Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Dissertação de Mestrado

Listas usadas como conjuntos: um estudo através de ferramenta de reescrita

Filipe Neves Cavalcante

Campina Grande – Paraíba – Brasil
Março de 2017

Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Listas usadas como conjuntos: um estudo através de ferramenta de reescrita

Filipe Neves Cavalcante

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande – Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Engenharia de Software

Tiago Lima Massoni
Adalberto Cajueiro de Farias
(Orientadores)

Campina Grande – Paraíba – Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

C376l Cavalcante, Filipe Neves.

Listas usadas como conjuntos : um estudo através de ferramenta de reescrita / Filipe Neves Cavalcante . – Campina Grande, 2017.

142 f. : il. color.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e informática, 2017.

"Orientação: Prof. Dr. Tiago Lima Massoni, Prof. Dr. Adalberto Cajueiro de Farias".

Referências.

1. *Java Collections Framework*. 2. Migração de API. 3. Ferramenta de Reescrita. I. Massoni, Tiago Lima. II. Farias, Adalberto Cajueiro de. III. Título.

CDU 004.43(043)

Resumo

Java Collections Framework (JCF) oferece um conjunto de Tipos Abstratos de Dados (TADs) na linguagem Java, sendo um *framework* bastante utilizado. Entretanto, as particularidades de uso de cada coleção são negligenciadas em favor da praticidade, sendo muito comum, por exemplo, o uso de listas sem permitir repetições de elementos, como se fossem um TAD conjunto.

Há abordagens semiautomáticas de correção desse tipo de problema, mas elas introduzem classes adaptadores ao código original, o que nem sempre é desejável. Já a Substituição Direta apenas troca uma chamada de método por outra(s) correspondente(s), realizando o mínimo de mudanças possíveis, parecendo ser uma opção melhor. Entretanto, ela é mais complexa, o que leva este trabalho a estudar as condições e consequências para que ela seja possível. Foram desenvolvidas transformações implementadas num ambiente de reescrita de programas, sendo estas testadas e analisadas por meio de um estudo de caso, a fim de verificar sua aplicabilidade, e também através de questionário destinado a programadores Java, que analisou a aceitabilidade das transformações em dois trechos de código. Como resultados do estudo de caso, tem-se a ocorrência de 14% de projetos fazendo uso de listas atuando como conjuntos. E a `ArrayList` apresentou maior incidência (90%) já `List` ocorreu nos demais casos (10%). Em se tratando das avaliações realizadas pelos desenvolvedores Java no questionário, cerca de 45% dos participantes propuseram sugestões consideradas adequadas, de acordo com a API e *JCF* [1], para um trecho de código que utilizava o método `add(E e)` de `List`, e 25% deles para outro trecho que realizava uma chamada do método `set(int index, E e)`. Com isso, percebe-se a relevância do problema, sendo necessário continuar desenvolvendo novas técnicas e ferramentas que façam um programa utilizar coleções de forma adequada, tornando-o mais legível e eficiente.

Palavras-Chave. *Java Collections Framework*, Migração de API, Ferramenta de Reescrita.

Abstract

Java Collections Framework (JCF) offers a set of Abstract Data Types (ADTs) in the Java language, being a widely used framework. However, because particularities of each collection the usage are neglected in favor of practicality, being very common, for example, the use of lists without allowing repetitions of elements, as if they were a set ADT. There are semi-automatic approaches to correct this type of problem, but they insert adaptive classes to the original code, which is not always desirable. Direct substitution, however, only exchanges a method call for another(s) corresponding(s), making as few changes as possible, appearing to be a better option. However, it is more complex, which leads this work to study the conditions and consequences to make it possible. It was developed transformations implemented in a program rewriting environment, which were tested and analyzed through a case study, an end to verify its application, and also through a questionnaire aimed at Java programmers, which analyzed the acceptability of transformations in two sections of code. As a case study results, there is an occurrence of 14% of the works that make use of lists acting as sets. And an `ArrayList` presented higher incidence (90%) and `List` itself cases (10%). Regarding the revisions made by the Java developers in the questionnaire, about 45% of the participants proposed suggestions considered appropriate, according to an API and JCF [1], for a section of code that used the `add (E e)` method of `List` , And 25% of them for another section that performed a call of the method `set (int index, E and)`. With this, we can see the relevance of the problem, being necessary to continue to develop new techniques and tools that will make a program using collection properly, making it more readable and efficient

Keywords. Java Collections Framework, API Migration, Rewriting Tool.

Agradecimentos

Agradeço a Deus por ter me dado forças para continuar e concluir mais esta etapa em minha vida.

Agradeço a Aline Araújo, minha companheira, por estar sempre ao meu lado nas horas boas e ruins, comemorando e celebrando os bons momentos das nossas vidas e sendo um porto seguro nos momentos de angústia e incerteza. Obrigado por me incentivar e ajudar a ser a minha melhor versão todos os dias. Sem você em minha vida, com certeza, tudo seria mais difícil. Em especial obrigado por me ajudar a terminar este mestrado, por ter estudado e entendido toda minha pesquisa para revisar e me ajudar a escrever a dissertação. TE AMO!

Agradeço ao tesouro que Deus me deu, meu filho Mateus, que dormiu no meu colo muitas vezes em quanto eu estudava para o mestrado. Ele mesmo sem saber, era meu combustível, quando me dava abraços, sorrisos e brincadeiras nos momentos em que eu estava mais estressado, me fazendo lembrar o que realmente importa na vida.

Agradeço a Edineis e Paulo, meus pais, por sempre terem proporcionado e incentivado o meu crescimento tanto profissional quanto pessoal, principalmente em se tratando de educação. Até mesmo abdicando de muitas coisas para me proporcionar um ensino de qualidade. Agradeço por terem "pegado no meu pé".

Sou grato a meus orientadores Adalberto e Tiago, por terem me mostrado o caminho certo a seguir e terem aberto minha mente para conhecimentos acadêmicos e de mundo. Também agradeço ao governo brasileiro, por ter apoiado financeiramente meu curso e ao pessoal da COPIN e do SPLAB.

Epígrafe

"Eu acredito que olhar pra trás também é seguir em frente. É quando a gente relembra tudo que já passou pra chegar até aqui e tem a certeza de que nunca é hora de parar a caminhada."

Bráulio Bessa

Dedicatória

À Aline, Mateus e toda minha família.

Conteúdo

Lista de Acrônimos

API	*Application Programming Interface*
IDE	*Integrated Development Environment*
JCF	*Java Collection Framework*
JDT	*Java Development Toolkit*

Lista de Figuras

Lista de Tabelas

Lista de Códigos Fonte

Capítulo 1

Introdução

Estruturas de Dados fazem parte de um sistema visando facilitar o armazenamento, recuperação e manipulação de dados, adequando-se a diferentes tipos de aplicações. Lista, conjunto e mapa são alguns dos exemplos clássicos dessas estruturas. Listas basicamente são usadas para armazenar elementos em uma ordem lógica, armazenados em certas posições, permitindo inserção de elementos repetidos. Conjuntos, por sua vez, armazenam elementos sem necessidade de posicioná-los em locais específicos e não se permite inserir elementos repetidos. E mapas são estruturas de dados onde cada um de seus elementos possuem duas partes chave e valor, e o mapeamento consiste na ideia de que para cada chave existe um valor associado de tal modo que não podem existir chaves duplicadas [5].

A implementação das estruturas de dados pelos desenvolvedores não é uma atividade comum (com exceção de fins didáticos), pois dispende tempo e atenção, além de ser suscetível a erros, comprometendo o desenvolvimento eficiente do software. A reutilização de códigos, em contrapartida, surge como alternativa, visto que contribui para a redução de testes, possibilidades de introdução de erros, tempo e esforço de desenvolvimento de novos softwares. E como os Tipos Abstratos de Dados (TADs) possuem características fixas, grande parte das linguagens de programação já oferecem *frameworks* que as implementam. Estes são definidos como um conjunto de classes que colaboram para realizar uma responsabilidade para um domínio de um subsistema da aplicação, possuindo implementações incompletas que, se estendidas, permitem produzir diferentes artefatos de software [6].

Em se tratando especificamente das estruturas de dados em Java, através de uma hierarquia de interfaces, são disponibilizadas classes que definem os mais variados tipos de

coleções a fim de auxiliar o desenvolvimento de aplicações. As classes prontas existentes no *framework* de coleções Java livram o programador da necessidade de implementar complicadas estruturas de dados para gerenciar coleções de objetos, facilitando, portanto, o desenvolvimento ao viabilizar uma codificação mais rápida, eficiente e padronizada com o reuso de coleções que são adotadas em Java. Alguns exemplos de *frameworks* para coleções Java são: *Apache Commons Collections [7], Guava Libraries [8] e Java Collections Framework (JCF)* [1], sendo este último o mais utilizado e objeto de estudo deste trabalho.

No *framework JCF*, as coleções são definidas e organizadas através de interfaces, além disso são oferecidas classes que implementam as características de cada uma. `Collection` é a interface absoluta na hierarquia de coleções, ela define métodos para adicionar e remover um elemento, verificar se ele está na coleção, entre outras operações. Uma coleção pode implementar diretamente a interface `Collection`, porém normalmente se usa uma das suas subinterfaces, `Set, Queue` (filas) e `List` (listas) que juntamente com `Map` (mapas), que não é subinterface de `Collection`, formam a base das coleções da linguagem Java [1].

Ao planejar um software que utilize coleções, torna-se importante o conhecimento das interfaces de *JCF* para uma aplicabilidade correta. Várias implementações de tais interfaces já são fornecidas, possibilitando assim um desenvolvimento mais eficiente, fazendo uso de soluções estáveis e apropriadas, focando nas funcionalidades principais e nas regras do sistema. Por exemplo, para utilizar uma lista encadeada, existe a classe `LinkedList`, ou quando precisa utilizar um conjunto em estrutura de árvore binária existe a classe `TreeSet`. Então, pode-se fazer uso da implementação que seja mais adequada ao programa em desenvolvimento, como também é possível substituí-la se os requisitos mudarem [5].

Shacham [9] afirma que o uso de uma coleção de maneira inconsistente pode afetar muito o desempenho do código, já que escolhendo certa coleção, torna fixa as dimensões de tempo de operação, utilização de espaço e sincronização. Xu [10] reafirma essa ideia, quando segundo ele em casos reais a escolha da implementação da coleção mais apropriada é um desafio, já que os desenvolvedores normalmente utilizam as mesmas coleções, independente do contexto, não se preocupando com os detalhes da mesma podendo escolher coleções inapropriadas.

Logo, o uso inadequado de classes *JCF* está intimamente ligado com a ineficiência e erros

do sistema, devendo-se substituir as coleções necessárias pelas mais apropriadas dependendo de cada contexto, além de transformar todas as chamadas de método que a nova coleção irá realizar, visto que não há uma coleção que seja melhor aplicável para todas as situações existentes [4].

Substituição e transformação são termos bastante utilizados nesta pesquisa. A primeira é definida, neste documento, como o ato de realizar uma simples troca entre tipos ou instanciações de atributos, por exemplo. Já transformação é uma série de substituições, onde for necessário para que o código se adeque à nova coleção.

1.1 Problema

Um ponto específico dessa problemática, relacionado à realização de transformações no código a fim de substituir coleções pelas mais adequadas, é quando se trata especificamente de `Set` e `List`. Cada uma destas apresentam características específicas e quando usadas inadequadamente também podem repercutir negativamente no entendimento e/ou no desempenho do código.

Estas interfaces foram escolhidas, pois normalmente os programas necessitam de coleções com as características delas, sendo assim mais utilizadas. `List` armazena elementos repetidos e os mantêm em posições (índices), às quais eles foram adicionados. Enquanto que `Set` não permite elementos duplicados, nem utiliza índices, similar a um conjunto matemático [1].

Quando se está em uma situação em que é possível saber a posição (índice) de um elemento é mais vantajoso utilizar `List`, pois sua adição, remoção e pesquisa serão em tempo O(1). Porém quando isso não ocorre, a busca em um `Set` pode ser mais rápida do que em um objeto do tipo `List`. Porque algumas classes que implementam `Set` se utilizam de tabelas de espalhamento (*hash tables*), realizando a operação em tempo (O(1)). Desta forma a utilização de `List` em detrimento de `Set` causará perda de desempenho além de um uso inadequado caso seja necessário armazenar elementos únicos. Outro exemplo ocorre na classe `TreeSet`, que mantém os elementos ordenados do menor para o maior, e pode adicionar, remover e pesquisar em um tempo logarítmico `O(log n)`. Já se fosse utilizar uma lista para manter os elementos ordenados, as inserções e remoções seriam em tempo

linear (`O(n)`), causando além da perda de desempenho um uso inadequado se também for necessário armazenar elementos únicos [5].

`List` pode ser utilizada de forma equivocada em vários casos, talvez pela popularidade de seu uso, sendo normalmente a primeira coleção estudada e por não possuir muitas restrições. Um desses casos especificamente é quando se usa uma lista como um conjunto, evidenciado no código com a execução de uma verificação de não existência do elemento antes que ele seja adicionado na lista, realizando assim o bloqueio de inserção de elementos repetidos.

Em um caso concreto, tem-se um sistema que utiliza uma lista de `disciplinas` onde antes era possível inserir objetos duplicados do tipo `Disciplina` e após uma evolução de código passou a conter uma verificação se o elemento a ser adicionado já existe na lista, como consta no Código Fonte 1.1.

```
//...
private List<Disciplina> disciplinas;
//...

public void adicionarDisciplina(Disciplina disc){
   if(!disciplinas.contains(disc)){
      disciplinas.add(disc);
   }
}
//...
```

Código Fonte 1.1: *Utilizando método `add` de `List` não podendo adicionar elementos repetidos*

Entretanto, o mais adequado seria substituir a coleção `List` por `Set`, caso não seja preciso usar os índices dos elementos da lista em nenhuma outra parte do programa, de acordo com as características de cada uma delas [1], pois `Set` é própria para ser utilizada no armazenamento de elementos únicos. O que modificaria o Código Fonte 1.1 para a forma apresentada no Código Fonte 1.2.

```
//...
private Set<Disciplina> disciplinas;
//...

public void adicionarDisciplina(Disciplina disc){
   disciplinas.add(disc);
}
//...
```

Código Fonte 1.2: *Utilizando método* `add` *de* `Set` *não podendo adicionar elementos repetidos*

Uma das maneiras mais comuns de realizar a transformação de código nesse caso é manualmente, onde o programador substitui as coleções e vai transformando o código em locais necessários. No entanto, tal transformação é susceptível a erros e ineficiência, por ser difícil a identificação dos casos de usos inadequados de coleções assim como as transformações de adequação do código a esta nova coleção, principalmente em sistemas de grande porte. Isso torna necessária a utilização de algum mecanismo que auxilie o programador nesse processo [4].

A automatização é possível, onde uma ferramenta realiza as mudanças no código do cliente sem a necessidade de intervenções manuais do programador, resultando em um código transformado para aquela situação. Porém, nesses casos onde o código original usa `List` e o transformado `Set`, existe uma mudança de semântica e pode existir mais de uma transformação possível. Isso depende do que o programador vai interpretar, necessitando a opinião ou intervenção do desenvolvedor para escolher, dentre algumas possíveis transformações, qual a que mais se adequaria para o contexto do seu software.

Como exemplo de ferramenta que pode ser utilizada automaticamente tem-se *Chameleon* [9], que visa auxiliar o programador na escolha da coleção apropriada para sua aplicação, focando especificamente na implementação. Ela utiliza análise dinâmica para calcular certas métricas de desempenho e capacidade de armazenamento das coleções, que posteriormente podem ser utilizadas em regras pré-definidas para realizar as trocas por coleções consideradas mais eficientes. Estas são utilizadas de forma a consumir menos memória e ter um processamento mais rápido.

Como exemplo, tem-se que após o cálculo das métricas de certo programa, a regra do

Código Fonte 6.1 será avaliada.

```
ArrayList:#contains > X ^ maxSize > Y ! LinkedHashSet
```

Código Fonte 1.3: Regra do Chameleon

Esta significa que quando o número total de chamadas, de uma coleção do tipo *ArrayList*, ao método `contains` for maior que uma constante X, e ainda o número máximo dessa coleção for maior que um certo Y, o tipo da coleção será substituído automaticamente por `LinkedHashSet`. Mas será que sempre que essa regra for verdadeira, é possível trocar uma lista por um conjunto? A resposta é não, pois se nenhum dos `contains` que forem executados tiver um comando de inserção (como um `add`) em seu corpo, não caracteriza o bloqueio de elementos repetidos.

Esses fatos levam a crer que seja interessante usar ferramentas semiautomáticas, onde em determinado momento na execução do código é solicitado que o usuário escolha uma das opções geradas automaticamente e/ou que ele faça uma mínima intervenção manual. Alguns exemplos de ferramentas semiautomáticas são aquelas utilizadas para migração de *Application Programming Interface* (API), as quais consistem em transformar um código que utiliza uma API A e deseja-se substituí-la por uma API B. Sendo a API B uma versão mais atualizada ou uma API similar à API A [11] [12] [13].

Nita e Notkin [2] propuseram uma nova técnica de migração de APIs, *Twinning*, que permitia aos programadores especificarem uma classe de mudanças no programa, sob a forma de um mapeamento, que auxiliará as transformações de duas abordagens: por **adaptadores** e **substituição direta**.

Uso de adaptadores

De acordo com Nita e Notkin [2], uma das maneiras de realizar estas transformações é utilizando adaptadores, que permite que os clientes das APIs possam usar tanto a API original quanto a API destino. O adaptador é uma classe ou interface intermediária, que realiza a chamada desejada.

Em um caso de troca entre duas APIs utilizando adaptadores, possui-se uma classe que faz a chamada da API A e outra que faz a chamada da API B. Elas implementam uma mesma interface, que é inserida no código original. Então, é realizada a instanciação da

classe que utiliza a API desejada. Esta forma de mapeamento também possui o nome de *Deep Adaptation*, pois ela faz uma adaptação "profunda" (*deep*), o que faz o desenvolvedor analisar com mais detalhes o código transformado, para entendê-lo.

Como exemplo de transformação entre APIs utilizando adaptadores, apresentado no trabalho de Nita e Notkin [2], tem-se a Figura 1.1. No código (a) é feita uma varredura nos elementos de um vetor, utilizando um objeto `Enumeration`, imprimindo qual o seu valor. E o código (c) é o código transformado de (a), realizando a mesma ação, mas utilizando adaptadores.

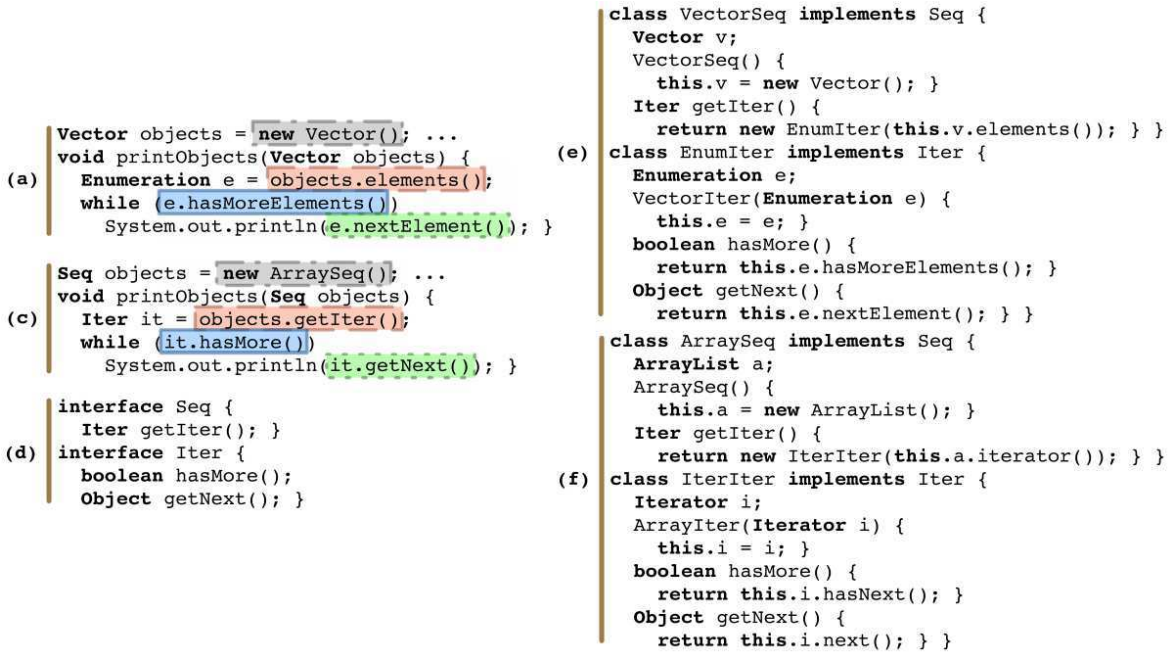

Figura 1.1: *Exemplo de Deep Adaptation [2]*

O resultado é a criação das classes `ArraySeq` e `IterIter` (f) que representam o uso de `ArrayList` (API B) e `VectorSeq` e `EnumIter` (e) que representam o uso de `Vector` (API A). E ainda foram criadas as interfaces `Seq` (d), que é implementada por `ArraySeq` e `VectorSeq`, e `Iter` (d), que é implementada por `IterIter` e `EnumIter`. Todas essas classes e interfaces criadas são adaptadores. As interfaces `Seq` e `Iter` são usadas no código transformado (c) para facilitar o uso das classes `ArraySeq`, `VectorSeq`, `IterIter e EnumIter`, onde para escolher qual classes usar (`Vector` ou `ArrayList`) basta substituir as instanciações dos objetos `Seq e Iter`.

O uso de Adaptadores promove um código mais fácil de manter a longo prazo, pois ele

fica mais desacoplado. Eles também são mais fáceis de serem aplicados semiautomaticamente por se comportarem de forma mais generalizada. Entretanto o código torna-se mais repetitivo, complexo e menos legível [2].

Maia [4] aplica essa abordagem especificamente em transformações entre as coleções de *JCF*, desenvolvendo a ferramenta *Collection Adapter*. De acordo com esta, o programador responde perguntas de alto nível sobre as características da coleção que ele deseja utilizar, não se preocupando em analisar detalhadamente a documentação de *JCF*. Estas perguntas são geradas pela ferramenta através de uma árvore de decisão, que vai selecionando as próximas perguntas baseadas nas respostas fornecidas. Após, é sugerida uma coleção mais adequada para o contexto do sistema em questão, seguida de possíveis adaptações para os trechos do código fonte. Então, o desenvolvedor pode escolher se deseja transformar os trechos de código ou não.

Abaixo segue um exemplo apresentado por Maia [4], onde é considerado o contexto que existe inicialmente um código utilizando um `HashMap`, e após a utilização da *Collection Adapter* é sugerida como coleção mais apropriada um `ArrayList`. Com isso, criando o adaptador `HashMapToArrayList` para ser inserido no código original onde antes continha a classe `HashMap`.

```
//...
public HashMapToArrayList(){
   container=new java.util.ArrayList<E>();
}
//...
public V put(K key, V value) {
   return container.add(value);
}
```

Código Fonte 1.4: *Método `put` do adaptador `HashMapToArrayList` [4]*

Percebe-se que a sintaxe do método `put(E e)` da classe `HashMapToArrayList` é idêntica à da classe `HashMap`, para que no momento da transformação do código não ocorram tantas modificações.

Substituição Direta

Outra abordagem existente para realizar a substituição entre APIs é utilizando Substituição Direta, que é chamada de *Shallow Adaptation* (Figura 1.2). Nesta adaptação ocorre uma substit uição das APIs de forma "superficial" (*shallow*), onde as chamadas de método da API antiga são substituídas para chamadas de método correspondentes na nova API, sem a necessidade de utilização de classes intermediárias.

Como exemplo, tem-se a Figura 1.2, onde (a) é o trecho do programa original, e (b) é o código transformado, resultado da aplicação da substituição direta.

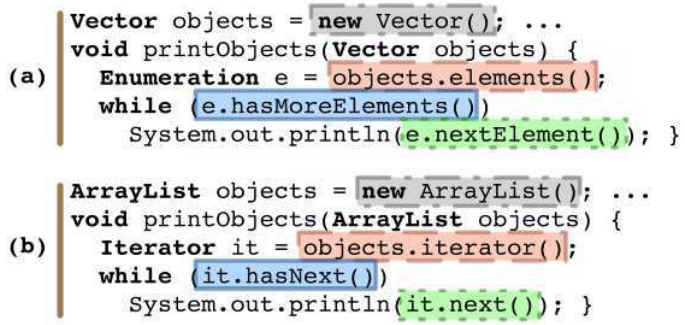

Figura 1.2: *Exemplo de Shallow Adaptation [2]*

O resultado é uma cópia do código original com mudanças das referências da API usada anteriormente para as da nova API. No caso citado, a classe `Vector` é substituída por `ArrayList` e `Enumeration` por `Iterator`. E ainda existem as trocas das chamadas de métodos `objects.elements()` por `objects.iterator()`, `e.hasMoreElements()` por `it.hasNext()` e de `e.nextElement()` por `it.next()`.

O uso da Substituição Direta promove um código mais difícil de manter a longo prazo, mas ela se torna menos custosa em termos de escrita e entendimento de código, pois troca às chamadas de uma API por outra, sem necessidade de classes ou interfaces intermediárias. Esta abordagem pode ser realizada pelo desenvolvedor manualmente ou semiautomaticamente, esta última se torna uma melhor opção por ser menos propícia a erros, porém ela é mais difícil de ser desenvolvida em uma ferramenta por só poder ser transformada em casos bem específicos, em que existam chamadas de métodos correspondentes nas duas APIs [2].

Em se tratando de *JCF* a proposta de uso de adaptadores desenvolvida por Maia [4],

resulta em muita inserção de código gerado, podendo ser indesejado e não bem aceito pelo programador, pois quanto mais códigos forem inseridos maiores são as chances de inelegibilidade e do código ser mais ineficiente.

Como não existe nenhuma abordagem utilizando Substituição Direta no contexto de *JCF*, o foco desta pesquisa é justamente este. Considerando a problemática do uso de um `List` sendo usado como um `Set`, e visando obter códigos mais legíveis e sem classes intermediárias (adaptadores), o uso de Substituição Direta pode ser um método possível para solucionar essas questões.

1.2 Objetivo

Nesta pesquisa pretende-se investigar as condições e consequências de um conjunto de transformações que realizam substituições entre as interfaces `List` e `Set`, quando uma lista está sendo usada como um conjunto (como nos Código Fonte 1.1 e Código Fonte 1.2), a fim que esse passe a utilizar a coleção mais adequada para o contexto.

A Questão de Pesquisa principal que este trabalho pretende responder é: **quais as condições e consequências das transformações nos códigos ao realizar substituições entre as interfaces `List` e `Set`, quando uma lista está sendo usada como um conjunto?**

1.2.1 Objetivos específicos

- Definir *templates* de transformações que modifiquem um código que usa uma lista como um conjunto
- Implementar tais transformações utilizando a ferramenta Spoofax [14]
- Possibilitar a replicação de transformações de trechos de código similares para realizar substituições entre outras coleções *JCF*.
- Analisar as consequências da aplicação das transformações em softwares reais.
- Investigar a percepção de desenvolvedores Java e *JCF* em relação às consequências das transformações aplicadas em pequenos trechos de código.

1.3 Solução

Para investigar as condições e consequências de transformações entre as interfaces `List` e `Set`, quando uma lista está sendo usada como um conjunto, utiliza-se a abordagem de Substituição Direta sendo implementadas num ambiente de reescrita de programas, utilizado o *framework* Spoofax [14], em especial sua linguagem auxiliar para definição de reescrita de programas, *Stratego*.

Visando assim, auxiliar o desenvolvimento de técnicas e ferramentas que sejam usadas pelos desenvolvedores ao realizarem substituições entre as tais interfaces, e as transformações dos demais trechos de código envolvidos com estas (incluindo métodos e atributos) com maior grau de adequabilidade, legibilidade, facilidade e eficiência. E estas transformações podem ser replicadas para as demais coleções de `JCF`.

Devido à grande complexidade em analisar as consequências das transformações de acordo com as dimensões de tempo de operação, utilização de espaço e sincronização, as análises são focadas na adequabilidade, legibilidade e complexidade dos códigos resultantes das transformações investigadas.

1.4 Contribuições

As contribuições que este trabalho proporciona são:

- Transformações que realizam substituições adequadas entre as interfaces `List` e `Set`, e transformam variáveis, atributos e métodos relacionados, a fim de promover uma adequação no uso das coleções.
- Transformações alternativas que visam realizar um melhoramento ainda maior no código original em relação às transformações "convencionais" citadas acima
- Um *survey* através do qual foi obtidas opiniões dos programadores e solicitado que eles fizessem mudanças em trechos de código utilizando listas.
- Um Estudo de Caso para avaliar a aplicabilidade das transformações em sistemas reais.
- Implementação de um protótipo de ferramenta de suporte em *JDT* [15]. Ele foi utilizado no estudo de caso para filtrar os projetos participantes que continham classes

que possuíam listas sendo usadas com conjuntos, se baseando em análise estática do código fonte dos projetos.

- Extensão na classificação dos tipos de substituições proposta por Maia [4], adicionando mais uma categoria, pois foi descoberto um caso que não se encaixava em nenhuma das categorias inicialmente citadas.

Através do Estudo de Caso realizado para avaliar a aplicabilidade das transformações em sistemas reais foi obtido que 14% dos projetos participantes do estudo apresentam situações em que é necessária à substituição do tipo de um objeto de `List` para `Set`, pois estas listas possuem um comportamento de conjunto. Porém, alguns projetos não podem ter suas classes transformadas devido a problemas no uso de padrões de projeto ou porque utilizam o índice das lista em outras partes do código, o que resultou em 8% que foram efetivamente transformados, do total inicial de projetos. Das listas que tiveram seus tipos substituídos para `Set` e seu restante do código transformado, 90% eram do tipo `ArrayList` e apenas 10% `List`. E também percebe-se que as transformações realizadas nunca inseriam mais linhas no código original e modificaram no máximo 3 linhas por atributo ou método transformado.

Já no *survey*, aplicado sob a forma de questionário não supervisionado através da internet, onde os participantes, apresentaram experiência considerável em Java e em *JCF* com mais da metade possuindo entre 4 e 12 anos de experiência, deveriam dar suas sugestões de mudanças acerca de dois trechos de código e posteriormente compará-las com transformações propostas. Cerca de 45% dos participantes propuseram sugestões consideradas adequadas, de acordo com a API e *JCF* [1], ao primeiro trecho de código analisado e 25% deles para o segundo. O primeiro utilizava o método `add(E e)`, já o segundo `set(int index, E e)` ambos pertencentes à interface `List`. Também foi evidenciada uma melhoria na legibilidade dos códigos transformados propostos pelo autor da pesquisa em relação aos originalmente apresentados. E ainda foi obtida a informação que 58% dos participantes não pesquisaram nem na API de JCF, nem em outras fontes para responder ao questionário, contra 26% que utilizaram API e 16% que pesquisaram sobre *JCF* em outras fontes.

1.5 Estrutura do Documento

Este documento está estruturado da seguinte forma: no **Capítulo 2 - Fundamentação Teórica** são apresentados conceitos necessários para o entendimento do trabalho. Tais conceitos abrangem Evolução de *Software*, *framework* do *Java Collections* e Migração de APIs. Já no **Capítulo 3 - Transformações de listas usadas como conjuntos** é explicado como é o estudo através das transformações que realizam a substituição das interfaces `List` e `Set` no problema em questão, que estão implementadas usando uma ferramenta de reescrita. No **Capítulo 4 - Avaliação por Estudo de Caso:** tem-se a primeira avaliação das transformações, realizada através de um Estudo de Caso com sistemas reais e no **Capítulo 5 - Avaliação por *Survey*** tem-se a segunda avaliação, realizada através de questionário respondido por programadores Java sobre a possível substituição de coleções em dois trechos de código. Também é apresentado o **Capítulo 6 - Trabalhos Relacionados**, que discute trabalhos da literatura que são relacionados a este, apontando semelhanças e diferenças. E por fim, o **Capítulo 7 - Conclusão** apresenta as contribuições da pesquisa desenvolvida, trabalhos futuros e considerações finais sobre o trabalho.

Capítulo 2

Fundamentação Teórica

Para melhor compreensão deste documento são apresentados os conceitos de alguns tópicos relacionados a esta pesquisa. Na Seção 2.1 é explanado sobre o conceito de Evolução de Software e quais tipos de manutenção de software existem; na Seção 2.2 é apresentada uma descrição sobre o *Java Collections Framework*, detalhando as características das principais coleções do *framework* e como é feito o uso adequado das interfaces; e na Seção 2.3 são descritas duas formas de realizar substituições entre APIs, além de técnicas e ferramentas existentes para realizar a migração de APIs e substituição entre coleções.

2.1 Evolução de Software

Evolução de Software é a constante manutenção e crescimento do sistema para que ele continue útil. "Software que não é tolerante a modificações está fadado ao abandono ou a substituição" [16] [17]. Outra evidência da importância da Evolução de Software é que mais de 90% dos custos de um sistema é proveniente da fase de manutenção. Necessitando assim de uma grande atenção na nessa fase do ciclo de vida do software [18].

Existem quatro tipos diferentes de manutenção de *software*: corretiva, adaptativa, preventiva e perfectiva. Na corretiva são feitas mudanças no código a fim de corrigir erros existentes. Já na adaptativa ocorrem mudanças no código para se adaptar a certo ambiente como hardware ou sistema operacional. A manutenção preventiva é aquela realizada no software a fim de detectar e corrigir problemas latentes, antes que estes se tornem erros. Por fim, a manutenção de software perfectiva ocorre quando se deseja modificar apenas a estrutura

do código, para que ele tenha uma qualidade melhor de acordo com critérios estabelecidos pelos desenvolvedores, mantendo o mesmo funcionamento anterior [19].

Sistemas desenvolvidos em Java, que estão em produção já há algum tempo, normalmente passam por várias das manutenções citadas, ocorrendo com mais frequência as manutenções adaptativas e perfectivas (cerca de 75%) e em torno de 21% corretivas [20]. Como as classes que utilizam coleções *JCF* pertencem ao sistema, elas também sofrem parte das manutenções ocorridas. Seja para corrigir um defeito imediato (bug), seja para substituir a coleção antiga por uma que se adeque mais ao novo contexto.

2.2 *Java Collections Framework (JCF)*

Um *collection framework* é uma arquitetura unificada para representar e acessar coleções, que são objetos destinados a armazenar outros objetos e posteriormente poder manipular tais objetos de forma facilitada. Ele objetiva manter utilizável implementações confiáveis, e bem testadas, porém seu objetivo principal é a facilitação do desenvolvimento por meio de reutilização do código, promovendo a recuperação e manipulação de dados. Em Java, existe o *JCF* que possui grande aplicabilidade com estruturas de coleção ricas, que reduzem o esforço de programação, fornecendo estruturas de dados e algoritmos úteis, juntamente com implementações de alto desempenho. Sua API provê vários Tipos Abstratos de Dados(TADs): Mapas, Conjuntos, Listas, Árvores, Matrizes e outras coleções. Os TADs são geralmente representados por interfaces: Mapa para interface *Map*, Conjunto para a interface *Set*, Lista para interface *List*, etc. [21] [5].

Para cada interface, tem-se ao menos uma implementação desenvolvida de acordo com as características do TAD que representa. Por exemplo, para a interface *List* existem as implementações *ArrayList, LinkedList, Vector e Stack*, onde um `Vector` é bem similar ao `ArrayList` possuindo tamanhos fixos, que aumentam dinamicamente quando o tamanho máximo é atingido. Já as diferenças são que a primeira é sincronizada enquanto a segunda não, e quando o tamanho máximo de elementos é atingido ele é aumentado em 100% na primeira enquanto na segunda é 50%. Por sua vez, `LinkedList` implementa uma lista duplamente encadeada, possuindo piores performances nos métodos `get(int index)` e `set(int index, E e)` em comparação com ArrayList, porém com melhores per-

formances nos métodos `add(E e)` e `remove(E e)`. E a classe `Stack` representa a implementação de uma pilha de objetos, LIFO (last in first out). Onde o método de inserção é o `push(E e)`, que insere um item no topo da pilha e o método `pop()` para remover o elemento do topo da pilha [1].

Na hierarquia das classes e interfaces de *JCF* temos como interface principal `Collection`, que herda da interface `Iterable`, o que significa que o uso de um iterador pode ser feito em qualquer objeto de `Collection`. Das interfaces que representam os principais TADs apenas `Map` não herda de `Collection`, `Set`, `List` e `Queue` herdam dela. A partir de cada uma dessas interfaces tem-se novas classes e interfaces que implementam ou herdam delas. A Figura 2.1 mostra com mais detalhes esta hierarquia.

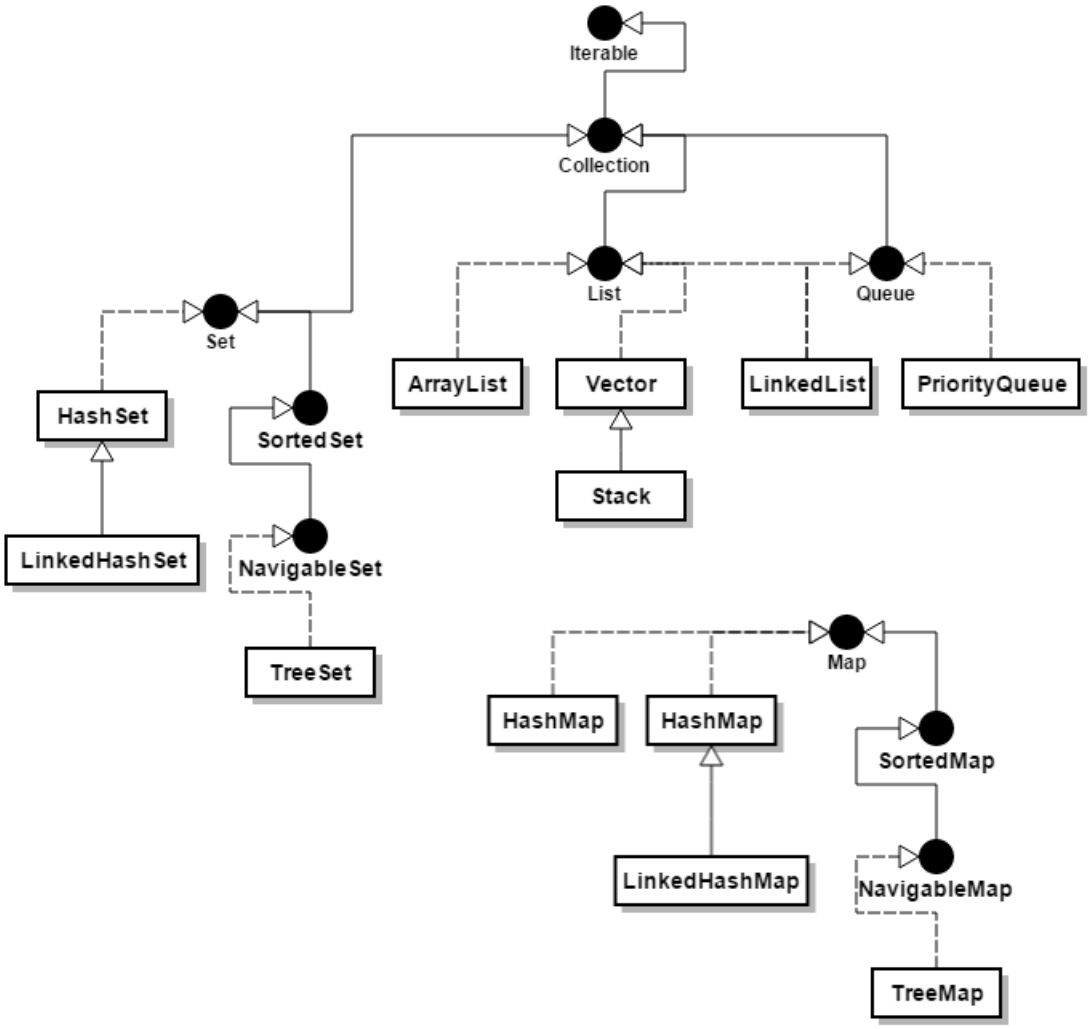

Figura 2.1: *Conjunto principal de classes e interfaces que forma o JCF. [3]*

Com uma quantidade considerável de classes e interfaces a serem utilizadas, vê-se a necessidade do programador conhecer bem as que ele vai utilizar. De acordo com Xu [10], os programadores tendem a utilizar sempre as mesmas coleções, normalmente as mais conhecidas e gerais. Principalmente no paradigma de orientação a objeto, onde é possível escrever código utilizando *frameworks* e bibliotecas sem entender profundamente os detalhes, devido ao encapsulamento e reutilização de código.

Já Maia [4] também afirmou que os programadores podem escolher coleções inadequadas, que pode acontecer devido à documentação da JCF ser insuficiente ou por falta de cuidado dos programadores não levando em consideração as exigências do contexto do seu projeto.

2.2.1 Uso adequado das interfaces

Com uma escolha adequada de qual coleção ser utilizada em cada parte de um sistema, este se torna bem mais eficiente em termos de tempo de operação, utilização de espaço e sincronização. O uso da coleção de maneira não consistente pode causar uma degradação significativa do desempenho [9]. Então, é interessante conhecer bem cada coleção do *JCF*

O *framework* possui como interface principal `Collection` que herda de `Iterable`. A partir de `Collection` existem as interfaces filhas que representam os TADs: `List, Set, Queue e Deque`, as quais (juntamente com `Map`) precisam ter suas características bem conhecidas para utiliza-las na situação que mais se adeque.

A interface genérica `Iterable` permite que a coleção que a instancie possa conter objetos que podem ser encontrados através de uma instrução *foreach*, o que significa dizer que esta coleção é iterável. Enquanto que, `Collection` representa uma coleção genérica, onde qualquer coleção (com exceção de `Map`) precisa instanciá-la. Ela é a interface raiz da hierarquia, herdando as características da interface `Iterable`, tornando qualquer coleção iterável. Não é possível realizar a implementação direta desta interface, somente as interfaces que vêm abaixo na hierarquia.

Já a interface `List` representa o TAD Lista, e nela são armazenados elementos repetidos e indexáveis. A utilização desses índices possibilita o acesso e remoção de elementos de forma extremamente rápida. A lista também preserva a ordem de inserção dos elementos.

Ao contrário da interface `List`, em `Set` não é possível armazenar elementos repetidos,

a própria coleção bloqueia internamente na instrução de inserção. E também não possuem índices em seus elementos, se assemelhando ao TAD Conjunto.

A interface `Queue` representa uma Fila, que normalmente possui a ordenação de seus elementos com uma política FIFO (*first-in, first-out*), porém existem exceções como filas de prioridade, que ordenam elementos de acordo com um comparador fornecido.

Por sua vez, a interface `Deque` representa uma coleção que permite ordenar seus elementos tanto pela política FIFO (first-in, first-out) como pela LIFO (last-in, first-out). Permitindo assim a inserção, remoção e recuperação de elementos pelos dois lados da coleção (cabeça ou calda).

Por fim, com a interface *Map*, que não implementa *Collection*, é possível inserir elementos (valores) que se relacionam/mapeiam com chaves. Uma restrição que existe neste tipo de coleção é a impossibilidade de conter chaves duplicadas, além de ser aconselhável o uso de objeto não mutável nas chaves. Temos ainda a interface *SortedMap* que implementa a interface *Map* e também tem uma característica adicional de manter suas chaves em ordem crescente ou decrescente.

2.2.2 Substituições entre coleções

O ato de substituir uma coleção, que esteja em funcionamento em determinado código por outra que seja mais apropriada, representa uma manutenção de software adaptativa ou preventiva, já que esta ação é feita para adaptar o sistema a um novo contexto ou para prevenir possíveis problemas de desempenho e/ou utilização de espaço. Além de realizar a substituição é necessário transformar outras partes do código, como variáveis, atributos e métodos relacionados à coleção substituída, para que estas se adequem à nova coleção.

Um desenvolvedor que necessita substituir coleções tem as seguintes alternativas: realizar a substituição das coleções e suas posteriores transformações de código manualmente, utilizando alguma ferramenta que faça esta troca e transformação dos códigos de forma automática (ou automatização total) ou de forma semiautomática (ou automatização parcial). A forma manual é desencorajada, pois é ineficiente e susceptível a inserção de outros erros no sistema. Já a automatização total vai modificar o código e o programador não terá que realizar nenhuma escolha ou intervenção no código durante o processo, diferentemente da automatização parcial que vai solicitando permissões para realizar as mudanças e dando ou-

tras opções ao programador, baseadas no *feedback* que ele passar para ferramenta. E ainda pode existir algum momento em que o programador precise realizar modificações manuais no código [2].

Pela pouca quantidade de ferramentas específicas para realizar essas substituições entre coleções *JCF*, e posteriormente transformações de código, pode-se fazer uso, em alguns casos, de ferramentas que são aplicadas no processo de Migração de APIs.

2.3 Migração de APIs

Migração de API é o processo de transformar o código de determinado software que utiliza uma certa API A, que precisa ser substituída por uma API B, sendo esta API B uma API semelhante ou uma nova versão da API A. Estas transformações são necessárias para evitar que o software cliente da API A adquira incompatibilidade com a API B.

Devida a dificuldade de adequação de um código cliente a uma nova versão de API, algumas APIs lançam métodos *deprecated* ou desencorajam a mudança da API. O que torna o sistema desatualizado. Para sanar esse problema algumas técnicas e ferramentas são usadas para auxiliar o programador.

2.3.1 Métodos para Migração de APIs

No trabalho de Nita e Notkin [2] são definidos dois métodos que podem ser utilizados para realizar a migração de APIs: Substituição direta e Adaptadores.

Substituição direta

Em um processo de migração de uma API A para uma API B, este método consiste na criação de uma cópia do código original que utiliza a API A, onde as chamadas de método são substituídas para aquelas correspondentes na API B. Essa abordagem é chamada *Shallow Adaptation* (ou adaptação rasa). Como exemplo, tem-se A Figura 2.2, onde (a) é o trecho do programa original, e (b) é o código transformado, resultado da aplicação da substituição direta.

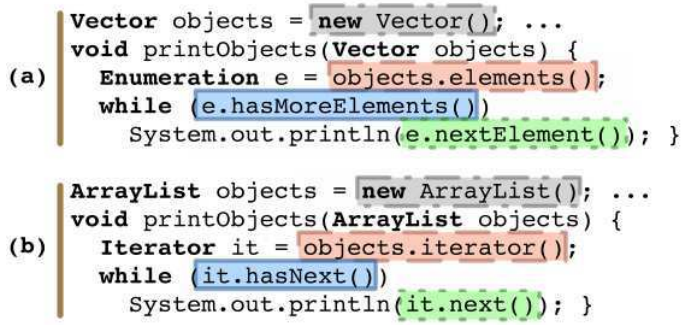

Figura 2.2: *Exemplo do método Shallow Adaptation [2]*

Neste caso, a classe `Vector` é substituída por `ArrayList` e `Enumeration` por `Iterator`. E ainda existem as trocas das chamadas de métodos `objects.elements()` por `objects.iterator()`, `e.hasMoreElements()` por `it.hasNext()` e de `e.nextElement()` por `it.next()`.

Este método de migração de APIs faz o mínimo de modificações no código original, não necessitando de geração de códigos auxiliares, não aumentando assim a complexidade do código bem como não o deixa menos legível nem duplicado [2].

Adaptadores

Em um processo de migração de uma API A para uma API B, este método consiste na criação de uma API intermediária, API C, onde o código cliente que utilizava a API A, passará a usar a API C. Então, esta ficará encarregada de redirecionar a chamada dos métodos envolvidos para a API B, que é a que se deseja usar. Porém, caso em algum momento decida-se voltar a usar a API A, está pode ser utilizada sem ser preciso fazer outras grandes modificações no código. Bastando a API C redirecionar as chamadas para a API A. A API C pode ser criada na própria biblioteca ou no código do cliente.

Na Figura 2.3 contém um exemplo de utilização de adaptadores na migração de APIs. No código (a) é feita uma varredura nos elementos de um vetor, utilizando um objeto `Enumeration`, imprimindo o seu valor. E o código (c) é o código transformado de (a), realizando a mesma ação, mas utilizado adaptadores.

Na adaptação são utilizadas as classes `ArraySeq e IterIter` (f), que implementam respectivamente as interfaces `Seq e Iter` (d), que também são utilizadas na transforma-

ção. Ainda tem o código (e) contendo os adaptadores da API original, caso queira utilizar as implementações `Vector` e `Enumeration` novamente.

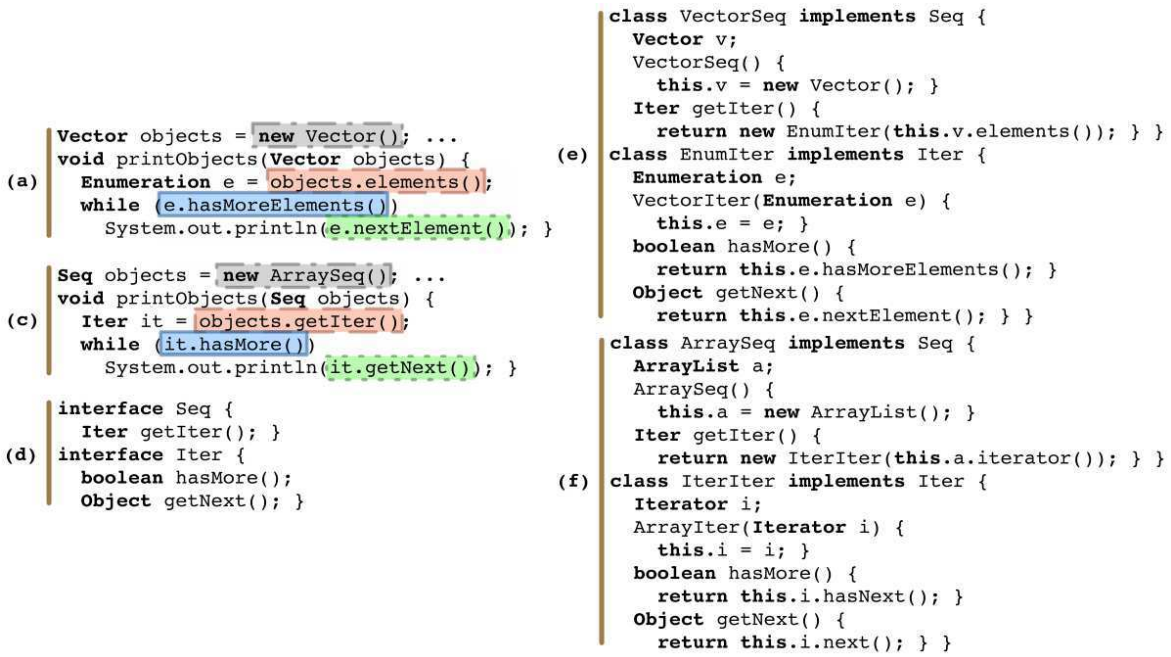

Figura 2.3: *Trecho de código retirado de Nita e Notkin [2] representando a técnica de Deep Adaptation*

Adaptadores promovem um código mais fácil de manter a longo prazo, pois ele fica mais desacoplado. Entretanto o código se torna mais repetitivo, complexo e menos legível. E ainda existe outro problema nessa abordagem, que ocorre quando o programa possuir n tipos na API A, deverão ser criadas `n` tipos de interfaces e o dobro de `n` de novas classes implementando essas interfaces. Dependendo da quantidade de tipos a serem migrados, pode haver uma sobrecarga sobre o código do cliente com a adição dessas novas classes e interfaces [2].

2.3.2 Ferramentas de Migração de APIs

ReBA [22] é um exemplo de ferramenta totalmente automatizada que gera uma camada intermediária entre a nova versão da API e a antiga. Sendo assim também um exemplo de ferramenta que utiliza adaptadores.

Outro exemplo é encontrado no trabalho Balaban et al. [23], que criou uma técnica automática de mapeamento onde o uso de classes legadas são relacionados para o uso de classes

a serem substituídas. O mapeamento é definido uma vez, podendo ser reutilizado em várias aplicações.

Como exemplos de ferramentas semiautomáticas (automatização parcial) existe o *Diff-CatchUp*, que juntamente com outra ferramenta chamada *UMLDiff*, identifica as mudanças nos *design-levels* das duas versões das APIs e apresenta ao programador um conjunto de propostas de transformação de código para que ele selecione a melhor opção para o seu caso específico [24].

Outro exemplo simples, mas que também se aplica nesse contexto, é o *Java Search* do próprio eclipse, que indica ao programador os locais referentes a alguma pesquisa no projeto. A posterior modificação ou não do código fica a cargo do desenvolvedor.

2.3.3 *Collection Adapter*

As ferramentas de migração de API não podem ser utilizadas com sucesso em todos os casos nas trocas entre coleções JCF, pois as classes e interfaces desse *framework* possuem características e comportamentos peculiares, que precisam ser levados em consideração no momento da transformação. Por isso é proposto por Maia [4], uma ferramenta chamada *Collection Adapter*, que utiliza adaptadores para realizar as transformações inerentes às substituições de coleções JCF.

Nesta ferramenta o programador responde perguntas de alto nível sobre as características da coleção que ele deseja utilizar, não se preocupando em analisar detalhadamente a documentação de *JCF*. Estas perguntas são geradas pela ferramenta através de uma árvore de decisão, que vai selecionando as próximas perguntas baseadas nas respostas fornecidas. Como por exemplo, se para as opções *"Contém elementos duplicados"* e *"Não contém elementos/chaves duplicadas"* se o programador selecionar a segunda opção, sabe-se que ele não deseja `List` nem `Queue`. Logo ele é redirecionado para as opções: *"Possui um elemento chave que servirá para mapear outro elemento"* e *"Possui apenas um conjunto linear de elementos"*, caso ele marque a primeira, a coleção mais adequada é um `Map`, caso contrário é um `Set`.

Após a sugestão da coleção mais adequada para o contexto do sistema em questão, são sugeridas possíveis adaptações para os trechos do código fonte. Então, o desenvolvedor pode escolher se deseja transformá-lo ou não.

O tipo de adaptador selecionado pelo *Collection Adapter* dependerá do tipo de substituição entre coleções *JCF* envolvido. Maia [4] definiu quatro tipos diferentes de substituição de chamadas de métodos entre duas classes ou interfaces: a que o programador utilizou a priori e outra que é mais adequada ao contexto do programa.

Dados **C** o conjunto de coleção existentes no JCF no qual $C_1 \epsilon C$ e $C_2 \epsilon C$, sendo $m_1()$ representando um dos métodos pertencentes a C_1 e $m_2()$ um dos métodos pertencentes a C_2, teremos a seguinte classificação dos métodos das classes envolvidas quanto à substituição:

- Tipo 1: $m_1()$ e $m_2()$ possuem a mesma sintaxe e semântica
- Tipo 2: $m_1()$ e $m_2()$ possuem a mesma semântica, mas têm sintaxe diferentes
- Tipo 3: $m_1()$ e $m_2()$ possuem a mesma sintaxe, mas têm semântica diferentes
- Tipo 4: $m_1()$ e $m_2()$ não possuem a nem sintaxe nem e semântica iguais

Entende-se por métodos com a mesma sintaxe aqueles que possuem o mesmo identificador, os mesmos parâmetros de entrada e de retorno, a mesma visibilidade e os mesmos modificadores. Como exemplo, existem os métodos `public boolean add(E e)` da classe `ArrayList` e `public boolean add(E e)` de `TreeSet`. Já métodos com mesma semântica são aqueles que possuem o mesmo efeito sobre a coleção independente da sintaxe envolvida, resultando nos mesmos elementos ao final da(s) chamada(s) desse(s) método(s) possuindo os mesmos elementos envolvidos, sem importar a ordem em que estes estão na coleção. Como por exemplo, os métodos `public boolean add(E e)` da classe `ArrayList` e `public void addLast(E e)` de `LinkedList`, onde ambos inserem um elemento no final da lista.

Para cada um dos quatro tipos de substituição entre coleções *JCF* existem diferentes formas de adaptação.

Adaptador do Tipo 1: Métodos sintática e semanticamente iguais

Neste tipo de substituição pode-se fazer a troca direta de uma coleção por outra, apenas redirecionando a chamada do método já que os métodos possuem a mesma sintaxe e a mesma semântica. Como exemplo, o adaptador entre as coleções `ArrayList` (C1) e `LinkedList` (C2), que é chamado de `ArrayListToLinkedList` (C1ToC2). Existe um método `add`

`(m1()')` com a mesma assinatura do mesmo método do `ArrayList (m1())`. A implementação do método `add` do adaptador está representado no Código Fonte 2.1.

```
//...
public ArrayListToLinkedList(){
   container=new java.util.LinkedList<E>();
}
//...
@java.lang.Override
public boolean add(E arg0) {
   return container.add(arg0);
}
```

Código Fonte 2.1: Método `add` do adaptador `ArrayListToLinkedList` [4]

Adaptador do Tipo 2: Métodos sintaticamente diferentes e semanticamente iguais

Já na substituição do tipo 2 é necessário que a sintaxe do método a ser inserido se adapte de tal forma que mantenha a mesma da coleção que foi substituída. Por exemplo, na adaptação de uma coleção `LinkedList` para uma coleção `ArrayList`, o método `addLast` do `LinkedList` (Código Fonte 2.2) possui a mesma semântica do método `add` do `ArrayList`, pois ambos adicionam o novo elemento no final da lista. Mas, os métodos possuem sintaxes diferentes, já que suas assinaturas diferem. O trecho do adaptador referente à adaptação de sintaxe, preservando a semântica, está exibido no Código Fonte 2.3.

```

public void addLast(E e) {...}
```

Código Fonte 2.2: *Método addLast do LinkedList*

```
//...
public LinkedListToArrayList(){
   container=new java.util.ArrayList<E>();
}
//...
@java.lang.Override
public void addLast(E arg0) {
   container.add(size(),arg0);
}
```

Código Fonte 2.3: *Método addLast do adaptador LinkedListToArrayList*

Adaptador do Tipo 3: Métodos sintaticamente iguais e semanticamente diferentes

Neste tipo deve ter um cuidado adicional ao realizar a adaptação, pois ela só é realizada com alteração na semântica. Como pode-se ver na substituição do método `iterator` da coleção `ArrayList` (Código Fonte 2.4) para o método `iterator` da coleção `HashSet` (Código Fonte 2.5). Os métodos possuem a mesma sintaxe mas, semânticas diferentes, já que não possuem o mesmo efeito sobre as coleções, pois o primeiro itera sobre a coleção pela ordem de inserção e segundo não possui ordem pré-definida.

```
public Iterator<E> iterator() {..}
```

Código Fonte 2.4: *Método iterator do `ArrayList`*

```
public Iterator<E> iterator() {...}
```

Código Fonte 2.5: *Método iterator do `HashSet`*

Uma possível adaptação para esta situação é vista no Código Fonte 2.6, onde é considerado que o método `iterator` do `ArrayList` se comportará como o método `iterator` do `HashSet` no adaptador. Porém, podem existir casos em que a adaptação não aconteça, pelo desejo do programador em não realiza-la, devido à ocorrência de mudança na semântica neste tipo de transformação.

```
//...
public ArrayListToHashSet(){
   container=new java.util.HashSet<E>();
}
//...

@java.lang.Override
public java.util.Iterator<E> iterator() {
   return container.iterator();
}
```

Código Fonte 2.6: *Método `iterator` do adaptador `ArrayListToHashSet`*

Adaptador do Tipo 4: Métodos sintaticamente e semanticamente diferentes

Este é o tipo mais difícil de adaptação, pois além de ter o cuidado adicional do caso anterior, por possuir semânticas diferentes, a sintaxe também muda. Porém, apesar das diferenças, assume-se que exista algum nível de semelhança semântica entre os métodos que torna possível relacioná-los.

A adaptação entre a coleção `HashMap` e a `ArrayList` é um exemplo, pois possuem os métodos `put`(Código Fonte 2.7) e `add` respectivamente, que apesar de serem diferentes em semântica e sintaxe têm um certo nível de relacionamento semântico se consideramos a ação de acrescentar um objeto à coleção presente em ambos. Mas, se for considerado que se pode fazer a adaptação, é preciso desconsiderar a chave que está associada ao objeto, que antes estava em um mapa e agora estará numa lista.

Assim passam a existem pelo menos duas soluções neste caso: considerar que não existe nenhuma relação semântica entre os métodos e não realizar a adaptação ou considerar que existe essa relação e que a adaptação é realizada com a perda semântica da chave de cada elemento do `HashMap` perdendo esses dados. No Código Fonte 2.8, pode-se verificar esta adaptação.

```
public V put(K key, V value) {...}
```

Código Fonte 2.7: *Método `put` do `HashMap`*

```
//...
public HashMapToArrayList(){
   container=new java.util.ArrayList<E>();
}
//...
public V put(K key, V value) {
   return container.add(value);
}
```

Código Fonte 2.8: *Método put do adaptador HashMapToArrayList*

2.4 Considerações Finais do Capítulo

Neste capítulo foram apresentadas as definições de alguns tópicos relacionados a esta pesquisa a fim de facilitar o entendimento deste trabalho. Alguns conceitos que não foram abordados neste capítulo serão apresentados nos capítulos seguintes, onde esses são usados e explicados especificamente no contexto deste trabalho, como *Minimal Core Java*, Spoofax e Equivalência Semântica Fraca.

Capítulo 3

Transformações de listas usadas como conjuntos

Diante da problemática exposta no Capítulo 1, neste capítulo é demonstrado o estudo realizado nesta pesquisa. Ele investiga as condições e consequências de transformações entre as interfaces `List` e `Set`, quando uma lista está sendo usada como um conjunto.

Para melhor compreensão são definidos *templates* de transformações entre as interfaces `List` e `Set`, que são implementados em uma ferramenta de reescrita, para poder observar mais detalhes sobre cada transformação. Esses *templates* de transformações são aplicados sob um *Minimal Core Java* que pode ser melhor entendido na próxima seção.

3.1 *Minimal Core Java*

Como são apresentadas nesta pesquisa transformações codificadas em Java, e esta é uma linguagem bastante complexa, existindo vários trechos de código diferentes para realizar a mesma ação semântica, se viu necessária à definição e utilização de uma linguagem reduzida: um *Minimal Core Java*. Este que também pode ser chamado de *Featherweight Java* (FJ) [25]), é uma linguagem reduzida de Java. Ela não possui todas as instruções e recursos de Java, que são descartadas para simplificar o contexto em questão e poder focar a análise sobre propriedades definidas como principais. Caso esta não fosse utilizada, tornaria os tipos de transformações de código muito extensas, e com muitas transformações similares.

Como exemplos de comandos equivalentes tem-se `while, do-while e for` que

são utilizados para representar um comando de repetição e podem, com algumas adaptações, serem substituídos um pelo outro. Mais um exemplo é o comando `if` que pode substituir o comando `switch-case`, e ainda o `if-else` que representa o operador condicional `? :`, que é escrito em uma só linha. Como é visto no Código Fonte 3.1 e Código Fonte 3.2.

```
max = (a > b) ? a : b;
```

Código Fonte 3.1: *Exemplo do uso do operador ?:*

```
if (a > b) {
  max = a;
}else {
  max = b;
}
```

Código Fonte 3.2: *Representação com `if-else`*

3.1.1 Definição do *Minimal Core Java* para esta pesquisa

A linguagem ficou definida da seguinte forma:

- Cada método pode conter somente um comando de retorno
- Os comandos de seleção utilizados são `if` ou `if/else`. Caso seja utilizado outro comando de seleção, deve-se trocar por um que seja aceito pela linguagem.
- O comando de repetição é o `while`, caso seja utilizado outro comando de repetição, como o `do-while` ou `for`, a troca para o `while` deve ser realizada.
- A expressão de verificação do `if` ou do `while` devem ser atômicas (com apenas uma expressão booleana). Caso exista uma expressão composta (com vários *ANDs* ou *ORs*) esta deve ser representada na forma de `ifs` e/ou `whiles` aninhados.
- Não é suportado `Enumeration`
- Não há suporte a *threads*

3.2 Condições para as transformações

Nesta seção são apresentadas as condições para realizar as transformações entre as interfaces `List` e `Set`, quando uma lista está sendo usada como um conjunto. A priori, a identificação

dessa situação através do código fonte se dá pela existência de algum bloqueio da inserção de um elemento que já existe naquela lista, através de um comando condicional, normalmente um `if`. Nesse comando comumente utiliza-se o método `contains(element)` ou `containsAll(collection)` e neste documento eles são tratados com **IfContains**, Seção A.2, que possui a seguinte definição: Dado C' o conjunto de todos os comandos **IfSimples** da linguagem reduzida de Java, tem-se que C'' é um outro conjunto de comandos **IfSimples** o qual $c'' \epsilon C''$ e $c'' \epsilon C'$, onde a expressão de verificação de c'' é: (1) a verificação de existência de um objeto o em uma coleção cl, ou (2) a verificação de existência de uma outra coleção cl_o em uma coleção cl. E caso o ou todos os objetos de cl_o não exista(m) em cl é executado os comandos b, que estão dentro do corpo de c''. Como é possível ver nos *templates* contidos no Código Fonte 3.3 e Código Fonte 3.4.

```
if(cl.contains(o)){
  b
}
```

Código Fonte 3.3: *Template para o IfContains utilizando o método contains*

```
if(cl.containsAll(clo)){
  b
}
```

Código Fonte 3.4: *Template para o IfContains utilizando o método containsAll*

E o **IfSimples** por sua vez, possui como definição: Dado C o conjunto de todos os comandos possíveis na linguagem reduzida de Java, tem-se que C' também é um conjunto de comandos, o qual $c' \epsilon C'$ e $c' \epsilon C$, onde c' possui uma expressão de verificação v, que caso seja verdadeira é executado um novo conjunto de comandos b que estão dentro do corpo de c'. Como pode-se visualizar no Código Fonte 3.5

```
if(v){
  b
}
```

Código Fonte 3.5: *Template para o IfSimples*

Existem cinco métodos em `List` que inserem pelo menos um elemento na lista. São eles: `add(element)`, `add(index, element)`, `addAll(collection)`, `addAll(index, collection)` e `set(index, element)`. Eles correspondem respectivamente a inserir `element` no fim da lista, inserir `element` na posição `index`

da lista, inserir todos os elementos de `collection` na lista, inserir todos os elementos de `collection` na lista a partir da posição `index` e substituir o elemento que está na posição `index` da lista por `element`. Exemplos dessas situações são demonstrados nas seções seguintes deste capítulo. Então, quando um código possuir um objeto `List` que utilize algum desses cinco métodos citados anteriormente dentro do corpo de um IfContains, o mais apropriado, em algumas situações, é esse objeto ser substituído por um `Set`.

Entretanto, esta troca de coleções não é tão simples, pois além de substituir `List` por `Set` é preciso eliminar a verificação do IfContains, já que é característico desta última evita a inserção de elementos duplicados internamente. Outros fatores podem influenciar as transformações nesse caso, porém para melhor entendimento é preciso analisar como os métodos de `List` e `Set` se relacionam.

3.3 Análise dos Métodos

Para a substituição direta no caso citado acima, além da retirada do IfContains, Seção A.2, é necessário modificar no restante do código as demais chamadas de método do objeto que era `List` e foi trocado para `Set`. Para isso são analisados como os métodos contidos em cada uma das duas interfaces estudadas se relacionam, resultando na Tabela 3.1.

Tabela 3.1: *Relação entre os métodos de `List` e `Set`*

Linha	**método de `List`**	**Relação**	**método de `Set`**
1	void:clear()	Correspondentes	void:clear()
2	boolean:contains(Object o)	Correspondentes	boolean:contains(Object o)
3	boolean:containsAll (Collection<?> c)	Correspondentes	boolean:containsAll (Collection<?> c)
4	boolean:equals(Object o)	Correspondentes	boolean:equals(Object o)
5	int:hashCode()	Correspondentes	int:hashCode()
6	boolean:isEmpty()	Correspondentes	boolean:isEmpty()
7	Iterator<E>:iterator()	Correspondentes	Iterator<E>:iterator()
8	boolean:remove(Object o)	Correspondentes	boolean:remove(Object o)

9	boolean:removeAll (Collection<?> c)	Correspondentes	boolean:removeAll (Collection<?> c)
10	boolean:retainAll (Collection<?> c)	Correspondentes	boolean:retainAll (Collection<?> c)
11	int:size()	Correspondentes	int:size()
12	default Spliterator<E>: spliterator()	Correspondentes	default Spliterator<E>: spliterator()
13	Object[]:toArray()	Correspondentes	Object[]:toArray()
14	<T> T[] toArray(T[] a)	Correspondentes	<T> T[] toArray(T[] a)
15	void: notify()	Correspondentes	void: notify()
16	void: notifyAll()	Correspondentes	void: notifyAll()
17	void: forEach (Consumer action)	Correspondentes	void: forEach (Consumer action)
18	Class<?>: getClass()	Correspondentes	Class<?>: getClass()
19	Stream: stream()	Correspondentes	Stream: stream()
20	Stream: parallelStream()	Correspondentes	Stream: parallelStream()
21	boolean: removeIf (Predicate filter)	Correspondentes	boolean: removeIf (Predicate filter)
22	String: toString()	Correspondentes	String: toString
23	void:wait()	Correspondentes	void:wait()
24	void:wait(long timeout)	Correspondentes	void:wait(long timeout)
25	void:wait(long timeout, int nanos)	Correspondentes	void:wait(long timeout, int nanos)
26	boolean: add(E e)	Correspondentes sob Condições	boolean: add(E e)
27	void: add(int index, Object o)	Correspondentes sob Condições	boolean: add(E e)
28	boolean: addAll (Collection<? extends E> c)	Correspondentes sob Condições	boolean: addAll (Collection<? extends E> c)

29	boolean: addAll (int index, Collection<? extends E> c)	Correspondentes sob Condições	boolean: addAll (Collection<? extends E> c)
30	Object: set(int index, Object o)	Correspondentes sob Condições	boolean: remove(Object o) boolean: add(Object o)
31	Object:get(int index)	Não Correspondentes	
32	int:indexOf(Object o)	Não Correspondentes	
33	int:lastIndexOf(Object o)	Não Correspondentes	
34	List<Object>: subList(int fromIndex, int toIndex)	Não Correspondentes	
35	ListIterator<Object>:listIterator()	Não Correspondentes	
36	ListIterator:listIterator (int index)	Não Correspondentes	
37	Object:remove(int index)	Não Correspondentes	
38	void:sort(Comparator c)	Não Correspondentes	
39	void:replaceAll (UnaryOperator operator)	Não Correspondentes	

Nesta tabela é observado que, com relação às trocas, os métodos se comportam da seguinte forma:

- **Correspondentes**: que possuem métodos correspondentes nas duas interfaces (das linhas 1 à 25)
- **Correspondentes sob Condições**: que possuem métodos correspondentes nas duas interfaces, no entanto, a troca só pode ocorrer mediante condições (das linhas 26 à 30)
- **Não Correspondentes**: são métodos de `List` que não possuem correspondentes na interface `Set` (das linhas 31 à 39)

Nos pares de métodos Correspondentes eles possuem a mesma sintaxe, mas semânticas diferentes, pois um é da interface `List` e outro da `Set`, já que a primeira pode manter

elementos duplicados e indexáveis e a segunda não. Então, caso o desenvolvedor deseje mudar de comportamento, a substituição direta entre eles pode acontecer. Como por exemplo trocar o método `equals(Object o)` de `List` pelo `equals(Object o)` de `Set`.

Já nos métodos Correspondentes sob Condições, a principal condição para que a transformação ocorra, além da aceitação do programador em mudar a semântica dos métodos, é que exista a chamada daquele método dentro do corpo de um IfContains, Seção A.2. É o que ocorre, por exemplo, no Código Fonte 3.6, que adiciona um certo `numero` na coleção caso ele já não esteja inserido nela.

```
List<Integer> colecaoEnvolvida = new ArrayList<Integer>();
//...
public void adicionaNumero(int numero){
   if(!colecaoEnvolvida.contains(numero)){
      colecaoEnvolvida.add(numero);
   }
}
//...
```

Código Fonte 3.6: *Exemplo do método `add(E e)` de `List` que se encaixa nas condições*

Outro exemplo, existe no Código Fonte 3.7, que além de adicionar um `numero` na lista caso ele ainda não tenha sido inserido nela, ele insere em um determinado índice da lista (`posicao`).

```
List<Integer> colecaoEnvolvida = new ArrayList<Integer>();
//...
public void adicionaNumeroEmPosicao(int numero, int posicao){
   if(!colecaoEnvolvida.contains(numero)){
      colecaoEnvolvida.add(posicao, numero);
   }
}
//...
```

Código Fonte 3.7: *Exemplo do método `add(int index, E e)` de `List` que se encaixa nas condições*

Já no Código Fonte 3.8 tem-se um exemplo de verificação de não existência e inserção

de vários objetos de uma vez, que estão contidos em uma coleção, `numeros`.

```
List<Integer> colecaoEnvolvida = new ArrayList<Integer>();
List<Integer> numeros = new ArrayList<Integer>();
//...
public void adicionaVariosNumeros(Collection numeros){
   if(!colecaoEnvolvida.containsAll(numeros)){
      colecaoEnvolvida.addAll(numeros);
   }
}
//...
```

Código Fonte 3.8: *Exemplo do método `addAll(Collection c) de List` que se encaixa nas condições*

Com exceção do método `set(int index, Object o)`, que para ser trocado precisa estar dentro do corpo do IfContains, Seção A.2, e ter a referência do objeto pertencente à lista que é substituído pelo objeto `o`. No Código Fonte 3.9 existe um exemplo desse caso.

```
List<Integer> colecaoEnvolvida = new ArrayList<Integer>();
//...
public void substituirNumeros(int numeroAntigo, int numeroNovo){
   if(!colecaoEnvolvida.contains(numeroNovo)){
      colecaoEnvolvida.set(colecaoEnvolvida.indexOf
      (numeroAntigo), numeroNovo);
   }
}
//...
```

Código Fonte 3.9: *Exemplo do método `set(int index, Object o) de List` que se encaixa nas condições*

Já como exemplo de um método Não Correspondente, tem-se o `get(int index)` que retorna o objeto pertencente aquele índice da lista, e quando usado em um conjunto ocorre um erro de compilação, pois este não possui índices. Como é visto no Código Fonte 3.10:

```
public Object getLast(){
   return collection.get(collection.size());
}
```

Código Fonte 3.10: *Exemplo do método `get(int index)` de `List`*

3.4 Transformações

A substituição da interface `List` para `Set` ocorre quando existir alguma chamada dos métodos Correspondentes sob Condições dentro de um IfContains, Seção A.2. Entretanto, apenas a existência desta situação não viabiliza a substituição direta dos tipos da coleção de `List` para `Set`, pois caso possua uma chamada na lista para um método Não Correspondente, a transformação do código não pode ser realizada, para que não aconteça um erro de compilação.

Como exemplo tem-se o Código Fonte 3.11 que possui uma chamada para o método `get(int index)` dentro do método `getUltimoElemento()`, que resultará em um erro de compilação caso seja transformado.

```
List<Integer> colecaoEnvolvida = new ArrayList<Integer>();
//...
public void substituirNumeros(int numeroAntigo, int numeroNovo){
   if(!colecaoEnvolvida.contains(numeroNovo)){
      colecaoEnvolvida.set(colecaoEnvolvida.indexOf
      (numeroAntigo), numeroNovo);
   }
}
//...
public int getUltimoElemento(){
   return colecaoEnvolvida.get(colecaoEnvolvida.size());
}
//...
```

Código Fonte 3.11: *Exemplo de código que não pode ser transformado pela existência do método `get(int index)` de `List`*

3.4.1 Tipos de substituições entre métodos das coleções *JCF*

Dados C, C_1 e C_2 conjuntos de coleções existentes, no *JCF* no qual $C_1 \epsilon C$ e $C_2 \epsilon C$, sendo $m_1()$ representando um dos métodos pertencentes a C_1 e $m_2()$ um dos métodos pertencentes a C_2, teremos a seguinte classificação dos métodos das classes envolvidas quanto à substituição: [4]

- **Tipo 1**: $m_1()$ e $m_2()$ possuem a mesma sintaxe e semântica
- **Tipo 2**: $m_1()$ e $m_2()$ possuem a mesma semântica, mas têm sintaxe diferentes
- **Tipo 3**: $m_1()$ e $m_2()$ possuem a mesma sintaxe, mas têm semântica diferentes
- **Tipo 4**: $m_1()$ e $m_2()$ não possuem a nem sintaxe nem e semântica iguais

Ao classificar os métodos da Tabela 3.1 é obtida a seguinte distribuição:

- Tipo 1 e 2: nenhum método classificado
- Tipo 3: os métodos das linhas 1 à 26 e 28, pois esses métodos de `List` possuem a mesma sintaxe dos correspondentes em `Set`, mas com semânticas diferentes, já que em uma coleção permite-se indexação e elementos repetidos e na outra não.
- Tipo 4: os métodos das linhas 27 e 29.

Contudo além desses, existe o caso dos métodos que não se enquadram com a classificação determinada, são estes:

- Os que não possuem transformação por não terem correspondentes em Set: os métodos das linhas 31 à 39.

- O método da linha 30, `set(int index, Object o)`, pois tem como correspondentes chamadas de dois métodos de `Set (add e remove)`, e tal classificação só considera trocas de um método para outro (1-para-1).

Por esse motivo, é alterada a representação das substituições para classificar esta nova troca de um método para vários (1-para-n), obtendo o seguinte resultado: Dados C, C_1 e C_2 conjuntos de coleções existentes no JCF, no qual $C_1 \epsilon C$ e $C_2 \epsilon C$, sendo $m_1()$ representando um dos métodos pertencentes a C_1, $m_2()$ um dos métodos pertencentes a C_2 e $m_2 * ()$ vários dos métodos pertencentes a C_2, tem-se dessa forma a adição de um novo tipo de classificação:

- **Tipo 5**: $m_1()$ e $m_2 * ()$ possuem sintaxe diferentes, já que é uma substituição de um método para vários, e possuem semânticas diferentes, porém com algum nível de relacionamento semântico, que torna possível transformá-los.

Com esta nova definição o método `set(int index, Object o)` é classificado como sendo do tipo 5.

3.4.2 Equivalência Semântica Fraca

Na seção anterior, como a transformação dos métodos entre `List` e `Set`, ocorre uma mudança de semântica (Tipos de 3 a 5), não é possível garantir que o resultado final das transformações seja seguro ou semanticamente idêntico. A fim de poder comparar os códigos transformados com os originais em relação a um grau de semântica menor é levada em consideração, nesta pesquisa, uma **Equivalência Semântica Fraca** entre eles, onde ao fim da execução dos dois códigos não é considerado o tipo da coleção ou os índices/posições dos elementos contidos nela, e sim a presença ou ausência dos elementos na coleção em questão.

Esta Equivalência Semântica Fraca é atestada no Estudo de Caso (Capítulo 4) através de testes que verificam se os elementos das coleções envolvidas no código original e transformado são os mesmos ao fim da execução de cada um. Como exemplo tem-se dois códigos, um que resulta ao final uma lista e outro um conjunto, por esse motivo estes dois códigos não podem ser semanticamente idênticos. Entretanto, caso as coleções resultem nos mesmos elementos, não importando os índices destes, pode-se considerar que esses dois códigos possuem Equivalência Semântica Fraca, como pode ser visto na Figura 3.1.

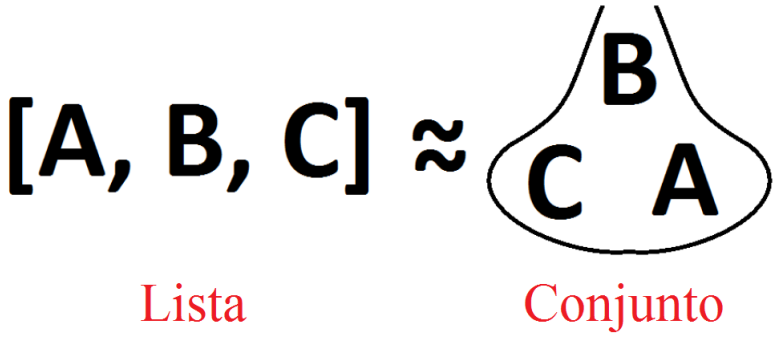

Figura 3.1: *Demonstração de uma lista e um conjunto que são resultados de dois códigos com Equivalência Semântica Fraca*

3.4.3 *Templates* das transformações

Nesta seção são apresentados os *templates* das transformações propostas para serem realizadas nos casos específicos em que ocorra a substituição direta entre as interfaces `List` e `Set` e consequente a transformação dos demais atributos, variáveis e métodos que se relacionam com a coleção modificada. Esses possuem dois lados, onde o lado esquerdo será transformado para o código do lado direito.

Estes *templates* são definidos através de uma metalinguagem, onde: Os `commands1, commands2,` etc. Já as expressões são representadas por `expr, expr1, expr2,` etc.

O `notContainsCall` representa a negação do resultado da chamada de qualquer um dos métodos que verificam se um elemento ou uma coleção de elementos já estão inseridos em uma lista, que é o IfContains, Seção A.2, explicado anteriormente (são `!list.contains(Object obj)` e `!list.containsAll(Collection c)`).

O `insertionMethod` representa os métodos `add(E e), add(int index, Object o), set(int index, Object o), addAll(Collection<? extends E> c)` e `addAll(int index, Collection<? extends E> c)` da interface `List`. Já o `insertionMethod'` representa o(s) método(s) correspondente(s) do `insertionMethod` em `Set`, de acordo com a Tabela 3.1.

Através dessa metalinguagem são definidos os padrões de código (*templates*), onde para cada um existe um código transformado, mediante a validação de certos provisos. Esse par de códigos original e transformado forma a transformação proposta. Também são exempli-

ficados casos de trechos de códigos onde podem ocorrer cada uma dessas transformações definidas.

3.4.4 Transformações Simples

As primeiras transformações, que foram chamadas de **Transformações Simples**, possuem como característica principal um IfContains, Seção A.2, que dentro do seu corpo possui um `insertionMethod`.

Transformação 1

Este tipo de transformação é aplicada quando o *template* do Código Fonte 3.12 for satisfeito.

```
commands1
if(notContainsCall){
  insertionMethod
}
commands2
```

Código Fonte 3.12: *Template para Transformação 1*

```
commands1

insertionMethod'

commands2
```

Código Fonte 3.13: *Resultado da Transformação 1*

provisos

1. `commands1` e `commands2` podem ser vazios
2. `commands1` e `commands2` não podem conter nenhum método Não Correspondente (Tabela 3.1) sendo chamado a partir da lista envolvida na instrução `notContainsCall`

No Código Fonte 3.13, a chamada do `insertionMethod'` não precisa mais ser precedida da verificação do IfContains, Seção A.2, pois a coleção passou a ser do tipo `Set`, que já evita a inserção de elementos duplicados internamente. Após essa ação o corpo do IfContains resulta em um `if` com um corpo vazio, por isso este é retirado por completo no código resultante da transformação. Como a instrução `notContainsCall` pode representar o método `lista.contains(E e)`, que possui complexidade de **O(n)** ou o método `lista.contains(Collection c)` com complexidade **O(n.m)**, onde n representa a

quantidade de elementos de `lista` e m a quantidade de elementos da coleção `c`, com a retirada dessa instrução é obtido um código mais eficiente. Um exemplo desse *template* de transformação sendo aplicado no método `add(E e)` se encontra no Código Fonte 3.14 e Código Fonte 3.15.

```
List configurationFiles;
//...
public void addFileToConfigurationFiles
    (File newFile){
  if (!configurationFiles.contains(
      newFile)){
    configurationFiles.add(newFile);
  }
}
```

Código Fonte 3.14: *Transformação 1: método* `add(E e)`

```
Set configurationFiles;
//...

public void addFileToConfigurationFiles
    (File newFile){

  configurationFiles.add(newFile);

}
```

Código Fonte 3.15: *Resultado da Transformação 1: método* `add(E e)`

Neste exemplo tem-se dentro do corpo do IfContains uma chamada do método `add(obj)` pela coleção `configurationFiles`. Então ocorre o casamento de padrão com o *template* definido para a Transformação 1 e as substituições diretas acontecem resultando no Código Fonte 3.15 que possui como método corresponde para `add(obj)` de `List` o `add(obj)` de `Set` (que é o `insertionMethod'`).

Exemplo da Transformação 1 aplicada no método `add(int index, Object o)`

Ocorre o mesmo código resultante quando o `insertionMethod` do *template* da Transformação 1 for `add(int index, Object o)`, se tornando igual ao método `add(E e)`, pois como em `Set` não tem índices, o parâmetro `index` é descartado.

```
List trackers;
//...
public void addTracker(String
    trackerAnnounceUrl) {
  if(!trackers.contains(
      trackerAnnounceUrl)){
    trackers.add(0,trackerAnnounceUrl);
  }
  saveList();
}
```

Código Fonte 3.16: *Transformação 1: método* `add(index, obj)`

```
Set trackers;
//...

public void addTracker(String
    trackerAnnounceUrl) {

  trackers.add(trackerAnnounceUrl);

  saveList();
}
```

Código Fonte 3.17: *Resultado da Transformação 1: método* `add(index, obj)`

Com a substituição do método `add(int index, Object o)` de `List`, que possui complexidade **O(1)**, pelo método `add(Object o)` de `Set`, que possui complexidade de **O(n)**, ocorre uma perca no desempenho, porém com a remoção da instrução `lista.contains(E e)`, que possui complexidade **O(n)**, ocorre um ganho no desempenho que resulta em códigos original e transformado com a mesma complexidade.

Exemplo da Transformação 1 aplicada no método `set(int index, Object o)`

Já quando o *template* é aplicado no método `set(int index, Object o)` tem-se que o `insertionMethod'` do código transformado é substituído pelas chamadas do método `add(novo)` em conjunto com `remove(antigo)`, como visto na Tabela 3.1. Uma vez que eles dois combinados possuem um certo grau de relacionamento semântico com o `set(collection.indexOf(antigo), novo)` de `List`. Isto pode ser visto no Código Fonte 3.18 e Código Fonte 3.19.

```
List bosses;
//...
public void changeBoss(Employee oldBoss
    , Employee newBoss){
  if (!bosses.contains(newBoss)){
    bosses.set(bosses.geIndexOf
    (oldBoss), newBoss);
  }
}
```

Código Fonte 3.18: *Transformação 1: método* `set(index, obj)`

```
Set bosses;
//...
public void changeBoss(Employee oldBoss
    , Employee newBoss){

  bosses.add(oldBoss);
  bosses.remove(newBoss);

}
```

Código Fonte 3.19: *Resultado da Transformação 1: método* `set(index, obj)`

Neste exemplo aparentemente percebe-se que ocorreu uma perda de desempenho no código já que o método `set(int index, Object o)`, que possui complexidade de **O(1)**, é substituído pelos métodos `add(Object o) e remove(Object o)`, que possuem complexidade **O(n)** cada. Entretanto, no exemplo é utilizado o método `bosses.geIndexOf(oldBoss)`, que possui complexidade de **O(n)**, para obter qual é o índice de `oldBoss` e ainda é feita a retirado do `bosses.contains(newBoss)` que também possui complexidade de **O(n)**, resultando em códigos original e transformado com complexidades iguais.

Vale ressaltar que antes de realizar estas transformações, que eliminam o uso dos índices, que ocorre com os métodos `set(int index, Object o)` e `add(int index, Object o)`, é mais cauteloso solicitar permissão do programador se ele realmente deseja realiza-las.

Transformação 2

Este tipo de Transformação Simples é aplicada quando o *template* do Código Fonte 3.20 for satisfeito.

```
commands1
if(notContainsCall){
  commands2
  insertionMethod
}
commands3
```

Código Fonte 3.20: *Template para Transformação 2*

```
commands1
if(notContainsCall){
  commands2
}
insertionMethod'
commands3
```

Código Fonte 3.21: *Resultado da Transformação 2*

provisos

1. `commands1` e `commands3` podem ser vazios
2. `commands2` não pode ser vazio
3. `commands1, commands2` e `commands3` não podem conter nenhum método Não Correspondente (Tabela 3.1) sendo chamado a partir da lista envolvida na instrução `notContainsCall`

Esta transformação é similar a Transformação 1, com a diferença de existir outros comandos (`commands2`) dentro do corpo do IfContains além do `insertionMethod`. O que resulta em um código transformado onde o `insertionMethod'` fica fora do corpo do IfContains, pois passou a ser usada uma interface `Set`, porém como ainda existe `commands2` dentro do IfContains, ele permanece no código.

A aplicação desse `template` é similar às aplicações mostradas na Transformação 1, substituindo o `insertionMethod` por um dos cinco métodos possíveis e o `insertionMethod'` pelo(s) seu(s) respectivo(s) correspondente(s). Como nesta transformação ainda é mantida a execução da instrução `notContainsCall` caso o `insertionMethod` seja algum dos métodos que possui `index` em seus parâmetros, ocorrerá um acréscimo na complexidade do código transformado em **O(n)**. Como exemplo o *template* da transformação é aplicado no método `addAll(collection)`, que se encontra dentro do corpo de um IfContains, Seção A.2, utilizando a verificação `containsAll(collection)`, como pode-se ver no Código Fonte 3.22, e o Código Fonte 3.23 é o resultado da transformação.

```
List<Disciplina> todasDisciplinas;
//...
public void matricular(List<Disciplina>
     disciplinasPeriodo){
  if(!todasDisciplinas.containsAll(
      disciplinasPeriodo)){
       checkDisciplinas(
           disciplinasPeriodo);
       todasDisciplinas.addAll(
           disciplinasPeriodo);
  }
}
```

Código Fonte 3.22: *Transformação 2: método* `addAll(Collection<? extends E> c)`

```
Set<Disciplina> todasDisciplinas;
//...
public void matricular(List<Disciplina>
     disciplinasPeriodo){
  if(!todasDisciplinas.containsAll(
      disciplinasPeriodo)){
    checkDisciplinas(disciplinasPeriodo
        );
  }
  todasDisciplinas.addAll(
      disciplinasPeriodo);
}
```

Código Fonte 3.23: *Resultado da Transformação 2: método* `addAll(Collection<? extends E> c)`

Outro exemplo de aplicação, similar à anterior, é no método `addAll(index, objetos)`, onde todos os objetos de uma coleção são adicionados na lista. Porém, como no código transformado é utilizado um `Set`, que não possuem índices, o `index` é desconsiderado, resultando exatamente no mesmo código transformado do exemplo anterior. Pode-se ver no exemplo abaixo, que é idêntico ao anterior com a alteração de inserir os elementos no início da lista ao invés do final.

```
List<Disciplina> todasDisciplinas;
//...
public void matricular(List<Disciplina>
     disciplinasPeriodo){
  if(!todasDisciplinas.containsAll(
      disciplinasPeriodo)){
    checkDisciplinas(disciplinasPeriodo
        );
    todasDisciplinas.addAll(0,
        disciplinasPeriodo);
  }
}
```

Código Fonte 3.24: *Transformação 2: método* `addAll(index, collection)`

```
Set<Disciplina> todasDisciplinas;
//...
public void matricular(List<Disciplina>
     disciplinasPeriodo){
  if(!todasDisciplinas.containsAll(
      disciplinasPeriodo){
    checkDisciplinas(disciplinasPeriodo
        );
  }
todasDisciplinas.addAll(
    disciplinasPeriodo);
}
```

Código Fonte 3.25: *Resultado da Transformação 2: método* `addAll(index, collection)`

3.4.5 Transformações Com 1 If Aninhado

Este tipo de transformação possui dentro do corpo do IfContains, Seção A.2, um IfSimples, Seção A.1, que por sua vez possui a chamada do `insertionMethod` no seu corpo.

Transformação 3

Neste caso a Transformações Com 1 If Aninhado é aplicada quando o *template* do Código Fonte 3.26 for satisfeito.

```
commands1
if(notContainsCall){
  commands2
  if(expr){
    commands3
    insertionMethod;
  }
}


commands4
```

Código Fonte 3.26: *Template para a Transformação 3*

```
commands1
if(notContainsCall){
  commands2
  if(expr){
    commands3
  }
}
if(expr){
  insertionMethod'
}
commands4
```

Código Fonte 3.27: *Resultado da Transformação 3*

provisos

1. `commands1`, `commands2` e `commands4` podem ser vazios
2. `commands3` não pode ser vazio
3. `commands3` não pode afetar nenhuma das variáveis presentes em `expr`
4. `commands1, commands2` e `commands3` não podem conter nenhum método Não Correspondente (Tabela 3.1) sendo chamado a partir da lista envolvida na instrução `notContainsCall`

O que é visto de diferente das transformações anteriores é a retirada do `insertionMethod'` juntamente com o IfSimples, que estava aninhado ao IfContains. Isso ocorre para tentar preservar o máximo de relacionamento semântico entre os trechos de

códigos original e transformado. Este também é o motivo pelo qual o `commands2` juntamente com o IfSimples com `commands3` permanecem dentro do corpo do IfContains.

Similarmente ao *template* de transformação anterior, nesta transformação ainda é mantida a execução da instrução `notContainsCall`, logo caso o `insertionMethod` seja algum dos métodos que possui `index` em seus parâmetros, ocorrerá um acréscimo na complexidade do código transformado em **O(n)** além do acréscimo da complexidade da execução da expressão `expr` ocorrido pela duplicação do IfSimples. Esta perda de desempenho também ocorrerá nas transformações 4 e 5, problema que é contornado através das Transformações Alternativas, Seção 3.4.7.

Transformação 4

Esta transformação é bastante similar à Transformação 3 com a única diferença do `commands3` ser vazio. Resultando em um código transformado onde o corpo do IfSimples fica vazio (sem nenhum comando), acarretando na sua retirada. Porém ele continua aparecendo junto com o `insertionMethod` fora do IfContains, Seção A.2. Como é possível visualizar no Código Fonte 3.28

```
commands1
if(notContainsCall){
  commands2
  if(expr){
    insertionMethod;
  }
}
commands4
```

Código Fonte 3.28: *Template para a Transformação 4*

⇨

```
commands1
if(notContainsCall){
  commands2
}
if(expr){
  insertionMethod'
}
commands4
```

Código Fonte 3.29: *Resultado da Transformação 4*

provisos

1. `commands1` e `commands4` podem ser vazios
2. `commands2` não pode ser vazio
3. `commands1, commands2` e `commands4` não podem conter nenhum método Não Correspondente (Tabela 3.1) sendo chamado a partir da lista envolvida na instrução

`notContainsCall`

Transformação 5

Já nesta transformação tem-se os mesmos elementos da Transformação 4 com a diferença de `commands2` ser vazio. Como é visto no Código Fonte 3.30

```
commands1
if(notContainsCall){
  if(expr){
    insertionMethod;
  }
}
commands3
```

Código Fonte 3.30: *Template para a Transformação 5*

```
commands1

if(expr){
  insertionMethod'
}

commands3
```

Código Fonte 3.31: *Resultado da Transformação 5*

provisos

1. `commands1` e `commands3` podem ser vazios
2. `commands1` e `commands3` não podem conter nenhum método Não Correspondente (Tabela 3.1) sendo chamado a partir da lista envolvida na instrução `notContainsCall`

Resultando em uma transformação onde o corpo do IfContains fica só com o IfSimples, só que este fica vazio (sem nenhum comando), acarretando na sua retirada, e consequentemente na retirada do IfContains, que também fica com seu corpo vazio. Porém o IfSimples continua aparecendo junto com o `insertionMethod` fora do IfContains

3.4.6 Transformações Com Vários Ifs Aninhados

Para as próximas transformações, que foram chamadas de **Transformações Com Vários Ifs Aninhados** por terem dentro do corpo do IfContains, Seção A.2, mais de um IfSimples, Seção A.1, aninhado possuindo o `insertionMethod` no corpo do último desses `Ifs`, é definida a seguinte representação para os `Ifs` aninhados.

```
IfOrInsertion -> if(exprA'){
                    commands1
                    IfOrInsertion
                 }
                       OR
                 insertionMethod
```

Código Fonte 3.32: *Representação para `Ifs` aninhados*

Onde o `commands1` é um conjunto de comandos aceitos pela *Minimal Core Java* definida. E o identificador `exprA'`, representa qualquer expressão aceita pela *Minimal Core Java*. E como já foi visto, o `insertionMethod` representa um dos métodos `add(E e)`, `add(int index, Object o)`, `set(int index, Object o)`, `addAll(Collection<? extends E> c)` e `addAll(int index, Collection<? extends E> c)` da interface `List`.

Percebe-se que existe uma recursão nesta representação, e o número de vezes que esta recursão é executada é também a quantidade de `Ifs` aninhados contidos na instrução `ifOrInsertion`.

Transformação 6

Esta transformação é aplicada quando o *template* do Código Fonte 3.33 for satisfeito. Neste *template* percebemos que existirá pelo menos dois `Ifs` aninhados dentro do corpo do IfContains, Seção A.2, já que o `IfOrInsertion` pode de imediato ser "substituído" pelo `insertionMethod`.

```
commands1
if(notContainsCall){
  if(expr1){
    commands2
    if(expr2){
      commands3
      IfOrInsertion
    }
  }
}




commands4
```

Código Fonte 3.33: *Template para transformação 6*

```
commands1
if(containsMethod){
  if(expr1){
    commands2
    if(expr2){
      commands3
      IfOrInsertion'
    }
  }
}
if(expr1){
  if(expr2){
    IfOrInsertion''
  }
}
commands4
```

Código Fonte 3.34: *Resultado da Transformação 6*

provisos

1. `commands1, commands2` e `commands4` podem ser vazios
2. `commands3` não pode ser vazio
3. `commands1, commands2, commands3` e `commands4` não podem conter nenhum método Não Correspondente (Tabela 3.1) sendo chamado a partir da lista envolvida na instrução `notContainsCall`

O código transformado contido no Código Fonte 3.34 segue a mesma lógica contida nas Transformações com um 1 If Aninhado, onde dentro do corpo do IfContains ficam os `Ifs` aninhados com os seus respectivos `commads`, e fora do IfContains ficam os `Ifs` aninhados com a chamada do `insertionMethod` no último `If`. A representação para a instrução `IfOrInsertion'`, que significa o resultado da transformação dos `Ifs` aninhados que ficam dentro do corpo de IfContains, é a seguinte:

```
IfOrInsertion' -> if(exprA'){
                      commands1
                      IfOrInsertion'
                  }
                        OR
                    emptyCommand
```

Código Fonte 3.35: *Representação para `Ifs` aninhados*

Onde `emptyCommand` representa um comando vazio (`;` ou simplesmente não possuindo nada). Já o `IfOrInsertion"` é o resultado da transformação dos `Ifs` aninhados que ficam fora do corpo do IfContains, e é representado pelo seguinte código:

```
IfOrInsertion'' -> if(exprA'){
                       IfOrInsertion''
                   }
                         OR
                    insertionMethod'
```

Código Fonte 3.36: *Representação para `Ifs` aninhados*

Deve-se levar em consideração que a quantidade de recursões existentes nas três instruções (`IfOrInsertion, IfOrInsertion' e IfOrInsertion"`) devem ser as mesmas, pois a quantidade de `Ifs` aninhados devem ser iguais. Como exemplo da aplicação dessa Transformação 7 em um pseudocódigo utilizando o método `add(obj)` existe o Código Fonte 3.37 que tem como transformação o Código Fonte 3.38.

```
commands1
if(!collection.contains(obj)){
  commands2
  if(expr1){
    commands3
    if(expr2){
      commands4
      if(expr3){
        commands5
        [return ][variavel = ]
        collection.add(obj);
      }
    }
  }
}



commands6
```

Código Fonte 3.37: *Transformação 7: método* `add(obj)`

```
commands1
if(!collection.contains(obj)){
  commands2
  if(expr1){
    commands3
    if(expr2){
      commands4
      if(expr3){
        commands5
      }
    }
  }
}
if(expr1){
  if(expr2){
    if(expr3){
      [return ][variavel = ]
      collection.add(obj);
    }
  }
}
commands6
```

Código Fonte 3.38: *Resultado da Transformação 7: método* `add(obj)`

Caso algum `if` após a transformação fique sem nenhum comando dentro de seu corpo ele deve ser eliminado no código transformado. Como exemplo tem-se o Código Fonte 3.39 que dentro do último `if` aninhado (`if(expr3)`) possui apenas a chamada do método `set(index, obj)`, então o Código Fonte 3.40 resulta na eliminação desse `if` dentro do corpo do IfContains, Seção A.2. Lembrando que esse mesmo `if` ainda deve permanecer nos `ifs` que se encontram fora do corpo do IfContains.

```
commands1
if(!collection.contains(novo)){
  commands2
  if(expr1){
    commands3
    if(expr2){
      commands4
      if(expr3){
        [return ][variavel = ]
            collection.set(collection.
            indexOf(antigo), novo);
      }
    }
  }
}

commands5
```

Código Fonte 3.39: *Transformação 7: método* `set(index, obj)`

```
commands1
if(!collection.contains(obj)){
  commands2
  if(expr1){
    commands3
    if(expr2){
      commands4
    }
  }
}
if(expr1){
  if(expr2){
    if(expr3){
      collection.remove(antigo)
      [return ][variavel = ]
      collection.add(novo);
    }
  }
}
commands5
```

Código Fonte 3.40: *Resultado da Transformação 7: método* `set(index, obj)`

3.4.7 Transformações Alternativas

Ao analisar os Código Fonte 3.26, Código Fonte 3.28 e Código Fonte 3.30 percebe-se que as transformações estão com pior desempenho e mais difíceis de entender do que os códigos originais. Isso pode prejudicar a transformação, porque provavelmente o programador não vai desejar grandes modificações no seu código original, por poder prejudicar a legibilidade do mesmo.

Como possível solução são definidas as **Transformações Alternativas**, que são códigos transformados que não possuem Equivalência Semântica Fraca em todas as situações possíveis com os respectivos códigos originais, mas que possivelmente, dado a mudança de contexto ocorrida com a substituição das coleções, pode ser uma opção mais interessante para o programador. Como exemplo vamos retornar ao Código Fonte 3.41, que possui a transformação contida no Código Fonte 3.42 e a transformação alternativa no Código Fonte 3.43.

```
commands1
if(!collection.contains(obj)){
  commands2
  [return][variavel = ] collection.
      add(obj);
}
commands3
```

Código Fonte 3.41: *Exemplo de código*

```
commands1
if(!collection.contains(obj)){
  commands2
}
[return][variavel = ] collection.
add(obj);
commands3
```

Código Fonte 3.42: *Transformação proposta para o Código Fonte 3.41*

```
commands1
commands2
[return][variavel = ] collection.
add(obj);



commands3
```

Código Fonte 3.43: *Transformação alternativa para o Código Fonte 3.41*

Entendendo que muito provavelmente o IfContains contido no código somente está realizando esta verificação por causa da execução do método `add(obj)`, não tendo nenhuma relação com `commands2`, fica desnecessário manter o IfContains no código transformado, resultando no Código Fonte 3.43.

Porém, por se tratar de uma transformação alternativa, que faz uma mudança maior na semântica do código, é interessante apresentar, em uma futura ferramenta, as duas possíveis transformações para que o programador decida qual ele deseja utilizar.

Este mesmo tipo de transformação alternativa pode acontecer com qualquer *template* de transformação definido anteriormente, bastando retirar por completo a chamada do IfContains, não necessitando realizar mais nenhuma ação. Como nos Código Fonte 3.44, Código Fonte 3.46 e Código Fonte 3.48 que possuem respectivamente as seguintes transformações alternativas: Código Fonte 3.45, Código Fonte 3.47 e Código Fonte 3.49.

```
commands1
if(!collection.notContainsCall(obj)){
  commands2
  if(expr){
    commands3
    insertionMethod
  }
}
commands4
```

Código Fonte 3.44: *Exemplo de padrão de código*

```
commands1
commands2
if(expr){
  commands3
  insertionMethod'
}


commands4
```

Código Fonte 3.45: *Transformação alternativa para o Código Fonte 3.44*

```
commands1
if(!collection.notContainsCall(obj)){
  commands2
  if(expr){
    insertionMethod
  }
}
commands3
```

Código Fonte 3.46: *Exemplo de padrão de código*

```
commands1
commands2
if(expr){
  insertionMethod'
}


commands3
```

Código Fonte 3.47: *Transformação alternativa para o Código Fonte 3.46*

```
commands1
if(!collection.notContainsCall(obj)){
  commands2
  if(expr1){
    commands3
    if(expr2){
      commands4
      if(expr3){
        insertionMethod
      }
    }
  }
}
commands5
```

Código Fonte 3.48: *Exemplo de padrão de código*

```
commands1
commands2
if(expr1){
  commands3
  if(expr2){
    commands4
    if(expr3){
      insertionMethod'
    }
  }
}


commands5
```

Código Fonte 3.49: *Transformação para o Código Fonte 3.48*

Como é retirada a execução da instrução `notContainsCall` ocorrerá uma eliminação na complexidade total do código em pelo menos **O(n)**, porque além disso evita o acréscimo da complexidade das execuções das expressões dos `if`s aninhados contidos dentro do IfCon-

tains. Isso melhora bastante o desempenho do código em comparação com as transformações convencionais.

Lembrando sempre que, tanto as transformações convencionais quanto as alternativas só podem ocorrer caso não exista em outra parte do código alguma chamada de métodos que não possuem correspondentes em `Set`.

3.4.8 Transformações de `Set` para `List`

Até o momento somente foi discutido características, condições e exemplos de transformações de códigos que utilizam `List` e passarão a usar `Set`. Agora trataremos do caminho contrário. Como transformar códigos que usam `Set` para passarem a usar `List`?

O indício de que um certo código se adequaria mais utilizando `Set` do que `List` era quando uma lista estava se comportando como um conjunto, não deixando inserir elementos repetidos (com o uso do IfContains, Seção A.2), e ainda não utilizava os índices da lista em outra parte do programa. Em `Set` não existe um indício no código de que aquele programador deseja utilizar `List`. Caso ele precise disso, deverá trocar o tipo das coleções manualmente, e a partir daí poderá utilizar os métodos de `List` que antes não podia.

Este caso não se trata de uma correção do código, para que ele use uma coleção mais adequada (como ocorria nas transformações anteriores), se trata de uma evolução de software adaptativa, visto na Seção 2.1. E ocorrendo essa adaptação do código, não sendo necessário mais nenhuma outra mudança nas chamadas dos métodos dessa coleção ao longo do código, porque como é visto na Tabela 3.1 todos os métodos de `Set` também existem em `List`, saindo do menos restrito pro mais restrito.

3.5 Implementação das transformações

A implementação das transformações apresentadas até agora é realizada utilizando uma ferramenta de reescrita, que a partir de uma certa estrutura de código satisfeita (casamento de padrão) é gerado um novo código reescrito de acordo com regras pré-estabelecidas, regras de reescrita. Ela foi bastante importante para realização dos *templates* das transformações citados até agora, pois proporcionou uma melhor visão sobre as situações a serem transformadas e nos proporcionou um maior formalismo para as transformações. Além de auxiliar

na etapa de avaliação das transformações através do estudo de caso.

A ferramenta de reescrita irá ser aplicada sobre uma *Domain-Specific Language*(DSL), que diferentemente das linguagens de programação de propósito geral (exemplo Java e Python), são linguagens de domínio específico, ou seja, são linguagens de programação que possuem um contexto e situação restrita para sua aplicação. E quando essa linguagem está fora do seu âmbito de atuação se torna incapaz de resolver problemas. Como exemplo existe *HyperText Markup Language* (HTML) que é aplicada em navegadores de internet para editar seu estilo, *Structured Query Language*(SQL) que é uma linguagem de manipulação de banco de dados e Verilog que é uma linguagem de descrição de *hardware* usada para modelar sistemas eletrônicos [26] [27].

Nesse trabalho o *Minimal Core Java* é implementado em uma *DSL*. Para isso é utilizado um *LW*. Esse é um tipo de *sistema* que cria e manipula *DSLs*, foi um conceito definido e popularizado por Martin Fowler em 2005.

Normalmente um *Language workbench* suporta especificações de **metamodelo**, ambientes de edição de *DSLs* e especificação da semântica de execução. Metamodelo é um modelo de um modelo, que no caso deste trabalho é o modelo da linguagem definida, onde contém as restrições e regras. E a especificação da semântica de execução é a geração de código a partir das regras definidas em cima da *DSL*, que é justamente a reescrita de código. Então a ferramenta de reescrita é uma parte integrante de um *Language workbench* [26].

Alguns exemplos de *Language workbench* existentes são: Rascal [28], Jetbrains MPS [29], Spoofax [14] e Xtext [30]. Dentre estes os que mais se adequavam para ser utilizado nesta pesquisa foram Xtext e Spoofax, pois eles são utilizados sob a IDE Eclipse, abrange todos os aspectos de uma infraestrutura de linguagem completa, desde o analisador até o interpretador e gerador de código [31].

Então Foi escolhido Spoofax, pois possui uma comunidade de desenvolvedores ativa, por ter uma sintaxe mais simples para suas definições, por estar em constante atualização de suas versões (teve inclusive grandes atualizações durante o desenvolvimento desta pesquisa), usa uma abordagem baseada em texto para *DSLs*, usa seu próprio *parser* internamente que permite combinações de linguagem e principalmente pois transformação é o conceito fundamental do Spoofax [32].

O Spoofax também provê uma linguagem declarativa mais poderosa para definição de

regras *name-binding*, que é importante nesta pesquisa para distinguir qual o nome da lista que esta sendo usada como conjunto. Outros motivos importantes para a escolha do Spoofax são: que ele possui uma sintaxe mais concreta e permite composição e reuso de *building blocks* melhor do que o Xtext onde a composição de gramáticas é limitada. [31]

Foi Utilizada a versão 1.5.0 do Spoofax, através de um *plug-in* para o eclipse, onde ao ser criado um novo projeto é retornado um pequeno exemplo de uma linguagem de programação, em que a partir dela pode-se acrescentar e modificar o que desejar.

Existem algumas linguagens auxiliares do Spoofax que possuem diferentes propósitos, as que mais foram utilizadas nesta pesquisa foram a *Syntax Definition Formalism* (SDF3) e o Stratego. A SDF3, se encontra em sua terceira versão, com ela é definida sintaticamente a linguagem em questão, definindo assim a gramática. Através da gramática definida em SDF3 é gerado um *parser*, que realizará a análise sintática do código original, convertendo-o em uma Árvore de Sintaxe Abstrata ou *Abstract Syntax Tree (AST)*. Esta arvore possui nós chamados de *ATerm*, que são termos definidos na gramática da linguagem.

O Código Fonte 3.50 é um exemplo desse formalismo de definição sintática, onde as definições de modificadores (`Modifier`), parâmetros (`Parameter`), métodos construtores(`FieldOrMethod.ConstructorMethod`) e comandos(`Command`) são criadas na forma de *ATerm*. Estes vão sendo verificadas a medida que o Spoofax varre o código original, então caso algum trecho combine com uma ou mais *ATerms*, este é incluído em uma *(AST)*.

```
//...
Modifier = <<Public>>
Modifier = <<Private>>
Modifier = <<Protected>>
//...
Public.Public = <public>
Private.Private = <private>
Protected.Protected = <protected>
//...
Parameter.Parameter = <<ParameterType> <ParameterId>>

ParameterList.ParameterManyList = <<{Parameter ","}*>>

FieldOrMethod.ConstructorMethod = <
<{Modifier " "}+> <ID>(<ParameterList>){
<{Command "\n"}*>

}>

Command.CommandAdd = <<ID>.add(<ParameterId>);>

Command.CommandSet = <<ID>.set(<ID>.getIndexOf(<ParameterId>),
<ParameterId>);>

Command.SimpleCommand = <<ID>.<MethodCall>;>

Command.DupleCommand = <<ID>.<ID>.<MethodCall>;>
//...
```

Código Fonte 3.50: Exemplo de formalismo de definição sintática

Como exemplo de um código e seu nó na AST tem-se os Código Fonte 3.51 e Código Fonte 3.52, onde é observado que cada elemento da representação textual inicial possui um nó correspondente na *AST*.

```
public Relationship(Object obj, List
    lista) {
lista.add(obj);
}
```

Código Fonte 3.51: Exemplo de código

```
ConstructorMethod(
[Public()]
,   "Relationship"
,     ParameterManyList(
[ Parameter(ParameterType("Object"),
    ParameterId("obj"))
, Parameter(ParameterType("List"),
    ParameterId("lista"))
]
)
,     [CommandAdd("lista", ParameterId(
    "obj"))]
)
```

Código Fonte 3.52: Nó na AST correspondente

Outra linguagem auxiliar do Spoofax, que é considerada a principal parte desta pesquisa, é a Stratego. Através dela é realizada a interpretação da *AST* e tradução que a transforma em versões diferentes até o ponto em que uma nova representação textual é gerada. Nela são definidas as regras de transformações com a seguinte sintaxe.

```
nome-da-regra:
  ATerm-original -> ATerm-resultante
```

Código Fonte 3.53: *Formato para escrever uma regra de transformação*

Esta parte é editada no arquivo `generate.str`. Como exemplo tem-se a execução da regra `analyse` que realiza uma mudança na *AST*, modificando o *ATerm* `SimpleCommandIfContains` para outro *ATerm* `CommandSet`. Esta regra se encontra no Código Fonte 3.54.

```
//...
analyse:
   SimpleCommandIfContains(
   collection
   , ParameterId(paramId)
   , [CommandSet(t1, t2,t3, t4)]) -> [CommandSet(t1, t2,t3, t4)]
   //...
```

Código Fonte 3.54: *Definição do* `CommandSet` *na gramática da linguagem*

Outra forma de escrever regras na linguagem Stratego é quando, ao invés de a partir de um *ATerm* ser resultado outro *ATerm*, ser resultado uma nova representação textual. Como pode ser visto no Código Fonte 3.55.

```
nome-da-regra:
  ATerm-original -> $[ nova_representacao_textual ]
```

Código Fonte 3.55: *Formato geral para escrever uma regra de transformação*

Este tipo de regra é chamada de **Regra de Reescrita**. Um exemplo dessa aplicada nesta pesquisa ocorre com a transformação para o nó `CommandSet`, que tem sua definição na gramática representada no Código Fonte 3.56.

```
//...
Command.CommandSet = <<ID>.set(<ID>.indexOf(<ParameterId>),
<ParameterId>);>
//...
```

Código Fonte 3.56: *Definição do* `CommandSet` *na gramática da linguagem*

Quando este é interpretado na varredura dos nós da *AST* executa a regra `to-java` gerando o código `[collection].remove([antiga]);`, onde `collection` é o ID da coleção envolvida no comando `set` e `antiga` é o elemento que é substituído, em conjunto com `[collection].add([nova]);`, onde `nova` é o elemento a ser inserido no lugar de `antiga`. Como pode ser visto no Código Fonte 3.57.

```
//...
to-java:
   CommandSet(collection,collection, ParameterId(antiga),
      ParameterId(nova))->
         $[  [collection].remove([antiga]);
            [collection].add([nova]);
         ]
         //...
```

Código Fonte 3.57: *Exemplo de regra de transformação para o nó CommandSet*

Estas regras descritas acima eram aplicadas no código fonte presente em um arquivo da extensão `.dir` (de *Direct Replacement*). Após a análise é gerado outro arquivo com o mesmo nome do primeiro contendo o código transformado em um arquivo `.java`.

3.6 Considerações Finais do Capítulo

Neste capítulo foi apresentado o estudo propriamente dito: quais são as consequência para realizar as modificações no código e quais os *templates* das transformações analisadas. Ainda foram apresentados conceitos que não foram abordados no capítulo anterior (como *Minimal Core Java*, Spoofax e Equivalência Semântica Fraca) e foram abordados neste pois foram explicados de acordo com sua aplicabilidade nesta pesquisa específica. Nos dois capítulos seguintes as transformações analisadas neste capítulo são avaliadas.

Capítulo 4

Avaliação: Estudo de Caso

Neste capítulo é apresentado um estudo de caso, que é definido de acordo com a literatura como sendo: "... um meio de organizar dados preservando o caráter unitário do objeto estudado" [33]. De outra forma, Tull [34] afirma que "...o estudo de caso refere-se a uma análise intensiva de uma situação particular.".

4.1 Objetivo do Estudo de Caso

Este estudo empírico é realizado com o objetivo de avaliar o comportamento em casos reais das transformações definidas neste documento. Por este motivo foi selecionado o Estudo de Caso como tipo de estudo, por necessitar de uma avaliação em ambiente real.

Esse teve a finalidade de medir o grau de aplicabilidade, eficácia e eficiência das transformações em resolver o problema de ter lista sendo usada como conjunto. A aplicabilidade das transformações foram medidas de acordo com a quantidade de casos por projeto de listas sendo usadas como conjunto e a quantidade de atributos do tipo `List`, que é o tipo de lista aplicado nas transformações. Já a eficácia foi medida através da corretude dos códigos transformados realizando a contagem dos erros deles em relação aos originais. E a eficiência visou medir quanto foi modificado e adicionado pelas transformações ao código original.

Uma visão geral de como as transformações analisadas foram aplicadas neste estudo empírico pode ser vista na Figura 4.1, que foi realizado de uma forma semiautomática, onde parte do processo foi realizado com auxílio de ferramentas e parte manualmente.

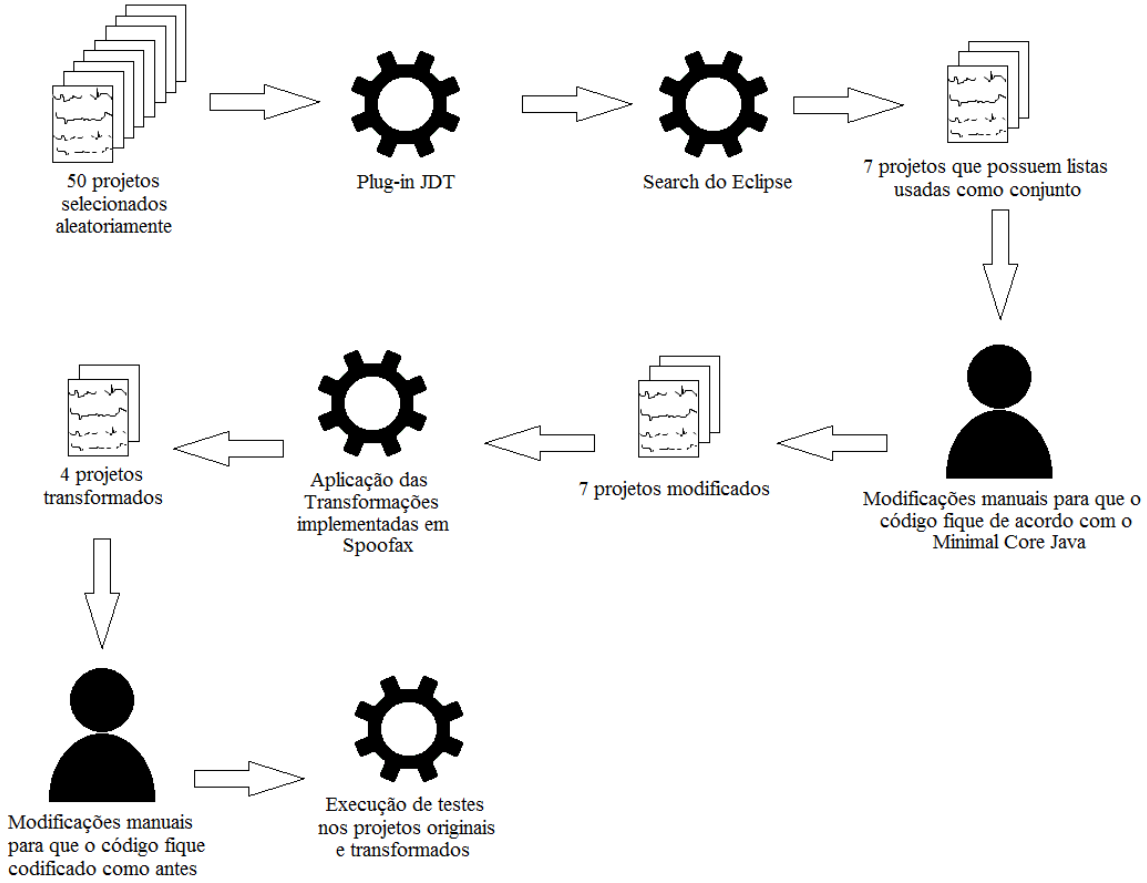

Figura 4.1: *Visão geral dos passos seguidos no Estudo de Caso*

4.2 Perguntas de Pesquisa

As perguntas de pesquisa que foram definidas para obter um melhor entendimento sobre as métricas que se pretendem avaliar, que são:

RQ1) Qual a proporção de projetos que tiveram pelo menos uma classe modificada pelas transformações?

RQ2) Dentre os projetos transformados, qual é a proporção de classes transformadas por projeto?

RQ3) Qual o percentual de uso dos tipos `List, ArrayList e LinkedList` nos atributos transformados?

RQ4) Foram detectados erros nas transformações realizadas?

RQ5) Quantas linhas foram modificadas no código original?

RQ6) Quantas vezes cada uma das transformações analisadas foram executadas?

4.3 Projetos Participantes

Os projetos, que foram objetos deste estudo, foram obtidos através do trabalho de Terra et al. [35], que disponibilizou 111 projetos Java *open source* pré-compilados que são utilizados em sua pesquisa. Por restrições de recursos disponíveis não foi possível realizar o estudo com todos eles, então foram selecionados aleatoriamente 50 projetos. Foram selecionados projetos *open source* por disponibilizarem o código fonte, e por muitos deles fazerem uso de coleções *JCF*. A Tabela 4.1 contém os 50 projetos selecionados.

Tabela 4.1: *Projetos open source selecionados para execução das transformações*

Projeto	**Versão Utilizada**	**Tamanho**
ant	1.8.2	37.1 MB
antlr	3.4	15 MB
aoi	2.8.1	9.4 MB
argouml	0.34	48 MB
aspectj	1.6.9	61 MB
axion	1.0-M2	2.5 MB
azureus	4.7.0.2	24 MB
batik	1.7	94 MB
castor	1.3.3	195 MB
cayenne	3.0.1	7.1 MB
checkstyle	5.6	53 MB
cobertura	1.9.4.1	11 MB
collections	3.2.1	2.4 MB
colt	1.2.0	4.9 MB
columba	1.0	45 MB
compiere	330	154 MB
c_jdbc	2.0.2	118 MB
derby	10.9.1.0	73 MB

displaytag	1.2	73 MB
drjava	20100913-r5387	10 MB
emma	2.0.5312	5.8 MB
exoportal	1.0.2	37 MB
findbugs	1.3.9	12 MB
fitjava	1.1	2.9 MB
hadoop	1.1.2	159 MB
heritrix	1.14.4	13 MB
informa	0.7.0-alpha2	12 MB
ireport	3.7.5	67 MB
itext	5.0.3	7.8 MB
jasml	0.1	66 MB
jedit	4.3.2	15 MB
jfreechart	1.0.13	13 MB
jmoney	0.4.4	1.4 MB
marauroa	3.8.1	4.2 MB
maven	3.0.5	18 MB
megamek	0.35.18	26 MB
mvnforum	1.2.2-ga	18 MB
myfaces_core	2.1.10	83 MB
quartz	1.8.3	12 MB
quickserver	1.4.7	3.1 MB
quilt	0.6-a-5	6.2 MB
struts	2.2.1	104 MB
tomcat	7.0.2	16 MB
velocity	1.6.4	26 MB
wct	1.5.2	151 MB
webmail	0.7.10	12 MB
weka	3.6.9	31 MB

xalan	2.7.1	67 MB
xerces	2.10.0	5.5 MB
xmojo	5.0.0	9.9 MB

4.4 Metodologia

Após a seleção dos 50 projetos, que foram objetos neste Estudo de Caso, foi verificado quais desses possuíam o problema estudado: possuir uma lista sendo usado como um conjunto. Para identificar esta característica através do código é preciso que ele satisfaça três condições: (1) que a classe possua uma variável do tipo lista, (2) que contenha uma verificação utilizando o método `contains(objeto)` e (3) que após esta verificação `objeto` seja adicionado caso ele já não esteja contido na lista. Estas duas últimas condições impede que ocorra a adição de elementos repetidos na lista.

Como seria um grande esforço verificar manualmente se as três condições, citadas no parágrafo anterior, são satisfeitas em alguma classe dos 50 projetos, foi criado um filtro, implementado como um *plug-in* JDT [15]. Quando este era executado, uma nova instância do eclipse era aberta, e nela eram importados os projetos participantes. Após, era selecionado um desses projetos do *workspace* e acionado um botão presente menu dessa nova instância do eclipse, assim era executado o *plug-in*. Então, em cada classe `.java` do projeto eram verificadas as condições (1) e (2), e caso fossem verdadeiras era impresso no *console* do eclipse o nome da classe, o tipo da lista em questão (`List, ArrayList, LinkedList,` etc) e qual o nome da variável.

O *plug-in* JDT realiza estas ações através do uso de *Visitors* que representam todos as listas (`ListVisitor`) e todos as invocações de métodos (`StatementVisitor`) e após realiza-se a filtragem de todas listas que realizam a invocação do método específico `contains`. Ao final é impresso no console do eclipse quais as classes que foram satisfeitas.

O passo seguinte era realizar a verificação manual da condição (3). Foi preferível fazer esta de formal manual pela dificuldade técnica em realiza-la de forma automática, por conta das várias situações de deveriam ser levadas em consideração. Após estas verificações foram

selecionados os seguintes sistemas para execução das transformações.

Tabela 4.2: *Projetos open source selecionados para execução das transformações*

Projeto	**Versão Utilizada**	**Tamanho**
aspectj	1.6.9	61 MB
azureus	4.7.0.2	24 MB
c_jdbc	2.0.2	118 MB
exoportal	1.0.2	37 MB
ireport	3.7.5	67 MB
struts	2.2.1	104 MB
xerces	2.10.0	5.5 MB

Então foram realizadas as seguintes etapas para estes:

1. Uma parte do sistema foi elegida para aplicar as transformações, que foram as classes retornadas através dos filtros citados anteriormente.

2. Duas versões dos códigos originais dos projetos foram salvas, uma para transformar e outra não.

3. As classes selecionadas foram modificadas manualmente para ficar de acordo com o *Minimal Core Java*, Seção 3.1, definido, o que as deixaram prontas para serem transformadas. Essas modificações eram basicamente as trocas dos comandos que não estavam definidos na linguagem por outros equivalentes que estavam. Como por exemplo os comandos `for` ou `do-while` que foram substituídos pelo `while`.

4. As transformações implementadas foram aplicadas, selecionando trechos das classes que satisfizeram as três condições citadas acima, e executando as transformações através do projeto desenvolvido no Spoofax. O trecho era colocado em um arquivo `.dir` e salvo o resultado em um arquivo `.java`, como foi descrito na Seção 3.5. Fazendo esse mesmo procedimento até que a classe fosse totalmente transformada.

5. As classes foram transformadas novamente para a codificação original, basicamente realizando a engenharia reversa do passo 3., retirando as mudanças feitas para que a classe ficasse de acordo com o `Minimal Core Java`. Caso a classe já estivesse escrita de acordo com o `Minimal Core Java` nenhuma ação seria necessária. Este passo juntamente com

os passos 3. e 4. teve um esforço em torno de 1 dia.

6. Foram executados os testes contidos nos projetos transformados, antes e depois das transformações, para verificar se a classe transformada e a original possuem uma Equivalência Semântica Fraca, Seção 3.4.2.

Entretanto, não foi obtido um resultado satisfatório, pois apenas em um projeto (*Azureus*) foi possível executar casos de teste. Os projetos *aspectj* e *xerces* não possuíam testes e no *c_jdbc* ocorreu um lançamento de exceção ao executarmos os casos de teste (`NullPointerException`). E ainda, no único projeto que conseguimos executar os casos de testes, só existia dois testes e estes não continham testes que cobrissem a classe que foi transformada. Isso levou a execução do próximo passo.

7. Foram aplicados testes gerados através do Randoop [36] para verificar se a classe transformada possui uma Equivalência Semântica Fraca com a classe original.

Para cada classe que seria modificada foi executado o comando Randoop para gerar seus testes durante 60 segundos. Por causa de dificuldades enfrentadas ao configurar o *classpath*, não foram gerados testes apenas nas classes `DistributedRequestManager` do projeto *c_jdbc* e `IncrementalImageBuilder` do projeto `aspect`, para as classes restantes os testes foram gerados normalmente.

8. Os resultados foram analisados como descritos posteriormente.

4.4.1 Instrumentação

As seguintes ferramentas foram utilizadas para este estudo:

1. Eclipse IDE: Desenvolvimento e execução do *software* de coleta dos dados.

2. Eclipse Java Development Tools (JDT): O *software* de coleta de dados foi criado como *plug-in*.

3. Spoofax: As transformações foram implementadas utilizando este *framework*.

4. Randoop: para gerar testes da classes transformadas, pois os testes contidos nos projetos não eram representativos para avaliar se as transformações continham Equivalência Semântica Fraca com o programa original. Isso será discutido com mais detalhes na Seção 4.5.4.

4.5 Resultados do Estudo de Caso

Nesta seção são apresentados os resultados obtidos neste estudo respondendo cada uma das perguntas de pesquisa 4.2 e ao final desta seção são apresentados outros resultados interessantes que não se encaixaram em nenhuma pergunta de pesquisa.

4.5.1 RQ1) Dentre todos os projetos, qual o percentual dos que tiveram pelo menos uma classe modificada pelas transformações?

Como foi visto, foram selecionados aleatoriamente 50 projetos (trechos de códigos reais). Dentre estes, 7 atenderam as condições de aceitabilidade para participação da pesquisa em pelo menos uma de suas classes, como mostrado na Tabela 4.2. Esse número corresponde a 14% dos projetos totais, o que representa um número significativo de projetos. Isso porque a utilização não consciente dessas coleções pode resultar em uma degradação significativa de desempenho, além de questões referentes à clareza de código e alocação de memória. De acordo com Mitchell [37] 90% do espaço consumido por coleções no programa é sobrecarga, ou seja apenas 10% da capacidade de armazenamento é utilizado.

Outro ponto que deve-se levar em consideração é o fato de que a presente pesquisa se aplica especificamente nos casos de trocas de interfaces `List` e `Set`, quando uma lista é usada como um conjunto. Esse número pode ser mais expressivo se avaliados outros possíveis problemas entre `List` e `Set`, e o universo de todas as coleções, resultando em um impacto ainda maior para o sistema.

Outra questão refere-se aos projetos selecionados, visto que não foi levado em consideração o processo de desenvolvimento, podendo haver uma maioria de projetos mais bem projetados dentre aqueles utilizados na pesquisa, podendo não representar a realidade dos sistemas como um todo.

Após a aplicação das transformações, nas classes que satisfizeram as três condições citadas no início deste capítulo, nos sete projetos da Tabela 4.2, foi verificado que várias dessas classes não puderam ser transformadas. Isso ocorreu devido à existência de métodos de `List` que não possuem correspondentes em `Set`. Os métodos Não Correspondentes citados na Seção 3.3.

O método Não Correspondente que teve maior frequência de uso foi o `get(index)`,

em que muitos casos apareciam numa varredura de elementos da lista, porém de acordo com os padrões de projeto seria mais adequado utilizar o método `iterator()`, demonstrando a existência de erros nos usos dos padrões de projeto, além dos usos inadequados das coleções. Isso afeta ainda mais a transformação dos atributos, variáveis e métodos associados à coleção *JCF* que terá seu tipo substituído. O que reforça o uso de uma técnica/abordagem que levem em consideração todos esses fatores no momento de auxílio do programador para a escolha da coleção *JCF* mais adequada ao seu contexto. Dos sete projetos, três não puderam ser transformados, resultando em um total de quatro projetos transformados.

Tabela 4.3: *Projetos open source transformados*

Projeto	**Versão Utilizada**	**Tamanho**
aspectj	1.6.9	61 MB
azureus	4.7.0.2	24 MB
c_jdbc	2.0.2	118 MB
xerces	2.10.0	5.5 MB

Finalizando com 4 projetos onde tiveram pelo menos uma classe modificada pelas transformações (classe transformada), que correspondem a 8% do total de projetos.

4.5.2 RQ2) Dentre os projetos transformados, qual é o numero de classes transformadas por projeto?

Uma classe transformada é aquela que foi selecionada através dos filtros citados anteriormente, e teve uma lista que estava sendo usada como conjunto transformada para `Set`, modificando também pelo menos um de seus métodos.

Antes de responder a pergunta de pesquisa, é visto que as distribuições de classes satisfeitas (que satisfizeram as três condições estudadas) por projetos se deu da seguinte forma:

Tabela 4.4: *Projetos open source selecionados para execução das transformações*

Projeto	**Versão Utilizada**	**Classes Satisfeitas**
aspectj	1.6.9	4
azureus	4.7.0.2	2
c_jdbc	2.0.2	2
exoportal	1.0.2	1
iReport	3.7.5	1
struts	2.2.1	1
xerces	2.10.0	6
Total		**17**

Os projetos contiveram uma quantidade de classe que variaram entre 1 a 6 classes. Sendo que a maior ocorrência se manteve entre 1 e 2 classes. Totalizando ao final 17 classes para todos os projetos. As quais foram submetidas às transformações, resultando em 7 classes que puderam ser transformadas e 10 não foi possível aplicar as transformações necessárias, como é visualizado nas Tabela 4.5 e Tabela 4.6.

Tabela 4.5: *Classes transformadas por Projeto*

Projeto	**Versão Utilizada**	**Classes Transformadas**
aspectj	1.6.9	3
azureus	4.7.0.2	1
c_jdbc	2.0.2	2
xerces	2.10.0	1
Total		**7**

Tabela 4.6: *Classes que não aceitaram as transformações por Projeto*

Projeto	**Versão Utilizada**	**Classes Não Transformadas**
aspectj	1.6.9	1
azureus	4.7.0.2	1
exoportal	1.0.2	1
iReport	3.7.5	1
struts	2.2.1	1
xerces	2.10.0	5
Total		**10**

Ao serem comparadas essas duas tabelas percebemos que 10 classes (quase 60% de todas as classes pré-selecionadas) não foram transformadas, apesar de possuírem lista(s) com comportamento(s) de conjunto(s). Esta diferença significativa se deu pelo fato de que nestas classes possuía, em algum trecho do código, pelo menos uma chamada ao método `get(index)` de `List` que pela Tabela 3.1 não possui método equivalente em `Set`, impossibilitando assim a transformação da coleção para evitar inserção de erros de compilação no código original.

Um exemplo dessa situação é vista no Código Fonte 4.1, onde a variável `fComponents` é uma lista que seria transformada para `Set`, mas não é por causa da chamada `get(i)` dentro do comando `for`. Outro é observado no Código-Fonte 4.2, que apenas retorna o objeto da posição `fileid` da lista `sourceFiles` que provavelmente é chamado em outra parte do projeto.

```
...
int count = fComponents.size();
for (int i = 0; i < count; i++) {
   XMLComponent c = (XMLComponent) fComponents.get(i);
   c.setFeature(featureId, state);
}
...
```

Código Fonte 4.1: *Trecho de código retirado do projeto xerces da classe `XML11Configuration` no método `setFeature`*

```
...
String getFile(final int fileid) {
   return (String) sourceFiles.get(fileid);
}
...
```

Código Fonte 4.2: *Trecho de código retirado do projeto structs da classe* `JspReader`

4.5.3 RQ3) Qual o percentual de uso dos tipos `List, ArrayList e LinkedList` nos atributos transformados?

Entende-se por atributo transformado aquele que mudou seu tipo ou instanciação através das transformações propostas. Analisando as 7 classes passiveis de transformação quanto à ocorrência das instanciações de lista `List, ArrayList e LinkedList` que são as mais comuns, observou a seguinte distribuição disposta no gráfico seguinte.

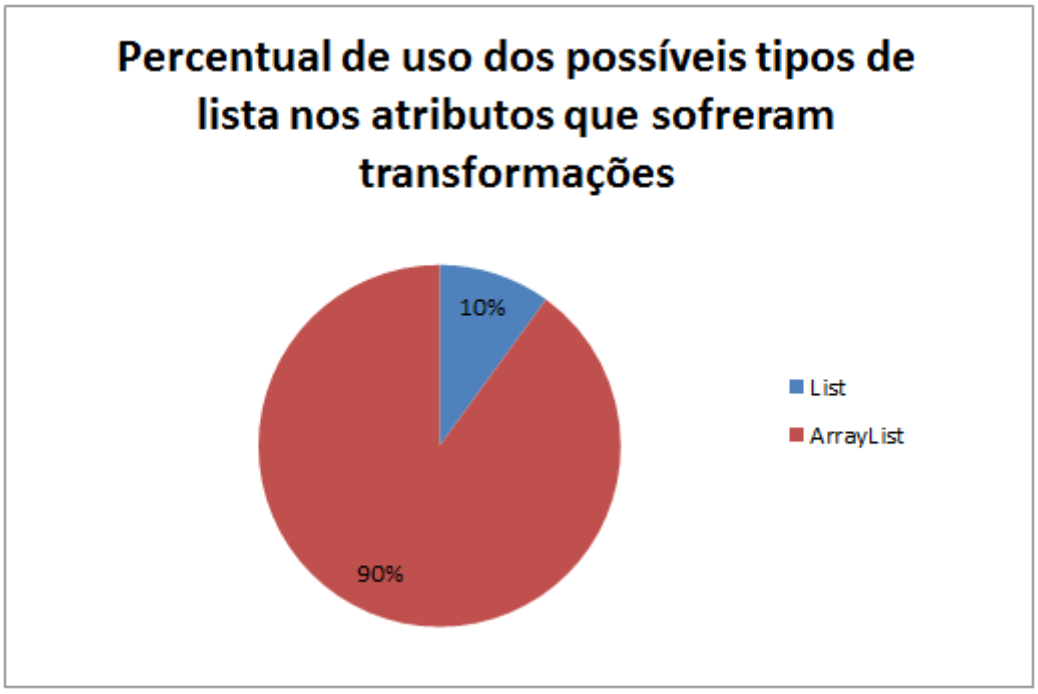

Figura 4.2: *Percentual de dos tipos* `List, ArrayList e LinkedList` *nos atributos transformados*

De acordo com esses dados são identificadas as frequências no uso destas instanciações

de `ArrayList` (90%) e `List` (10%), inexistindo ocorrência para `LinkedList`. Como as transformações propostas neste documento consideram apenas substituições entre as interfaces `List e Set`. E observando o dado que mostra que a grande maioria das listas transformadas são do tipo `ArrayList`, faz-se necessário realizar uma adequação futura nas transformações, para que elas passem a considerar também substituições entre classes, principalmente envolvendo `ArrayList`.

4.5.4 RQ4) Foram detectados erros nas transformações realizadas?

A quantidade de testes gerados através do Randoop [36], como foi descrito no passo 7. da Metodologia, Seção 4.4, para cada classe está apresentada na Tabela 4.7, além disso nesta tabela temos a porcentagem de cobertura desses testes para a classe que eles testam e se a(s) parte(s) da classe que contém um IfContains, Seção A.2, é (são) testada(s). Estas informações foram obtidas com a execução da ferramenta Eclemma [38] sob os testes gerados no Randoop.

Tabela 4.7: *Quantidade e Cobertura dos Testes Por Classe*

Classe	**Quantidade de Testes Gerados**	**Porcentagem de Cobertura da Classe**	**O(s) IfContains foi(foram) coberto(s) pelos testes**
Aspectj			
Relationship.java	8740	90,5%	Não
MessageHandler.java	1892	49,8%	Sim
IncrementalImageBuilder.java	0	0%	Não
Azureus			
TrackersUtil.java	330	54%	Sim
c_jdbc			
GUISession.java	4330	49,8%	Sim
DistributedRequestManager.java	0	0%	Não
Xerces			
ParserConfigurationSettings.java	262	98,4%	Sim
Total de Testes	**15554**	-	-

Através da Tabela 4.7 percebe-se que a classe `Relationship` apesar de possuir testes

um alto grau de cobertura nenhum desses cobre a parte do código que executa o método de inserção de um elemento em uma lista, precedido por uma verificação (`if`), o que caracteriza uma lista sendo usada como conjunto. Ao ser analisado o motivo dessa não cobertura de código, foi visto que este método não é executado em nenhuma outra parte do projeto, sendo portanto considerado um código morto (*dead code*).

As classes ao serem transformadas não continham nenhum erro de compilação, com exceção da classe `GUISession` do sistema `c_jdbc` ao realizar a transformação das coleções `controllerItems` e `configurationFiles` que eram `ArrayList` para `HashSet`, apresentou dois erros em duas chamadas de um método externo, `writer.write`, que esperava objetos `ArrayList`. Isto é visto no Código Fonte 4.3 nas linhas 10 e 13.

```
1  public void saveSessionToFile(File sessionFile) throws
       IOException{
2      BufferedWriter writer = new BufferedWriter(new FileWriter(
           sessionFile));
3      if (saveDatabaseInfoToDisk){
4          writer.write("### DATABASES           ###"
5          + System.getProperty("line.separator"));
6          writer.write(ReadWrite.write(databaseItems, false));
7      }
8      writer.write("### CONTROLLERS         ###"
9      + System.getProperty("line.separator"));
10     writer.write(ReadWrite.write(controllerItems, "controller",
           false));
11     writer.write("### CONFIGURATION FILES ###"
12     + System.getProperty("line.separator"));
13     writer.write(ReadWrite.write(configurationFiles, "
           configuration", false));
14     writer.close();
15 }
```

Código Fonte 4.3: *Código da classe `GUISession` com erro de compilação*

Erros como estes não foram possíveis de ser contornados pelas transformações propostas. Para solucionar este impasse pode-se no futuro pesquisar formas de análise estática

mais completas que possam identificar tais problemas de checagem de tipos das variáveis envolvidas na transformação.

Foi realizada manualmente a correção necessária na classe `GUISession`, passando como parâmetro para os métodos que deram errado um `new ArrayList(controllerItens)`, já que `controllerItens` após a transformação ficou do tipo `Set`. Esforço necessário foi de duas horas para esta tarefa.

Então, foram gerados e executados testes antes das classes serem transformadas e executados os mesmos testes depois de suas transformações. Realizando apenas algumas adequações manuais nos testes para considerar objetos `Set ou HashSet` e não mais `List ou ArrayList`. As transformações não foram aplicadas nos testes pois os mesmos não são originalmente classes do projeto e porque eles simplesmente realizam as chamadas dos métodos da classe transformada, não possuindo as características necessárias para ter uma lista usada como conjunto.

Como os testes tiveram uma porcentagem de cobertura considerável, e todos os testes foram bem sucedidos tanto antes quanto depois das transformações, tem-se um indício que os códigos Transformados possuem uma Equivalência Semântica Fraca com o código original. Dessa forma os únicos erros apresentados foram os ocorridos na classe `GUISession`, após as transformações, que foram mencionados anteriormente.

4.5.5 RQ5) Quantas linhas foram modificadas no código original?

Para verificar quantas linhas foram modificadas iremos mostrar a Tabela 4.8 que também mostra o total de linhas de cada trecho no código original e no transformado.

Tabela 4.8: *Quantidade de Linhas Modificadas*

Classe	**Atributo ou Método**	**Quantidade de Linhas no Código Original**	**Quantidade de Linhas Removidas**	**Quantidade de Linhas Adicionadas**	**Quantidade de Linhas Modificadas**
Aspectj					
Relationship.java	atributo1	1	0	0	1
	método1	20	0	0	1
	método2	3	0	0	1
	método3	4	1	0	1
MessageHandler.java	atributo2	1	0	0	1
	método4	6	0	0	1
	método5	5	0	0	1
IncrementalImageBuilder.java	atributo3	1	0	0	1
	método6	33	1	0	1
	método6	53	0	0	2
	método7	21	0	0	1
Azureus					
TrackersUtil.java	atributo4	1	0	0	1
	método8	6	0	0	1
	método9	6	0	0	2
	método10	6	2	0	3
	método11	6	0	0	1
c_jdbc					
GUISession.java	atributo5	1	0	0	1
	atributo6	1	0	0	1
	método12	6	0	0	2
	método13	4	0	0	1
	método14	5	0	0	1
DistributedRequestManager.java	atributo7	1	0	0	1
	método15	12	0	0	1
	método16	13	1	0	1
	método17	14	1	0	1
	método18	27	1	0	1
	método19	13	1	0	1
Xerces					
ParserConfigurationSettings.java	atributo8	1	0	0	1
	atributo9	1	0	0	1
	método20	14	0	0	2
	método21	12	1	0	1
	método22	12	1	0	1

Com a análise desses dados percebe-se que as modificações feitas pelas transformações foram mínimas, nunca inserindo código diferente do original e com uma quantidade muito pequena de linhas modificadas, que em sua vasta maioria foi 1 linha e só existiu um caso com 3 linhas modificadas. Com isso, tem-se um indício de que o código transformado possa ser mais bem aceito pelo desenvolvedor, pois não modifica muito o seu código. Mesmo assim ocorre uma melhora, pois fará com que o código utilize a coleção mais apropriada.

4.5.6 RQ6) Quantas vezes cada uma das transformações analisadas foram executadas?

Nesta pergunta de pesquisa foi analisado todos os trechos de código que continham algum indício de uma lista sendo usada como um conjunto (um IfContains, Seção A.2) verificando em qual tipo de transformação, Seção 3.4.3, ela se encaixa. Como resultado temos a Tabela 4.9 onde todas as transformações aplicadas foram a Transformação 1 (do Tipo Simples).

Tabela 4.9: *Quantidade de Transformações aplicadas Por Classe*

Classe	Quantidade de execuções da Transformação 1
Aspectj	
Relationship.java	1
MessageHandler.java	1
IncrementalImageBuilder.java	3
Azureus	
TrackersUtil.java	1
c_jdbc	
GUISession.java	2
DistributedRequestManager.java	4
Xerces	
ParserConfigurationSettings.java	2

Este resultado dá um indício de que as outras transformações investigadas não possuem uma aplicabilidade considerável, já a Transformação 1 é a que foi unicamente executada neste estudo de caso, devendo assim ter mais enfoque do que as demais. Poderia ser interessante em trabalhos futuros desmembrar este tipo de Transformação para que ela seja aplicada

em qualquer tipo de codificação, e não somente naqueles que estejam escritas de acordo com o *Minimal Core Java* definido.

4.5.7 Outros resultados

Nesta seção são apresentados outros resultados relevantes que foram encontrados no estudo de caso, mas que não foram abordados por nenhuma *pergunta de pesquisa*.

Um resultado interessante, encontrado ao executar este estudo de caso, é a variedade de implementações diferentes para definir a situação de existência de uma lista funcionando como um conjunto. A forma de codificar definida na *Minimal Core Java* 3.1 para este caso, que é chamado de IfContains, Seção A.2, foi a do Código Fonte 4.4, porém nos Códigos Fonte 4.5, 4.6 e 4.7 existem outros casos que podem ser reduzidos/transformados para o primeiro caso, podendo realizar então as transformações deste trabalho.

```
...
if (!fRecognizedFeatures.contains(featureId)) {
   fRecognizedFeatures.add(featureId);
}
...
```

Código Fonte 4.4: *1º Caso de IfContains*

```
public void addTarget(String handle) {
   if (targets.contains(handle)) return;
      targets.add(handle);
   }
```

Código Fonte 4.5: *2º Caso de IfContains*

```
...
next : for (int i = 0, l = valueTable.length; i < l; i++) {
   ...
   if (sourceFiles.contains(sourceFile)) continue next;
   ...
   sourceFiles.add(sourceFile);
}
...
```

Código Fonte 4.6: *3º Caso de IfContains*

```
...
public void addTracker(String trackerAnnounceUrl) {
if(trackers.contains(trackerAnnounceUrl))
   return;
trackers.add(0,trackerAnnounceUrl);
saveList();
}
...
```

Código Fonte 4.7: *4º Caso de IfContains*

Entretanto, foi encontrado um caso, no Código Fonte 4.8 que se parece bastante com o IfContains, mas ao realizar uma análise percebe-se que ele verifica a não existência na lista `ignoring` de um atributo do objeto `message` e adiciona este último em outra lista, `messages`. Situações como esta podem confundir o programador, que pode inserir erros no seu código ou desistir de fazer a substituição para a coleção mais adequada, o que provavelmente não aconteceria se ele utilizasse as transformações propostas por este estudo.

```
...
if (!ignoring.contains(message.getKind())) {
   messages.add(message);
}
...
```

Código Fonte 4.8: *Caso parecido com IfContains*

Outro resultado interessante, não avaliado pelas perguntas de pesquisa, é a quantidade de modificações necessárias no restante do código do projeto para que ele se adequasse à nova coleção, desconsiderando as modificações nas classes de teste geradas pelo Randoop [36]. Em algumas classes não foi feita nenhuma adequação de código de outras classes, como a classe `MessageHandler` do projeto aspectj e `DistributedRequestManager` do projeto c_jdbc, o que significa que a transformação feita foi local. Já em outras classes, poucas mudanças nas demais partes do projeto foram necessárias (como a classe `GUISession` do projeto c_jdbc e a `TrackersUtil` do projeto azureus). E por fim em duas classes, a `Relationship` do projeto aspectj e a `ParserConfigurationSettings` do projeto xerces, ocasionaram diversos erros de compilação em outras partes do projeto após a transformação, o que necessitou uma série de adequações de código que precisam de bastante cuidado ao serem feitas para não inserção de erros no sistema.

Observa-se então que seria preciso uma análise mais criteriosa nesse aspecto e pesquisar mais possibilidades de transformações, considerando não apenas uma classe, e sim todo um projeto Java.

4.6 Discussão

Através desse estudo percebe-se que a situação de ter uma lista sendo utilizada como um conjunto possui uma porcentagem de aparição de 14% dos 50 projetos participantes. O que é um número significativo, levando em consideração que só estão sendo analisadas substituições entre as interfaces `List` e `Set`, de um universo de várias coleções de *JCF*.

Também observa-se que boa parte dos códigos que não foram transformados continham erros de padrão de projeto, por apresentar, em sua maioria, a chamada para o método `get(index)`. Em muitos casos aparecia numa varredura de elementos da lista, porém de acordo com os padrões de projeto seria mais adequado utilizar o método `iterator()`.

Conclui-se que não é tão relevante realizar transformações apenas entre interfaces `List` e `Set`, pois o uso da interface `List` é explicitamente baixo, somente 10% dos atributos transformados eram desse tipo. O melhor seria criar transformações envolvendo também, pelo menos a classe `ArrayList`, que correspondeu aos outros 90% dos casos.

Outro fato é que as transformações realizadas tiveram um impacto pequeno no código,

pois modificou em cada transformação de atributo ou de método, na maior parte dos casos uma linha, chegando só em um caso a modificar 3 linhas. E nunca adicionando mais linhas (todos os casos foi zero). O que dá um indício da boa aceitabilidade dessas transformações por parte dos programadores, já que não irá realizar muitas mudanças e nem inserir nenhum tipo de código gerado, como acontece com os adaptadores.

Este Estudo de Caso foi muito importante na consolidação e melhoramento das transformações. E ainda na concepção de novas situações ainda não observadas. Como por exemplo, considerar na implementação da ferramenta de reescrita declarações e instruções. Na **categoria de declarações** tem-se: *imports* com vários pontos e.*, implementações de interface, nomes de pacotes com vários pontos, métodos construtores com parâmetros, variáveis *Long*, declarações de variáveis do tipo *array*. Já na **categoria instruções** existem: anotações, *casts*, utilizar uma classe com todo o caminho do seu pacote, como por exemplo: `if (org.aspectj.org.eclipse.jdt.internal.core.util.Util.isJavaLikeFileName(proxy.getName()))`.

Percebe-se também que as substituições devem ocorrer somente entre interfaces, e somente entre classes. Como por exemplo, não substituir `ArrayList` para `Set`, pois existem métodos não correspondentes que caso fosse feita a troca por `HashSet` existiria (método `clone()`). E pensar nas substituições entre as coleções envolvidas de um modo mais completo. Uma vez que ao analisar um estudo de caso com situações e objetos reais, não existem abstrações e percebendo os fatores como eles são na realidade.

E também auxiliou a observação de uma transformação alternativa para os trechos de código que utilizam o método `get(index)` dentro de um `for` que faz a varredura da lista. Esta situação foi abordada na RQ2), Seção 4.5.2, que foi um dos casos que impossibilitou a transformação de várias classes que continham uma lista sendo usada como um conjunto.

Uma solução, que se enquadraria como sendo uma **Transformação Alternativa** 3.4.7, para possibilitar a realização da transformação das classes que possuíssem tal situação seria a que contem o *template* no Código Fonte 4.9 para o código transformado do Código Fonte 4.10.

```
List collection;
commands1
var1 = collection.size();
for(int i=0; i< var1; i++){
  commands2
  var2 = collection.get(i);
  commands3
}
commands4
```

Código Fonte 4.9: *Exemplo de padrão de código*

```
Set collection;
commands1
Iterator iter = collection.iterator();
while(iter.hasNext()){
  commands2
  var2 = iter.next();
  commands3
}
commands4
```

Código Fonte 4.10: *Transformação alternativa proposta*

provisos

1. `commands1`, `commands2`, `commands3` e `commands4` podem ser vazios
2. `commands2`, `commands3` e `commands4` não podem conter manipulação das variáveis `i` e `var1`
3. `commands1`, `commands2`, `commands3` e `commands4` não podem conter nenhum método Não Correspondente (Tabela 3.1) sendo chamado a partir da lista envolvida na instrução `notContainsCall`

Que no exemplo citado na RQ2) resultaria na seguinte transformação:

```
List fComponents;
//...
int count = fComponents.size();
for (int i = 0; i < count; i++) {
  XMLComponent c = (XMLComponent)
  fComponents.get(i);
  c.setFeature(featureId, state);
}
//...
```

Código Fonte 4.11: *Exemplo de código usando `get(index)`*

```
Set fComponents;
//...
Iterator iter = fComponents.iterator();
while (iter.hasNext()) {
  XMLComponent c = (XMLComponent)
  iter.next();
  c.setFeature(featureId, state);
}
//...
```

Código Fonte 4.12: *Transformação alternativa*

4.7 Ameaças à validade

Dentre os tipos de ameaças à validade descritos em Wohlin et al. [39], abaixo selecionamos algumas delas que podem afetar o estudo de caso.

1. Ameaça à validade interna:
   Instrumentation - caso os artefatos usados para realização do estudo estejam errados ou com vícios, ele é afetado negativamente.

   (a) *Transformações*: As transformações criadas, assim como qualquer software, já que foram implementas utilizando Spoofax, estão suscetíveis a erros, o que pode vir a invalidar as classes transformadas, assim como a coleta de dados realizada que foi realizada pelo plug-in JDT para filtrar os projetos que continham as condições (1) e (2).

   (b) Estudo de Caso: Por ser um estudo de caso, a análise e a coleta estão mais abertas a um viés do pesquisador.

2. Ameaça à validade externa:
   Interaction of selection and treatment - Se os sujeitos da população não forem representativas da população que queremos generalizar.

   (a) Projetos escolhidos - Os projetos *open source* selecionados podem ser não representativos da população.

   (b) Estudo de caso - Por ser um estudo de caso, isto o torna menos flexível fazendo com que seja mais difícil generalizar para outras populações.

4.8 Considerações Finais do Capítulo

Como foi visto neste capítulo, o Estudo de Caso realizado dá um indício de que uma parte considerável dos sistemas reais participantes possuem uma lista sendo usada como conjunto, podendo também existir uso de outras coleções de forma inadequada. Então, as transformações analisadas podem ter uma grande utilidade nesses casos, já que elas deixaram o código mais adequado sem modifica-lo muito. No capitulo seguinte as transformações também são avaliadas, porém de acordo com a perspectiva de programadores Java e *JCF*.

Capítulo 5

Avaliação: *Survey*

Neste capítulo é descrito o *Survey* que, de acordo com Andrade [40], é definido como sendo "um conjunto de questões, feito para gerar os dados necessários para se verificar se os objetivos de um projeto foram atingidos.".

5.1 Objetivo do *Survey*

Este método de pesquisa empírica foi utilizado objetivando avaliar as transformações propostas para trechos de códigos nas substituições das interfaces de `List` para `Set`, quando uma lista é usada como conjunto, pela perspectiva dos desenvolvedores *JFC*. Ele tem a finalidade de medir a aceitabilidade e o grau de legibilidade das transformações perante os programadores, a partir da análise de trechos de código propostos no questionário.

5.2 Perguntas de Pesquisa

As perguntas de pesquisa que deseja-se responder com este *Survey* são:

RQ1) Quantos anos de experiência em Java e em `JCF` os participantes possuem?

RQ2) Qual a porcentagem das sugestões consideradas mais adequadas para o código 1 e o código 2?

RQ3) Qual o grau de legibilidade do código original e transformado?

RQ4) O que os participantes perceberam dos resultados das transformações?

RQ5) Após o participante ver a transformação sugerida pelo autor, qual sua escolha de transformação a ser feita para o código 1?

RQ6) Após o participante ver a transformação sugerida pelo autor, qual sua escolha de transformação a ser feita para o código 2?

RQ7) Os participantes consultaram a API de *JCF* para responder o questionário?

5.3 Metodologia

O *survey* foi realizado pela aplicação de um Questionário Não-Supervisionado disponível online [41]. Ele foi dividido em quatro partes: **Informações Profissionais, Transformações, Análise das Transformações e Conclusão**.

Antes do participante começar a responder as perguntas era apresentado para ele a seguinte mensagem: *"Neste questionário você irá responder algumas perguntas sobre certas situações onde coleções pertencentes ao Java Collections Framework (JCF) são utilizadas. Leia atentamente as instruções contidas nas demais seções deste questionário. Ao enviar sua resposta neste questionário você estará de acordo com os termos de consentimento que se encontram neste link."* O link em questão se encontra no seguinte *site* [42].

5.3.1 Questionário: Informações Profissionais

Nesta seção o participante respondeu duas questões de caráter profissional:

1. Quantos anos de experiência em Java?

2. Quantos anos de experiência em *JCF*?

Caso o participante não tenha experiência em Java ou *JCF* não seria possível prosseguir o questionário.

5.3.2 Questionário: Transformações

Aqui foram apresentados 2 trechos de código e depois indagadas perguntas sobre possíveis mudanças nesse código a critério do participante.

Código 1:

Diante deste código é importante considerar que em nenhuma outra parte do programa é importante à ordem e indexação dos elementos das coleções `messages` e `ignoring`.

```
protected final ArrayList messages;
protected final ArrayList ignoring;
...
messages = new ArrayList();
ignoring = new ArrayList();
...
public void ignore(IMessage.Kind kind) {
   if ((null != kind) && (!ignoring.contains(kind))) {
      ignoring.add(kind);
   }
}
...
```

Código Fonte 5.1: Código 1 do questionário: 1ª parte

O Código Fonte 5.1 representa a 1ª parte do código 1. Nela existem duas listas como atributos `messages` e `ignoring`, ambas `ArrayList`. O participante precisará analisar se é necessária a mudança de algum desses atributos. Através do método `ignore` percebe-se que ocorre um bloqueio de elementos `kind` repetidos na lista `ignoring` através de um IfContains A.2, o que caracteriza um comportamento de conjunto a ela, sendo necessária sua transformação.

```
...
public boolean handleMessage(IMessage message) {
   ...
   if (null == message) {
      throw new IllegalArgumentException("null message");
   }
   if (!ignoring.contains(message.getKind())) {
      messages.add(message);
   }
   return handleMessageResult;
}
...
```

Código Fonte 5.2: Código 1 do questionário: 2ª parte

Já o Código Fonte 5.2 é a 2ª parte do Código 1, e possui o método `handleMessage`, que lança uma exceção do tipo `IllegalArgumentException` caso o objeto `message` passado por parâmetro seja nulo. Depois existe um IfSimples A.1 que por ser muito parecido com um IfContains pode ser confundido com tal. Neste `if` é verificado, também na coleção `ignoring`, se não existe o objeto `message.getKind()`, e no corpo desse `if` é adicionado o objeto `message` na coleção `messages`. Isto não caracteriza o comportamento de conjunto, não devendo modificar o tipo da coleção `messages`. Juntamente com as transformações da 1ª parte, é resultado o código contido no Código Fonte 5.3, que é considerado mais adequado.

```
...
protected final ArrayList messages;
protected final Set ignoring;
...
ignoring = new HashSet();
messages = new ArrayList();
...
public void ignore(IMessage.Kind kind) {
   if ((null != kind)) {
      ignoring.add(kind);
   }
}

public boolean handleMessage(IMessage message) {
   ...
   if (null == message) {
      throw new IllegalArgumentException("null message");
   }
   if (!ignoring.contains(message.getKind())) {
      messages.add(message);
   }
   return handleMessageResult;
}
...
```

Código Fonte 5.3: Transformação para o código 1 do questionário

O participante respondeu questões subjetivas e objetivas, referente ao código 1, apresentadas a seguir:

Questões subjetivas:

- Você substituiria o tipo do atributo "`protected final ArrayList messages`"? Se sim, por qual?

- Você substituiria o tipo do atributo "`protected final ArrayList ignoring`"? Se sim, por qual?
- Você substituiria o tipo da instanciação nessa linha "`messages = new ArrayList();`"? Se sim, por qual?
- Você substituiria o tipo da instanciação nessa linha "`ignoring = new ArrayList();`"? Se sim, por qual?

Questões objetivas:

1. "O que você modificaria no método "`public void ignore(IMessage.Kind kind)`" Levando em consideração as modificações das perguntas anteriores?", com as opções:
   (a) Nada
   (b) Retirava o trecho "`(null != kind) &&`" da verificação do if
   (c) Retirava o trecho "`&& (!ignoring.contains(kind))`" da verificação do if
   (d) Retirava o if deixando no corpo do método apenas a linha "`ignoring.add(kind);`"
2. "O que você modificaria no método "`public boolean handleMessage(IMessage message)`" Levando em consideração as modificações das perguntas anteriores?", com as opções:
   (a) Nada
   (b) Retirava o 3º if deixando a instrução "`messages.add(message);`" ser executada sem verificação

Código 2:

Diante desse código deve ser levado em consideração que em nenhuma outra parte do programa é importante a ordem e indexação dos elementos das coleções `bosses`.

```
private List bosses;
bosses = new LinkedList();

public void changeBoss(Employee oldBoss, Employee newBoss){
   if (!bosses.contains(newBoss)){
      bosses.set(bosses.indexOf(oldBoss), newBoss);
   }
}
...
```

Código Fonte 5.4: Código 2 do questionário

Analisando o método `changeBoss` percebe-se que existe um IfContains A.2, onde em seu corpo existe o método `set(bosses.indexOf(oldBoss), newBoss)`, que adiciona o `newBoss` no lugar de `oldBoss`. A lista `bosses` se tornaria mais adequada se fosse transformada para um `Set` e o IfContains é retirado do código com a transformação do método `set(int index, E e)` para as chamadas de método `bosses.add(newBoss);` e `bosses.remove(oldBoss)`. O que resulta no código transformado contido no Código Fonte 5.5.

```
private Set bosses;
...
bosses = new HashSet();
...

public void changeBoss(Employee oldBoss, Employee newBoss){
   bosses.remove(oldBoss);
   bosses.add(newBoss);
}
```

Código Fonte 5.5: Transformação para o código 2 do questionário

O participante respondeu questões subjetivas e objetivas referente ao código 2 apresentadas a seguir.

Questões subjetivas:

- Você substituiria o tipo do atributo "`private List bosses;`"? Se sim, por qual?
- Você substituiria o tipo da instanciação nessa linha "`bosses = new LinkedList();`"? Se sim, por qual?

Questão objetiva: (Que pode selecionar mais de uma alternativa)

- "O que você modificaria no método "`public void changeBoss(Employee oldBoss, Employee newBoss)`", levando em consideração as modificações das perguntas anteriores?" alternativas (podendo ser selecionada mais de uma alternativa):
  (a) Nada
  (b) Retiraria o "if (!bosses.contains(newBoss))"
  (c) Substituiria a instrução "bosses.set(bosses.indexOf(oldBoss), newBoss);"

5.3.3 Questionário: Análise das Transformações

Nesta parte do questionário o participante analisou transformações propostas para os Códigos 1 e 2 e respondeu questões referentes à sua percepção diante de tais transformações.

Transformação do código 1:

Foi apresentado o código 1 (Código Fonte 5.1 e Código Fonte 5.2), que foi chamado de código original, juntamente com o código transformado (resultado da aplicação das transformações definidas neste documento no código 1), que se encontra no Código Fonte 5.3.

A partir dessa análise do código original e do transformado, o participante da pesquisa respondeu a seguinte questão:

1. "Qual transformação você faria no Código 1?", opções:
   - A transformação proposta pelo autor do questionário
   - As suas sugestões de mudança feitas anteriormente

- As suas sugestões de mudança são iguais a transformação proposta pelo autor do formulário.
- Não sei dizer

Também possuía um campo de texto para que ele justificasse esta resposta.

A seguir ele devia marcar em uma escala de 1 a 5 "Qual o grau de facilidade de entendimento do código original (Quanto mais estrelas, mais fácil de entender o código é)", e o mesmo para o código transformado.

Transformação do código 2:

Foi apresentado o Código Fonte 5.4, o código 2, que foi chamado de código original, juntamente com o Código Fonte 5.5, que foi chamado de código transformado (resultado da aplicação das transformações definidas neste documento no código 2).

O participante respondeu as mesmas questões citadas na situação anteriormente só que agora aplicadas ao código 2.

5.3.4 Questionário: Conclusão

E por fim, na conclusão do questionário o participante precisava responder "Você acessou a API de Java Collections Framework para realizar este questionário?", com as opções:

- Sim
- Não
- Não, mas pesquisei sobre Java Collections em outras fontes

E caso ele deseje obter os resultados deste questionário deixasse seu email num campo de texto.

5.3.5 Etapas do *Survey*

As etapas realizadas nesse estudo empírico foram as seguintes:

1. Definição do objetivo principal do *survey*.

2. Formulação das perguntas do questionário.

3. Execução piloto do questionário com dois desenvolvedores Java do SPLab (laboratório da UFCG).

4. Modificação do questionário para atender às críticas apresentadas na execução piloto, como por exemplo, modificar a maneira de realizar as perguntas, indo mais direto ao ponto, para que o participante não tivesse muita dificuldade. Isso ocorreu nas perguntas sobre os atributos lista, que foi solicitado que dividisse em duas perguntas, sobre qual tipo e instanciação respectivamente seria preferível realizar a transformação, e inserindo uma explicação adicional na pergunta sobre o grau de facilidade de entendimentos dos códigos original e transformado, que foi: "(Quanto mais estrelas, mais fácil de entender o código é)".

5. Envio do link do questionário para listas de email e grupos de redes sociais que provavelmente possuíssem programadores Java. Como por exemplo, a lista dos alunos da graduação e pós-graduação da UFCG, as comunidades dos alunos da UFPB, UEPB e IFPB.

6. Os dados do questionário foram coletados entre as datas 20/12/2016 e 04/01/2017. Foram obtidas respostas de 43 participantes diferentes, que nomeou-se **Respostas A**.

7. Realização de uma filtragem a fim de desconsiderar as resposta daqueles que demonstraram não ter domínio suficiente em *JCF*, o que resultou em 15 programadores, que nomeou-se **Respostas B**.

8. Os resultados foram analisados e comparados entre as *Respostas A* e as *Repostas B* como descritos posteriormente.

5.3.6 Instrumentação

As seguintes ferramentas foram utilizadas neste *Survey*:

1. Eclipse IDE: para definir os códigos a serem analisados

2. Jotformz: para criação do questionário e armazenamento das respostas

3. Planilha eletrônica: para auxílio na análise dos dados coletados e criação dos gráficos.

4. R Studio: para auxílio na análise dos dados coletados e criação dos gráficos.

5.4 Resultados do *Survey* e Discussões

Para melhor compreensão são apresentados os resultados obtidos respondendo cada uma das perguntas de pesquisa 5.2

5.4.1 RQ1) Quantos anos de experiência em Java e em `JCF` os participantes possuem?

Em se tratando das questões referentes às Informações Profissionais obtivemos os seguintes dados dispostos nos gráficos seguintes, Figura 5.1 e Figura 5.2

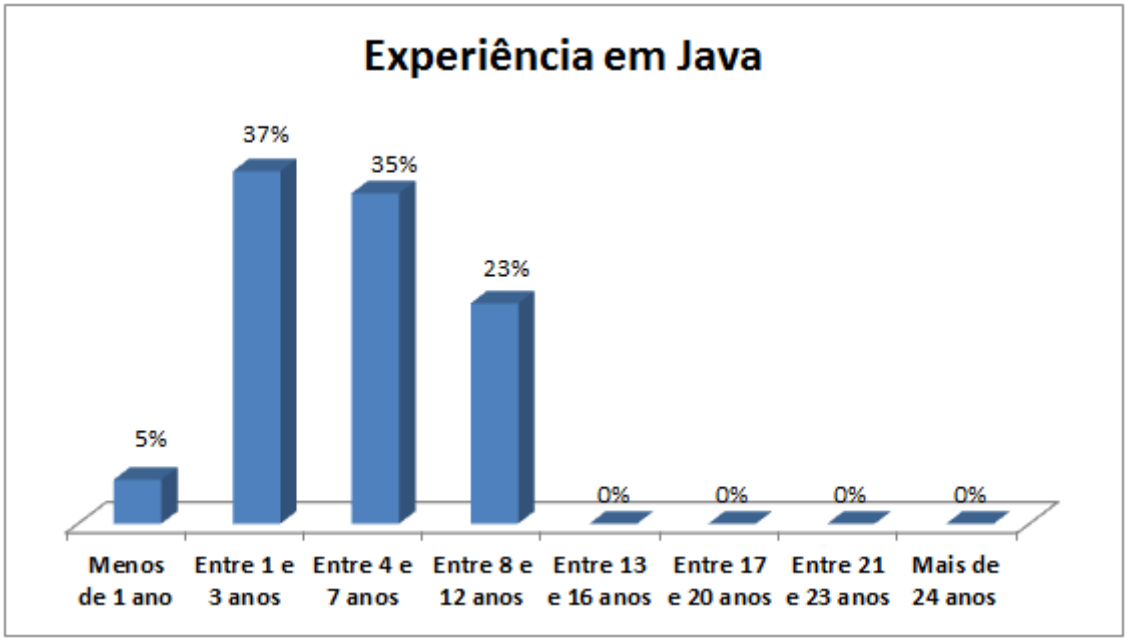

Figura 5.1: Experiência em Java

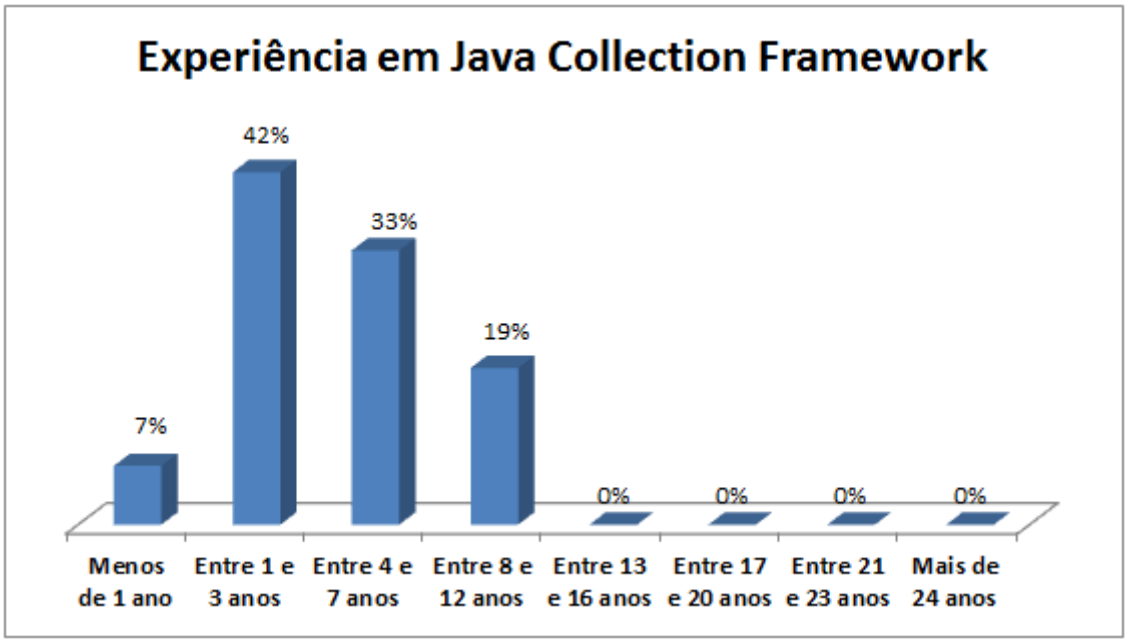

Figura 5.2: Experiência em *JCF*

É observado que boa parte dos participantes possuíam certa experiência em Java (23% entre 8 e 12 anos), assim como em *JCF* (com 19% entre 8 e 12 anos). E o restante deles possuindo experiência entre 1 e 7 anos.

Assim que a população participante da pesquisa possuía um grau de experiência em Java e JCF considerável, o que leva a crer que eles possuíam um certo conhecimento sobre programação orientada e capacidade de usar da forma adequada as coleções *JCF*.

Participantes que dominavam mais o *JCF*

Ao analisar esta pergunta de pesquisa nas *Respostas B* percebe-se, através da Figura 5.3 e da Figura 5.4, que nível de experiência da maioria deles é mantido que ficam entre 1 e 7 anos de experiências tanto em Java quanto em *JCF*.

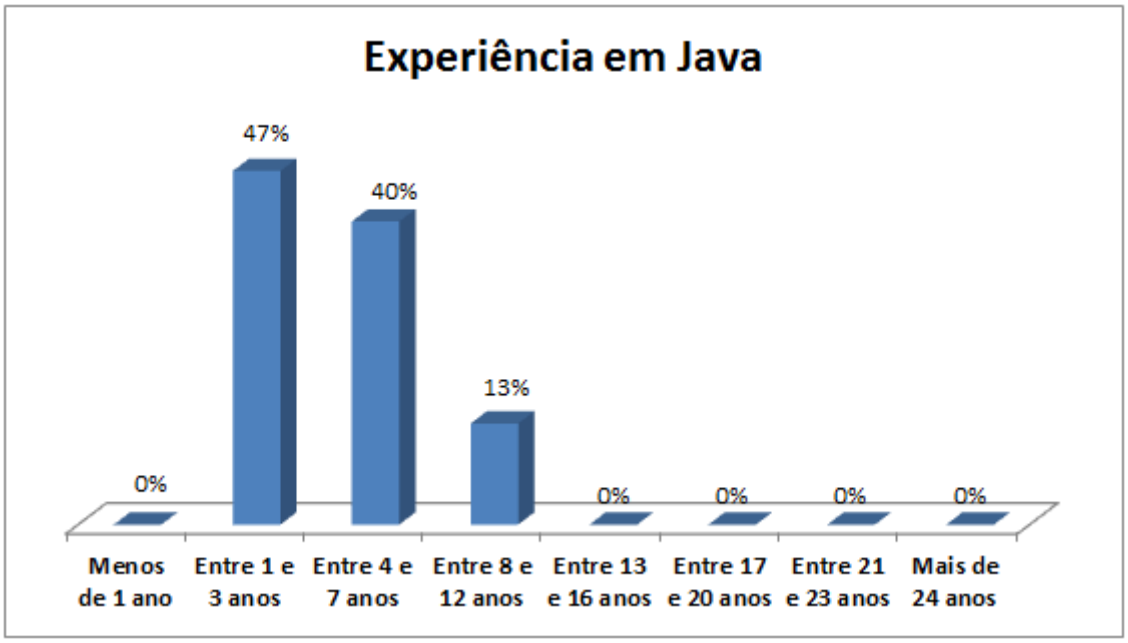

Figura 5.3: Experiência em Java para os participantes que dominam mais *JCF*

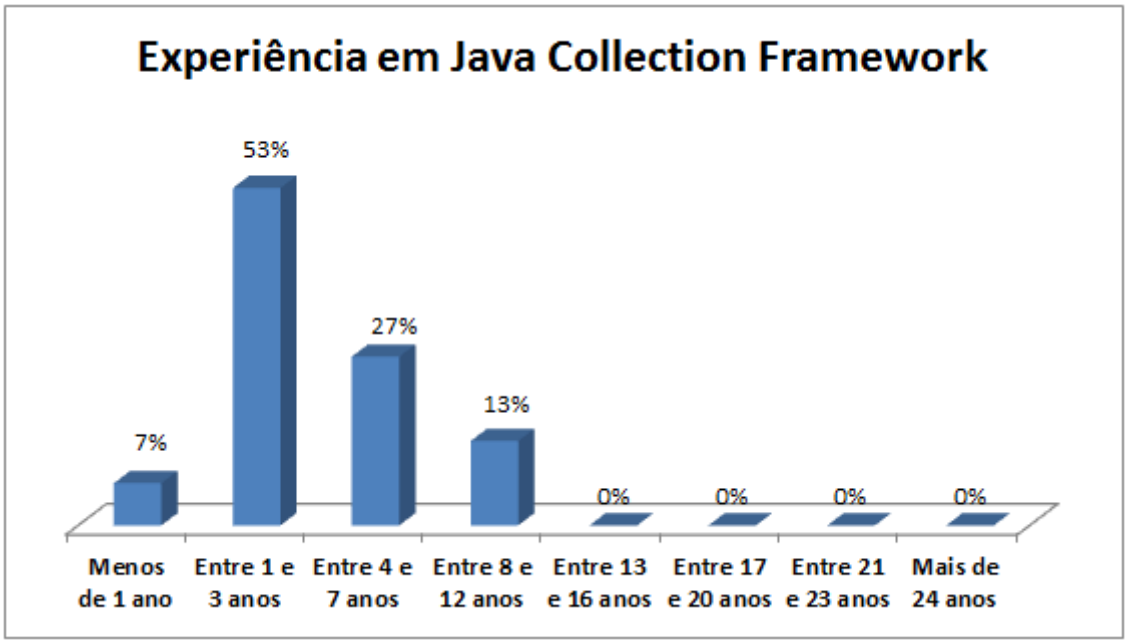

Figura 5.4: Experiência em *JCF* para os participantes que dominam mais *JCF*

5.4.2 RQ2) Qual a porcentagem das sugestões consideradas mais adequadas para o código 1 e o código 2?

Nesta questão foram levadas em consideração todas as respostas subjetivas e objetivas referentes ao código 1 , Seção 5.3.2.

As questões subjetivas, que indagavam quais os tipos e as instanciações dos atributos `ignoring` e `messages`, eram as seguintes:

1. Você substituiria o tipo do atributo "`protected final ArrayList messages`"? Se sim, por qual?
2. Você substituiria o tipo do atributo "`protected final ArrayList ignoring`"? Se sim, por qual?
3. Você substituiria o tipo da instanciação nessa linha "`messages = new ArrayList();`"? Se sim, por qual?
4. Você substituiria o tipo da instanciação nessa linha "`ignoring = new ArrayList();`"?Se sim, por qual?

Através delas foram obtidas as seguintes respostas, onde aquelas que estão em negrito são consideradas mais adequadas:

Tabela 5.1: *Quantidade de repostas para o atributo* `messages`

Valor da resposta	**Quantidade dos Tipo**	**Quantidade das Instanciações**
Não	**9**	**17**
Set	12	-
List	**16**	**-**
ArrayList	**5**	**7**
Collection	1	-
HashSet	-	10
LinkedList	**-**	**7**
Removeria	-	1
TreeSet	-	1
Total	43	43

Tabela 5.2: *Quantidade de repostas para o atributo* `ignoring`

Valor da resposta	**Quantidade dos Tipo**	**Quantidade das Instanciações**
Não	6	16
Set	**20**	**-**
List	11	-
ArrayList	4	6
HashSet	**2**	**17**
LinkedList	-	1
Removeria	-	1
TreeSet	**-**	**2**
Total	43	43

Já nas questões objetivas, que indagavam quais as modificações realizar nos métodos `ignore(IMessage.Kind kind)` e `handleMessage(IMessage message)`, obtivemos as seguintes respostas:

Tabela 5.3: Quantidade de repostas para o método `ignore`

Valor da resposta	**Quantidade de respostas**
Retirava o trecho "(null != kind) &&" da verificação do if	1
Retirava o trecho "&& (!ignoring.contains(kind))" da verificação do if	**19**
Retirava o if deixando no corpo do método apenas a linha, "ignoring.add(kind);"	6
Nada	17
Total	43

Valor da resposta	**Quantidade de respostas**
Retirava o 3º if deixando a instrução "messages.add(message);" ser executada, sem verificação	9
Nada	**34**
Total	43

Tabela 5.4: Quantidade de repostas para o método `handleMessage`

A fim de uma análise comparativa, classificamos as respostas dos participantes em *Adequadas* e *Diversas*. Consideramos *Adequadas* aquelas que estão de acordo com o que preconiza a API de *JCF* e similares às transformações propostas no Código Fonte 5.3 (respostas em negrito nas tabelas acima). Já as respostas incluídas na classificação *Diversas* foram as que não se enquadraram como *Adequadas*(respostas que não estão em negrito nas tabelas acima). Com essas as classificações os gráficos gerados foram:

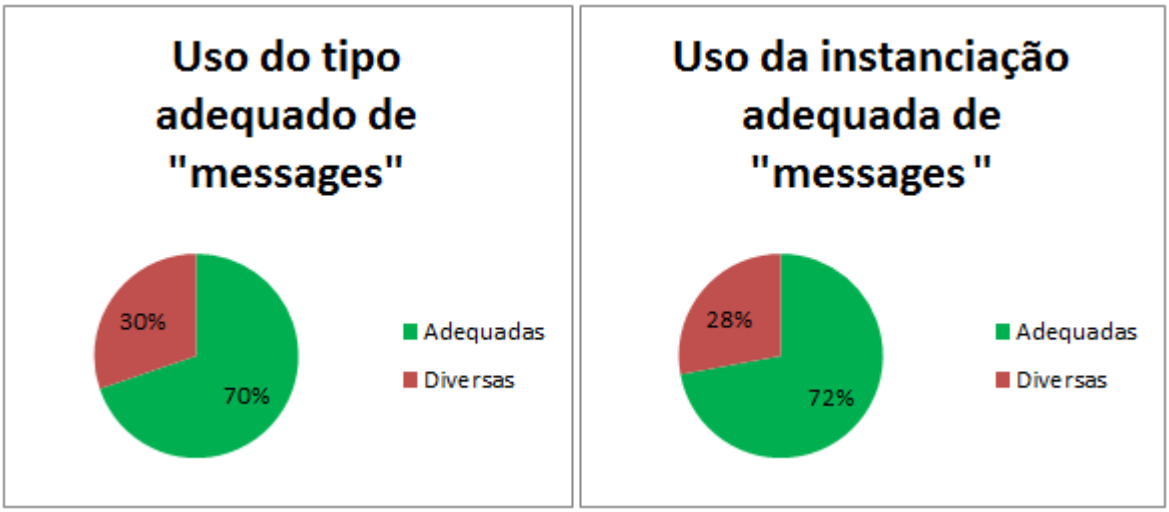

Figura 5.5: Uso adequado do atributo messages

Nestes dois gráficos são apresentados as porcentagens de uso adequado do tipo e da instanciação da coleção `messages`. Neles é percebido que o uso adequado do tipo é de 70% e da instanciação é de 72%.

Observa-se que cerca de 30% dos participantes realizaram transformações desnecessárias na coleção `messages` de `ArrayList` para `Set`, se enquadrando na classificação **Diversas**, como é percebido nesses exemplos retirados das perguntas *1.* e *3.* do início desta seção: *"Sim. Por um Set, já que a indexação e ordem não importam, e, por não permitir elementos repetidos, não seria preciso verificar se a coleção não continha o kind antes de guardá-lo."* e *"Sim, ao invés de usar um ArrayList eu mudaria para algum tipo de Set, que não permita a repetição de dados."*. Nestas respostas os participantes não perceberam que no `if` contido no método `handleMessage(IMessage message)` a verificação do objeto `message.getKind()` era feita em uma coleção (`ignoring`) e a adição do objeto `message` em outra (`messages`), permitindo assim elementos duplicados na coleção `messages`. Essa não percepção pode ter ocorrido por falta de capacidade do participante em identificar situações mais complexas sobre o uso das coleções *JCF*, uma vez que eles mostraram conhecer as características das coleções `List` e `Set`, porém mesmo assim não sugeriram transformações *Adequadas*.

Já no gráfico abaixo observa-se as transformações adequadas do método

`handleMessage(IMessage message)`, que manipula a coleção `messages` em seu corpo, que foi 79%.

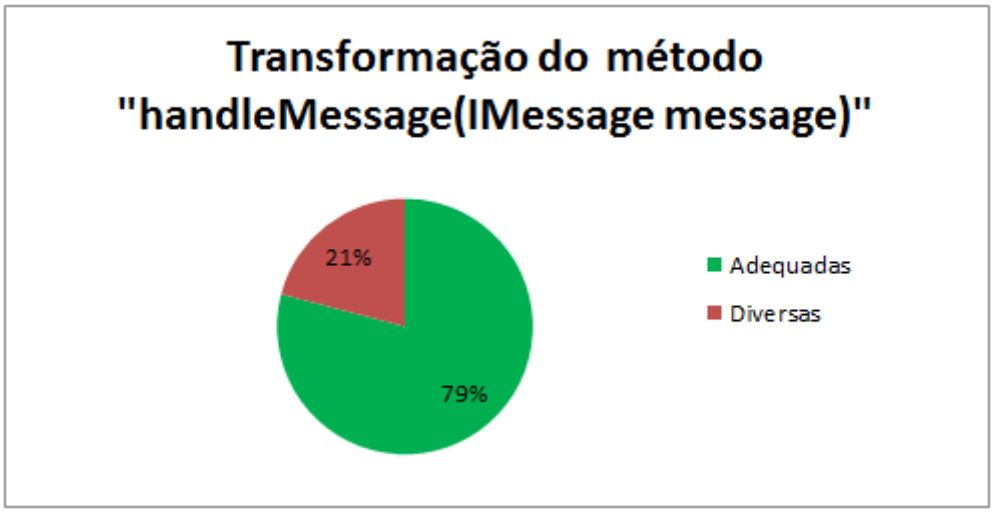

Figura 5.6: Transformação do método `handleMessage(IMessage message)`

Nos dois gráficos abaixo são apresentados as porcentagens de uso adequado do tipo e da instanciação da coleção `ignoring`. Neles percebe-se que o uso adequado do tipo é de 49% e o da instanciação é de 42%.

No restante dos participantes, que realizaram transformações **Diversas**, tivemos mais índices de mudanças de `ArrayList` para `List`, e não para `Set`, o que seria mais adequado. Como nesses exemplos retirados das perguntas *2.* e *4.* do início desta seção: *"protected final List ignoring"* e apenas *"List"*. Dando um indício que eles não tinham conhecimento necessário para identificar através do código fonte que uma lista estava se comportando como conjunto, e sentindo a necessidade de trocar o tipo `ArrayList` por `List`, que em alguns casos seria o mais adequado a se fazer, porém nessa situação deveria substituir por `Set`.

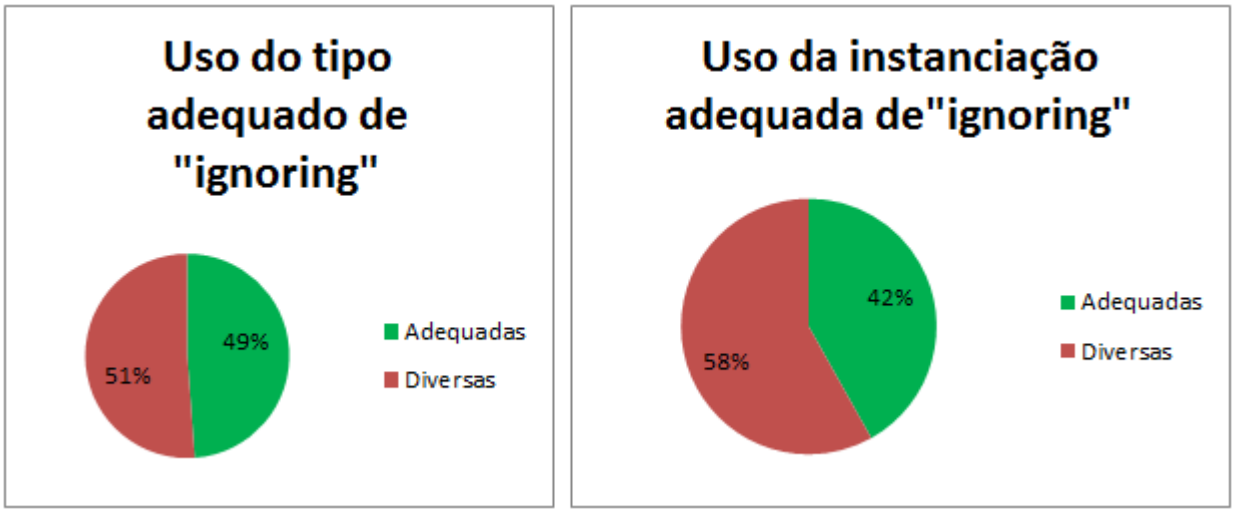

Figura 5.7: Uso adequado do atributo ignoring

Já no gráfico abaixo são observadas as transformações adequadas do método `ignore(IMessage.Kind kind)`, que manipula a coleção `ignoring` em seu corpo, que foi 44%. E nos outros 56% restantes que não propuseram transformações **Adequadas** teve como principal resposta "Nada", onde o participante julgou não ser necessário nenhuma mudança nesse método.

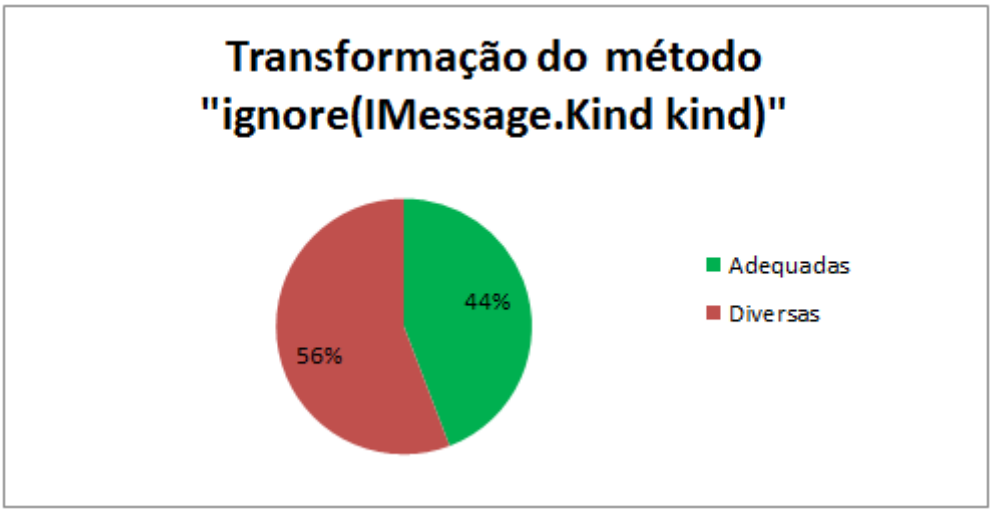

Figura 5.8: Transformação do método `ignoring(IMessage.Kind kind)`

Em resumo, nos casos envolvendo `messages` tivemos uma taxa de respostas **Diversas**

de 25%, que foi um número inferior nos casos envolvendo `ignoring`, em torno de 55%. Estes números são bem expressivos visto que o código 1 é um trecho de código pequeno, que já possuía opções pré-definidas de resposta, e mesmo assim teve um índice alto de mudanças **Diversas**, percebendo a dificuldade dos programadores participantes em realizar transformações de código nas substituições entre as coleções `List` e `Set`.

No código 2 (Código Fonte 5.5), similarmente ao código 1, foram levadas em consideração todas as respostas subjetivas e objetivas. Nas questões subjetivas, que indagavam qual o tipo e a instanciação do atributo `bosses`, que eram as seguintes:

- Você substituiria o tipo do atributo "`private List bosses;`"? Se sim, por qual?
- Você substituiria o tipo da instanciação nessa linha "`bosses = new LinkedList();`"? Se sim, por qual?

Foram obtidas as seguintes respostas, onde as linhas da tabela em negrito são aquelas consideradas mais adequadas:

Tabela 5.5: Quantidade de repostas para o atributo `bosses`

Valor da resposta	**Quantidade dos Tipo**	**Quantidade das Instanciações**
Não	24	19
Set	**10**	**-**
Map	2	-
List	5	-
Generics	1	1
HashSet	**1**	**10**
HashMap	-	1
ArrayList	-	5
LinkedList	-	5
LinkedHashSet	**-**	**2**
Total	43	43

Já nas questões objetivas, que indagavam quais as modificações realizar no método `changeBosses(Employee oldBoss, Employee newBoss)`, obtivemos as seguintes respostas:

Valor da resposta	**Quantidade de respostas**
Retiraria o "if (!bosses.contains(newBoss))	3
Substituiria a instrução "bosses.set(bosses.indexOf(oldBoss), newBoss);	5
Nada	26
Retiraria o "if (!bosses.contains(newBoss)) + Substituiria a instrução "bosses.set(bosses.indexOf(oldBoss), newBoss);	**9**
Total	43

Tabela 5.6: Quantidade de repostas para o método `changeBosses`

A fim de uma análise comparativa, classificamos as respostas dos participantes em *Adequadas* e *Diversas*. Consideramos *Adequadas* aquelas que estão de acordo com o que preconiza a API de *JCF* e similares às transformações propostas no Código Fonte 5.5. (respostas em negrito nas tabelas acima). Já as respostas incluídas na classificação *Diversas* foram as que não se enquadraram como *Adequadas* (respostas que não estão em negrito nas tabelas acima). A partir dessas classificações os gráficos gerados foram:

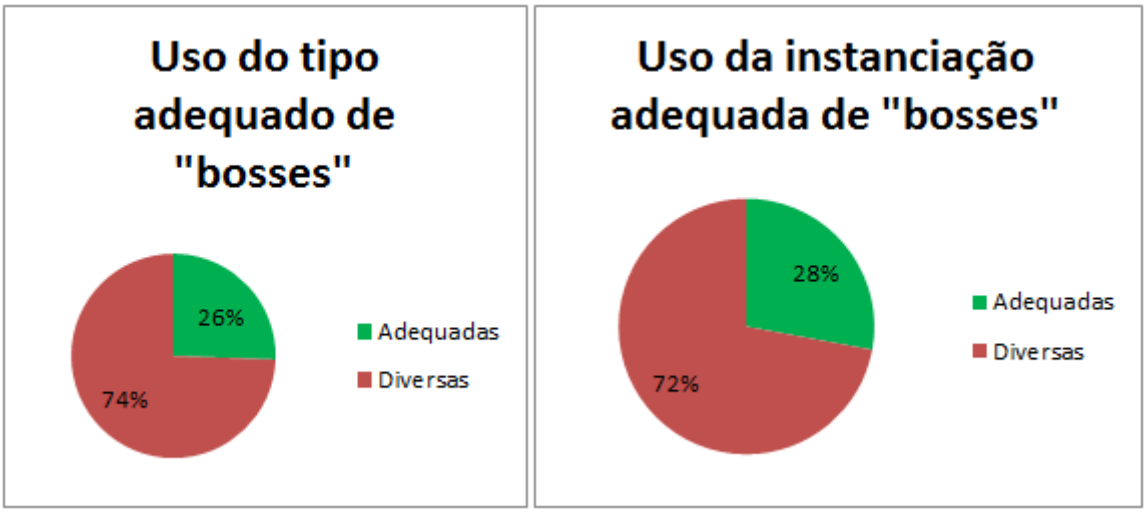

Figura 5.9: Uso adequado do atributo `bosses`

Nestes dois gráficos são apresentados as porcentagens de uso adequado do tipo e da instanciação da coleção `bosses`. Neles, o uso adequado do tipo é de apenas 26% e o uso adequado da instanciação é de 28%.

No restante dos participantes, que realizaram transformações **Diversas**, tivemos em sua maioria a não modificação do código modificando em alguns casos apenas a instanciação de `ArrayList` para `LinkedList`, o que é visto nas respostas: *"Apenas iria inserir o tipo de dado manipulado. Ficando: bosses = new LinkedList();"* e *"Sim. Aparentemente ArrayList é melhor para a situação."*.

Já no gráfico abaixo são observadas as transformações **Adequadas** do método `changeBosses(Employee oldBoss, Employee newBoss)`, que manipula a coleção `bosses` em seu corpo, que foi apenas 16%. As transformações que estão contidas nos demais 84% são principalmente a não mudança da coleção bosses para `Set` e a transformação do método `set(int index, E e)` para outros trechos de código diferente dos comandos `add(newBoss) + remove(oldBoss)`. Como por exemplo: *"bosses.add(newBoss);"* e *"if (!bosses.contains(newBoss) && bosses.remove(oldBoss)) bosses.add(newBoss); "*.

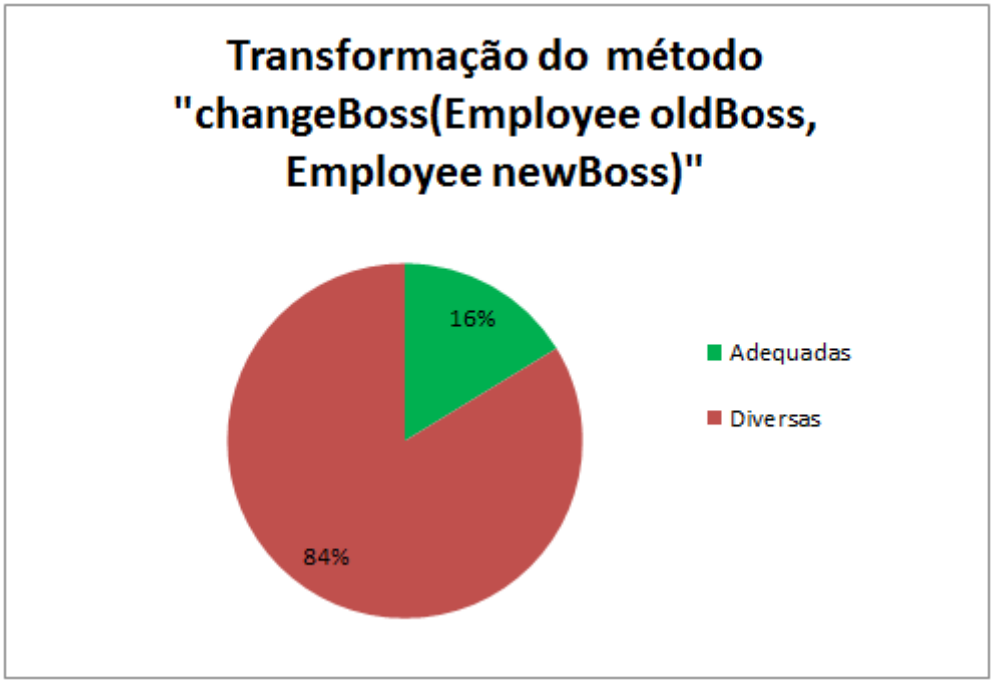

Figura 5.10: Transformação do método `changeBosses`

Em resumo os casos envolvendo `bosses` (nos três gráficos desta pergunta de pesquisa), que de acordo com as transformações *Adequadas* precisava ser substituída por `Set`, resultou em torno de apenas 25% dos participantes realizando as transformações **Adequadas**.

Esse fato preocupa ainda mais, pois as porcentagens são ainda menores que as da pergunta de pesquisa anterior. Dando um indício de que a transformação do método `set(int index, E e)` de `List` para os métodos `add(E e)` e `remove(E e)` de `Set`, retirando do código o IfContains é ainda mais incomum, aumentando assim sua dificuldade, que a de `add(E e)` de `List` para `add(E e)` de `Set`. Reforçando a fragilidade dos participantes em identificar e ter capacidade de escolher a coleção mais adequada.

Participantes que dominavam mais o *JCF*

Analisando as *Respostas B* nesta pergunta de pesquisa pode-se perceber que no Código 1 o grau de respostas Adequadas para o uso da lista `messages` (80%) foi bem próximo ao que foi verificado nas *Respostas A* (70%). Do mesmo modo com o método `handleMessage` que é 87% para as *Respostas B* contra 80% das *Respostas A*. Esses dados são mostrados na Figura 5.11 e Figura 5.12.

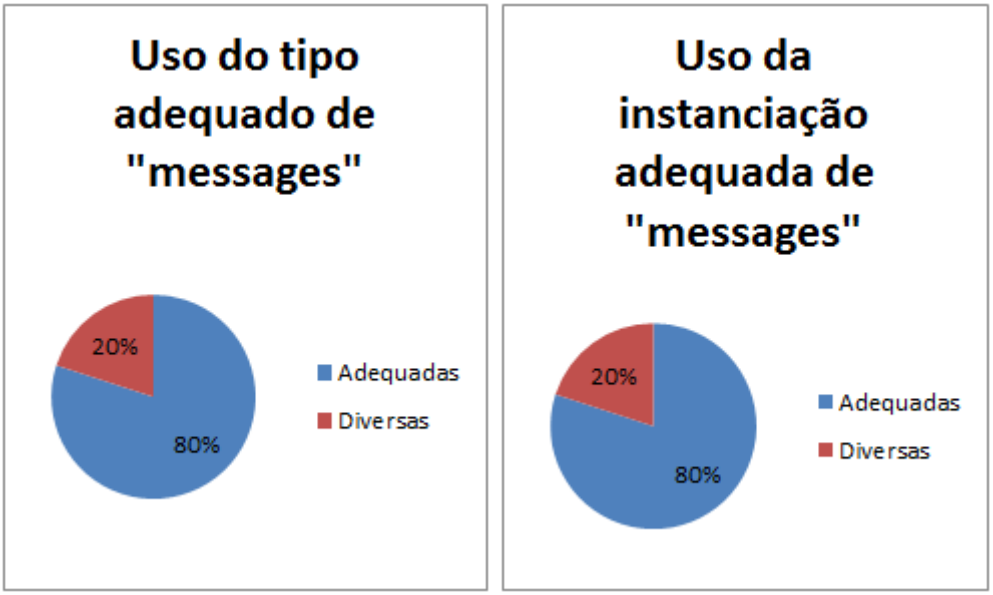

Figura 5.11: Uso adequado do atributo messages para os participantes que dominam mais *JCF*

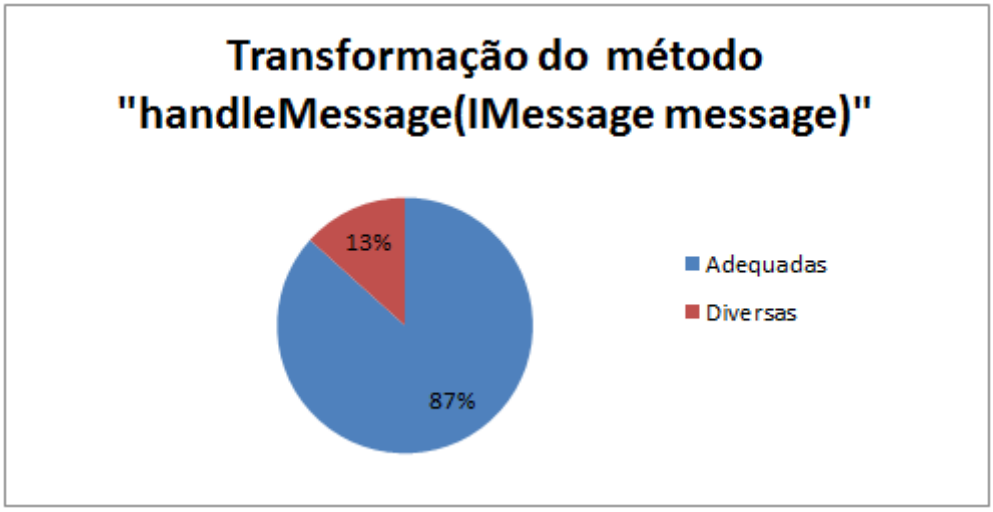

Figura 5.12: Transformação do método `handleMessage(IMessage message)` para os participantes que dominam mais *JCF*

Já se for analisado, ainda no Código 1, as transformações propostas consideradas Adequadas para a lista `ignoring` vê-se que houve uma diferença nas porcentagens, onde nas *Respostas A* tem o valor em torno de 45% e nas *Respostas B* um valor de 60%, assim como o método `ignore(IMessage.Kind kind)` que manipula essa lista. Esses dados podem

ser vistos nos gráficos da Figura 5.13 e da Figura 5.14.

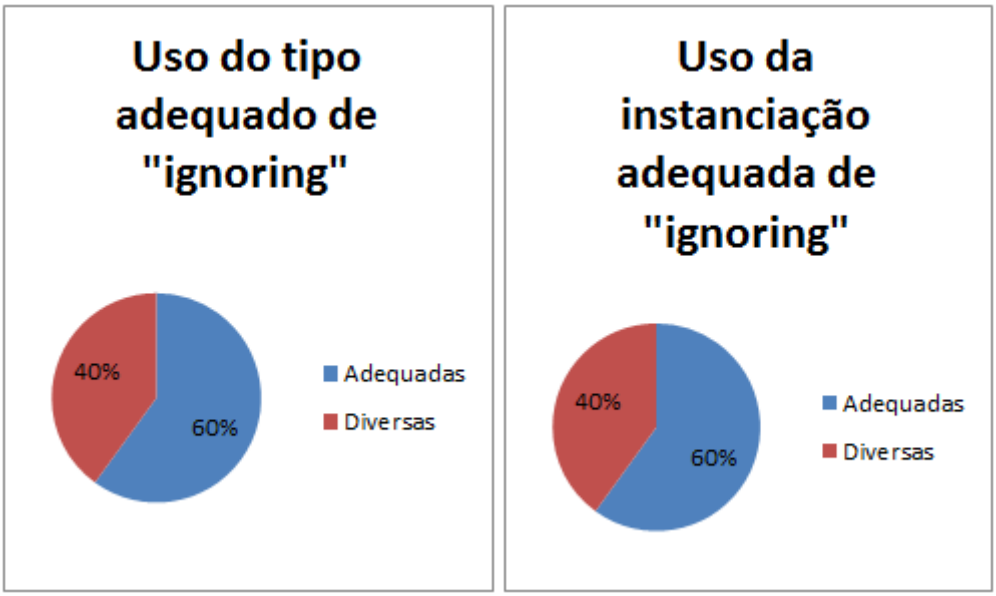

Figura 5.13: Uso adequado do atributo `ignoring` para os participantes que dominam mais *JCF*

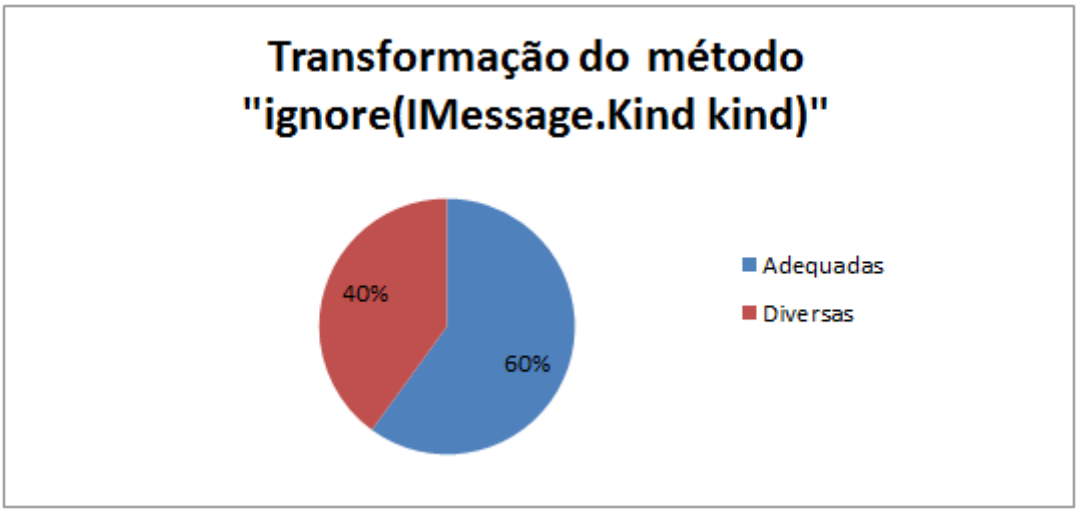

Figura 5.14: Transformação do método `ignoring(IMessage.Kind kind)` para os participantes que dominam mais *JCF*

A porcentagem de respostas Adequadas nas *Respostas B* também aumentaram em relação às sugestões de transformação para o Código 2, na a lista `bosses` e o mé-

todo `changeBoss(Employee oldBoss, Employee newBoss)` que eram cerca de 25% e passaram a ser 40%. Vê-se na Figura5.15 e FiguraFigura5.16.

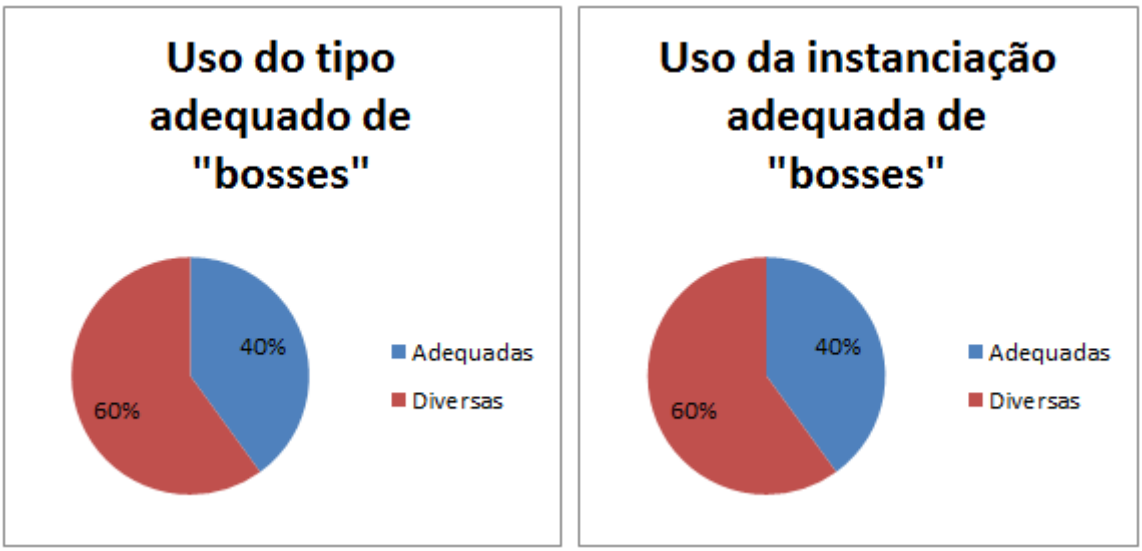

Figura 5.15: Uso adequado do atributo `bosses` para os participantes que dominam mais *JCF*

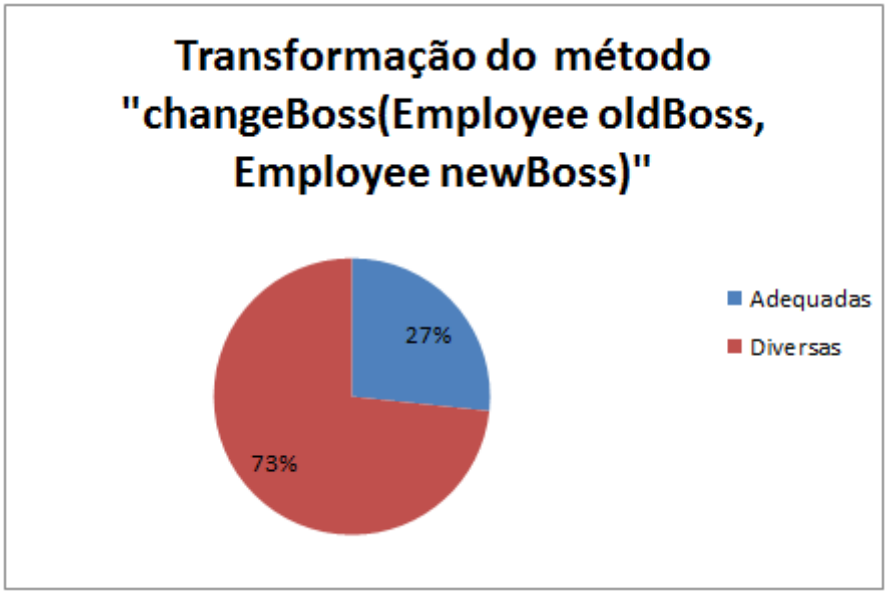

Figura 5.16: Transformação do método `changeBosses` para os participantes que dominam mais *JCF*

Esses aumentos nas porcentagens de respostas Adequadas nas transformações para as

listas `ignoring e bosses` dão um indício que uma maior parte dos participantes souberam identificar e transformar uma situação de uma lista sendo usada como conjunto, possivelmente pelo fato dos participantes escolhidos nas *Respostas B* dominarem mais os conhecimentos em *JCF*. Entretanto estes valores ainda podem ser considerados baixos, pois se for levado em consideração que o questionário aplicado possuía opções pré-definidas e os códigos para análise são bem pequenos a quantidade de respostas Adequadas deveria ser maior.

5.4.3 RQ3) Qual o grau de legibilidade do código original e transformado?

O grau de legibilidade do código foi medido através de uma escala de 1 a 5, onde o participante selecionava a quantidade de estrelas correspondentes. Quanto maior o valor respondido mais legível aquele código era.

Então foram feitas contagens das respostas dos códigos originais em comparação com os transformados, tanto para o código 1 quanto para o código 2. Esses dados se encontram nos gráficos abaixo:

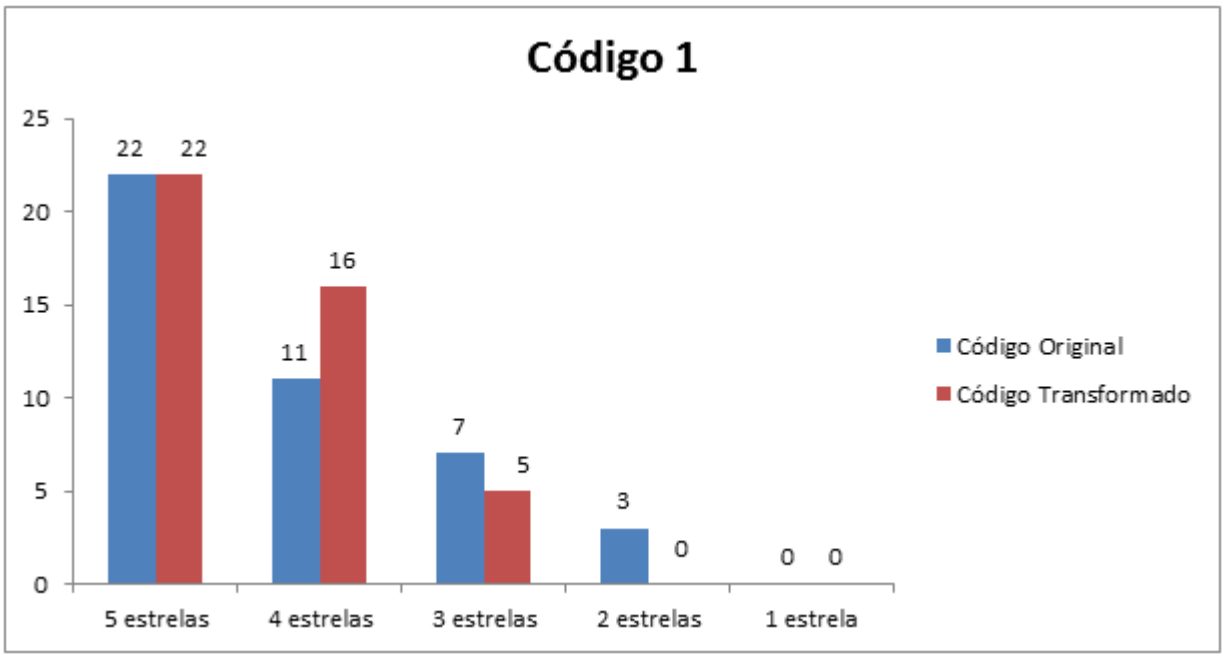

Figura 5.17: Resultados do grau de legibilidade do Código 1

Neste primeiro gráfico, Figura 5.17, a quantidade de avaliações **5 estrelas** tanto para o Código Original quanto para o Transformado foram iguais (22). Porém nas **4 estrelas** percebe-se uma diferença considerável de avaliações do código original (11) para o transformado (16), que pode evidenciar uma melhoria no código transformado. Este fato ainda é reforçado pelas valores das **3 estrelas**, que possuem menos avaliações para o código transformado (5) em detrimento do original (7). E mais ainda na categoria **2 estrelas**, que possui 3 avaliações para o código original e nenhuma para o transformado.

O que demonstra que para os valores maiores o código transformado obteve maior frequência do que o original. Dando indícios de uma melhora no grau de facilidade de entendimento do código 1 transformado.

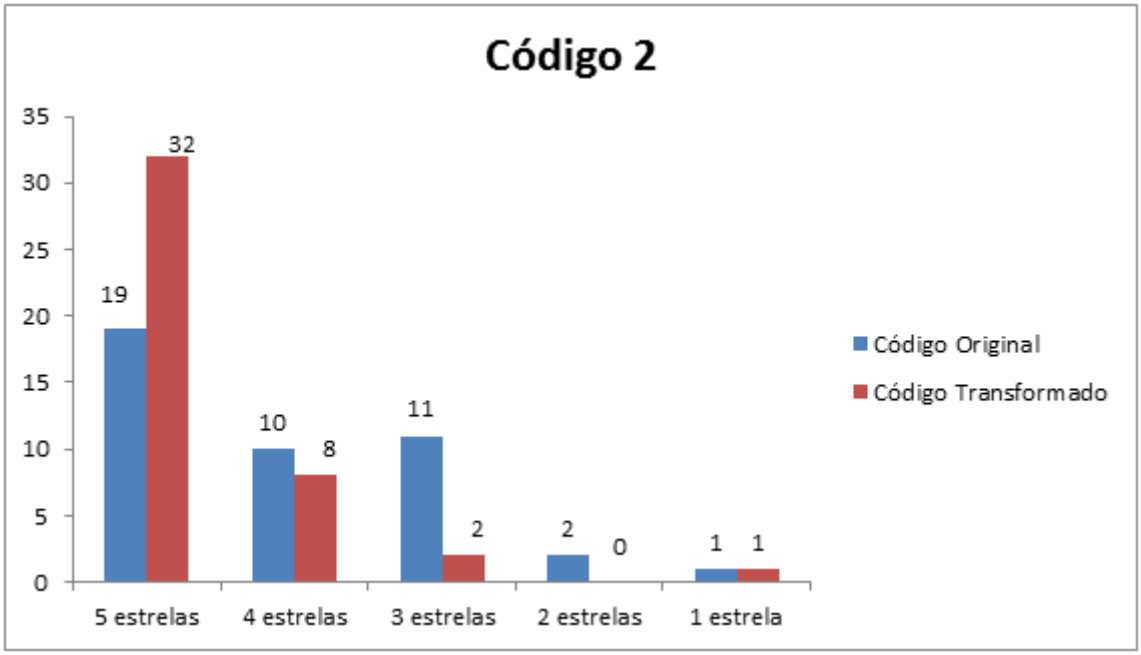

Figura 5.18: Resultados do grau de legibilidade do Código 2

Analisando o código 2, tem-se uma diferença ainda maior, onde na categoria **5 estrelas** possui 19 ocorrências no código original, contra 32 no transformado. Nas demais categorias tem-se sempre uma quantidade maior de ocorrências no código original do que no transformado, com exceção da categoria **1 estrela** que teve uma ocorrência de cada código. Demonstrando que para o maior valor possível o código transformado obteve frequência expressiva em relação ao original. Dando indícios de uma melhora no grau de facilidade de

entendimento do código 2 transformado.

Participantes que dominavam mais o *JCF*

Analisando as *Respostas B* nesta pergunta de pesquisa é visto que o grau de legibilidade tanto no Código 1 quanto no Código 2 o Código Transformado continua tendo maiores frequências nos valores 5 e 4 estrelas em relação ao Código Original, como é visto na Figura 5.19 e na Figura 5.20.

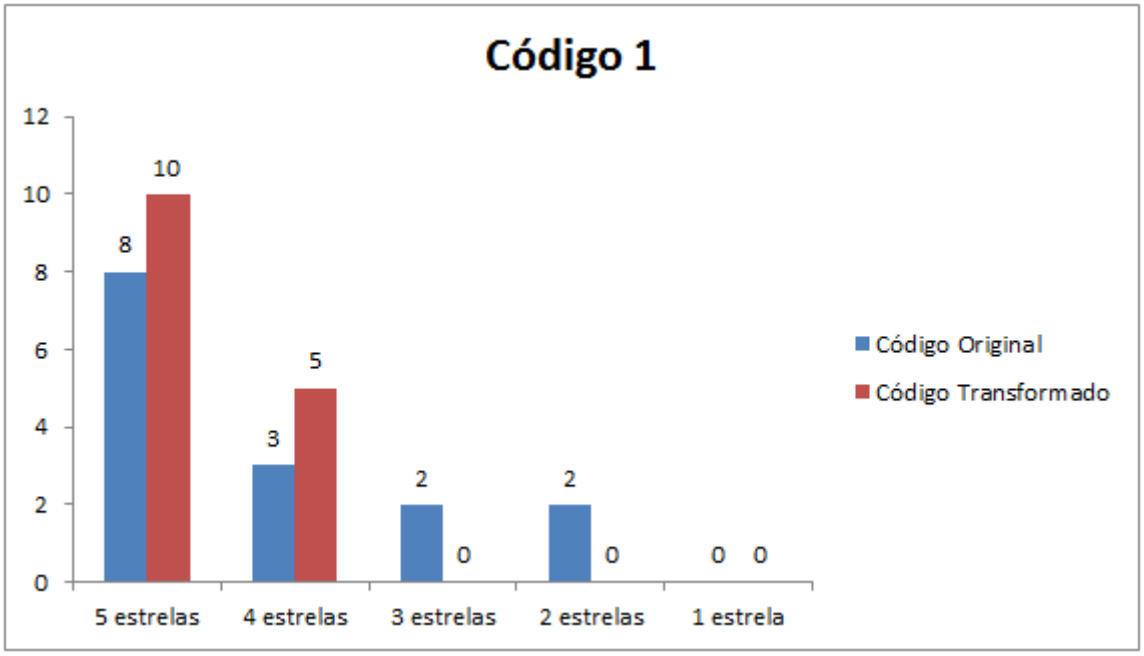

Figura 5.19: Resultados do grau de legibilidade do Código 1 para os participantes que dominam mais *JCF*

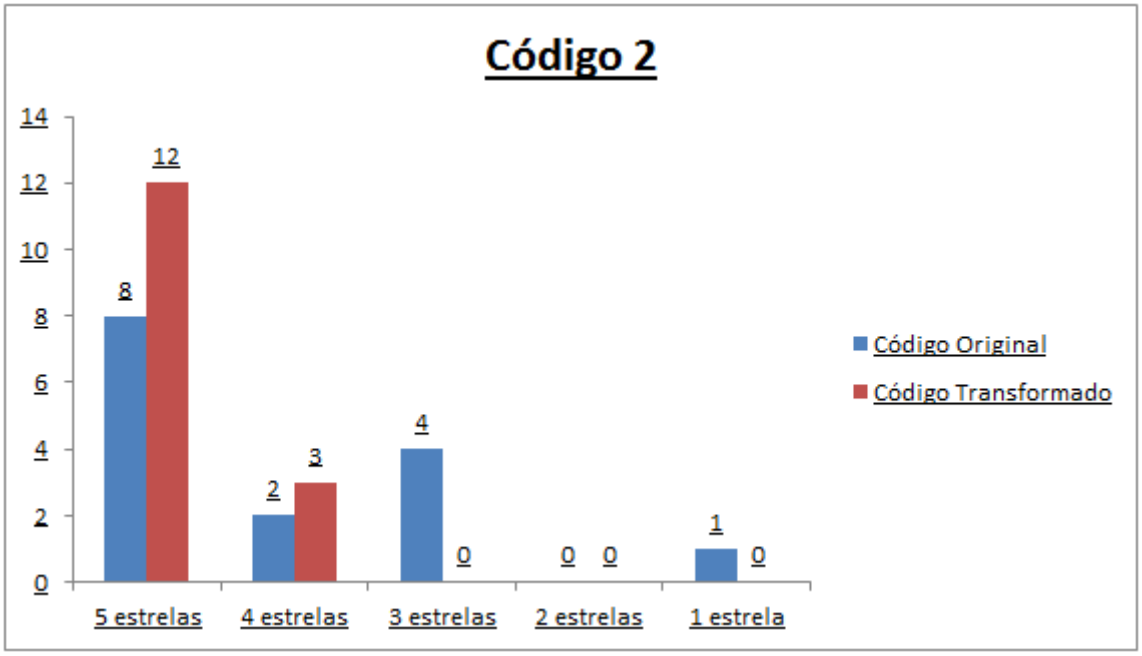

Figura 5.20: Resultados do grau de legibilidade do Código 2 para os participantes que dominam mais *JCF*

Isso dá um indício que o código Transformado é mais legível do que o Original também nas *Respostas B*.

5.4.4 RQ4) O que os participantes perceberam dos resultados das transformações?

Na resposta dessa pergunta de pesquisa continuou-se analisando as respostas dos graus de facilidade de entendimento dos códigos original e transformado, porém nesse caso classificamos cada par de respostas nas seguintes opções:

- **Não mudou**: quando o valor do código original for igual ao do transformado.
- **Piorou**: quando o valor do código original for maior que o do transformado.
- **Melhorou**: quando o valor do código original for menor que o do transformado.

Sendo feito tanto para o código 1 quanto para o código 2, resultando nos seguintes dados:

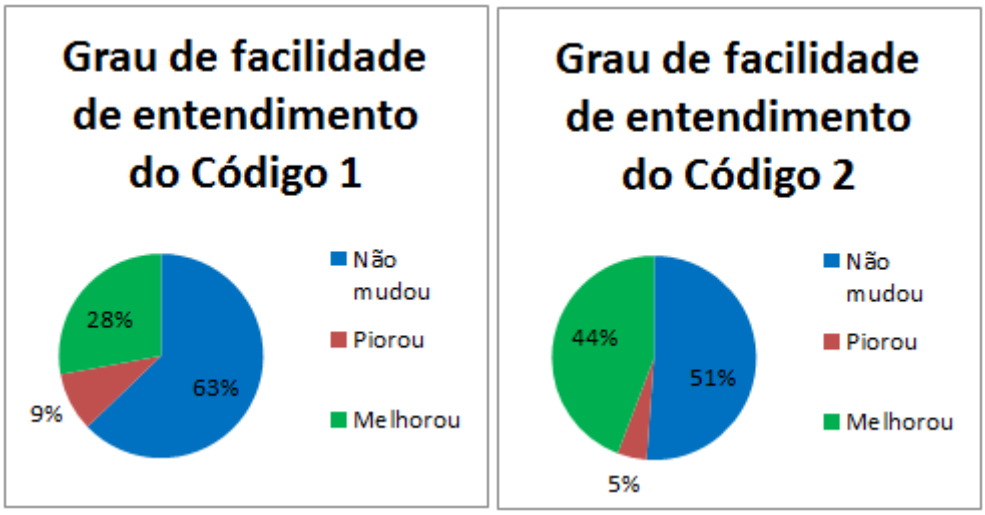

Figura 5.21: Percepção de melhora dos códigos

Esses gráficos reforçam as conclusões feitas na RQ3, visto que tanto no gráfico do código 1 quanto no do código 2 ocorreram pequenas porcentagens de participantes que consideraram que o código depois de transformado **Piorou** (9% e 5% respectivamente).

Analisando a porcentagem da categoria **Melhorou** percebe-se que no código 2 ela foi maior que a do código 1, que é evidenciada pela igualdade ocorrida na categoria **5 estrelas** no gráfico da Figura 5.17, e a diferença maior para o código transformado nessa mesma categoria no gráfico da Figura 5.18. Isto pode ter ocorrido pois o código 2 transformado retirou a verificação do comando `if` dentro do método `changeBosses(Employee oldBoss, Employee newBoss)`, diferentemente do que ocorreu no código 1 transformado que o `if` dentro do método `ignore(IMessage.Kind kind)` permaneceu, retirando a verificação `&& (!ignoring.contains(kind))`. Já a categoria **Não mudou** sobressaiu sobre as demais, obtendo um valor mais expressivo no código 1 (63%) do que no código 2 (51%).

Logo, analisando em conjunto as RQ3 e RQ4, tem-se um indício que as transformações propostas não pioraram o código original e possuindo uma parcela de melhora em termos de legibilidade.

Participantes que dominavam mais o *JCF*

Ao analisar as *Respostas B* para esta pergunta de pesquisa é visto que a percepção de melhora na legibilidade dos códigos Transformados foi ainda maior que a das *Respostas A* inexistindo percepções de piora no código, como é vista na Figura 5.22.

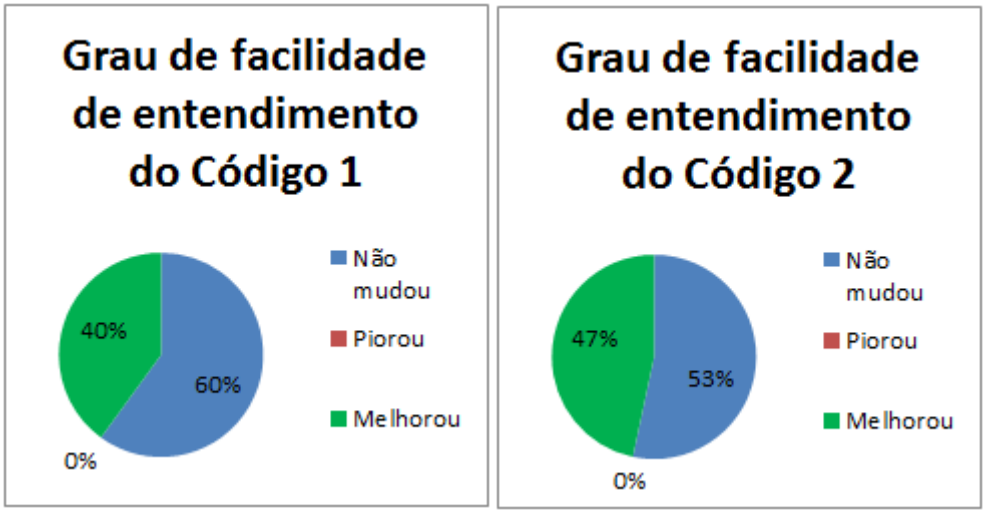

Figura 5.22: Percepção de melhora dos códigos para os participantes que dominam mais *JCF*

Esse fato aumenta ainda mais o indício de que as transformações propostas não pioraram o código original possuindo uma boa percepção de melhora em termos de legibilidade.

5.4.5 RQ5) Após o participante ver a transformação sugerida pelo autor, qual sua escolha de transformação a ser feita para o código 1?

Os dados que respondem esta pergunta estão apresentados na Figura 5.23.

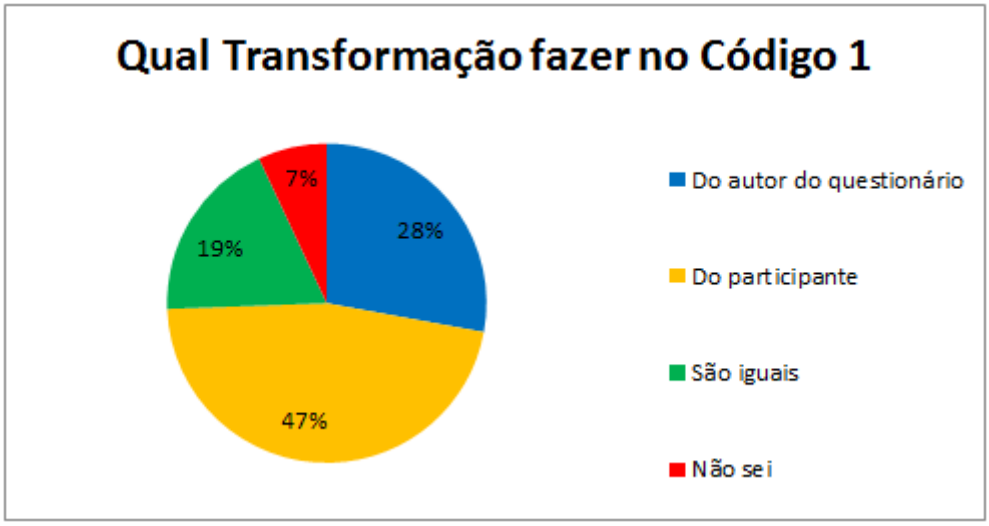

Figura 5.23: Qual transformação fazer no código 1

Entretanto, como o os participantes que definiram suas transformações sendo iguais às do autor do questionário também "escolheram"a transformação proposta, é possível reorganizar os dados da seguinte forma:

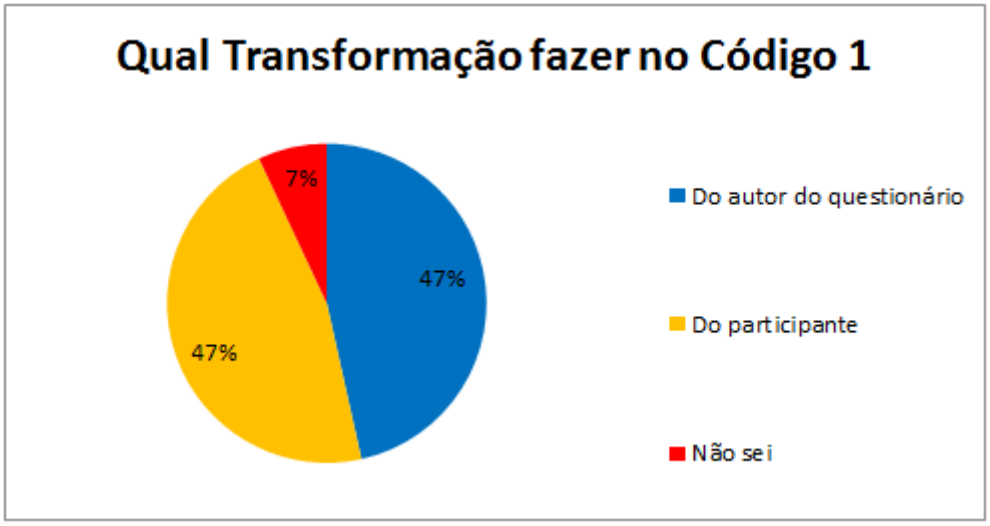

Figura 5.24: Qual transformação fazer no código 1

O que mostra exatamente a mesma porcentagem (47%) de escolha da transformação do autor e do participante, e 7% não souberam opinar qual transformação preferiam. Esse dado demonstra que mesmo após a apresentação de uma transformação mais adequada ao contexto do código 1, mais da metade dos participantes não a escolheram como mais adequada.

Ao analisar o campo de justificativas dessa escolha vemos alguns dos motivos que os participantes consideraram para não escolher a transformação proposta, como neste caso que ele não percebeu que a coleção `messages` poderia receber elementos duplicados, não devendo assim ser substituída por um `Set`: *"Usar ambas variáveis como Set evitaria o if no método handleMessage verificando se ignoring contém message.getKind(); desse modo, tornaríamos o corpo do método handleMessage mais simples e fácil de manter."*. Outras justificativas que evidenciam esse fato são: *"Se as mensagens não necessitam estar em ordem e sua indexação é irrelevante, também transformaria em Set.", "Minha sugestão é ter dois HashSet pois evitaria a verificação se o objeto já está contido na collection." e "O autor falou que ordem e indexação de "messages" também não é importante, então não tem por que deixar esses objetos numa lista ao invés de um conjunto."*

Já nesses comentários os participantes não conseguem, mesmo após a apresentação da transformação, perceber que `Set` é mais adequado para ser utilizado na coleção `ignoring`: *"Não vejo necessidade de trocar o tipo de 'ignoring' para um Set.", "Não há necessidade de usar a interface Set."* e *"Não há necessidade de mudar ArrayList."*. Este fato é ainda pior que o primeiro citado, pois evidencia uma falta de percepção, por parte dos participantes, de identificar a lista `ignoring` esta bloqueando a inserção de elementos repetidos, devendo assim ser substituída para um conjunto e eliminando a necessidade da verificação do IfContains, Seção A.2.

Já nesse próximo comentário o participante marcou preferir as mudanças dele pois a transformação proposta não utilizou `List` na coleção `messages`: *"Na verdade, mais adequado que utilizar ArrayList ao meu ver seria ideal List. Já a mudança para Set sugerida pelo autor encontra-se correta ao meu ver..."*. Percebendo assim que não só as trocas entre as interfaces `List e Set` são importantes como também entre `List` e as demais classes de que a implementam, como `ArrayList` e `LinkedList`.

Esse fato também foi evidenciado neste comentário: *"O uso de Set nesse caso realmente é sensato, mas também é uma boa prática nomear um atributo com o tipo mais primitivo quando possível, que nesse caso é a interface "List". Isso contribui para o desacoplamento do código e explica a quem está lendo exatamente quais são as regras mínimas que você quer ter governando aquele atributo."*. Então, caso as transformações levassem em consideração essas trocas entre as coleções derivadas de `List` teríamos um número maior de participantes

que escolheriam a transformação proposta pelo autor do questionário como mais adequada.

Alguns comentários de participantes que perceberam que a transformação proposta é mais adequada ao contexto do código 1 foram: *"Resolve o problema adequadamente e com pouca modificação no código.", "A estrutura de dados Set já faz a verificação para não adicionar elementos repetidos, deixando o código mais legível." e "parece plausível"*

Então, é preciso encontrar formas diferentes para instruir os programadores que utilizam *Java Collection Framework (JCF)* para que estes entendam a fundo cada situação em que cada coleção é mais adequada. Podendo ser através de exemplos comentados e de uma melhor explicação na API, por exemplo, ou desenvolvendo ferramentas que auxiliem o desenvolvedor. Esta última sugestão parece ser mais viável, pois caso seja bem feita ajudará qualquer tipo de programador, até os que desconhecem os detalhes sobre as coleções *JCF*.

Participantes que dominavam mais o *JCF*

Ao analisar as *Respostas B* para esta pergunta de pesquisa, na Figura 5.25, é visto que a porcentagem de participantes que escolheu suas sugestões de transformação, mesmo após a apresentação de uma transformação considerada mais adequada para aquele contexto, continuou exatamente a mesma em comparação com as *Respostas A* (47%) o que mudou foi o percentual de participantes que não sabiam qual decisão tomar, escolhendo a transformação proposta pelo autor do questionário.

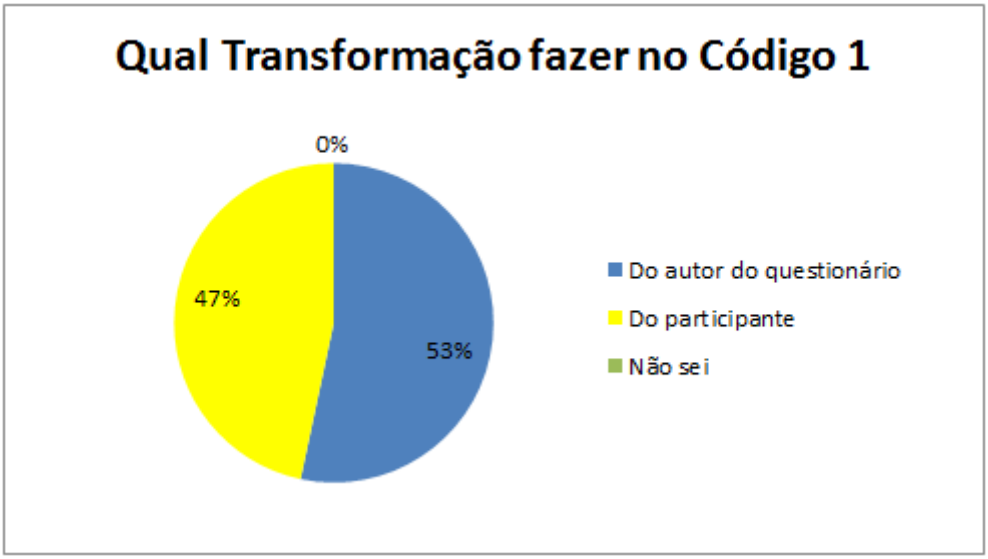

Figura 5.25: Qual transformação fazer no código 1 para os participantes que dominam mais *JCF*

Então pode-se continuar com os mesmos indícios que foram obtidos nas *Respostas A*, em que mesmo após a apresentação de uma transformação mais adequada ao contexto do código 1, cerca de metade dos participantes não a escolheram como mais adequada, sendo necessário encontrar formas diferentes para instruir os programadores que utilizam as coleções *JCF* de forma mais apropriada.

5.4.6 RQ6) Após o participante ver a transformação sugerida pelo autor, qual sua escolha de transformação a ser feita para o código 2?

Os dados que respondem esta pergunta estão apresentados na Figura 5.26.

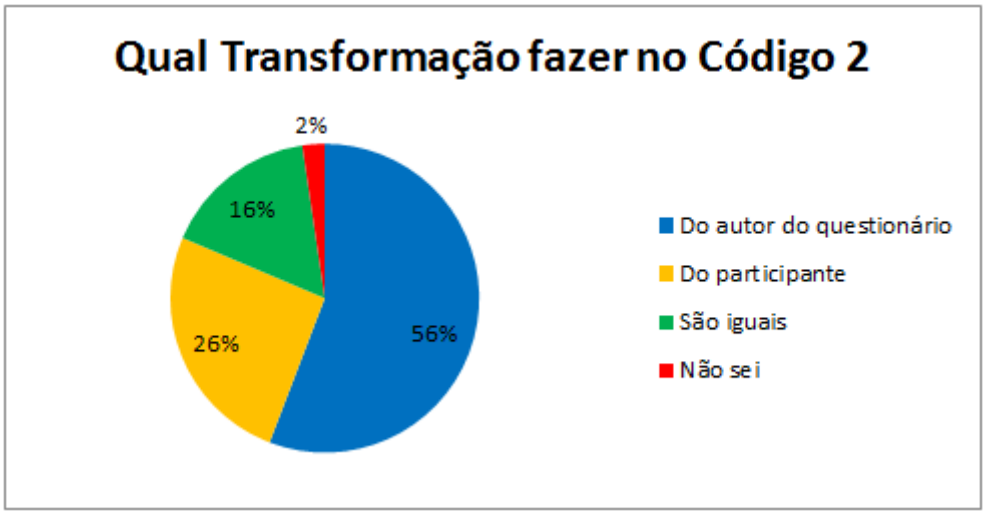

Figura 5.26: Qual transformação fazer no código 2

Entretanto, os participantes que definiram que suas transformações eram iguais às do autor do questionário também "escolheram" a transformação proposta, é possível reorganizar os dados da seguinte forma:

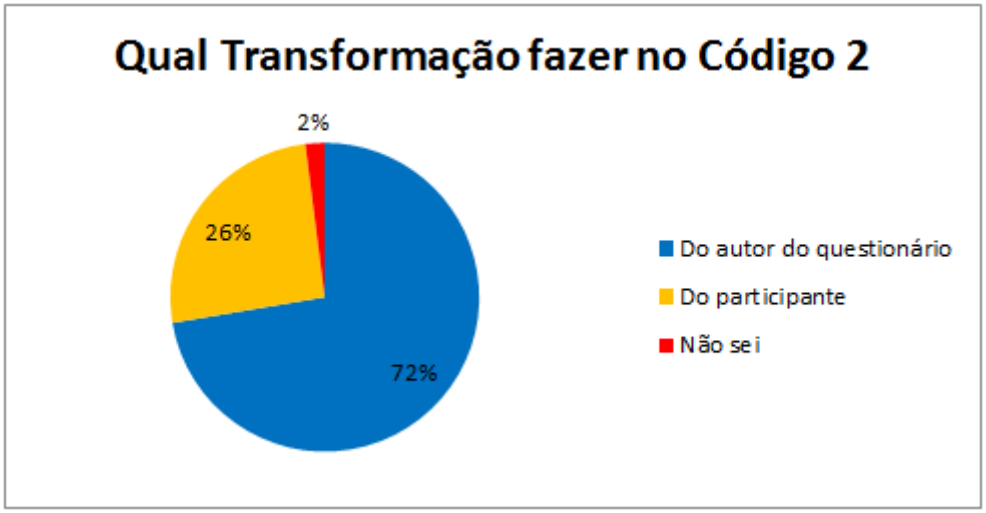

Figura 5.27: Qual transformação fazer no código 2

Diferentemente das respostas do código 1, a transformação do código 2 foi mais bem aceita, onde 72% dos participantes a preferiram. Outro fato é que também diminuiu a quantidade de participantes que não sabiam qual transformação escolher em relação ao código 1. Esses fatores podem ter ocorrido pela taxa muito baixa de sugestões *Adequadas* neste código,

fazendo com que após a apresentação da transformação sugerida pelo autor do questionário os participantes tenham percebido que existia uma transformação mais adequada que a deles.

Essa aceitabilidade é evidenciada pelos seguintes comentários: *"parece plausível", "Não pensei em usar um Set para esse caso. Funciona bem." e "Embora eu tenha pensando em manter o código original, a mudança proposta pelo autor faz sentido...".*

Já alguns os comentários negativos que foram obtidos são: *"Apesar de ser mais fácil de entender o código modificado mostrado acima, fazer a operação da forma como está será muito mais custoso do que se fosse com um ArrayList.", "Na transformação do autor, a posição do objeto não será preservada. Se o item a ser modificado não estiver no final da estrutura sua remoção acarretará na realocação de alguns itens.".* Realmente, pode-se perder desempenho ao substituir o comando `set(int index, E e)` de `List` pelos comando `add(E e) + remove(E e)` de `Set`, já que este primeiro remove o elemento antigo e adiciona o novo através do índice do mesmo, sendo *O(1)* e o remove(E e) de Set remove com *O(n)*. Entretanto levando em consideração que no código são usados os métodos `contains(Object o)` e `indexOf(Object o)` que ambos são *O(n)*, o que demonstra uma melhora no desempenho utilizando o código transformado. Estes apresentam preocupações a respeito do uso de `Set` ser mais custoso do que uma lista, que vão de encontro a este comentário: *"O Set resolve elementos não repetidos e melhora a alocação dinâmica de memória. As operações remove e add são mais rápidas que as operações para List.".*

O comentário seguinte evidência a falta de atenção do participante, que não percebeu que na parte do questionário da transformação do código 2 tinha uma informação que nenhuma outra parte do programa é importante a ordem e indexação dos elementos da coleção `bosses`. *"Se a função é para mudança de chefe a hash não se encaixaria, pois a mudança deveria conservar o índice ou algo que relacione o antigo ao novo. No caso da hash ele apenas remove e adiciona, assim o chefe está na lista, mas não se sabe qual o chefe ele substituiu. Se ao invés de "bosses.set(bosses.indexOf(oldBoss), newBoss);" fosse um "bosses.remove (oldBoss); bosses.add (newBoss);" teria sentido a substituição pela a hash"*

Já este participante demonstrou em seu comentário que ocorreu uma falha na noção de equivalência semântica do código original com o transformado, pois caso fossemos substituir um `oldBoss` por um `newBoss` que já existe na coleção, no código original ele não iria substituir, pois o IfContains A.2 iria resultar falso, já no código transformado o elemento

`oldBoss` iria ser removido e o `newBoss` ao tentar ser adicionado não conseguiria, resultando em elementos diferentes nas mesmas coleções para as mesmas ações: *"o autor mudou a semântica do método changeBoss. O método só deveria remover oldBoss se o newBoss não estiver na lista. Com a alteração proposta acima, essa semântica muda."*.

Realmente, esse participante do questionário conseguiu encontrar uma solução ainda mais adequada da transformação proposta, que se encontra no Código Fonte 5.6:

```
private set bosses;
...

bosses = new HashSet();
...
public void changeBoss(Employee oldBoss, Employee newBoss){
   if (bosses.add(newBoss)){
      bosses.remove(oldBoss);
   }
}
...
```

Código Fonte 5.6: Transformação modificada para o código 2 do questionário

Como o comando `add(E e)` de `Set` retorna um booleano informando se o elemento foi adicionado com sucesso na coleção ou não, é utilizado esse dado como condição para execução do comando `remove(E e)`, evitando assim que o objeto antigo (no caso `oldBoss`) seja removido sem que o objeto novo (`newBoss`) tenha sido inserido normalmente.

Participantes que dominavam mais o *JCF*

Visualizando os dados das *Respostas B* para esta pergunta, contidos na Figura 5.28, percebe-se que o gráfico permaneceu quase o mesmo que o das *Respostas A* na Figura 5.27, com a única diferença de que os 2% dos participantes que não sabiam qual transformação escolher não existem mais.

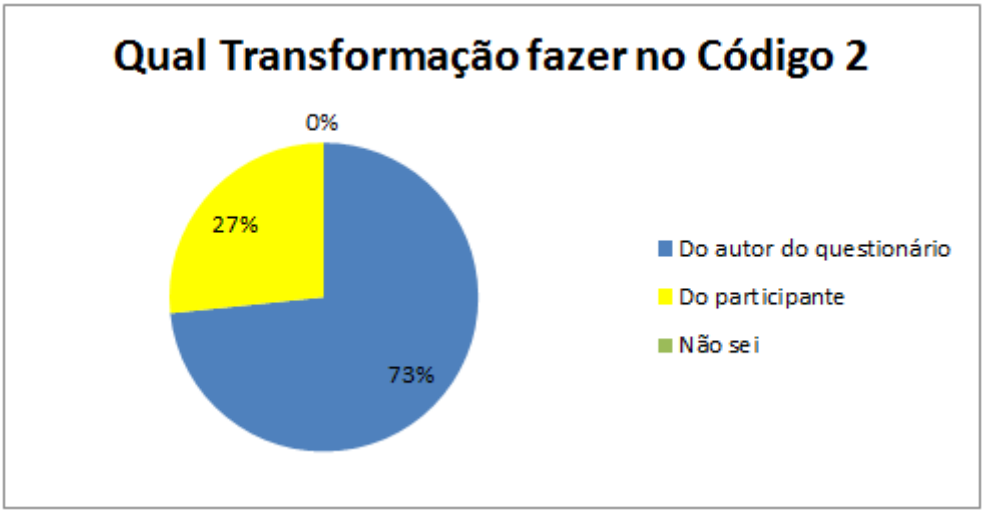

Figura 5.28: Qual transformação fazer no código 2 para os participantes que dominam mais *JCF*

Com isso a análise realizada nas *Respostas A* continua a mesma para as *Respostas B* também.

5.4.7 RQ7) Os participantes consultaram a API de *JCF* para responder o questionário?

Para esta pergunta apresentamos os dados da Figura 5.29.

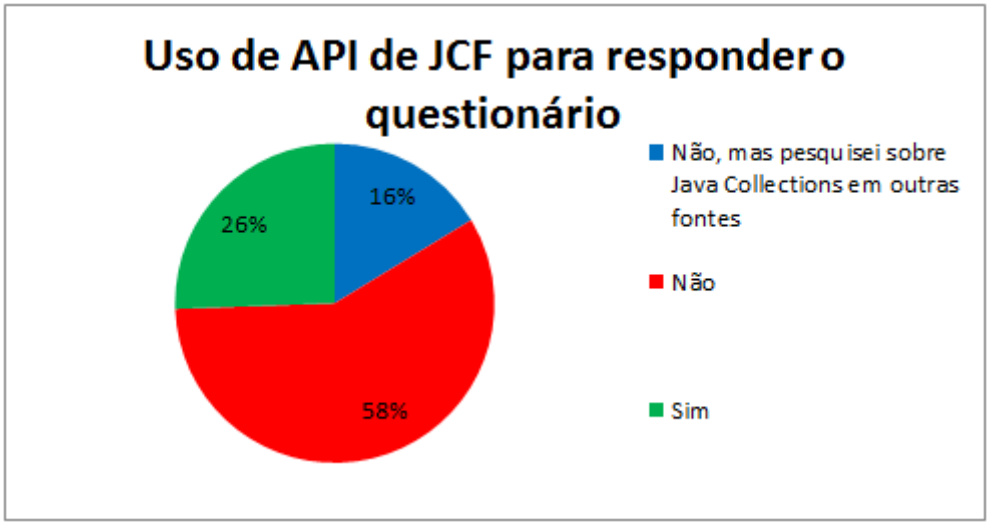

Figura 5.29: Uso de API de *JCF* para responder o questionário

Através dos dados percebe-se que a grande maioria dos participantes (quase 60%) não pesquisou sobre JCF nem na API nem em outras fontes. Sobre o restante dos participantes, 26% pesquisou sobre *JCF* na própria API e 16% em outras fontes de informação.

Esse fato aumenta ainda mais a preocupação em relação aos conhecimentos dos participantes acerca da *JCF*. Boa parte deles não realizou as transformações adequadas no código 1 (cerca de 55%, informação contida na RQ2) nem no código 2 (cerca de 75%, informação contida na RQ3). De modo geral, classificaram uma melhora na facilidade de entendimento do código transformado em relação ao original. Boa parte deles, principalmente no código 1, RQ5, ainda assim preferiram as transformações propostas por eles próprios, isso com a convicção que estavam corretos, mesmo sem terem, em sua maioria, pesquisado sobre JCF na API ou em outras fontes para confirmarem se estava mesmo certos.

Isso reforça a fragilidade da maioria dos programadores em pesquisar mais sobre JCF a fim de utilizar a coleção mais apropriada em seus sistemas. O que é levado a considerar mais a necessidade de elaboração de uma ferramenta que auxilie o programador de forma adequada no momento da escolha e da correção das coleções em seus projetos.

Participantes que dominavam mais o *JCF*

Ao analisar os dados contidos na Figura 5.30, referentes às *Respostas B* para esta pergunta de pesquisa, tem-se que a porcentagem de participantes que não acessaram a API de *JCF* para responder ao questionário foi praticamente a mesma das *Respostas A*. Ocorreu uma diferença na porcentagem de participantes que pesquisaram sobre *JCF* em outras fontes diferentes da API, porém esta porcentagem "migrou" para a porcentagem daqueles que utilizaram a API.

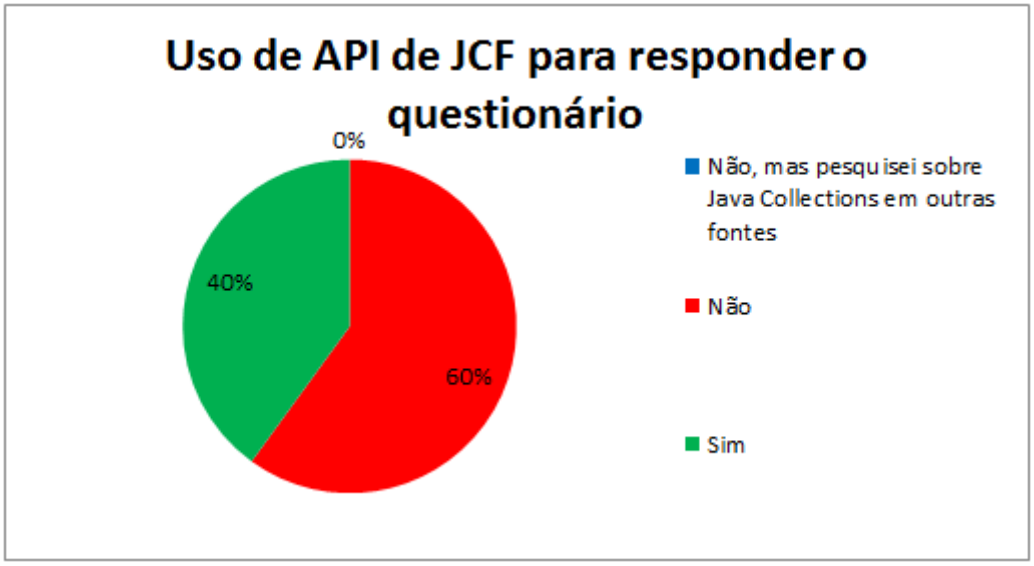

Figura 5.30: Uso de API de *JCF* para responder o questionário para os participantes que dominam mais *JCF*

Com estes dados tem-se que os indícios verificados para as *Respostas A* podem ser os mesmos para estas *Respostas B*.

5.5 Ameaças à validade

Dentre os tipos de ameaças à validade descritos em Wohlin et al. [39], abaixo selecionamos algumas ameaças à validade que podem afetar as conclusões do nosso *survey*.

1. Ameaça à validade interna:

   Instrumentation - Caso os instrumentos para realização da coleta e análises dos dados do *survey* estejam errados ou com vícios, ele é afetado negativamente.

   (a) *Questionário*: O instrumento criado para coleta dos dados do *survey* esta suscetível a erros de design, não coletando os dados necessários para avaliar às perguntas de pesquisa, o que pode vir a invalidar a coleta de dados realizada.

   Para avaliar quão bem o questionário captura aquilo que deseja-se medir (sua validade) foi realizado apenas o teste de *Content Validity*, onde foi feita um avaliação subjetiva por revisores (professores orientadores) com conhecimento na

área. Como os demais testes de validade precisam ter como base um questionário já existente e aplicado anteriormente, e o questionário deste *survey* foi feito especificamente para este caso, foi realizado somente um teste.

(b) Não supervisionado: Pelo fato da coleta de dados ter sido realizada de forma não supervisionada, pode ter ocorrido algum mal entendido em alguma questão que pode ter afetado os resultados do *survey*.

(c) Análise do pesquisador: a análise dos dados realizada pelo pesquisador com o auxílio de uma ferramenta de análise estatística (R) e uma planilha eletrônica, pode ter ocorrido falhas, que podem afetar negativamente o *survey*

2. Ameaça à validade externa:
   Interaction of selection and treatment - Se os sujeitos da população não forem representativas da população que queremos generalizar.

   (a) Participantes escolhidos - Os participantes do questionário foram selecionados de acordo com a sua conveniência, não foi baseado em nenhum critério para seleção da amostra (*Convenience Sampling e Focus groups*), o que a faz não ser representativa da população, tornando a generalização dos resultados para toda população de programadores que utilizam *JCF* não tão direta.

   (b) Teste estatísticos não foram feitos, pois a amostra não é representativa da população de programadores Java e *JCF*. Por esse motivo não é possível inferir conclusões para essa população, nos resumindo a suposições.

   (c) *Survey* - Por ser um *survey*, só pode analisar informações de maneira retrospectiva, podendo deixar de avaliar características importantes que provavelmente poderiam ser capturadas através de outro tipo de pesquisa, como o experimento.

   Existem certos testes (*intra-observer reliability, Alternate form reliability, intra-observer reliability*) para ser verificada a confiabilidade do *survey*(se seriam obtidos os mesmos dados com uma nova aplicação), porém por falta de tempo de uma reaplicação do *survey* não foi possível a realização de nenhum desses testes consequentemente não garantindo a confiabilidade.

5.5.1 Melhorias Futuras

Algumas melhorias que poderiam ser realizadas no questionário para futuras replicações dessa pesquisa seriam:

- Além de apresentar a transformação proposta utilizando a técnica de migração de API substituição direta, apresentaria outra transformação utilizando adaptadores. Perguntando aos participantes quais das três transformações eles preferiam? A propostas por eles, a de substituição direta ou a usando adaptadores.
- Além de solicitar os participantes para avaliar a facilidade de entendimento entre os códigos original e transformado, poderia solicitar a avaliação entre transformações diretas e utilizando adaptadores
- Modificar a transformação sugerida do código 2 da forma contida no Código Fonte 5.6.
- Aplicar o *survey* com uma quantidade maior e mais variada de programadores Java, para pelo menos poderem ser feitas inferências sobre os programadores Java brasileiros.

5.6 Considerações Finais do Capítulo

Através do *Survey* executado neste capítulo é visto que não existem diferenças muito significativas entre os dados obtidos nas *Respostas A* e nas *Respostas B*. Vê-se também que os programadores possuem dificuldade em transformar um código que contém uma lista sendo usada como um conjunto, para que ele utilize a coleção mais adequada, e mesmo após a apresentação de um código mais adequado, boa parte ainda não o escolheu como melhor opção. Isso dá um indício de que é preciso pensar em novas formas de auxílio quando os desenvolvedores precisam utilizar uma coleção *JCF*.

Capítulo 6

Trabalhos Relacionados

Neste capítulo, são apresentados trabalhos de pesquisa relacionados a este, apontando semelhanças e diferenças. Começando abordando o assunto de Migração de APIs 6.1 e posteriormente sobre a Seleção e substituição de coleções *JCF* 6.2.

6.1 Migração APIs

Com a escassez de trabalhos relacionados com o ponto principal dessa pesquisa, que é a substituições entre coleções *JCF* a fim de tornar o código mais adequado, foram considerados artigos e pesquisas sobre Migração de APIs, que se assemelha com a substituição dentre coleções *JCF*, sendo este último um ponto mais específico e uma parte ///integrante do primeiro.

Apesar de na teoria APIs de bibliotecas de software e *frameworks* não devessem mudar com frequência, na prática elas são bastante mutáveis, principalmente quando novos requisitos são adicionados ou refatorações são feitas. Refatoramentos causam cerca de 80% das mudanças em APIs [12], por isso é de suma importância ferramentas que auxiliem e/ou automatizem esse processo, que é custoso e suscetível a erros quando feito de forma puramente manual.

A *RefactLib*, apresentado por Taneja et al. [13], é um exemplo desse tipo de ferramenta que detecta automaticamente refatorações em bibliotecas, melhorando assim o auxílio aos programadores. Porém, a *RefactLib* não pode ser utilizada com eficácia para o contexto dessa pesquisa, visto que ela apensa detecta refatorações, como por exemplo mudanças de

assinaturas de método e renomeação de classes, não tratando das especificidades de cada coleção *JCF*.

Nesse contexto existe o trabalho de Nita e Notkin [2] que propuseram duas abordagens para substituir uma API por outra, a *Deep e Shallow Adaptation* (adaptação profunda e rasa) onde representam o uso de adaptadores e substituição direta respectivamente. Nessas abordagens são necessárias configurações que devem ser definidas pelo programador do programa a fim de adequar a abordagem ao contexto específico do programa. Isso demanda conhecimento especializado e cuidado, por esse motivo esta pesquisa estudou transformações de código, para o contexto estudado, sem a necessidade de configuração prévia. E essas transformações não utilizarem a abordagem *Shallow*, por ser mais prejudicial ao código original em termos de legibilidade e possivelmente desempenho.

Se a API sofre mudanças os clientes que a utilizam podem ser afetados, tendo que realizar alterações para se adaptar à nova versão. Essas mudanças são denominadas *ripple effects*, que podem impactar de várias formas o sistema que utiliza tal API. Normalmente são necessários vários *commits* para adequar o que foi mudado na API para o software cliente, o que mostra que essas mudanças não são triviais, como foi dito por Robbes et al. [43]. Para evitar este problema algumas APIs desencorajam os seus clientes de atualizarem para a nova versão ou lançam versões com elementos *deprecated* como alerta de futura remoção do mesmo. Visando um código adequado sob o contexto estudado neste trabalho foram estudadas as transformações para que o código se tornasse adequado e que não fossem necessários muitas mudanças no código, para evitar os *ripple effects*, utilizando a abordagem de Substituição Direta.

Ainda em relação à API, seu uso pode ser feito de modo incorreto prejudicando o entendimento do código e ainda o desempenho. Um exemplo de situação onde ocorre uso incorreto da API é quando o código cliente implementa algum método que já existe na API que está sendo utilizada. Para auxiliar o desenvolvedor a não cometer esse tipo de erro, Kawrykow et al. [44] propôs uma ferramenta que detecta automaticamente, através de análise estática, as partes do software que usam APIs de uma forma ineficiente. Essa ferramenta, além de poder gerar códigos que não são melhorias claras tornando o código incorreto, não se trata diretamente ao assunto de estudo desta pesquisa, não podendo ser aplicada para substituição entre coleções *JCF*.

Em relação ao problema abordado por nosso trabalho a técnica citada acima se torna onerosa de ser feita, pois as ferramentas geralmente solicitam que o próprio programador faça os mapeamentos necessários para a migração das APIs, e caso ele não saiba, o uso da ferramenta ocasionará em novos erros.

6.2 Seleção e substituição de coleções *JCF*

No trabalho de Shacham et al. [9] é apresentada *Chameleon*, uma ferramenta que, assim como as transformações propostas neste, também visa auxiliar o programador na escolha da coleção apropriada para sua aplicação, focando especificamente na implementação. *Chameleon* se utiliza de análise dinâmica para calcular certas métricas de desempenho e capacidade de armazenamento das coleções que posteriormente serão utilizadas em regras pré-definidas para realizar as trocas por coleções consideradas mais eficientes, que serão utilizadas de forma a consumir menos memória e ter um processamento mais rápido.

Como exemplo, tem-se que após o cálculo das métricas de certo programa, a regra do Código Fonte 6.1 será avaliada.

```
ArrayList:#contains > X ^ maxSize > Y ! LinkedHashSet
```

Código Fonte 6.1: Regra do Chameleon

Esta significa que quando o número total de chamadas, de uma coleção do tipo *ArrayList*, ao método `contains` for maior que uma constante X, e ainda o número máximo dessa coleção for maior que um certo Y, o tipo da coleção será substituído automaticamente por `LinkedHashSet`.

Diferentemente, as transformações estudadas nesta pesquisa realizam uma análise estática e apontam o local exato no código em que pode ocorrer uma substituição entre as interfaces `List e Set`. As transformações de código também só ocorrem quando certas condições de codificação forem verdadeiras.

Voltando ao exemplo do *Chameleon*, será que apenas pelo número total de chamadas ao método `contains`, da coleção envolvida, e pelo tamanho total dela, é possível trocar uma lista por um conjunto? A resposta é não, pois se nenhum dos `contains` que forem executados tenha um comando de inserção (como um `add`) em seu corpo, não caracteriza o bloqueio de elementos repetidos. Não sendo um IfContains A.2. Conclui-se que as transformações

propostas se forem aplicadas juntamente com a ferramenta *Chameleon* [9], para contexto de uso de interfaces `List e Set`, melhorará bastante o código em termos de desempenho, alocação de memória, uso adequado de coleções e clareza de codificação.

Já os trabalhos de Maia [4] e Xu [10], apresentam respectivamente as ferramentas *Collection Adapter* e *CoCo*, que também realizam trocas entre uma coleção *JCF* por outra mais adequada à situação, utilizando técnicas de adaptadores, onde nessa última recebe o nome de *glue code* e *Combo Classes*.

Como a técnica da abordagem apresentada nessa pesquisa foi a substituição direta, tem-se a não inserção de adaptadores no código cliente, que pode proporcionar mais em clareza no código. Porém, o espectro de casos de códigos possíveis a serem transformados diminui, pois as transformações estudadas abrange alguns casos entre *List e Set*. Outra vantagem é que as transformações fazem a análise do código e a Ferramenta de Maia [4] precisa obter respostas corretas do programador sobre quais características ele deseja naquela coleção, que foi citada por ela como sendo um trabalho de melhoria futuro pois alguns usuários relataram ter dificuldades com as características selecionadas para representar cada coleção na árvore de decisão.

Capítulo 7

Conclusão

A área que estuda as substituições entre coleções, em especial do *JCF*, ainda está em processo de produção de técnicas e ferramentas utilizáveis. Existe uma gama de detalhes e particularidades referentes às coleções e como o código se transforma quando uma é substituída por outra. Quando existe o problema de ter uma lista usada como conjunto, o código fica menos legível e pode ocorrer problemas de desempenho, a fim de ajudar a resolver esse problema, este trabalho apresentou uma análise de condições e consequências de um conjunto de transformações que substituem os tipos das coleções entre as interfaces `List` e `Set`, transformando variáveis, atributos e métodos necessários do restante do código. Estas transformações foram implementadas em Spoofax, sobre uma *Minimal Core Java*, Seção 3.1, com o objetivo de realizar uma adequação no uso dessas coleções, retornando um código mais apropriado com o mínimo de modificação do código original, fazendo-se uso para tal, a abordagem de Substituição Direta de migração de API.

Com esta pesquisa foi observado que o problema atacado, lista sendo usada como conjunto, faz parte de um problema maior: as transformações ocorridas quando substituímos uma lista por um conjunto e vise versa. Este por sua vez pertence a um conjunto ainda maior que é o das transformações necessárias entre as trocas de qualquer tipo de coleção *JCF*, que ainda pode ser feito uma analogia com o problema de realizar migração de APIs. O que ressalta ainda mais a importância deste trabalho para definição de um caminho viável para que todos essas problemáticas possam ser solucionadas. Caminho este que é a utilização da abordagem de Substituição Direta no contexto de coleções *JCF*.

Foram realizadas duas formas de pesquisas empíricas para avaliar as transformações.

Primeiramente foi feita uma execução controlada das transformações em códigos de softwares reais, e posteriormente a análise e discussão dos resultados por meio de um Estudo de Caso. Obteve-se através deste que a quantidade de aparições de classes que continham uma lista sendo utilizada como um conjunto (e por esse motivo deveria ter seu tipo trocado para `Set`) foi um número razoável, visto que nesta pesquisa foi considerada um pequeno ponto da problemática maior que é a realização de substituições adequadas entre todas as coleções de *JCF*. E esta problemática pode ser ainda maior, se considerarmos as migrações de APIs de modo geral. Isso se tratando de sistemas reais e relativamente estáveis.

Outro resultado interessante obtido através do Estudo de Caso foi a impossibilidade de transformação de quase 60% de todas as classes pré-selecionadas, apesar de possuírem lista(s) com comportamento(s) de conjunto(s). Esta diferença significativa se deu pelo fato de que nestas classes possuía, em algum trecho do código, pelo menos uma chamada ao método `get(index)` de `List` que pela Tabela 3.1 não possui método equivalente em `Set`, impossibilitando assim a transformação de todo código para evitar inserção de erros de compilação. Na grande maioria desses casos o mais adequado de acordo com as boas práticas de padrão de projeto seria a utilização de um iterador, que dá um indício de que além de ter o uso inadequado da lista como um conjunto, existe o uso errado dos padrões de projeto.

Também obteve-se o resultado que das transformações analisadas apenas a Transformação 1, Seção 3.4.4, foi executada, dando um indício que as situações das outras transformações estudadas não aparecem tão comumente como a primeira. O Estudo de Caso também deu um indício de que essa transformação foi executada com sucesso já que ela deixou os códigos mais legíveis, eficientes e os modificou minimamente. Isso indica que o uso da abordagem de Substituição Direta pode ser um caminho a ser seguido para resolver o problema de existir programas que não utilizam a coleção *JCF* mais adequada, ao invés de utilizar adaptadores que tornam o código resultante menos legível e eficiente.

Já através da aplicação do *survey*, por meio de um questionário não supervisionado pela internet, é visto que os participantes não propuseram, em sua maioria, transformações do trecho de código de forma adequada. E mesmo após a apresentação de uma transformação considerada mais adequada para o contexto (que foi proposta pelo autor da pesquisa) grande parte dos participantes não percebeu que as mudanças foram realmente necessárias. O que dá um indício que o problema estudado é relevante e que ainda pode ocorrer com uma boa

parcela dos programadores. Esse problema pode ser solucionado conscientizando os programadores a compreender todas os detalhes de todas as coleções de *JCF* para que eles as utilizem da forma mais adequada, ou implementando ferramentas que os auxiliem no momento da utilização das coleções. Esta última parece ser mais plausível para o problema, pois não necessita que o programador tenha conhecimento profundo sobre as coleções.

As transformações avaliadas neste documento não podem ser consideradas totalmente seguras, visto que quando trocamos uma lista por um conjunto, ou vice versa, mudamos a semântica daquela parte do programa. O que pode existir é uma Equivalência Semântica Fraca, como foi visto na Seção 3.4.2, que dá um certo grau de confiança que a transformação ocorrida pode resultar em um código válido para o contexto envolvido.

De modo geral, as transformações propostas podem ser muito úteis para o programador que possui mínimos conhecimentos em *JCF*, aumentando a legibilidade de seu código e diminuindo seu tempo de codificação caso estas transformações fossem ser feitas manualmente.

7.1 Trabalhos futuros

Com isso tem-se como principal trabalho futuro sugerido a implementação de uma ferramenta que implemente as transformações estudadas neste trabalho, pois não necessita que o programador tenha conhecimento profundo sobre as coleções *JCF* para utilizar a mais apropriada, a ferramenta o auxiliaria nessa tarefa. Ressaltando que é muito improvável executar ferramentas totalmente automatizadas para resolução desse tipo de problema, pois com a substituição de uma lista para um conjunto ocorre a perda da possibilidade de inserir elementos repetidos e também a indexação desses elementos. Com essa mudança de semântica não é possível a ferramenta por si só definir qual é o resultado correto, já que dependerá da opinião do programador informar o que será mais apropriado para seu software.

Então, tem-se que o mais apropriado seria a implementação de uma ferramenta semi-automática, que realizasse uma analise estática para encontrar os possíveis ponto passíveis de transformação. A partir daí seria apresentado ao programador possíveis transformações (inclusive as Transformações Alternativas) para os pontos encontrados contendo explicações mais diretas sobre qual coleção seria mais adequada a ser utilizada, porém ficando na respon-

sabilidade do desenvolvedor escolher a transformação que ele acredita ser mais apropriada.

Poderiam ser realizadas novas pesquisas sobre análises estáticas mais complexas para serem implementadas na ferramenta, pois as transformações estudadas não foram capazes de identificar erros de compilação presentes em códigos externos aos das classes transformadas, como ocorreu no Estudo de Caso na classe `GUISession`. Também através deste obteve-se o resultado que das transformações analisadas apenas a Transformação 1 foi executada, dando um indício que as outras transformações estudadas não são tão comuns assim. Então, em trabalhos futuros seria interessante dar um enfoque maior à essa transformação específica, podendo estende-la para que ela seja aplicada em qualquer tipo de programa Java, não somente aqueles escritos de acordo com a *Minimal Core Java*.

Também seria interessante aumentar o espectro das transformações, para que elas considerem também outras substituições além das entre `List` e `Set`. Como por exemplos incluir as classes que implementam `List` (`ArrayList`, `LinkedList`, etc) e as que implementam `Set` (`HashSet`, `TreeSet`, etc). E também pode abranger para transformações entre `List` e `Map` e/ou entre `Set` e `Map`.

Também poderia ser feita uma nova execução do *survey* realizado neste documento para o número bem maior de programadores *JCF*, realizando as melhorias sugeridas na Seção 5.5.1, e fazendo uma análise de validade e confiabilidade melhor já que será a segunda vez que o mesmo questionário será aplicado. E ainda realizar uma comparação entre códigos transformados através de adaptadores e de substituição direta.

Por fim é sugerido como possíveis trabalhos futuros, que reforçaria a relevância do problema e da efetividade das transformações e da ferramenta, a expansão do Estudo de Caso considerando todos os 111 projetos contidos em Terra et al. [35], e a execução de um estudo empírico mais poderoso, um experimento, com programadores utilizando a ferramenta sugerida anteriormente a fim de avaliar se ela realmente os auxiliam na tomada da decisão da coleção mais adequada para o seu contexto.

Referências Bibliográficas

[1] ORACLE. Java se documentation. http://docs.oracle.com/javase/7/docs/technotes/guides/collections/, Junho 2016.

[2] NITA, M.; NOTKIN, D. Using twinning to adapt programs to alternative apis. In: . ICSE '10. ACM, c2010. p. 205–214.

[3] BIEL, M. Marcus biel modificado. http://www.marcus-biel.com/wp-content/uploads/2016/03/java-collections-framework.pdf, Janeiro 2017.

[4] MAIA, M. A. O. Uma abordagem para adaptação de clientes do java collections framework baseada em técnicas de migração de apis. 8 2014.

[5] COLLINS, W. J. *Data structures and the java collections framework*. Third. ed. Ohn Wiley & Sons, INC, 2010.

[6] FAYAD, M.; SCHMIDT, D. C. Object-oriented application frameworks. *Commun. ACM*, New York, NY, USA, v. 40, n. 10, p. 32–38, Oct. 1997.

[7] COLLECTONS, C. A. Xtext main website. https://commons.apache.org/proper/commons-collections/, Janeiro 2017.

[8] LIBRARIES, G. Guava libraries website. https://github.com/google/guava/wiki, Janeiro 2017.

[9] SHACHAM, O.; VECHEV, M.; YAHAV, E. Chameleon: Adaptive selection of collections. *SIGPLAN Not.*, New York, NY, USA, v. 44, n. 6, p. 408–418, June 2009.

[10] XU, G. Coco: Sound and adaptive replacement of java collections. In: . ECOOP'13. Berlin, Heidelberg: Springer-Verlag, c2013. p. 1–26.

[11] LI, J.; WANG, C.; XIONG, Y.; HU, Z. Swin: Towards type-safe java program adaptation between apis. In: . PEPM '15. New York, NY, USA: ACM, c2015. p. 91–102.

[12] DIG, D.; JOHNSON, R. How do apis evolve? a story of refactoring: Research articles. *J. Softw. Maint. Evol.*, 2006.

[13] TANEJA, K.; DIG, D.; XIE, T. Automated detection of api refactorings in libraries. In: . ASE '07. New York, NY, USA: ACM, c2007. p. 377–380.

[14] SPOOFAX. Spoofax main website. `http://metaborg.spoofax.com`, Janeiro 2017.

[15] JDT. Jdt website. `http://www.eclipse.org/jdt/`, Janeiro 2017.

[16] MENS, T.; DEMEYER, S. *Software evolution*. 1. ed. Springer Publishing Company, Incorporated, 2008.

[17] BRCINA, R.; BODE, S.; RIEBISCH, M. Optimisation process for maintaining evolvability during software evolution. In: . c2009. p. 196–205.

[18] BROOKS JR, F. P. The mythical man-month (anniversary ed.). 1995.

[19] LIENTZ, B. P.; SWANSON, E. B. *Software maintenance management*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1980.

[20] LEHMAN, M. M. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, v. 68, n. 9, p. 1060–1076, 1980.

[21] EUGSTER, P. T.; GUERRAOUI, R.; SVENTEK, J. *Distributed asynchronous collections: Abstractions for publish/subscribe interaction*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000. p. 252–276.

[22] DIG, D.; NEGARA, S.; MOHINDRA, V.; JOHNSON, R. Reba: Refactoring-aware binary adaptation of evolving libraries. In: . ICSE '08. New York, NY, USA: ACM, c2008. p. 441–450.

[23] BALABAN, I.; TIP, F.; FUHRER, R. Refactoring support for class library migration. *SIGPLAN Not.*, p. 265–279, 2005.

[24] XING, Z.; STROULIA, E. Api-evolution support with diff-catchup. *IEEE Trans. Softw. Eng.*, 2007.

[25] IGARASHI, A.; PIERCE, B. C.; WADLER, P. Featherweight java: A minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, New York, NY, USA, v. 23, n. 3, p. 396–450, May 2001.

[26] FOWLER, M. *Domain specific languages*. 1st. ed. Addison-Wesley Professional, 2010.

[27] FOWLER, M. A pedagogical framework for domain-specific languages. *IEEE Softw.*, Los Alamitos, CA, USA, v. 26, n. 4, p. 13–14, July 2009.

[28] RASCAL. Rascal main website. `http://www.rascal-mpl.org/`, Janeiro 2017.

[29] MPS, J. Jetbrains mps main website. `https://www.jetbrains.com/mps/`, Janeiro 2017.

[30] XTEXT. Xtext main website. `http://www.eclipse.org/Xtext/`, Janeiro 2017.

[31] ERWIG, M.; PAIGE, R. F.; VAN WYK, E. The state of the art in language workbenches-conclusions from the language workbench challenge. In: . c2013. v. 8225. p. 197–217.

[32] STOFFEL, R. Comparing language workbenches. In: . c2010. p. 18–24.

[33] GOODE, W. J.; HATT, P. K., W. *Métodos em pesquisa social.* 3. ed. Cia Editora Nacional, 1969.

[34] TULL, D. S.; HAWKINS, D. I. *Marketing research, meaning, measurement and method*. Macmillan Publishing Co., Inc., 1976.

[35] TERRA, R.; MIRANDA, L. F.; VALENTE, M. T.; BIGONHA, R. S. Qualitas.class corpus: A compiled version of the qualitas corpus. *Software Engineering Notes*, v. 38, n. 5, p. 1–4, 2013.

[36] RANDOOP. Randoop website. `https://randoop.github.io/randoop/`, Janeiro 2017.

[37] MITCHELL, N.; SEVITSKY, G. The causes of bloat, the limits of health. In: . OOPSLA '07. New York, NY, USA: ACM, c2007. p. 245–260.

[38] ECLEMMA. Eclemma main website. `http://www.eclemma.org/`, Janeiro 2017.

[39] WOHLIN, R.; HUST, O.; REGNELL, W. *Experimentation in software engineering*. Kluwer Academic Publishers, 2000.

[40] ANDRADE, M. M. D. Introdução à metodologia do trabalho científico. *São Paulo: Atlas*, v. 7, 1999.

[41] CAVALCANTE, F. Questionário sobre substituição de coleções java. `https://form.jotformz.com/63308302885659`, Janeiro 2017.

[42] CAVALCANTE, F. Termos de consentimento do questionário sobre substituição de coleções java. `https://www.dropbox.com/s/3mfwflrfki6gbib/TERMO%20DE%20CONSENTIMENTO.pdf?dl=0`, Janeiro 2017.

[43] ROBBES, R.; LUNGU, M.; RöTHLISBERGER, D. How do developers react to api deprecation?: The case of a smalltalk ecosystem. In: . FSE '12. New York, NY, USA: ACM, c2012. p. 56:1–56:11.

[44] KAWRYKOW, D.; ROBILLARD, M. P. Improving api usage through automatic detection of redundant code. In: . ASE '09. Washington, DC, USA: IEEE Computer Society, c2009. p. 111–122.

Apêndice A

Lista de Regras

A.1 IfSimples

Dado **C** o conjunto de todos os comandos possíveis na linguagem reduzida de Java, temos que:

IfSimples: é um conjunto C', o qual $c' \epsilon C'$ e $c' \epsilon C$, onde c' possui uma expressão de verificação v que caso seja verdadeira será executado um novo conjunto de comandos b que estão dentro do corpo de c'.

A.2 IfContains

Dado **C´** o conjunto de todos os comandos **IfSimples** da linguagem reduzida de Java, temos que:

IfContains: é um conjunto C'', o qual $c'' \epsilon C''$ e $c'' \epsilon C'$, onde a expressão de verificação v de c'' é: (1) a verificação de existência de um objeto o em uma coleção cl, ou (2) a verificação de existência de uma outra coleção de objetos cl_o em uma coleção cl. E caso o ou todos os objetos de cl_o não exista(m) em cl será executado os comandos b, que estão dentro do corpo de c''.