover detalhes de implementação (SILVA et al., 2011). Aplicaram alinhamento de rastros entre funcionalidades similares em uma mesma versão de um sistema, para separar partes do código comum de partes específicas. Os desalinhamentos foram usados para ajudar o desenvolvedor implementar novas funcionalidades (manutenção perfectiva). De maneira similar, neste trabalho, também é aplicado alinhamento de rastros. No entanto, o alinhamento é feito para a mesma funcionalidade mas entre versões diferentes de um mesmo sistema. Além disso, o objetivo deste trabalho está em detectar mudanças estruturais que ocorreram entre duas versões no sistema.

Riva e Rodriguez apresentaram uma técnica que combina informações estáticas e dinâmicas para ajudar a reconstruir a arquitetura do sistema (RIVA; RODRIGUEZ, 2002). Sua abordagem permite criar visões arquiteturais do sistema. Enquanto estão focados na reconstrução da arquitetura, o objetivo da abordagem proposta neste trabalho de dissertação é auxiliar na compreensão de mudanças estruturais entre duas versões do sistema que afetam as relações entre os pacotes. Além disso, é possível identificar quais foram as mudanças e quais funcionalidades foram afetadas.

Roehm, Nosovic e Bruegge coletaram interações em rastros de usuários para ajudar a resolver *bug* e travamento de sistemas (ROEHM; NOSOVIC; BRUEGGE, 2015). Fazem uso de dois algoritmos (primeiro sozinhos e depois combinados) para reduzir a quantidade de informações para o mínimo possível que esteja relacionado com o processo de falha. No trabalho proposto nesta dissertação, os rastros coletados para uma funcionalidade específica do sistema são também reduzidos para facilitar a análise dos rastros (descartando laços, chamadas recursivas e chamadas de baixa granularidade).

Wloka Ryder e Tip desenvolveram *JUnitMX*, uma ferramenta de testes de unidade – extensão do plug-in *JUnit* para Eclipse – que captura comportamento de programa e calcula a cobertura de mudança para cada teste (WLOKA; RYDER; TIP, 2009). *JUnitMX* provê várias funcionalidades, como explorar os efeitos de mudanças não testados e adicionar novos testes exercitando as características adicionadas no sistema. *JUnitMX* se difere deste trabalho porque a ferramenta classifica as mudanças somente como adição e mudanças enquanto que a abordagem proposta classifica as mudanças estruturais em cinco padrões e permite visualizar quantas e quais funcionalidades são afetadas.

Chern e Volder investigaram o impacto de acoplamento dinâmico-estrutural em uma linguagem de programação sobre o esforço necessário para evoluir a estrutura modular de programas escritos nesta linguagem (CHERN; VOLDER, 2008). Foi definido o conceito

de acoplamento dinâmico-estrutural como sendo o grau de mudanças em uma estrutura modular estática de um programa, que implica mudanças na sua estrutura dinâmica. Os resultados apresentaram alguns indícios de como a complexidade accidental surge na remodularização de programas Java. Em outras palavras, remodularização em Java envolve mais estratégias de separação complexas (estratégias para modificar a estrutura dinâmica somente para permitir a separação de código). Holmes e Notkin apresentaram uma abordagem para identificar dependências de chamadas específicas que sugerem aos desenvolvedores, quando uma mudança no sistema e os comportamentos subsequentes do programa talvez seja menos consistente que o esperado (HOLMES; NOTKIN, 2011). Para isso, extraíram grafos de dependências estáticas e dinâmicas de pares de versões e particionaram as dependências baseando-se em suas presenças ou ausências em cada um dos grafos. Como resultado, observaram que algumas partições específicas contém dependências que podem ser difíceis de identificar sucintamente através de outras abordagens. Enquanto o objetivo das duas abordagens citadas acima é de verificar como mudanças alteram o comportamento dinâmico de programas, o foco deste trabalho é de detectar e classificar mudanças estruturais que acontecem em um sistema de uma versão pra outra.

McCamant e Ernst propuseram uma técnica automática para predizer problemas causados por atualizações de componentes (MCCAMANT; ERNST, 2003). A técnica é aplicada antes de integrar o novo componente ou executar testes no sistema, permitindo identificar os problemas de maneira mais rápida e simples. O resultado gerado é uma abstração operacional para o componente antigo no contexto do sistema e uma abstração operacional para o novo componente no contexto da sua suíte de testes. Os resultados apresentaram que o novo componente não garante fazer tudo que o antigo fazia e, portanto, a atualização pode afetar o comportamento do sistema e não deveria ser executado sem uma avaliação cuidadosa.

Conclusão

Este trabalho apresentou uma abordagem para localizar mudanças estruturais em versões consecutivas de sistemas de software, tendo em vista a necessidade de os desenvolvedores entenderem mais facilmente o impacto de mudanças em outras funcionalidades. Foi usado o alinhamento de rastros de execução, alinhando-se rastros de duas versões de um mesmo sistema. Os rastros foram coletados a partir de cenários de execução, documentados e disponíveis junto com as aplicações. Em seguida foram comprimidos para reduzir a quantidade de informação e facilitar seu armazenamento. Em seguida os rastros foram separados por cenários e um par de versões do sistema foi alinhado para extrair mudanças, que foram armazenadas em um banco de dados relacional. Em seguida, um novo filtro foi feito usando análise estática. Na última etapa, as mudanças resultantes foram classificadas em cinco padrões: *Move Class*, *Move Method*, *Add Method*, *Remove Method* e *Access Modifier Change*.

Foram feitos seis estudos de caso, sendo três no *Tomcat*, dois no *jFreeChart* e um no *JHotDraw*. No caso do *jFreeChart* mudanças de relacionamento de pacotes foram detectadas entre as versões *0.7.0* e *0.9.5* e essas mudanças foram validadas usando precisão e revocação. A precisão foi validada por inspeção no código fonte e a revocação pela documentação disponível de mudanças no *jFreeChart*. O resultado foi uma precisão de 100% e uma revocação de 83%. Além disso, para todos os sistemas, em média, os padrões de mudanças mais detectados foram os padrões relacionados com *Add Method* e *Move Method*. Também foram detectadas as classes alteradas que mais impactam funcionalidades dos sistemas.

6.1 Principais Contribuições

As principais contribuições deste trabalho são:

- ❑ **Localização de Mudanças estruturais:** A detecção de mudanças estruturais nos sistemas foi feita avaliando uma ou mais funcionalidades específicas. A vantagem

desse método é que o desenvolvedor não precisa avaliar o código fonte inteiro do sistema, como nas abordagens que fazem somente análise estática.

- ❑ **Categorização automática das mudanças estruturais:** As mudanças detectadas pela análise dinâmica e estática são classificadas em cinco padrões: i) movimentação de classes; ii) movimentação de métodos; iii) adição de métodos; iv) remoção de métodos; v) mudança de modificador de acesso.
- ❑ **Detecção do impacto de mudanças estruturais em funcionalidades:** A abordagem também mede o impacto das mudanças em cenários do sistema, ao mostrar quantas funcionalidades são afetadas por uma mudança em um método ou em uma classe. Os resultados dos experimentos mostraram as classes alteradas que mais impactam funcionalidades do sistema. Assim, o desenvolvedor poderá, a partir dos resultados, prever mudanças que poderão acontecer quando chegarem novas solicitações de mudanças no sistema. Os resultados mostram, para uma classe modificada, se houve mudanças em outras classes e em outras funcionalidades. Desta forma, ao conhecer o comportamento de mudanças olhando para o passado do sistema, o desenvolvedor pode, por exemplo, preparar casos de testes para as funcionalidades que foram impactadas por mudanças em uma class e previamente. Por exemplo, suponha que a abordagem detectou mudanças estruturais na classe *HostConfig* para o cenário *Acessar o Manager* que afetaram também os cenários *Fazer depuração de aplicação* e *Verificar Status do Servidor* e outras classes, para o *Tomcat*. Quando chegar uma nova mudança que for afetar novamente a funcionalidade *Acessar o Manager* o desenvolvedor poderá prever que podem acontecer mudanças para os outros dois cenários novamente, *Fazer depuração de aplicação* e *Verificar status do servidor* e inclusive quais classes que também podem ser modificadas junto. Assim ele pode fazer um melhor planejamento diante desta mudança e preparar casos de testes para estes cenários e classes.
- ❑ **Detecção de mudança de relacionamento entre os pacotes:** É possível detectar mudanças de relacionamentos entre pacotes do sistema. Os resultados dos experimentos mostraram as mudanças classificadas nos padrões, as classes alteradas que mais impactam funcionalidades do sistema e mudanças de relacionamento entre pacotes do *jFreeChart*.
- ❑ **Permitir ao desenvolvedor fazer uma previsão do impacto das mudanças:** Com o resultado do impacto das mudanças de cada classe em funcionalidades, o desenvolvedor é capaz de, ao olhar para o passado, prever como mudanças futuras afetarão o sistema. Por exemplo, diante de um requisito de mudança que afeta uma determinada funcionalidade, o desenvolvedor terá uma previsão de quantas classes serão modificadas e quais as outras funcionalidades serão afetadas por tais

modificações. Com isso, ele pode mensurar melhor os recursos empregados nas mudanças solicitadas assim como preparar casos de testes mais abrangentes.

6.2 Trabalhos Futuros

Como trabalho futuro, deseja-se avaliar a abordagem para versões mais antigas de sistemas para avaliar outras categorias de mudanças e validar mudanças nos pacotes que podem ser arquiteturais. Além disso, pode ser relevante avaliar a abordagem para novas funcionalidades implementadas, alinhando os rastros de execução com funcionalidades semelhantes já existentes.

A abordagem detectou mudanças de relacionamentos de pacotes. Essas mudanças podem ser ainda mais evidentes para versões mais antigas do sistema. Pois, as versões mais novas dos sistemas possuem muitas mudanças com relação às versões mais antigas. E essas mudanças podem ficar ainda mais evidentes se forem comparadas com versões mais distantes. Por exemplo, ao invés de avaliar as versões 6 e 7 do Tomcat, versões mais próximas, pode-se tentar fazer uma avaliação entre o Tomcat 3 e Tomcat 7, por exemplo. Assim, para avaliar se a mudança é arquitetural com maior grau de assertividade, a avaliação de versões mais antigas pode ser uma saída, pois essas mudanças seriam ainda mais evidentes.. Além disso, outros tipos de mudanças podem ser, também adicionados na classificação feita pela abordagem.

Como a abordagem determina o alinhamento de rastros de execução por funcionalidade, quando uma nova funcionalidade é implementada, é preciso avaliar as mudanças e os impactos causados por essa nova funcionalidade. Como ela é nova e não possui uma funcionalidade correspondente, ela pode ser alinhada com uma funcionalidade semelhante. O alinhamento de rastros de execução é uma técnica capaz de comparar funcionalidades similares, e não só as mesmas. Em um trabalho anterior, foram usadas funcionalidades similares (e não a mesma funcionalidade) no sistema para saber onde adicionar código fonte quando novos requisitos forem implementados. Assim, se um componente ou cenário for adicionado para uma versão, ele ainda pode ser comparado com cenários semelhantes para detectar mudanças.

Para tornar a tarefa de coleta dos rastros de execução mais fácil, pode-se abstrair um cenário como sendo um *bug report*, um *commit* ou textos de um *issue tracker*. Os passos e os dados descritos para reproduzir o *bug* podem ser usados para montar um cenário de execução, assim como os textos de um *commit* ou a descrição em um *issue tracker* também o podem. Essas três abstrações podem ser usadas para montar classes de teste e assim facilitar o trabalho de coleta dos rastros de execução. Um trabalho anterior foi feito usando esses itens para representar cenários de execução. Assim esse conceito pode ser aplicado também nessa abordagem de forma a facilitar a coleta dos rastros de execução.

Referências

- ARISHOLM, E.; BRIAND, L. C.; FOYEN, A. Dynamic coupling measurement for object-oriented software. **IEEE Transactions on Software Engineering**, v. 30, n. 8, p. 491–506, 2004. Disponível em: <<https://doi.org/10.1109/TSE.2004.41>>.
- AUSTIN, R. D. The effects of time pressure on quality in software development: An agency model. **Information Systems Research**, v. 12, n. 2, p. 195–207, 2001. Disponível em: <<https://doi.org/10.1287/isre.12.2.195.9699>>.
- BECK, F. et al. Rethinking user interfaces for feature location. In: **23rd International Conference on Program Comprehension**. IEEE Press, 2015. p. 151–162. Disponível em: <<https://doi.org/10.1109/ICPC.2015.24>>.
- BOOCH, G. et al. **Object-oriented Analysis and Design with Applications**. [S.l.]: Addison-Wesley Professional, 2007.
- BRIAND, L.; LABICHE, Y.; LEDUC, J. Toward the reverse engineering of uml sequence diagrams for distributed java software. **IEEE Transactions on Software Engineering**, v. 32, n. 9, p. 642–663, 2006. Disponível em: <<https://doi.org/10.1109/TSE.2006.96>>.
- BUSE, R. P.; WEIMER, W. R. Automatically documenting program changes. In: **10th Proceedings of the International Conference on Automated Software Engineering (ASE)**. [s.n.], 2010. p. 33–42. Disponível em: <<https://doi.org/10.1145/1858996.1859005>>.
- CHEN, A. et al. Cvssearch: searching through source code using cvs comments. In: **IEEE International Conference on Software Maintenance (ICSM)**. [s.n.], 2001. p. 364–373. Disponível em: <<https://doi.org/10.1109/ICSM.2001.972749>>.
- CHERN, R.; VOLDER, K. D. The impact of static-dynamic coupling on modularization. In: **23rd Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA)**. [s.n.], 2008. p. 261–276. Disponível em: <<https://doi.org/10.1145/1449764.1449786>>.
- CODOBAN, M. et al. Software history under the lens: A study on why and how developers examine it. In: **31st International Conference on Software Maintenance and Evolution (ICSME)**. IEEE, 2015. p. 1–10. Disponível em: <<https://doi.org/10.1109/ICSME.2015.7332446>>.

- CORNELISSEN, B. et al. A systematic survey of program comprehension through dynamic analysis. **IEEE Transactions on Software Engineering**, v. 35, n. 5, p. 684–702, 2009. Disponível em: <<https://doi.org/10.1109/TSE.2009.28>>.
- DIT, B. et al. Can better identifier splitting techniques help feature location? In: **19th International Conference on Program Comprehension (ICPC)**. [s.n.], 2011. p. 11–20. Disponível em: <<https://doi.org/10.1109/ICPC.2011.47>>.
- DIT, B.; MORITZ, E.; POSHYVANYK, D. A tracelab-based solution for creating, conducting, and sharing feature location experiments. In: **20th International Conference on Program Comprehension (ICPC)**. [s.n.], 2012. p. 203–208. Disponível em: <<https://doi.org/10.1109/ICPC.2012.6240489>>.
- DIT, B. et al. Feature location in source code: a taxonomy and survey. **Journal of Software: Evolution and Process**, v. 25, p. 53–95, 2013. Disponível em: <<https://doi.org/10.1002/smr.567>>.
- DIT, B.; REVELLE, M.; POSHYVANYK, D. Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. In: **Empirical Software Engineering**. [s.n.], 2013. v. 18, n. 2, p. 277–309. Disponível em: <<https://doi.org/10.1007/s10664-011-9194-4>>.
- DIT, B. et al. Impactminer: A tool for change impact analysis. In: **Proceedings of the 36th International Conference on Software Engineering**. [s.n.], 2014. p. 540–543. Disponível em: <<https://doi.org/10.1145/2591062.2591064>>.
- DUCASSE, S.; POLLET, D. Software architecture reconstruction: A process-oriented taxonomy. **IEEE Transactions on Software Engineering**, v. 35, n. 4, p. 573–591, 2009. Disponível em: <<https://doi.org/10.1109/TSE.2009.19>>.
- ECKEL, B. **Thinking in Java**. 4. ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005. ISBN 0131872486.
- EICK, S. et al. Does code decay? assessing the evidence from change management data. **IEEE Transactions on Software Engineering**, v. 27, n. 1, p. 1–12, 2001. Disponível em: <<https://doi.org/10.1109/32.895984>>.
- GUIMARÃES, E.; GARCIA, A.; CAI, Y. Architecture-sensitive heuristics for prioritizing critical code anomalies. In: **14th International Conference on Modularity**. [s.n.], 2015. (MODULARITY), p. 68–80. Disponível em: <<https://doi.org/10.1145/2724525.2724567>>.
- HAMOU-LHADJ, A.; LETHBRIDGE, T. Compression techniques to simplify the analysis of large execution traces. In: **10th International Workshop on Program Comprehension**. [s.n.], 2002. p. 159–168. Disponível em: <<https://doi.org/10.1109/WPC.2002.1021337>>.
- HASHIMOTO, M.; MORI, A.; IZUMIDA, T. A comprehensive and scalable method for analyzing fine-grained source code change patterns. In: **22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)**. [s.n.], 2015. p. 351–360. Disponível em: <<https://doi.org/10.1109/SANER.2015.7081845>>.

- HOLMES, R.; NOTKIN, D. Identifying program, test, and environmental changes that affect behaviour. In: **33rd International Conference on Software Engineering (ICSE)**. [s.n.], 2011. p. 371–380. Disponível em: <<https://doi.org/10.1145/1985793.1985844>>.
- HU, J. et al. Modeling the evolution of development topics using dynamic topic models. In: **22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)**. [s.n.], 2015. p. 3–12. Disponível em: <<https://doi.org/10.1109/SANER.2015.7081810>>.
- JIANG, Q. et al. Summarizing evolutionary trajectory by grouping and aggregating relevant code changes. In: **22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)**. [S.l.: s.n.], 2015. p. 361–370.
- KIM, M.; NOTKIN, D. Discovering and representing systematic code changes. In: **31st International Conference on Software Engineering (ICSE)**. [s.n.], 2009. p. 309–319. Disponível em: <<https://doi.org/10.1109/ICSE.2009.5070531>>.
- KIM, S. et al. Predicting faults from cached history. In: **29th International Conference on Software Engineering (ICSE)**. [s.n.], 2007. p. 489–498. Disponível em: <<https://doi.org/10.1109/ICSE.2007.66>>.
- KOBORI, K.; MATSUSHITA, M.; INOUE, K. Evolution analysis for accessibility excessiveness in java. In: **22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)**. [S.l.: s.n.], 2015. p. 83–90.
- KPODJEDO, S. et al. Madmatch: Many-to-many approximate diagram matching for design comparison. **IEEE Transaction on Software Engineering**, v. 39, n. 8, p. 1090–1111, 2013. Disponível em: <<https://doi.org/10.1109/TSE.2013.9>>.
- LAVALLEE, M.; ROBILLARD, P. Why good developers write bad code: An observational case study of the impacts of organizational factors on software quality. In: **37th International Conference on Software Engineering (ICSE)**. [S.l.: s.n.], 2015. v. 1, p. 677–687.
- LIU, D. et al. Feature location via information retrieval based filtering of a single scenario execution trace. In: **22nd International Conference on Automated Software Engineering (ASE)**. [s.n.], 2007. p. 234–243. Disponível em: <<https://doi.org/10.1145/1321631.1321667>>.
- MAQBOOL, O.; BABRI, H. Hierarchical clustering for software architecture recovery. **IEEE Transactions on Software Engineering**, v. 33, n. 11, p. 759–780, 2007. Disponível em: <<https://doi.org/10.1109/TSE.2007.70732>>.
- MARUYAMA, K. et al. Slicing and replaying code change history. In: **27th International Conference on Automated Software Engineering (ASE)**. [s.n.], 2012. p. 246–249. Disponível em: <<https://doi.org/10.1145/2351676.2351713>>.
- MCCAMANT, S.; ERNST, M. D. Predicting problems caused by component upgrades. In: **9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)**. [s.n.], 2003. p. 287–296. Disponível em: <<https://doi.org/10.1145/940071.940110>>.

- MEYER, B. **Object-Oriented Software Construction**. [S.l.]: Prentice-Hall, 2000.
- NAYROLLES, M.; MOHA, N.; VALTCHEV, P. Improving soa antipatterns detection in service based systems by mining execution traces. In: **20th Working Conference on Reverse Engineering (WCRE)**. [s.n.], 2013. p. 321–330. Disponível em: <<https://doi.org/10.1109/WCRE.2013.6671307>>.
- NEEDLEMAN, S. B.; WUNSCH, C. D. A general method applicable to the search for similarities in the amino acid sequence of two proteins. p. 443–453, 1970.
- OIZUMI, W. et al. When code-anomaly agglomerations represent architectural problems? an exploratory study. In: **28th Brazilian Symposium on Software Engineering (SBES)**. [S.l.: s.n.], 2014. p. 91–100.
- PAIXÃO, K. V. R. **Alinhamento de Rastros de Execução de Programas para Compreensão de Pontos de Variação em Código Fonte**. Dissertação (Mestrado) — Universidade Federal de Uberlândia, 2009.
- PAUW, W. D. et al. Web services navigator: Visualizing the execution of web services. **IBM Systems Journal**, v. 44, n. 4, p. 821–845, 2005. Disponível em: <<https://doi.org/10.1147/sj.444.0821>>.
- PIORKOWSKI, D. et al. To fix or to learn? how production bias affects developers' information foraging during debugging. In: **31st International Conference on Software Maintenance and Evolution (ICSME)**. [S.l.]: IEEE, 2015. p. 11–20.
- POSHYVANYK, D. et al. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. **IEEE Transactions on Software Engineering**, v. 33, n. 6, p. 420–432, 2007. ISSN 0098-5589.
- REN, X. et al. Chianti: A tool for change impact analysis of java programs. In: **19th Object-Oriented Systems, Languages, Programming, and Applications (OOPSLA)**. [s.n.], 2004. p. 432–448. Disponível em: <<https://doi.org/10.1145/1028976.1029012>>.
- REVELLE, M.; DIT, B.; POSHYVANYK, D. Using data fusion and web mining to support feature location in software. In: **18th International Conference on Program Comprehension (ICPC)**. [s.n.], 2010. p. 14–23. Disponível em: <<https://doi.org/10.1109/ICPC.2010.10>>.
- REVELLE, M.; POSHYVANYK, D. An exploratory study on assessing feature location techniques. In: **17th International Conference on Program Comprehension**. [s.n.], 2009. p. 218–222. Disponível em: <<https://doi.org/10.1109/ICPC.2009.5090045>>.
- RISI, M.; SCANNIELLO, G.; TORTORA, G. Using fold-in and fold-out in the architecture recovery of software systems. **Formal Aspects of Computing**, v. 24, n. 3, p. 307–330, 2012. Disponível em: <<https://doi.org/10.1007/s00165-011-0199-y>>.
- RIVA, C.; RODRIGUEZ, J. Combining static and dynamic views for architecture reconstruction. In: **16th European Conference on Software Maintenance and Reengineering**. [s.n.], 2002. p. 47–55. Disponível em: <<https://doi.org/10.1109/CSMR.2002.995789>>.

- ROEHM, T.; NOSOVIC, S.; BRUEGGE, B. Automated extraction of failure reproduction steps from user interaction traces. In: **22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)**. [s.n.], 2015. p. 121–130. Disponível em: <<https://doi.org/10.1109/SANER.2015.7081822>>.
- SANCHEZ, H.; ROBBES, R.; GONZALEZ, V. An empirical study of work fragmentation in software evolution tasks. In: **22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)**. [s.n.], 2015. p. 251–260. Disponível em: <<https://doi.org/10.1109/SANER.2015.7081835>>.
- SARTIPI, K.; DEZHKAM, N. An amalgamated dynamic and static architecture reconstruction framework to control component interactions 259. In: **14th Working Conference on Reverse Engineering (WCRE)**. [S.l.: s.n.], 2007. p. 259–268.
- SAVAGE, T.; REVELLE, M.; POSHYVANYK, D. Flat3: Feature location and textual tracing tool. In: **32nd International Conference on Software Engineering (ICSE)**. [S.l.: s.n.], 2010. p. 255–258.
- SEACORD, R. C.; PLAKOSH, D.; LEWIS, G. A. **Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices**. [S.l.: s.n.], 2003.
- SILVA, L. et al. Software evolution aided by execution trace alignment. In: **24th Brazilian Symposium on Software Engineering (SBES)**. [s.n.], 2010. p. 158–167. Disponível em: <<https://doi.org/10.1109/SBES.2010.25>>.
- _____. Developers’ perception of co-change patterns: An empirical study. In: **31st International Conference on Software Maintenance and Evolution (ICSME)**. [S.l.: s.n.], 2015. p. 21–30.
- SILVA, L. L. et al. On the use of execution trace alignment for driving perfective changes. In: **15th European Conference on Software Maintenance and Reengineering (CSMR)**. [s.n.], 2011. p. 221–230. Disponível em: <<https://doi.org/10.1109/CSMR.2011.28>>.
- TAO, Y. et al. How do software engineers understand code changes?: An exploratory study in industry. In: **International Symposium on the Foundations of Software Engineering (FSE)**. [S.l.: s.n.], 2012. p. 1–11.
- TUFANO, M. et al. When and why your code starts to smell bad. In: **37th International Conference on Software Engineering (ICSE)**. [s.n.], 2015. p. 403–414. Disponível em: <<https://doi.org/10.1109/ICSE.2015.59>>.
- VAHABZADEH, A.; FARD, A. M.; MESBAH, A. An empirical study of bugs in test code. In: **International Conference on Software Maintenance and Evolution (ICSME)**. [s.n.], 2015. p. 1–10. Disponível em: <<https://doi.org/10.1109/ICSM.2015.7332456>>.
- VIEGA, J. et al. Statically scanning java code: Finding security vulnerabilities. **IEEE Transactions on Software Engineering**, v. 17, n. 5, p. 68–74, 2000. Disponível em: <<https://doi.org/10.1109/52.877869>>.

- WEISER, M. Program slicing. In: **5th International Conference on Software Engineering (ICSE)**. [S.l.: s.n.], 1981. p. 439–449.
- WILLIAMS, B. J.; CARVER, J. C. Characterizing software architecture changes: A systematic review. **Information and Software Technology**, v. 52, n. 1, p. 31–51, 2010. Disponível em: <<https://doi.org/10.1016/j.infsof.2009.07.002>>.
- WLOKA, J.; RYDER, B.; TIP, F. Junitmx - a change-aware unit testing tool. In: **31st International Conference on Software Engineering (ICSE)**. [S.l.: s.n.], 2009. p. 567–570.
- WONG, S. et al. Detecting software modularity violations. In: **33rd International Conference on Software Engineering (ICSE)**. [s.n.], 2011. p. 411–420. Disponível em: <<https://doi.org/10.1145/1985793.1985850>>.
- XIE, X.; POSHYVANYK, D.; MARCUS, A. Support for static concept location with sv3d. In: **3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)**. [S.l.: s.n.], 2005. p. 1–6.
- XING, Z.; STROULIA, E. Analyzing the evolutionary history of the logical design of object-oriented software. **IEEE Transactions on Software Engineering**, v. 31, n. 10, p. 850–868, 2005. Disponível em: <<https://doi.org/10.1109/TSE.2005.106>>.
- ZHAO, W. et al. Sniafl: towards a static non-interactive approach to feature location. In: **26th International Conference on Software Engineering (ICSE)**. [S.l.: s.n.], 2004. p. 293–303.