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

Dissertação de Mestrado

Uma Abordagem para Adaptação de Clientes do *Java Collections Framework* Baseada em Técnicas de Migração de APIs

Mikaela Anuska Oliveira Maia

Campina Grande – Paraíba – Brasil
Agosto de 2014

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

Uma Abordagem para Adaptação de Clientes do *Java Collections Framework* Baseada em Técnicas de Migração de APIs

Mikaela Anuska Oliveira Maia

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

M217a Maia, Mikaela Anuska Oliveira.

Uma abordagem para adaptação de clientes do *Java Collections Framework* baseada em técnicas de migração de APIs / Mikaela Anuska Oliveira Maia. – Campina Grande, 2014.

106 f. : il. color.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2014.

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

Referências.

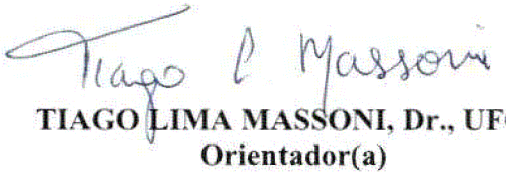
1. *Java Collections Framework*. 2. Migração de API. 3. Linguagem de Programação. I. Massoni, Tiago Lima. II. Farias, Adalberto Cajueiro de. III. Título.

CDU 004.43(043)

"UMA ABORDAGEM PARA ADAPTAÇÃO DE CLIENTES DO JAVA COLLECTIONS FRAMEWORK BASEADA EM TÉCNICAS DE MIGRAÇÃO DE APIs"

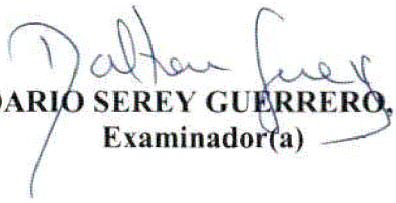
MIKAELA ANUSKA OLIVEIRA MAIA

DISSERTAÇÃO APROVADA EM 28/08/2014



TIAGO LIMA MASSONI, Dr., UFCG
Orientador(a)

ADALBERTO CAJUEIRO DE FARIAS, Dr., UFCG
Orientador(a)



DALTON DARIO SEREY GUERRERO, D.Sc, UFCG
Examinador(a)

MÁRCIO LOPES CORNÉLIO, D.Sc., UFPE
Examinador(a)

CAMPINA GRANDE - PB

Resumo

Apesar da diversidade que a API do *Java Collections Framework*(JCF) provê, com uma variedade de implementações para várias estruturas de dados, os desenvolvedores podem escolher interfaces ou classes inadequadas, em termos de eficiência ou propósito. Isto pode acontecer devido à documentação da API ser insuficiente ou a falta de análise ponderada pelo desenvolvedor de acordo com exigências do contexto. É possível a substituição manual, em paralelo com uma análise do contexto do programa. No entanto, isso é cansativo e suscetível a erros, desestimulando a modificação. Neste trabalho, nós definimos uma abordagem semi-automática para a seleção de interfaces e implementações dentro do JCF e a modificação de clientes do JCF, com base em técnicas de migração de API. A abordagem ajuda o usuário a escolher a coleção mais apropriada, com base em requisitos coletados por meio de perguntas mais intuitivas para o usuário. A seleção é resolvida com uma árvore de decisão que, a partir das respostas dadas pelo desenvolvedor, decide qual é a interface e implementação mais adequada do JCF. Após esta decisão, a modificação do programa é realizado por meio de adaptadores, minimizando a modificação do código fonte. Nós avaliamos a abordagem, implementada em uma ferramenta de apoio, com um estudo experimental que compreende estudantes de Ciência da Computação distribuídos aleatoriamente em grupos, os quais realizaram mudanças para clientes do JCF por diferentes métodos: manualmente, utilizando-se do Eclipse *Java Search* e nossa abordagem. Os resultados foram avaliados na qualidade, esforço e tempo gasto. Descobrimos que a maioria dos usuários teve dificuldades em escolher a interface ou implementação apropriada para os requisitos apresentados. Nossa abordagem evidenciou uma melhora no esforço de selecionar a melhor coleção para a exigência, poupando algum tempo no processo. Sobre a qualidade da coleção selecionada, encontramos o mesmo comportamento usando as duas ferramentas.

Palavras-Chave. Java Collections Framework, Migração de API.

Abstract

Despite the API diversity that the Java Collections Framework (JCF) provides, with diverse implementations for several data structures, developers may choose inappropriate interfaces or classes, in terms of efficiency or purpose. This may happen due to insufficient API documentation or the lack of thoughtful analysis by the developer according to context requirements. A possible solution is manual replacement, in parallel with an analysis of the program context. However, this is tiresome and error-prone, discouraging the modification. In this work, we define a semi-automatic approach for (i) the selection of interfaces and implementation within the JCF and (ii) the modification of JCF clients, based on API migration techniques. The approach helps the user in choosing the most appropriate collection, based on requirements collected by means of simple yes/no questions. The selection is resolved with a decision tree that, from the answers given by the developer, decides which is the most adequate interface (and implementation) from the JCF. After this decision, the actual program modification is performed by means of adapters, minimizing the source code modification. We evaluate the approach, as implemented in a supporting tool, with an experimental study comprising computer science students randomly distributed into groups, whose task was performing changes to JCF clients by different methods (manually, using Eclipse's Java Search and our approach); the results were evaluated on quality, effort and time spent. We found that most students had a hard time choosing the right interface or implementation for the given requirements. Our approach seemed to improve the effort of selecting the best collection for the requirement, saving some time in the process. Regarding the quality of the collection selected, we found the same behavior using both tools.

Keywords. Java Collections Framework, API migration.

Agradecimentos

Agradeço aos meus pais Cleone e Braulio, por não medirem esforços para me dar a melhor educação que suas condições permitiam. Também agradeço a minha irmã Mirna por me fazer entrar no curso de ciência da computação por "acidente", mas bem que poderia ter me guiado para fazer medicina. Agradeço ao meu marido Igor, por ser minha calma nos momentos de estresse, por me dar dicas de português, só não agradeço por fazer todo dia a pergunta "defende quando?".

Sou agradecida às minhas amigas de infância Clarissa, Galhães, Ferreira, Raissa, Gabriela, Jovana, Maitê e Mariana por me relaxarem com suas conversas diárias no Whatsapp e por fazerem bullying no colégio por eu sentar na primeira fila. Aos meus amigos do trabalho no STI Willian, Ianna, Camila, Marzina, Leandro, Leonel, Zé e Diogo, por acreditarem na minha capacidade multithread (Trabalho + Mestrado). Agradeço ao meu amigo Diego por me ajudar quando eu estava apanhando do Latex ou do Mac.

Sou grata aos meus orientadores, Tiago e Adalberto, por terem me recebido como orientanda, por acreditarem no meu trabalho e por todo suporte e ensinamentos que me deram durante essa temporada na qual trabalhamos juntos. Grata também ao aluno Fábio pelo o imenso apoio no desenvolvimento do meu trabalho e a todo o grupo do SPLAB.

Por fim, agradeço ao governo brasileiro, por ter apoiado financeiramente toda a minha pesquisa e ao pessoal da COPIN.

Epígrafe

"A menos que modifiquemos a nossa maneira de pensar, não seremos capazes de resolver os problemas causados pela forma como nos acostumamos a ver o mundo"

Albert Einstein

Dedicatória

À minha família, pelo incentivo e apoio.

Conteúdo

Lista de Acrônimos

API	*Application Programming Interface*
IDE	*Integrated Development Environment*
JCF	*Java Collection Framework*
JDT	*Java Development Toolkit*
UML	*Unified Modeling Language*
XML	*Extensible Markup Language*

Lista de Figuras

Lista de Tabelas

Lista de Códigos Fonte

Capítulo 1

Introdução

1.1 Contextualização

Durante o ciclo de vida de desenvolvimento de produtos de *software*, as modificações no código fonte podem afetar a qualidade em relação ao projeto inicial. Essa perda de qualidade é geralmente causada por modificações com objetivos de curto prazo, como correções de defeitos, ou por alterações realizadas sem a clara compreensão da concepção do sistema. O problema pode ser também causado pela necessidade de mudanças no *design* da aplicação devido à inclusão de novos requisitos.

Considerando a evolução de um *software* desenvolvido em *Java*, dentre as mudanças, pode existir uma alteração do tipo de coleção que estava sendo utilizado em algum trecho de código. Na alteração, precisa-se substituir a coleção anterior por um tipo de coleção que seja mais apropriada para o novo contexto. Entretanto, não existe uma interface ou uma implementação de uma coleção que seja melhor aplicável para todas as situações. Sendo assim, é necessário que o desenvolvedor analise e escolha a melhor opção para cada caso em questão.

Quando se trata de *software* em *Java*, o *Java Collections Framework*(JCF) é o *framework* utilizado para representação de coleções. Este *framework* é formado por uma variedade de interfaces e classes exaustivamente testadas. As classes representam estruturas de dados e algoritmos [5]. O *framework* provê uma API com vários Tipos Abstratos de Dados: Mapas, Conjuntos, Listas, Árvores, Matrizes e outras coleções. Os tipos abstratos de dados são geralmente representados por interfaces e, para cada interface, tem-se ao menos uma

Figura 1.1: Exemplo de substituição de um uso inadequado para outra versão utilizando adequadamente a coleção.

implementação desenvolvida de acordo com as características do tipo abstrato de dados que representa. Por exemplo, o tipo abstrato de dado Lista é representado pela interface `List` e para ela existem as implementações `Vector`, `ArrayList`, `LinkedList`, `Stack` e `CopyOnWhiteArrayList`.

Apesar da diversidade da API que o *Java Collections Framework* fornece, definindo um comportamento específico e diferentes implementações para as coleções existentes, é possível que, no ciclo de vida de uma aplicação, uma estrutura inadequada em termos de propósito ou eficiência seja adotada. Isto pode acontecer pela documentação da API não ser suficiente para o desenvolvedor ou pelo fato dele normalmente não analisar as características de cada implementação de coleção mais profundamente antes de selecioná-la.

Uma amostra de seleção de estrutura inadequada, em nível de código, está abordada na Figura 1.1. Neste caso, deseja-se utilizar uma implementação de coleção que não possua elementos repetidos porém, ao invés de utilizar uma estrutura que representa um conjunto, como pode ser visto no lado direito da Figura 1.1, fez-se uso de estrutura que representa uma lista e adicionou-se uma verificação antes da adição para saber se o elemento já existe na implementação de coleção.

A seleção de implementação de coleção a partir das suas características em alto nível de abstração é uma possível solução para facilitar indisposição do desenvolvedor de consultar documentação. Utilizando o exemplo da Figura 1.1, bastaria o desenvolvedor selecionar uma característica que explicitasse a ausência de elementos repetidos na instância da coleção para diferenciar o uso de uma lista e um conjunto.

Após a análise do contexto em que a implementação de coleção se encontra e da seleção da mais apropriada a partir de suas características, é necessário realizar a substituição da coleção pela selecionada. Uma solução seria a substituição manual da coleção. Porém, essa

substituição é cansativa e suscetível a erros, desencorajando a modificação. Uma melhor abordagem seria realizada com uma substituição de forma automatizada.

Existem atividades relacionadas que podem ser aplicadas como suporte na automação das transformações, como por exemplo, a migração de APIs. Esta atividade visa, em sua maioria, resolver o problema das mudanças que envolvem a evolução de *frameworks* e bibliotecas de software e que podem implicar em alterações na API. Os clientes que utilizam a API podem ser afetados por essas mudanças tendo que realizar alterações para se adaptar à nova versão. Desta forma, a migração de API consiste na adaptação da aplicação cliente para a nova versão da API, mantendo o comportamento da aplicação.

1.2 Objetivo

Neste trabalho, pretendemos definir uma abordagem para adaptação de uma melhor utilização do *Java Collections Framework* em *software Java*, com a colaboração do usuário, baseada em técnicas de migração de APIs e adaptadores. Os desenvolvedores poderão definir qual coleção eles desejam utilizar a partir de características de alto nível de abstração, implementadas em uma árvore de decisão, sem se preocupar com a documentação do JCF nem com as especificações associadas a cada tipo de estrutura de dados.

1.3 Solução

O trabalho de seleção da melhor estrutura de dados para determinado contexto foi resolvido com uma árvore de decisão que, a partir das características dadas pelo desenvolvedor, decide qual o melhor tipo de implementação da estrutura de dados. Adaptadores foram utilizados para a substituição no código fonte da instância de coleção, anteriormente usada, para a nova instância apropriada para o novo contexto. O mapeamento entre a instância anterior e a nova, para posterior substituição, foi realizado com a associação dos elementos que as compõem, diferenciando os que possuem diferença semântica e/ou sintática.

1.4 Avaliação

Avaliamos nossa abordagem, implementando uma ferramenta de suporte, juntamente com um estudo experimental compreendendo estudantes de ciência da computação selecionados randomicamente e distribuídos em grupos. A tarefa consistiu na realização de alterações de aplicações clientes do JCF por diferentes métodos: manualmente, utilizando-se do *Java Search* e nossa abordagem. Os resultados foram avaliados segundo critérios de qualidade, esforço e tempo gasto.

1.5 Resultados

Em um estudo de caso prévio sobre a utilização das implementações de coleções pelos desenvolvedores obtivemos que, em todos os projetos analisados, a implementação de `ArrayList` foi aplicada de forma inapropriada segundo nossas métricas. Diferindo do comportamento encontrado para o uso da implementação `LinkedList` que teve, em mais de 80% dos projetos, a aplicação apropriada. No estudo experimental realizado, obtivemos que 67% dos desenvolvedores tiveram receio de alterar uma implementação de coleção por outra. Ainda neste experimento, 75% dos desenvolvedores não selecionaram a implementação esperada para o contexto analisado. Com relação a abordagem aplicada por nós, esta melhorou o esforço de selecionar a melhor coleção para a exigência, poupando algum tempo no processo. Sobre a qualidade da coleção selecionada, encontramos o mesmo comportamento usando duas ferramentas, tanto a nossa, *Collection Adapter* como o *Java Search*.

1.6 Contribuições

Como contribuições do nosso trabalho podemos destacar:

- a realização de um estudo do comportamento dos desenvolvedores quanto à utilização de um subconjunto de coleções do JCF;
- a criação de uma abordagem, baseada em árvores de decisão, para selecionar a mais apropriada interface ou implementação de um elemento do JCF a partir de características de alto nível de abstração;

- a criação de uma técnica, baseada em adaptadores, para realizar uma modificação semi-automática em um cliente do JCF selecionando a implementação de coleção mais apropriada;
- a implementação de uma ferramenta de suporte para esta atividade.

1.7 Estrutura do Documento

Este documento está estruturado da seguinte forma:

Capítulo 2 - Fundamentação Teórica: apresenta conceitos necessários para o entendimento do trabalho apresentado. Tais conceitos abrangem o *framework* do *Java Collections* e o processo de Migração de APIs;

Capítulo 3 - Estudo Motivação: apresenta dados que nos motivaram a realizar o processo de seleção da melhor estrutura de coleções para cada contexto;

Capítulo 4 - Processo de Adaptação e a Ferramenta:apresenta como é definido o processo de adaptação das coleções e como ele foi implementado na ferramenta;

Capítulo 5 - Experimento: apresenta o experimento realizado com a ferramenta implementada e outras formas de transformação de coleções. Além da discussão geral sobre os resultados encontrados e delineia ideias para trabalhos futuros, baseados em nossa experiência;

Capítulo 6 - Trabalhos Relacionados: apresenta trabalhos da literatura que são relacionados ao nosso.

Capítulo 7 - Conclusão: apresenta as contribuições da pesquisa desenvolvida, trabalhos futuros e considerações finais sobre o trabalho.

Capítulo 2

Fundamentação Teórica

Para o bom entendimento do estudo apresentado neste documento, faz-se necessário o conhecimento de alguns tópicos relacionados ao trabalho desenvolvido. Neste capítulo, são apresentados brevemente conceitos importantes referentes a este trabalho. Na Seção 2.1 apresentamos uma breve descrição dos elementos do *framework* do *Java Collections*; enquanto que na Seção 2.2, descrevemos conceitos e tipos de migração existentes na área.

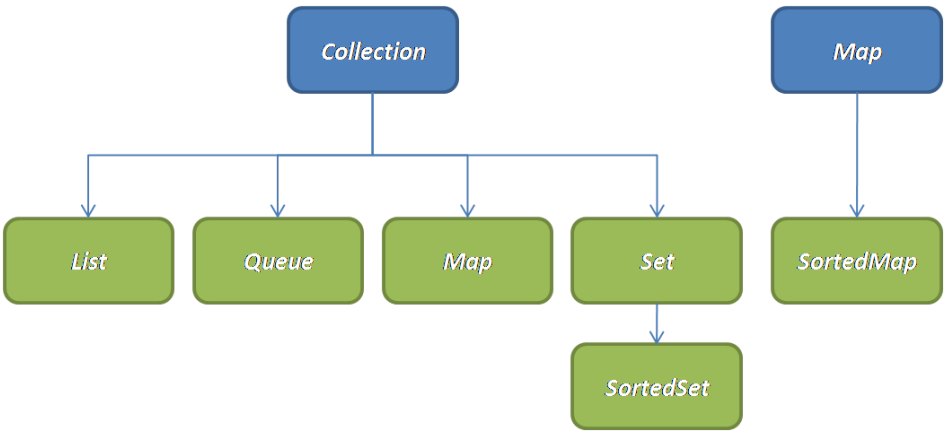
2.1 *Java Collections Framework*

O *Java Collections Framework* [4] define uma arquitetura unificada para representação e manipulação de coleções, permitindo que elas sejam manipuladas independentemente dos detalhes da sua representação. O *framework* é formado por uma variedade de interfaces e classes representando estruturas de dados e algoritmos [5] e provê uma API com vários Tipos Abstratos de Dados: Mapas, Conjuntos, Listas, Árvores, Matrizes, entre outros. Os tipos abstratos de dados são geralmente representados por interfaces e, para cada interface, tem-se ao menos uma implementação desenvolvida de acordo com as características do tipo abstrato de dados que representa.

O conjunto principal de interfaces que forma o *framework* e representa os diferentes tipos abstratos de dados são: `Collection`, `List`, `Set`, `Queue`, `Deque`, `SortedSet`, `Map` e `SortedMap`. As interfaces estão exibidas na Figura 2.1 de acordo com sua hierarquia. Existem duas árvores hierárquicas, já que o *Map* não é considerado uma coleção por ter um comportamento diferente dos elementos que estendem a interface

Collection.

Figura 2.1: Conjunto principal de interfaces que forma o JCF.



2.1.1 Selecionando a Interface do JCF

A seleção da interface mais adequada depende das características do tipo abstrato de dados que ela representa. Nas subseções seguintes, exibimos as características das principais interfaces do JCF e algumas de suas implementações de acordo com a documentação do *framework* [4].

`Collection`

A interface `Collection` representa a forma mais generalizada de se manipular uma coleção por ser o denominador comum entre todas as coleções. Não existe nenhuma implementação direta desta interface, somente de interfaces abaixo desta hierarquicamente.

`List`

A interface `List` representa as coleções ordenadas, podendo conter elementos duplicados, que fornecem métodos de acesso aos elementos de forma indexada. As duas implementações desta interface mais comumente utilizadas são a `LinkedList` e a `ArrayList`.

`Set`

A interface `Set` representa uma coleção que não permite ter elementos duplicados. As três implementações mais comuns desta interface são `HashSet`, `TreeSet` e `LinkedHashSet`.

Abaixo hierarquicamente da interface `Set` existe a interface `SortedSet` que mantém os elementos do conjunto em ordem crescente.

`Queue`

A interface `Queue` representa uma coleção que adota a ordenação dos elementos de acordo com a política FIFO (first-in, first-out), mas não é obrigatório que a ordenação aconteça seguindo a política, pode acontecer de acordo com um comparador fornecido ou na ordenação natural dos elementos.

`Deque`

A interface `Deque` representa uma coleção que permite utilizar tanto a política FIFO (first-in, first-out) como a LIFO (last-in, first-out). Seus métodos permitem adição, recuperação e remoção de elementos pelos dois lados da coleção. As duas implementações mais comuns desta interface são `ArrayDeque` e `LinkedList`.

`Map`

A interface `Map` representa um objeto que mapeia chaves para valores. Não pode ter chaves duplicadas e cada uma das chaves está associada a pelo menos um valor. Já a interface `SortedMap` representa também um mapa que mantém suas chaves em ordem decrescente. As três implementações mais comuns da interface `Map` são `HashMap`, `TreeMap` e `LinkedHashMap`.

2.1.2 Selecionando a Implementação de Coleção

Cada implementação de coleção tem uma performance específica para certas situações e conhecer esses comportamentos ajuda o desenvolvedor a selecionar melhor a coleção para

cada uma delas. Iremos exibir nesta subseção como se comporta implementações de uma seleção de coleções em algumas operações.

Classificação de Desempenho

A Tabela 2.1 faz um resumo da classificação do tempo das operações que derivam da complexidade algorítmica das operações de acesso, inserção, remoção e busca de elementos nas coleções `LinkedList` e `ArrayList` e nos mapas `TreeMap` e `HashMap`. Porém, iremos explicar nos parágrafos seguintes em quais casos o desempenho exibido na tabela é obtido.

Tabela 2.1: Tabela com a complexidade das operações das coleções baseados na Documentação do JCF [4]

	Acesso	**Inserção**	**Remoção**	**Busca**
`LinkedList`	Constante	Constante	Constante	Linear
`ArrayList`	Constante	Linear	Linear	Linear
`TreeMap`	Logarítmico	Logarítmico	Logarítmico	Logarítmico
`HashMap`	Constante	Constante	Constante	Constante

Na implementação de coleção `LinkedList` a operação de acesso, se realizada para o elemento do inicio da lista ou para o elemento corrente durante uma iteração, possui tempo constante. Porém, acessar um elemento em um determinado índice demanda tempo linear. A operação de inserção, se realizada no início da lista ou de um elemento depois do elemento corrente numa iteração, demanda tempo constante. Já inserir um elemento em um determinado índice custa tempo linear. Remover elementos do início da lista e remover o elemento corrente numa iteração acontece em tempo constante, entretanto remover elemento de um dado índice demanda tempo linear. A operação de busca por elementos custa um tempo linear.

Para implementação `ArrayList` a operação de acesso a elementos, dado um determinado índice, demanda tempo constante, já a operação de inserção também utilizando índice provoca mudanças dos elementos para um índice acima do que eles estavam (*shift*), por isso demanda tempo linear. Da mesma forma acontece para remoção em um dado índice, já que os elementos serão movidos para um índice a menos. Na operação de busca, se os elemen-

tos estiverem em ordem o tempo de execução é logarítmico, caso contrário demanda tempo linear.

Para o mapa `TreeMap`, dada a chave para realizar as operações, todas elas, acesso, inserção, remoção e busca, demandam tempo logarítmico. Já para o mapa `HashMap` a performance das operações depende da UHA (*Uniform Hashing Assumption*). UHA é uma suposição de que o conjunto de todas as chave possíveis é distribuído uniformemente sobre o conjunto de todos os índices da tabela [6]. Se o mapa suporta a UHA então acesso e busca dado uma chave, inserção e remoção acontecem em tempo constante considerando o caso médio. No pior caso, mesmo seguindo a UHA, as operações demandam tempo linear. Analisando estes dados é possível que o desenvolvedor tire algumas conclusões sobre a aplicação das coleções, por exemplo, se o contexto em que você vai utilizar a coleção demande muito acesso e substituição de elementos em um determinado índice, é melhor utilizar a coleção `ArrayList` pois ela terá melhor performance, caso o contexto consiste em iterar sobre a lista e na iteração inserir ou remover elementos, a `LinkedList` terá melhor performance. Por fim, podemos concluir que não existe uma coleção perfeita. Cada coleção é melhor aplicável para um certo contexto.

2.2 Migração de APIs

Nesta seção iremos abordar o conceito e algumas técnicas existentes para a Migração de APIs.

2.2.1 Conceito

Apesar de na teoria APIs de bibliotecas de *software* e *frameworks* ser estáveis, na prática elas mudam [7]. Quando novos requisitos são adicionados, *bugs* são corrigidos e são incorporados a uma nova versão da API. Se a API das bibliotecas possui mudanças, os clientes que a utilizam podem ser afetados tendo que realizar alterações para se adaptar à nova versão. Para evitar que as aplicações clientes adquiram incompatibilidades com a nova versão da API, 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. Vários exemplos podem ser encontrados no JCF: a Classe `Hashtable` foi substituída pela

`HashMap` que é semelhante mas permite o acesso não sincronizado a elementos de uma tabela *hash*, bem como o uso de chaves nulas e valores; a classe `ArrayList` é semelhante à classe `Vector` exceto pelo fato de que permite o acesso não sincronizado de elementos da lista. Fora do JCF, o Java 1.5 suporta a classe `java.io.StringBuilder` que provê a mesma funcionalidade da classe `java.io.StringBuffer` mas não é *thread-safe*, ou seja, permite acesso não sincronizado.

Nem sempre os elementos passam pela rotulação de *deprecated* e podem mudar de repente na nova versão. Danny Dig e seus colaboradores [7] mostraram em seu trabalho alguns tipos de mudanças que podem ocorrer em APIs e que podem prejudicar aplicações clientes da mesma.

- Remoção em APIs: No caso de remoção de elementos da API em uma nova versão, como uma classe ou um método por exemplo, qualquer cliente da versão anterior que fizer uso de algum desses elementos removidos não irá funcionar nesta versão mais nova.

- Refatoramentos de métodos: No caso de refatoramentos do código da API pode acontecer, por exemplo, a mudança da assinatura de um método: alterar de um para ter dois argumentos. Desta forma, os clientes da API que estão fazendo uso da versão do método com um argumento deixarão de compilar.

- Refatoramento de tipos: Para refatoramento de tipos podemos exemplificar que, com uma simples renomeação de uma classe da API, o cliente fazia uso desta classe irá deixar de compilar.

Esses casos exibidos são apenas alguns dos problemas que podem surgir com as mudanças e evolução de APIs. Como é impossível evitar as mudanças da API em alguns casos, existem processos que ajudam a aplicação cliente a fazer o processo de adaptação para a nova versão da API, processo chamado de **Migração de API**.

2.2.2 Técnicas de Migração

Na evolução de versão de uma API, existem diferenças entre as versões que nem sempre são documentadas dificultando sua determinação. O apoio ao processo de migração da aplicação

cliente para a nova API, tanto na detecção de mudanças como no processo de migração propriamente dito, pode ser realizada com o suporte de ferramentas que automatizam parte ou totalmente este processo.

Diferentes formas de determinação e migração das mudanças das versões foram utilizadas em publicações anteriores e as classificamos como: manual, automatização parcial e a automatização total.

Manual

Consideramos a migração manual de API quando a transformação do código cliente para se adaptar a nova versão da API é feita sem auxílio de ferramentas: o desenvolvedor manualmente corrige os erros que surgiram após a mudança de versão. Por exemplo, no caso exibido na Figura 2.2, se a API do *Java Collections Framework* alterar a assinatura do método `add` para conter três parâmetros em vez de apenas um, a aplicação cliente da API, para adaptar-se à nova versão da ferramenta, teria que alterar todos os locais em que esse método é utilizado com apenas um parâmetro. A alteração manual é a forma mais prática para pequenas mudanças porém, dependendo da modificação realizada na API, as aplicações clientes terão que realizar muitas alterações que, se feitas manualmente, são cansativas e suscetíveis a erros.

```
public class CrReservedName extends CrUML {

    private static List<String> names;

    private static List<String> umlReserved = new ArrayList<String>();

    static {
        umlReserved.add("none");
        umlReserved.add("interface");
        umlReserved.add("sequential");
        umlReserved.add("guarded");
        umlReserved.add("concurrent");
        umlReserved.add("frozen");
        umlReserved.add("aggregate");
        umlReserved.add("composite");
    }

    static {
        // TODO: This could just work off the names in the UML profile
        // TODO: It doesn't look like it matches what's in the UML 1.4 spec
        umlReserved.add("becomes");
        umlReserved.add("call");
        umlReserved.add("component");
        //umlReserved.add("copy");
        //umlReserved.add("create");
```

Figura 2.2: Trecho de código do projeto ArgoUML [1]

No trabalho de Kapur et al. [8], o autor exibe uma determinação e migração das mu-

danças entre duas versões da API da biblioteca JDOM, JDOM-b9 e JDOM-b10, de forma manual. Primeiro, foi alterada a versão da API na aplicação cliente e exibiu-se 467 erros de compilação. Para completar a identificação e a migração da API para a nova versão, foram gastos aproximadamente 2 dias inteiros de esforço envolvendo alteração manual pelos desenvolvedores, de cerca de 274 arquivos Java.

Automatização Parcial

Consideramos como automatização parcial a transformação com ajuda de ferramentas no processo, porém com alguma fase do processo feita de forma manual. Uma abordagem exemplo é a utilização de uma busca sintática automatizada, como a provida através do *Java Search* do Eclipse, que foi exposta por Kapur et al. [8]. Para migração de APIs, as diferenças são apontadas pelo compilador, após alteração da aplicação cliente para usar a nova versão da API e, com a ferramenta *Java Search*, busca-se as substituições que devem ser realizadas. A ferramenta apenas ajudará a localizar os lugares onde devem ser alterados, não auxiliando na alteração do código fonte. Neste caso, se as alterações na nova versão da API refletirem em muitas partes do código da aplicação cliente essa opção de automatização parcial não é a mais eficiente.

Xing e Stroulia [9] apresentaram em seu trabalho a ferramenta *Diff-CatchUp* que, com ajuda de outra ferramenta chamada UMLDiff, automaticamente identifica todos as mudanças nos *design-levels* das duas versões das APIs mas, depois que o problema de migração de API for reportado pelo compilador, um conjunto de substituições e propostas de exemplos de uso são formulados e apresentados para o cliente para que ele selecione a melhor substituição para o caso em questão. Sendo assim, temos parte do processo automático e parte do processo manual. Este tipo de migração pode ser a melhor opção para os casos em que necessita-se da opinião do usuário no processo de migração diminuindo a limitação da ferramenta.

Automatização Total

Apresentamos a automatização total como sendo um processo de migração da aplicação cliente para a nova versão da API de forma totalmente automatizada por uma ferramenta.

Na ferramenta apresentada por Dig et al. [7], a aplicação da API não necessita fazer

nenhuma mudança ou recompilação de código, a ferramenta, *ReBa*, automaticamente gera uma camada de compatibilização entre a nova versão da API e a antiga no código binário. Por Balaban et al. [10], foi criada uma técnica automática para migração de aplicações que usam classes legadas de APIs. Uma especificação de migração define como o uso das classes legadas são mapeadas para o uso de suas classes de substituições. Cada mapeamento é definido uma vez, e pode ser usada para migrar qualquer número de aplicações.

Algumas ferramentas implementadas, como a presente no trabalho de Dig et al. [7], consideram as mudanças que devem ser realizadas no código para adaptar-se a nova versão da API como refatoramentos. As ferramentas geralmente não abrangem todos os refatoramentos possíveis, não conseguindo migrar toda a aplicação cliente para a nova versão da API, ou não consideram as mudanças que não são refatoramentos, mudanças estas que possuem alteração semântica da aplicação, limitando assim a migração.

2.2.3 Adaptações e Transformações Suportadas

Existem diversos métodos para realizar a adaptação ou transformação da aplicação cliente para a nova API. Separamos os métodos em dois: transformação direta e adaptadores. O refatoramento também pode ser considerado uma técnica para realizar as transformações e pode acontecer em conjunto com as outras técnicas.

Transformação Direta

A transformação direta é um método de migração caracterizado pela substituição direta do trecho com a versão antiga da API para o trecho com a versão nova da API. Na Figura 2.3, é exibida uma das formas de transformação direta, nomeada como *Shallow Adaptation* por Nita e Notkin [2]. Supondo que (a) é o trecho do programa com a API na versão anterior, (b) é o resultado da aplicação da transformação direta, *Shallow Adaptation*, nesta versão. A aplicação resulta em uma cópia do programa no qual as referências da versão anterior da API é substituída com a referência da nova versão. No caso exemplo, o *ArrayList* e *Iterator* foram substituídos pelos *Vector* e *Enumeration*, respectivamente.

A transformação direta é uma técnica mais fácil de ser aplicada, porém ela é limitada por não poder se comportar de forma mais generalizada. Apenas em casos bem específicos e de

```
Vector objects = new Vector(); ...
void printObjects(Vector objects) {
(a)   Enumeration e = objects.elements();
      while (e.hasMoreElements())
        System.out.println(e.nextElement()); }

ArrayList objects = new ArrayList(); ...
void printObjects(ArrayList objects) {
(b)   Iterator it = objects.iterator();
      while (it.hasNext())
        System.out.println(it.next()); }
```

Figura 2.3: Trecho de código retirado de [2] representando a técnica de *Shallow Adaptation*

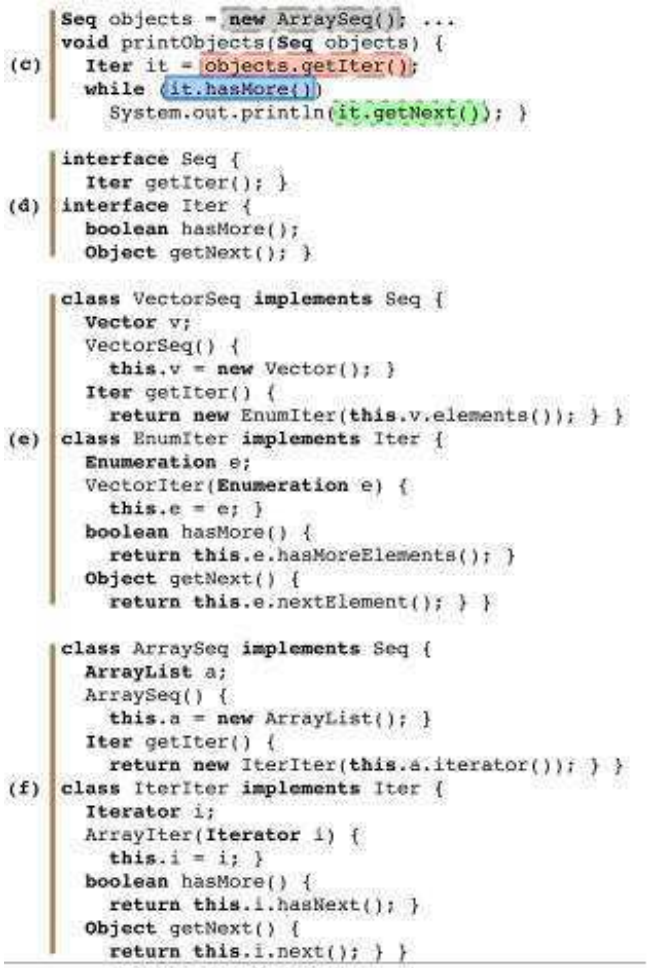
domínio de conhecimento do programador podem ser transformados.

Adaptadores

O método de migração de APIs utilizando adaptadores é uma forma de permitir que os clientes das APIs possam usar tanto a nova como a antiga versão da API da biblioteca. No trabalho de Dig et al. [7], a alteração é realizada na própria biblioteca, sendo criada uma *adapted-library* que contém o mesmo código na nova versão da biblioteca, mas suporta tanto a antiga como a nova versão da API.

Na Figura 2.4, podemos ver o resultado da aplicação para a mesma situação referenciada na Figura 2.3 porém, utilizando uma técnica de adaptação chamada *Deep Adaptation* mostrada no trabalho de Nita e Notkin [2]. No caso exemplo, a adaptação foi feita com interfaces e classes em Java, tendo como resultado a possível utilização de ambas as versões da API.

A técnica utilizando adaptadores é mais difícil de ser implementada do que a transformação direta, já que, geralmente, é necessária a criação de uma nova API, seja na própria biblioteca, ou no código do cliente, que suporte ambas as versões da API. No caso da técnica de *Deep Adaptation*, dada a **API A** como a antiga versão e a **API B** como a nova versão, se o programa possuir **n** tipos na **API A**, deverão ser criadas **n** tipos de interfaces ou classes abstratas 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.



```
    Seq objects = new ArraySeq(); ...
    void printObjects(Seq objects) {
(c)   Iter it = objects.getIter();
      while (it.hasMore())
        System.out.println(it.getNext()); }

    interface Seq {
      Iter getIter(); }
(d) interface Iter {
      boolean hasMore();
      Object getNext(); }

    class VectorSeq implements Seq {
      Vector v;
      VectorSeq() {
        this.v = new Vector(); }
      Iter getIter() {
        return new EnumIter(this.v.elements()); } }
(e) class EnumIter implements Iter {
      Enumeration e;
      VectorIter(Enumeration e) {
        this.e = e; }
      boolean hasMore() {
        return this.e.hasMoreElements(); }
      Object getNext() {
        return this.e.nextElement(); } }

    class ArraySeq implements Seq {
      ArrayList a;
      ArraySeq() {
        this.a = new ArrayList(); }
      Iter getIter() {
        return new IterIter(this.a.iterator()); } }
(f) class IterIter implements Iter {
      Iterator i;
      ArrayIter(Iterator i) {
        this.i = i; }
      boolean hasMore() {
        return this.i.hasNext(); }
      Object getNext() {
        return this.i.next(); } }
```

Figura 2.4: Trecho de código retirado de Nita e Notkin [2] representando a ténica de Deep Adaptation

Refatoramento

Refatoramento é o processo de alteração de um sistema de *software* de modo que o comportamento externo do código não mude, mas que sua estrutura interna seja melhorada [11]. Estudos anteriores mostraram que 80% das mudanças de APIs são causadas por refatoramentos [12]. Em vários trabalhos de migração de APIs, considera-se que as mudanças existentes entres as versões de APIs são refatoramentos. Os trabalhos de Dig et al. [7], Kapur et al. [8], Balaban et al. [10], Savga et al. [13] e Taneja et al. [14] utilizam essa referência para migração de API. Porém, trabalhos mostraram visões diferentes destas mudanças. No trabalho de Murphy-Hill et al. [15] descobriu-se que os desenvolvedores, na maioria das vezes, não realizam refatoramentos ditos puros, eles intercalam mudanças de refatoramento com outras mudanças que modificam o comportamento do *software*. Adicionalmente, Schäfer et al. [16], mostraram que mudanças conceituais são definidas associação de responsabilidades a blocos de códigos existentes que não preservam o comportamento anterior do código. Foram encontrados, no seu estudo de caso, que um significante número de divergência de versões eram mudanças que não se classificam como refatoramentos. Se tratando de mudanças conceituais, se na migração de API forem utilizadas ferramentas que usam técnicas focadas em descobrir e migrar apenas refatoramentos, elas não serão consideradas na migração.

Cada técnica e método utilizado tem seu comportamento melhor em alguns contextos que deve ser analisado antes da decisão da realização da migração da API. Além do contexto, deve-se estudar as limitações de cada técnica e método para saber se as mesmas não irão prejudicar a migração.

Capítulo 3

Estudo exploratório sobre a utilização de coleções em projetos *open source*

Neste capítulo, apresenta-se um estudo da aplicação de implementações interface *List* do *Java Collections Framework*, no contexto de projetos *open source* em *Java*. O objetivo é verificar como as implementações são utilizadas, considerando o comportamento esperado para a estrutura de dados que a compõe. A análise é realizada através da criação e coleta de métricas para determinar o contexto em que as implementações de coleções foram utilizadas e qual seria a melhor aplicada para essa situação. O estudo nos traz o conhecimento da existência de projetos que possuem formas inadequadas de aplicação de implementações de coleções, sendo este a motivação para a criação de um processo de auxílio à seleção das implementações e à utilização das mesmas exibido no Capítulo 4.

Consideramos um exemplo de seleção de estrutura inadequada, em nível de código, o caso em que se deseja uma implementação de coleção que não possua elementos repetidos e, para isso, ao invés de utilizar uma implementação que representa um conjunto, utiliza-se uma coleção que representa uma lista e faz-se uma verificação se o elemento já existe na coleção ao adicioná-lo. Este cenário está exemplificado na Figura 1.1.

Neste estudo, selecionamos para análise nos projetos apenas três possibilidades de coleções do JCF que implementam a interface *List*: `ArrayList` e `LinkedList`, por serem as implementações mais comumente utilizadas; e `Vector` por ser uma implementação considerada legada.

3.1 Metodologia

Nesta seção abordamos a metodologia utilizada para a realização deste estudo.

3.1.1 Tipo de Pesquisa Empírica e o Contexto da Pesquisa

A pesquisa executada é um estudo de caso exploratório. A nossa avaliação está relacionada com o *Java Collections Framework* portanto, selecionamos entre os projetos *open source* desenvolvidos em *Java*. Os projetos escolhidos estão exibidos na Tabela 3.1, os dados foram retirados do trabalho de Terraet al. [17] que disponibiliza os projetos *open source* pré-compilados.

Tabela 3.1: Projetos *open source* utilizados no estudo de caso exploratório

Projeto	**Descrição**	**KLOC**	**Nº de Classes**	**Versão Utilizada**
ArgoUML [1]	Ferramenta *open source* de modelagem UML	192	2560	0.34
Apache Ant [18]	Ferramenta *open source* de principal uso na construção de aplicações Java	105	1290	1.8.4
AspectJ [19]	Extensão orientada a aspectos para a linguagem de programação *Java*	412	2624	1.6.9
FindBugs [20]	Programa *open source* que usa análise estética para buscar *bugs* no código *Java*	109	1715	1.0.2
JEdit [21]	Programa *open source* de edição de texto para programadores *Java*	107	1128	4.3.2

3.1.2 Proposições de estudo

A realização deste estudo de caso exploratório procura responder se a aplicação da coleção do JCF presente nos projetos *open source* analisados está de acordo com o comportamento esperado para sua utilização. Para cada elemento do JCF considerado neste estudo temos uma proposição de estudo a ser verificada em cada projeto analisado.

1. O elemento `ArrayList` foi utilizado de acordo com o comportamento esperado para sua utilização.

2. O elemento `LinkedList` foi utilizado de acordo com o comportamento esperado para sua utilização.

3. O uso de implementações sincronizadas foi realizado de acordo com o comportamento esperado para sua utilização.

Cada proposição de estudo será refutada ou não a partir do resultado das métricas coletadas.

3.1.3 Variáveis de Pesquisa

No Capítulo 2 na Seção 2.1, exibimos o tempo de execução de operações em algumas implementações de coleções. É a partir da performance das operações nas implementações `LinkedList` e `ArrayList` que definimos qual a melhor para aquele contexto. Por exemplo, se existe uma instância de coleção do tipo lista que tende a ter mais operações de inserção e remoção que operações de acesso, essa instância pelo seu desempenho é mais indicada a ser `LinkedList`. Caso existam mais operações de acesso, a implementação tende a ser um `ArrayList`. Para a implementação de coleção `Vector` não há uma conclusão do que é melhor aplicado nos casos de acesso sincronizado, se a utilização da implementação que por si só já é sincronizada ou se a utilização de blocos ou métodos sincronizados com implementações de coleções não *thread-safe* como `ArrayList` e `LinkedList`. Porém, se existe bloco sincronizado em uma implementação do tipo `Vector` isto é caracterizado como um mau comportamento, já que a ela é *thread-safe* e não necessitaria deste bloco.

As nossas variáveis de pesquisa são métricas implementadas para determinação de boas e más aplicações das implementações de coleções. As métricas estão explicitadas nas Fórmulas 3.1, 3.4 e 3.7.

Nas instâncias da coleção `ArrayList` e `LinkedList`, utilizamos uma combinação de métricas para avaliar o contexto em que a implementação de coleção foi aplicada: se foi apropriado para aquela implementação ou não. O cálculo das métricas para implementações do tipo `ArrayList` e `LinkedList` é realizado através da contabilização da quantidade de chamadas das variáveis a métodos pré-determinados. Estes métodos podem ter o tempo de acesso constante ($O(1)$) , amortizado($O(n)$ no pior caso em que o *array* precisa ser redefinido e copiado) ou linear ($O(n)$).

O contexto ($CONTEXT$), presente nas Fórmulas 3.1, 3.4, é criado a partir da razão entre a quantidade de chamadas das variáveis aos métodos com tempo constante de acesso ($CONST\_TIME\_CALLS$) e a soma da quantidade total de chamadas de variáveis aos métodos com tempo de acesso constante e amortizado ($AMORT\_TIME\_CALLS$) ou linear ($LINEAR\_TIME\_CALLS$). Caso as chamadas aos métodos das variáveis estejam inseridos em um laço, é atribuído um peso a mais para o valor dessa chamada já que com análise estática não é possível determinar quantas vezes esta chamada será repetida no laço.

Realizamos também a análise do uso de implementações sincronizadas: sejam `LinkedList` e `ArrayList` em blocos ou métodos sincronizados ou o uso do `Vector`. Para esta análise foi utilizada uma métrica sintetizada na Fórmula 3.7 que verifica se a implementação de coleção instanciada está inserida em um bloco ou método sincronizado.

As métricas foram desenvolvidas a partir da implementação de um *software* com um auxílio do JDT (*Java Development Toolkit*), ferramenta que fornece API para navegar na árvore de elementos Java. O JDT permitiu que realizássemos uma análise estática do código por possibilitar a separação de cada fragmento do código fonte dos projetos selecionados, exibindo-os em formado de árvore, tornando possível a captação dos dados para realização do cálculo das métricas.

Métrica para a implementação de coleção `ArrayList`

O contexto para a implementação de coleção `ArrayList` é determinado pela Fórmula 3.1.

$$CONTEXT = \frac{CONST\_TIME\_CALLS}{AMORT\_TIME\_CALLS + CONST\_TIME\_CALLS} \tag{3.1}$$

Determinamos os elementos $CONST\_TIME\_CALLS$ e $AMORT\_TIME\_CALLS$

da fórmula $CONTEXT$ a partir dos conjuntos VA, MC, MA e M e das funções $C(va, m)$, $L(c)$ e $Q(x)$. O conjunto VA representa as variáveis que referenciam uma instância da coleção `ArrayList`; MC representa o conjunto de métodos executados com tempo constante para coleções do tipo `ArrayList`, consideramos no estudo os métodos `get, set e setElementAt`, métodos mais utilizados dependentes de índices; o conjunto MA representa os métodos executados com tempo amortizado para a implementação de coleção `ArrayList`, consideramos para o estudo os métodos `add, addAll, addFirst e addLast`; M é o conjunto de métodos acessados para a implementação de coleção `ArrayList`, tanto constantes como amortizados.

$VA = \{va_1, va_2...va_n\}$
$MC = \{get, set, setElementAt\}$
$MA = \{add, addAll, addFirst, addLast\}$
$M = MA \cup MC = \{m_1, m_2, ..., m_k\}$

A função $C(va, m)$ é a que calcula as chamadas da variável va pertencente a VA ao método m pertencente a M; $L(c)$ é a função peso de um laço atribuído à chamada c do conjunto C.

$C(va, m) = \{c_1, c_2, ..., c_z\}$
$$L(c) = \begin{cases} 0, & \text{se} \quad c \notin Loop, \\ 1, & \text{se} \quad c \in Loop. \end{cases}$$

$Q(x)$ é a função quantidade que retorna o cálculo da quantidade de chamadas de uma variável a um método, já atribuindo o peso do laço caso esta chamada esteja dentro de um laço.

$Q : \mathbb{N} \to \mathbb{N}$
$Q(x) = COUNT(x) + 2 * L(x)$

Então, $\forall va \in VA$, $mc \in MC$, $ma \in MA$, temos:

$$CONST\_TIME\_CALLS = Q(C(va, mc)) \tag{3.2}$$

$$AMORT\_TIME\_CALLS = Q(C(va, ma)) \tag{3.3}$$

Métrica para a implementação de coleção `LinkedList`

O contexto para a implementação de coleção `LinkedList` é determinado pela Fórmula 3.4.

$$CONTEXT = \frac{LINEAR\_TIME\_CALLS}{LINEAR\_TIME\_CALLS + CONST\_TIME\_CALLS} \tag{3.4}$$

Determinamos os elementos $CONST\_TIME\_CALLS$ e $LINEAR\_TIME\_CALLS$ da fórmula $CONTEXT$ a partir dos conjuntos VL, MC, ML e M e as funções $C(vl, m)$, $L(c)$ e $Q(x)$. O conjunto VL é formado pelas variáveis que referenciam uma instância da coleção `LinkedList`; o conjunto de métodos executados em tempo constante nas instâncias é representado por MC, consideramos para o estudo os métodos `add`, `addAll`, `addFirst`, `addLast`, métodos mais utilizados neste caso; já ML é o conjunto de métodos executados com tempo linear para a implementação de coleção `LinkedList`, considerados no estudo os métodos `get`, `set` e `setElementAt`; o conjunto M são os métodos executados tanto em tempo linear como em tempo constante para instâncias da coleção `LinkedList`.

$VL = \{vl_1, vl_2 ... vl_n\}$
$MC = \{add, addAll, addFirst, addLast\}$
$ML = \{get, set, setElementAt\}$
$M = ML \cup MC = \{m_1, m_2, ..., m_k\}$

A função C(vl, m) calcula as chamadas da variável vl pertencente a VL ao método m pertencente a M; L(c) é a função peso de um laço atribuído à chamada c do conjunto C.

$C(vl, m) = \{c_1, c_2, ..., c_z\}$

$$L(c) = \begin{cases} 0, & \text{se} \quad c \notin Loop, \\ 1, & \text{se} \quad c \in Loop. \end{cases}$$

$Q(x)$ é a função quantidade que retorna o cálculo da quantidade de chamadas de uma variável a um método, já atribuindo o peso do laço caso esta chamada esteja dentro de um laço.

$Q : \mathbb{N} \rightarrow \mathbb{N}$
$Q(x) = COUNT(x) + 2 * L(x)$

Então, $\forall vl \in VL, mc \in MC, ml \in ML$:

$$CONST\_TIME\_CALLS = Q(C(vl, mc)) \tag{3.5}$$

$$LINEAR\_TIME\_CALLS = Q(C(vl, ml)) \tag{3.6}$$

`ArrayList` X `LinkedList`

A aplicação das métricas revela um valor para o contexto que a instância de coleção foi aplicada, e o valor resultante está dentro de um intervalo de 0 a 1. No contexto do intervalo $[0, 0.5)$ a implementação `LinkedList` é mais apropriada, pois significa que existe mais uso de métodos de tempo de execução constante($CONST\_TIME\_CALLS$) do que métodos de tempo de execução linear($LINEAR\_TIME\_CALLS$) para esta coleção. Já no contexto do intervalo $(0.5, 1]$ a aplicação da implementação `ArrayList` é a mais apropriada, pois existem mais métodos de performance constante($CONST\_TIME\_CALLS$) para o `ArrayList` do que métodos de performance amortizada($AMORT\_TIME\_CALLS$). Para o contexto que resultar no valor 0.5, podemos considerar qualquer uma das duas implementações.

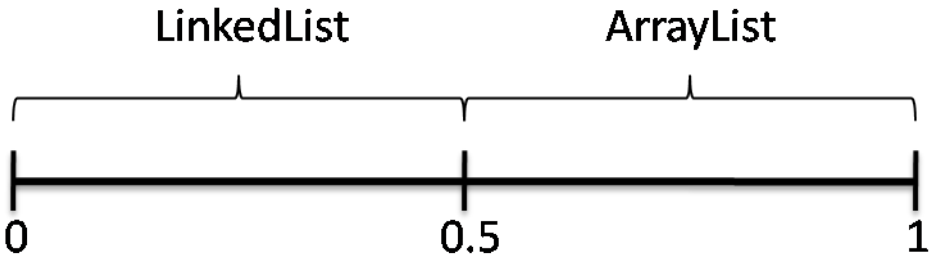


Figura 3.1: Intervalo do contexto das coleções

Outras coleções X `Vector`

O contexto para a determinação do uso de implementações de coleções sincronizadas é determinado pela Fórmula 3.7.

$$CONTEXT = SYNC\_BLOCKS \tag{3.7}$$

Determinamos o elemento $SYNC\_BLOCKS$ da fórmula $CONTEXT$ a partir dos conjuntos V e VS e a Função $S(v)$. O conjunto de variáveis que referenciam uma instância da coleção `ArrayList`,`LinkedList` e `Vector` é determinado pelo conjunto V; o conjunto de variáveis inseridas em blocos e métodos sincronizados é representado por VS; e a função S determina se a variável está inserida em contexto sincronizado ou não, se estiver retorna 1, caso contrário retorna 0.

Dados:

$$V = \{v_1, v_2, ..., v_n\}$$
$$VS = \{vs_1, vs_2, ..., vs_z\}$$
$$S(x) = \begin{cases} 0, & \text{se} \quad x \notin VS, \\ 1, & \text{se} \quad x \in VS. \end{cases}$$
$$S : V \rightarrow \mathbb{N}$$

Então, $\forall v \in V$ temos:

$$SYNC\_BLOCKS = S(v) \tag{3.8}$$

3.1.4 Configuração do Estudo de Caso

Forma de Coleta de Dados e análise

Desenvolvemos um programa em *Java* que acessou o código fonte dos projetos *open source* selecionados, implementamos e medimos as métricas pré-definidas. Na Figura 3.1, definimos um intervalo de valores através da fórmula do contexto para as coleções `ArrayList` e `LinkedList`. Fizemos o teste de normalidade e simetria nos dados coletados nas métricas. Os dados não são normais e não são simétricos, então selecionamos o *SIGN-TEST* para verificar em que parte do intervalo da Figura 3.1 os valores computados como contexto nas Fórmulas 3.1 e 3.4 se enquadravam: inferiores, superiores ou iguais a 0.5.

Instrumentação

As seguintes ferramentas foram utilizadas para nosso estudo:

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

2. Software R: Análise estatística.

3. Eclipse Java development tools (JDT): O *software* de coleta de dados foi criado como *plugin*.

3.2 Resultados

Nesta seção exibem-se tabelas com os resultados coletados através das métricas implementadas.

3.2.1 Resultados das coleções `ArrayList` e `LinkedList`

A Tabela 3.2 exibe os resultados da análise da aplicação das métricas para as coleções instanciadas como `ArrayList` e `LinkedList`. No campo total está resumido os resultados coletados de todos os projetos. Na tabela, o termo *Métodos Dependentes de índice* refere-se aos métodos de tempo de acesso linear para o `LinkedList` e de tempo constante para o `ArrayList`. Já o termo *Métodos Independentes de índice* refere-se aos métodos de tempo de acesso constante para o `LinkedList` e de tempo amortizado para o `ArrayList`.

O tipo definido como *Undefined* representa situações em que não foi possível a distinção de qual implementação de coleção foi usada, apenas foi possível a determinação que a variável é do tipo da coleção *List*. O Código Fonte 3.1 é um exemplo de uma situação em que não conseguimos fazer esta distinção. Como a coleção $fSelectionListeners$ é instanciada através do retorno do método $createList()$ da classe $CollectionsFactory$ e este retorno tem o tipo *List*, através da analise estática não é possível classificar a coleção instanciada neste caso como `ArrayList`, `Vector` ou `LinkedList`. Nos resultados que serão exibidos nas seções subsequentes, os casos em que consideramos o tipo *List* são os casos em que não foi possível classificar a variável estudada.

Tabela 3.2: Resultados da análise da aplicação das métricas para as coleções instanciadas como `ArrayList` e `LinkedList`

	Tipo	**Total de variáveis Instanciadas**	**Chamadas de Métodos Dependentes de índice**		**Chamadas de Métodos Independentes de índice**	
Apache Ant	`ArrayList`	79	31	21,7%	112	78,3%
	`LinkedList`	11	0	0%	13	100%
	Undefined	14	9	47,4%	10	52,6%
ArgoUML	`ArrayList`	256	136	17,2%	657	82,8%
	`LinkedList`	13	1	1,4%	75	98,6%
	Undefined	82	145	80,5%	35	19,5%
AspectJ	`ArrayList`	380	119	14,4%	713	85,6%
	`LinkedList`	13	28	65,2%	15	34,8%
	Undefined	101	95	56,9%	72	43,1%
Find Bugs	`ArrayList`	150	83	24,5%	256	75,5%
	`LinkedList`	71	8	7,3%	102	92,7%
	Undefined	75	64	57,7%	47	42,3%
Jedit	`ArrayList`	93	77	39,9%	116	60,1%
	`LinkedList`	26	1	2,6%	38	97,4%
	Undefined	31	32	52,5%	29	47,5%
Total	ArrayList	958	446	19,3%	1854	80,7%
	LinkedList	134	38	13,5%	243	86,5%
	Undefined	303	345	64,1%	193	35,9%

```
1  public class Test {
2  ...
3  List fSelectionListeners=CollectionsFactory.current().createList();
4  ...
5  }
6
7  public abstract class CollectionsFactory {
8  ...
9     public abstract List createList(Collection initList);
10 ...
11 }
```

Código Fonte 3.1: Situação em que não foi possível a distinção de qual implementação de coleção foi usada

3.2.2 Resultados da implementação de coleção `Vector` e das variáveis Sincronizadas

Os resultados do uso de implementações sincronizadas está representado na Tabela 3.3. A coluna *Synchronized* representa a taxa percentual do total de variáveis de cada tipo de instância de coleção que está inserida em blocos ou métodos sincronizados. Para esta análise foi utilizada uma métrica que verifica se a coleção instanciada está inserida em um bloco ou método sincronizado.

3.2.3 Resultados do SIGN TEST

A Tabela 3.4 representa os resultados do *SIGN-TEST* para verificar em que parte do intervalo representado pela Figura 3.1 os valores computados como contexto para `ArrayList` e `LinkedList` se enquadravam. Realizamos o teste com 95% de confiança.

Para o teste de verificação se o valor do contexto é válido tanto para implementação do `ArrayList` como para a do `LinkedList`, temos as seguintes proposições:

1. $P_0 : CONTEXT = 0.5$

Tabela 3.3: Resultados da análise da aplicação das métricas para as coleções instanciadas como `Vector`

	Tipo	**Total de variáveis Instanciadas**	***Synchronized*** (% do Total)
Apache Ant	`ArrayList`	79	6,3%
	`LinkedList`	11	9%
	Undefined	14	35,7%
	`Vector`	54	11%
ArgoUML	`ArrayList`	256	8,6%
	`LinkedList`	13	7,6%
	Undefined	82	1,2%
	`Vector`	2	0%
AspectJ	`ArrayList`	380	2,6%
	`LinkedList`	13	7,6%
	Undefined	101	0%
	`Vector`	3	0%
Find Bugs	`ArrayList`	150	2%
	`LinkedList`	71	0%
	Undefined	75	0%
	`Vector`	7	5,7%
JEdit	`ArrayList`	93	3,2%
	`LinkedList`	26	11,5%
	Undefined	31	16,1%
	`Vector`	33	9%

2. $P_A : CONTEXT \neq 0.5$

Para o teste verificação se o valor do contexto é válido para implementação do `ArrayList` ou para a do `LinkedList`, temos as seguintes proposições:

1. $P_0 : CONTEXT < 0.5$

2. $P_A : CONTEXT > 0.5$

O resultado *Não rejeitada* significa que a proposição nula (P_0) foi considerada e *Rejeitada* significa que a proposição alternativa (P_A) foi aceita e a proposição nula rejeitada.

Tabela 3.4: Resultados do SIGN-TEST

Tipo	**SIGN TEST P/ CONTEXT = 0.5**	**SIGN TEST P/ CONTEXT < 0.5**
Apache Ant		
ArrayList	Rejeitada	Não Rejeitada
LinkedList	Rejeitada	Não Rejeitada
ArgoUML		
ArrayList	Rejeitada	Não Rejeitada
LinkedList	Rejeitada	Não Rejeitada
AspectJ		
ArrayList	Rejeitada	Não Rejeitada
LinkedList	Não Rejeitada	-
Find Bugs		
ArrayList	Rejeitada	Não Rejeitada
LinkedList	Rejeitada	Não Rejeitada
Jedit		
ArrayList	Rejeitada	Não Rejeitada
LinkedList	Rejeitada	Não Rejeitada

3.3 Discussão

Este seção realiza uma discussão dos resultados obtidos e as proposições levantadas neste estudo.

3.3.1 Proposição: O elemento `ArrayList` foi utilizado de acordo com o comportamento esperado para sua utilização.

Em todos os projetos o uso de `ArrayList` aparece em maior proporção que as outras implementações consideradas. Esta parece ser a primeira opção do desenvolvedor dentre as implementações de coleções existentes, levantamos duas possíveis justificativas para a popularidade da coleção. Um dos motivos seria sua semelhança estrutural com os *Arrays* primitivos e outro possível motivo seria sua semelhança estrutural com a implementação `Vector` em situações que não precisam ser *thread-safe*.

Nos projetos analisados, a implementação `ArrayList` foi aplicada de forma inapropriada, como pode ser visto da Tabela 3.4 na qual, em todos os projetos, a proposição de que a fórmula do contexto tinha valor inferior a 0,5 não foi rejeitada. Em outras palavras, as instâncias das coleções `ArrayList` possuem mais chamadas a métodos com tempo amortizado do que chamadas a métodos constantes.

Um motivo para estes números pode ser a falta de análise prévia do cenário de uso da implementação `ArrayList`, sendo esta a mais popular dentre as implementações de coleções, os desenvolvedores estariam aplicando-a sem analisar se este seria o cenário mais apropriado. Outro motivo seria que, na maioria das vezes, o uso de listas tem apenas o objetivo de adição de elementos e, como essa operação tem tempo amortizado para a implementação `ArrayList`, o seu uso excessivo seria inapropriado. Diante dos resultados coletados, rejeitamos nossa proposição de estudo.

3.3.2 Proposição: O elemento `LinkedList` foi utilizado de acordo com o comportamento esperado para sua utilização.

A implementação de coleção `LinkedList` aparece em menores proporções que a implementação `ArrayList` nos projetos, porém parece ser usada com mais consideração por parte do desenvolvedor pois, na maioria dos projetos analisados a coleção foi aplicada de forma apropriada. Esse dado pode ser visto na Tabela 3.4 na qual a proposição de que a fórmula do contexto tinha valor inferior a 0,5 não foi rejeitada. Ou seja, as instâncias das coleções `LinkedList` possuíam mais chamadas a métodos com tempo constante do que

chamadas a métodos com performance linear. Sendo assim, aceitamos a nossa proposição de estudo.

3.3.3 Proposição: O uso de implementações sincronizadas foi realizado de acordo com o comportamento esperado para sua utilização

Na Tabela 3.3 podemos verificar a existência de blocos sincronizados em implementações de coleções do tipo `Vector` em três projetos, o que não é necessário já que a implementação é *thread-safe*. Poderia ser realizado nesses casos o refatoramento das instâncias do tipo `Vector` para outros tipos como `LinkedList` ou `ArrayList` dependendo do contexto em que está inserida.

Em alguns projetos a utilização da implementação de coleção `Vector` se classificou como a segunda mais usada, atrás apenas do `ArrayList`. Este resultado pode indicar a existência de casos em que `Vector` não seria a implementação mais indicada por não haver necessidade de acesso sincronizado no local. Implementações de coleções como `ArrayList` e `LinkedList` poderiam substituir os casos em que não é preciso utilizar uma implementação *thread-safe*.

3.4 Ameaças à validade

Dentre os tipos de ameaças à validade descritos em Wohlin et al. [6], abaixo selecionamos algumas ameaças à validade que podem afetar as conclusões do nosso estudo de caso.

1. Ameaça à validade interna:
   Instrumentation - Se o design do efeito causado pelos artefatos usados para a execução do experimento, como a forma da coleta de dados, documento a ser inspecionado, inspeção do documento, etc. for feito de forma ruim, o estudo é afetado negativamente.

   (a) Métricas: Se as métricas definidas para definir as proporções de boas e más aplicações das coleções não forem representativas deste comportamento, podemos ter uma ameaça à validade interna.

   (b) *Software*: O *software* criado, assim como qualquer *software*, esta suscetível a erros, o que pode vir a invalidar a coleta de dados realizada.

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

(d) Tipo *Undefined*: O tipo que representa situações em que não foi possível a distinção de qual implementação de coleção foi usada, pode invalidar as conclusões sobre cada projeto se as variáveis deste tipo forem de uma proporção suficiente a influenciar um resultado inverso ao encontrado para o `ArrayList` ou para o `LinkedList` no SIGN-TEST.

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.

3.5 Conclusões

Através do nosso estudo de caso, conseguimos analisar a aplicação de coleções que implementam a interface *List* do *Java Collections Framework* em cinco projetos *open source* em *Java* selecionados. Implementamos um programa que possui métricas para avaliar o contexto em que as coleções foram inseridas. Através da determinação do contexto, conseguimos determinar qual implementação de coleção é a mais apropriada para a situação encontrada.

Os resultados do estudo inferem que a existência de projetos que possuem formas inadequadas de aplicação de coleções, passíveis de transformação para uma melhor seleção das mesmas. Também nos infere a deficiência de conhecimento dos desenvolvedores sendo plausível a criação de um processo de auxílio à seleção das coleções e à utilização das mesmas exibido no Capítulo 4.

Como trabalhos futuros pretendemos estudar uma forma de minimizar a quantidade de coleções do tipo *Undefined* encontradas visando ter mais confiança dos resultados encontrados. Também pretendemos aplicar as métricas em uma amostra maior de projetos *open*

source, incluir novos tipos de coleções no estudo, e consequentemente novos tipos de métricas para avaliá-las. Novos estudos possíveis seria a implementação de um *plugin* para sugerir ao desenvolvedor que implementação de coleção utilizar no contexto do trecho de código desenvolvido e ainda refatorar de forma automática aquele trecho de código para o desenvolvedor.

Capítulo 4

Processo de Adaptação: Abordagem e a Ferramenta

Neste capítulo, apresenta-se o processo adotado para a adaptação e transformação de programas clientes das coleções. Todas as fases do processo que serão descritas nas seções seguintes, considera-se o usuário do processo como sendo o desenvolvedor ao codificar.

4.1 Seleção e inclusão de uma coleção

Como citado no Capítulo 1, é um do objetivos deste trabalho permitir que os desenvolvedores definam as características da coleção que eles desejam utilizar, sem se preocupar com a documentação do JCF nem com as especificações associadas a cada tipo de estrutura de dados.

O processo de transformação de uma referência a uma instância de coleção a partir de suas características pode acontecer em dois cenários: no primeiro cenário, o usuário ainda não usa a coleção instanciada e irá incluí-la; no segundo cenário, a instanciação da coleção já existe e o usuário deseja alterá-la para acrescentar novos comportamentos.

O cenário de inclusão de uma coleção é uma situação mais simples porque não envolve alteração de uma estrutura de código e sim uma adição na estrutura. Dado que a coleção ainda não existe, os trechos de código existentes ainda não são dependentes de sua implementação. O desenvolvedor deve analisar o contexto que a nova coleção irá representar para que a melhor seja selecionada. Para um desenvolvedor que conhece bem a documentação do

Java Collections ou que não tem dificuldade de consultar a sua documentação, a atividade de analisar o contexto e inserir a coleção mais apropriada deveria ser corriqueiro. Poderíamos relacionar a dificuldade em selecionar a coleção com desenvolvedores mais novos, sem muita experiência. Porém, o estudo do Capítulo 3 em projetos *open source* que nos mostrou que, mesmo considerando diferentes níveis de experiência em desenvolvimento, as coleções que seriam as mais apropriadas não foram selecionadas para o contexto analisado.

4.1.1 Definir pela Árvore de Decisão a coleção mais apropriada

Supondo que essa dificuldade de seleção apropriada do contexto da coleção está relacionada com a falta de conhecimento da documentação ou a falta de tempo de consultá-la, uma solução é elevar o nível de abstração das características de cada coleção para o usuário e agregá-las em uma forma rápida de definição das características desejadas. Dessa forma, o usuário não precisaria se preocupar com a documentação do JCF nem com as especificações associadas a cada tipo de estrutura de dados.

Levantamos, então, características das coleções presentes no *Java Collections* e montamos uma **árvore de decisão**(AD). Cada nó da árvore de decisão possui uma característica de uma ou mais coleções, com ramos rotulados com afirmação ou negação da característica do nó e as folhas sendo as coleções resultantes da aceitação ou negação das características dos nós no caminho seguido. A Figura 4.1 mostra um ramo da árvore de decisão que define qual a interface selecionada para a coleção a partir de características predefinidas. Cada folha desta árvore possui subárvores para definição das implementações das coleções de cada interface a partir das características disponibilizadas.

O usuário define as características desejadas. Consideramos o usuário no processo como sendo o desenvolvedor durante a mudança de um sistema em nível de código. No caso da Figura 4.1, se o usuário selecionar para sua coleção as características *'Pode adicionar elementos repetidos na coleção'* e, sequencialmente, *'Lista de elementos'* a árvore de decisão determina que a coleção mais apropriada faz parte do conjunto de coleções que implementam a interface *List* e então parte para a subárvore que representa a decisão para as implementações dessa interface. Este processo pode ser feito através da seleção das características por níveis da árvore, como realizamos na ferramenta exibida na Seção 4.5.

No caso da inclusão de uma nova instância de coleção, o processo é finalizado sem a

Figura 4.1: Árvore de decisão

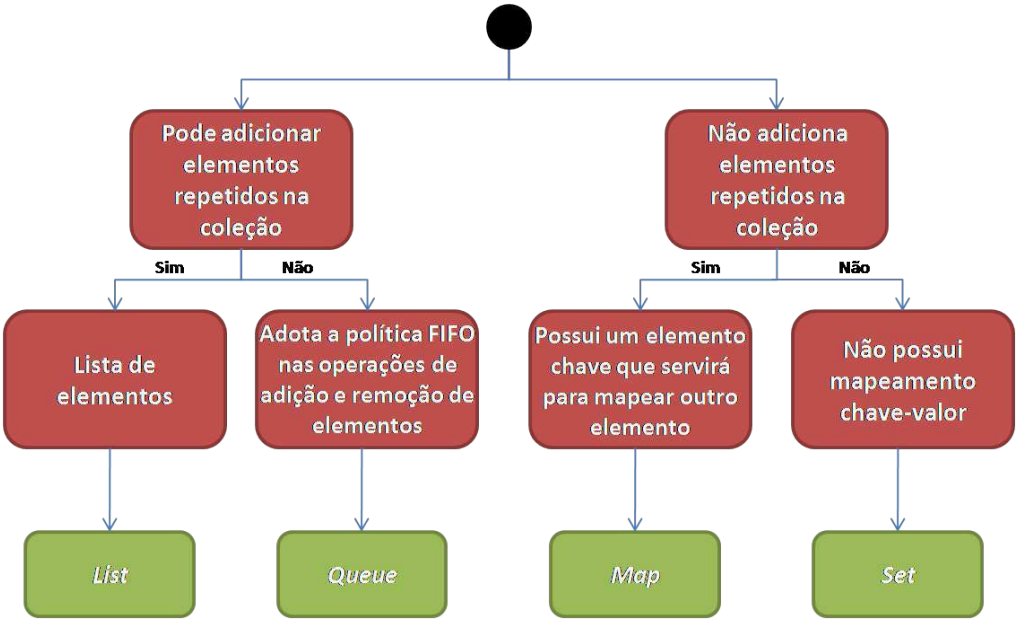


Figura 4.2: Diagrama do processo de adaptação

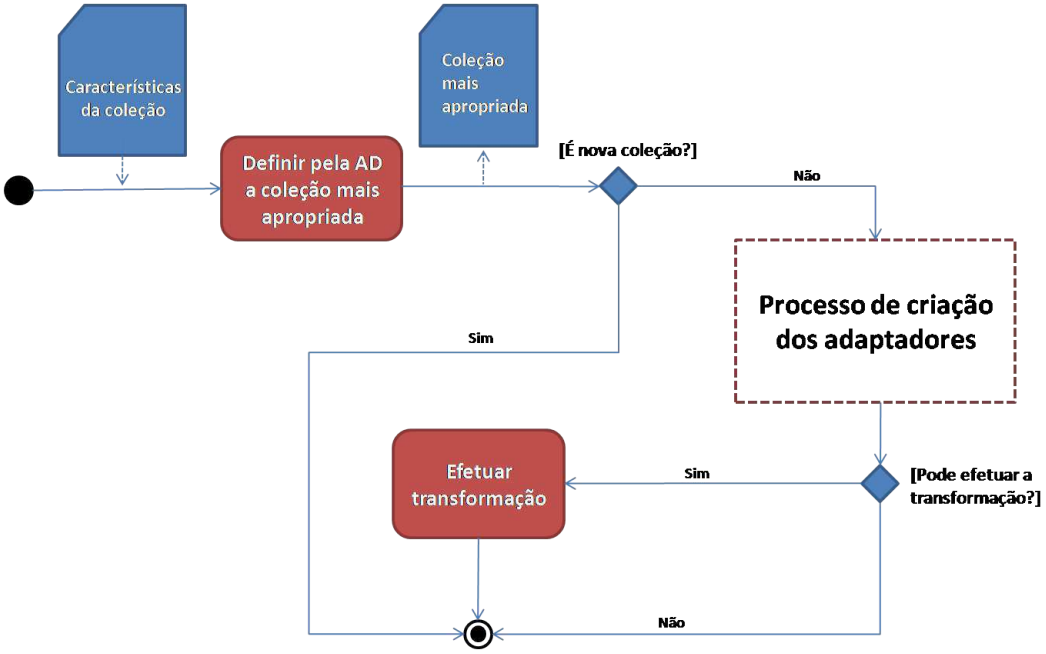
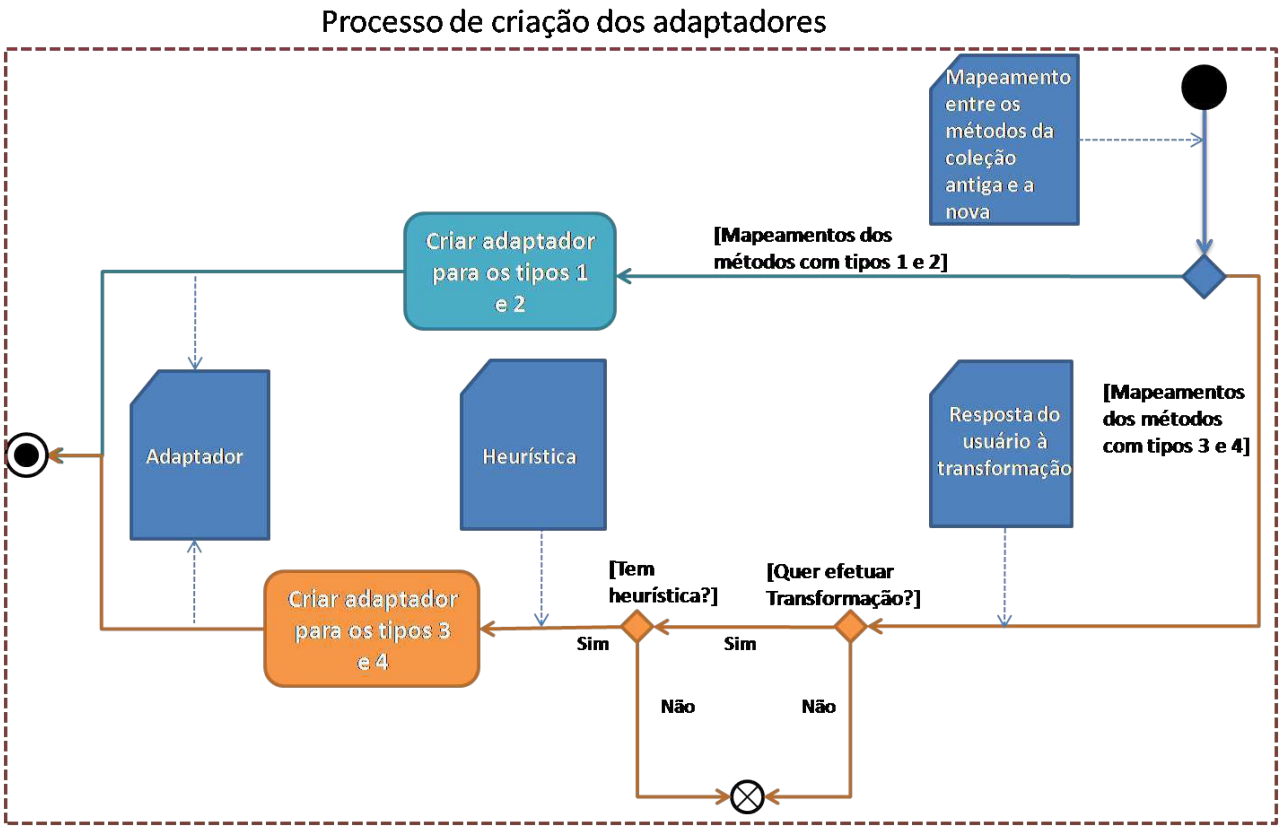


Figura 4.3: Diagrama do processo de criação dos adaptadores



necessidade da existência de adaptadores, considerando que a partir das características o usuário selecionou a melhor coleção que pode ser incluída no cenário. Seguindo o diagrama da Figura 4.2, a inclusão passa pela fase '*Usuário solicita nova coleção e sistema requisita as suas características*', na qual o usuário define as características que ele gostaria que estivesse presente na coleção, e pela fase *'Definição pela AD'* na qual o sistema define qual a melhor coleção a partir das características dadas pelo usuário, chegando ao final do fluxo.

4.2 Processo de criação dos adaptadores

Assumindo que o usuário quer modificar uma referência de coleção já instanciada, após a definição de como selecionaremos a coleção mais apropriada pela árvore de decisão, iremos determinar como realizamos a associação entre os elementos das coleções envolvidas para criação dos adaptadores e como é feita a criação e transformação de uma antiga instância de uma coleção para uma nova.

4.2.1 Mapeamento entre os métodos da coleção antiga e a nova

Os trabalhos de migração de API mostram diversas formas de efetuar a transformação do código cliente para a nova API. No trabalho de Nitas e Notkin [2], por exemplo, como se trata de uma migração genérica de qualquer API, ficou a cargo do usuário que vai efetuar a transformação a definição do mapeamento entre o código antigo que será substituído e o código novo que irá substituí-lo. No nosso caso, estamos tratando de um conjunto finito de opções (conjunto de coleções do JCF), assim não deixamos a cargo do usuário a definição do mapeamento da transformação. Criamos o mapeamento prévio entre duas coleções envolvidas na transformação: a coleção que está em uso e a coleção selecionada pelo usuário para substituí-la. Como previamente não temos como saber qual as coleções selecionadas para efetuar a transformação, o ideal seria a realização do mapeamento duas a duas de todas as coleções existentes do JCF. Porém, realizamos o mapeamento de um subconjunto de coleções do JCF que será especificado no experimento.

O mapeamento associa os elementos da coleção já instanciada com o elemento da coleção selecionada pela árvore de decisão. Por exemplo, quando se trata dos construtores das classes que representam as coleções envolvidas na transformação, o mapeamento con-

tém uma associação entre cada construtor da coleção atual com um construtor da coleção futura. Da mesma forma, quando se trata de métodos, seus parâmetros, modificadores e tipo de retorno, serão mapeados através de uma associação entre todos os elementos do método da coleção atual com um método da coleção futura. Um exemplo de mapeamento pode ser visto no Apêndice B e a explicação dos seus elementos é realizada na Seção 4.5.

4.3 Substituição da coleção com os adaptadores

Utilizando a abordagem com a utilização de adaptadores temos uma classificação das adaptações de uma coleção, por ordem de complexidade, considerando a sintaxe e a semântica dos métodos a serem adaptados.

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

- Tipo 1: $m_1() =_{synt} m_2()$ & $m_1() =_{sem} m_2()$;
- Tipo 2: $m_1() \neq_{synt} m_2()$ & $m_1() =_{sem} m_2()$;
- Tipo 3: $m_1() =_{synt} m_2()$ & $m_1() \neq_{sem} m_2()$;
- Tipo 4: $m_1() \neq_{synt} m_2()$ & $m_1() \neq_{sem} m_2()$;

Considera-se que métodos sintaticamente iguais ($=_{synt}$) possuem a mesma assinatura que inclui mesmos parâmetros de entrada e de retorno, mesma visibilidade e mesmo modificadores. Por exemplo, o método `add` tanto do `ArrayList`, presente no Código Fonte 4.1, quanto o método `add` do `LinkedList`, no Código Fonte 4.2, possuem a assinatura iguais, portanto consideramos como sendo sintaticamente iguais. Já o método `add` do `Vector`, exibido no Código Fonte 4.3, não é considerado sintaticamente igual já que tem o modificador que o torna *thread-safe*, diferentemente dos outros métodos `add`.

```
1 public boolean add(E e) {..}
```

Código Fonte 4.1: Método `add` do `ArrayList`

```
1 public boolean add(E e) {...}
```

Código Fonte 4.2: Método `add` do `LinkedList`

```
1 public synchronized boolean add(E e) {...}
```

Código Fonte 4.3: Método `add` do `Vector`

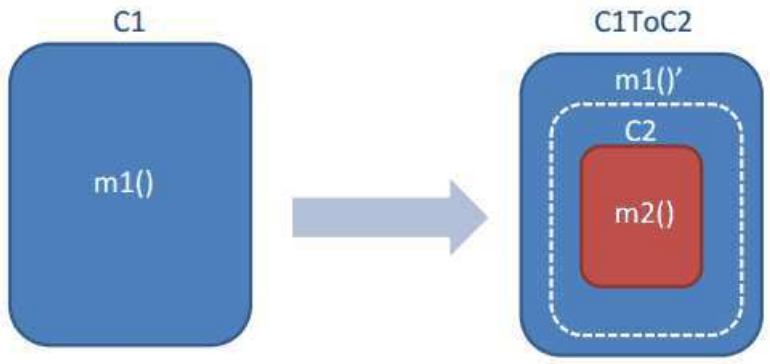
Os métodos semanticamente iguais ($=_{sem}$) possuem o mesmo efeito sobre a coleção independentemente da sintaxe envolvida. É o caso do método `add` tanto do `ArrayList` e do método `add` do `LinkedList`, presentes nos Código Fonte 4.1 e no Código Fonte 4.2 respectivamente, nos quais o efeito sobre a coleção é adicionar um elemento no final de uma coleção, podendo ser um elemento já existente, permitindo assim multiplicidade. Diferentemente do método `add` da coleção `HashSet`, presente no Código Fonte 4.4, que apesar de ter a mesma sintaxe do método `add` do `LinkedList` e `ArrayList`, possui um efeito diferente sobre a coleção não permitindo duplicidade de elementos.

```
1 public boolean add(E e) {...}
```

Código Fonte 4.4: Método `add` do `HashSet`

Dentro de uma adaptação entre duas coleções, pode existir adaptação de métodos dos quatro tipos citados. Cada caso exige uma abordagem diferente. A adaptação está representada na Figura 4.4, onde C_1ToC_2 é o adaptador da coleção C_1 para uma nova coleção C_2 e $m_1()'$ o método de C_1ToC_2 que adaptará o método $m_1()$ de C_1 para o método $m_2()$ de C_2.

Figura 4.4: Adaptador entre métodos do tipo 1



Para o desenvolvedor que solicita uma mudança, se ela inclui uma mudança semântica ele precisa estar ciente das consequências dessa mudança semântica. Por exemplo, uma adição

que antes da mudança verificava a existência de um elemento na instância para não permitir elementos repetidos, agora pode não ter mais esta verificação. Por este motivo, deixamos claro, antes de completar a transformação de uma instância para outra, as consequências da realização daquela transformação. A partir dessas informações o desenvolvedor decidirá se continua com a transformação ou não.

4.3.1 Tipo 1: Métodos sintática e semanticamente iguais

No Tipo 1, podemos fazer uma delegação direta, ou seja, apenas redirecionar a chamada do método já que os métodos possuem a mesma sintaxe e a mesma semântica. Por exemplo, no adaptador entre as coleções `ArrayList` (C_1) e `LinkedList` (C_2), nomeado de `ArrayListToLinkedList` ($C_1 To C_2$), existe um método `add` ($m_1()'$) com a mesma assinatura do mesmo método do `ArrayList` ($m_1()$). A implementação do método `add` do adaptador está representado no Código Fonte 4.5.

```
1 //...
2 public ArrayListToLinkedList(){
3    container=new java.util.LinkedList<E>();
4 }
5 //...
6 @java.lang.Override
7 public boolean add(E arg0) {
8    return container.add(arg0);
9 }
```

Código Fonte 4.5: Método `add` do adaptador `ArrayListToLinkedList`

4.3.2 Tipo 2: Métodos sintaticamente diferentes e semanticamente iguais

No Tipo 2, é necessário que a sintaxe do método do adaptador coleção se adapte de uma forma que ela se mantenha a mesma da coleção que ele substituirá. Por exemplo, no caso da adaptação de uma coleção `LinkedList` para uma coleção `ArrayList`, o método `addLast` do `LinkedList`, visto no Código Fonte 4.6, possui mesma semântica do mé-

todo `add` do `ArrayList`, visto no Código Fonte 4.1: ambos adicionam o novo elemento no final da lista. Porém, os métodos possuem sintaxes diferentes: mesmos parâmetros de entrada e de retorno, mesma visibilidade e mesmo *modifiers* mas assinaturas diferentes. O trecho do adaptador referente a adaptação de sintaxe, preservando a semântica, está exibido no Código Fonte 4.7.

```
1
2 public void addLast(E e) {...}
```

Código Fonte 4.6: Método `addLast` do `LinkedList`

```
1 //...
2 public LinkedListToArrayList(){
3    container=new java.util.ArrayList<E>();
4 }
5 //...
6 @java.lang.Override
7 public void addLast(E arg0) {
8    container.add(size(),arg0);
9 }
```

Código Fonte 4.7: Método `addLast` do adaptador `LinkedListToArrayList`

4.3.3 Tipo 3: Métodos sintaticamente iguais e semanticamente diferentes

No tipo 3, verificamos um impedimento com a possível adaptação. Em outras palavras, ou a adaptação será realizada com alteração na semântica ou não será possível realizá-la. Um exemplo de adaptação desse tipo está presente na transformação do método `iterator` da coleção `ArrayList` para o método `iterator` da coleção `HashSet`. Como podemos ver no Código Fonte 4.8 e no Código Fonte 4.9, os métodos possuem a mesma sintaxe. Porém, eles possuem semântica diferente se nos atentarmos ao fato de que o `ArrayList` itera sobre a coleção pela ordem de inserção e o iterador do `HashSet` não possui ordem pré-definida de iteração.

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

Código Fonte 4.8: Método `iterator` do `ArrayList`

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

Código Fonte 4.9: Método `iterator` do `HashSet`

O exemplo de uma possível adaptação para essa situação, considerando que haverá alteração semântica, está exemplificado no Código Fonte 4.10: foi considerado que o método `iterator` do `ArrayList` se comportará como o método `iterator` do `HashSet` no adaptador. Porém, existem outras opções possíveis como impedir a transformação em casos que acontece uma mudança semântica.

```
1 //...
2 public ArrayListToHashSet(){
3    container=new java.util.HashSet<E>();
4 }
5 //...
6 @java.lang.Override
7 public java.util.Iterator<E> iterator() {
8    return container.iterator();
9 }
```

Código Fonte 4.10: Método `iterator` do adaptador `ArrayListToHashSet`

4.3.4 Tipo 4: Métodos sintática e semanticamente diferentes

Consideramos um impedimento com possível adaptação também no tipo 4. Neste caso, apesar das diferenças, assumimos que exista algum nível de relacionamento semântico entre os métodos que torna possível adaptá-los. Na adaptação entre uma coleção `HashMap` e um `ArrayList`, por exemplo, temos os métodos `put` e `add` das respectivas coleções que possuem um certo nível de relacionamento semântico se consideramos a ação de acrescentar um objeto à coleção presente em ambas. Porém, se considerarmos que o `HashMap`, por ser um mapa, necessita da adição de uma chave associada ao objeto, comportamento este não existente no `ArrayList`, pode-se dizer que não existe nenhum nível de relacionamento

semântico entre os métodos.

Existem duas soluções neste caso: considerar que não existe nenhuma relação semântica entre os métodos e impossibilitar a adaptação entre esses tipos de coleções; considerar que existe uma relação e que a adaptação será realizada porém não mantendo a semântica da coleção anterior, como no exemplo representado pelo `HashMap`, podendo haver até perda de dados. No Código Fonte 4.11, podemos verificar que a criação da adaptação o método `put` perde o conceito de mapeamento e não possui mais chaves, apenas os valores.

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

Código Fonte 4.11: Método `put` do `HashMap`

```
1 //...
2 public HashMapToArrayList(){
3    container=new java.util.ArrayList<E>();
4 }
5 //...
6 public V put(K key, V value) {
7    return container.add(value);
8 }
```

Código Fonte 4.12: Método `put` do adaptador `HashMapToArrayList`

Tanto a adaptação do tipo 3 como do tipo 4 possuem consequências decisivas para a realização ou não da adaptação entre as coleções. A decisão é atribuída ao desenvolvedor criador do mapeamento, pois no momento de criação ele deve decidir se inclui uma implementação possível, mesmo com perda de dados e de semântica, ou não, como a exibida nos Códigos Fontes 4.10 e 4.12. A essa possível implementação, demos o nome de heurística. Também é decisão do usuário que está solicitando a transformação da instância da coleção, pois são exibidas para ele as possíveis consequências da transformação, como a perda da semântica.

A interface `Collection` tem `E` de *"element"* como seu *"generics"*, o que torna as classes de coleções que a implementam classes parametrizáveis. Isto permite que, no momento da declaração da variável que instância a coleção, os tipos dos objetos que a compõem sejam definidos previamente evitando erros de `ClassCastException` em tempo de execução: uma coleção definida para ter apenas objetos do tipo `Integer` não pode inserir uma

`String`. Os adaptadores criados também são classes parametrizáveis, é possível utilizar *generics* ou não.

4.4 Criação dos Adaptadores

Os adaptadores são criados a partir do mapeamento da classe origem, sendo a representação de coleção a ser substituída, e a classe destino, sendo a representação da coleção selecionada para substituir a antiga representação.

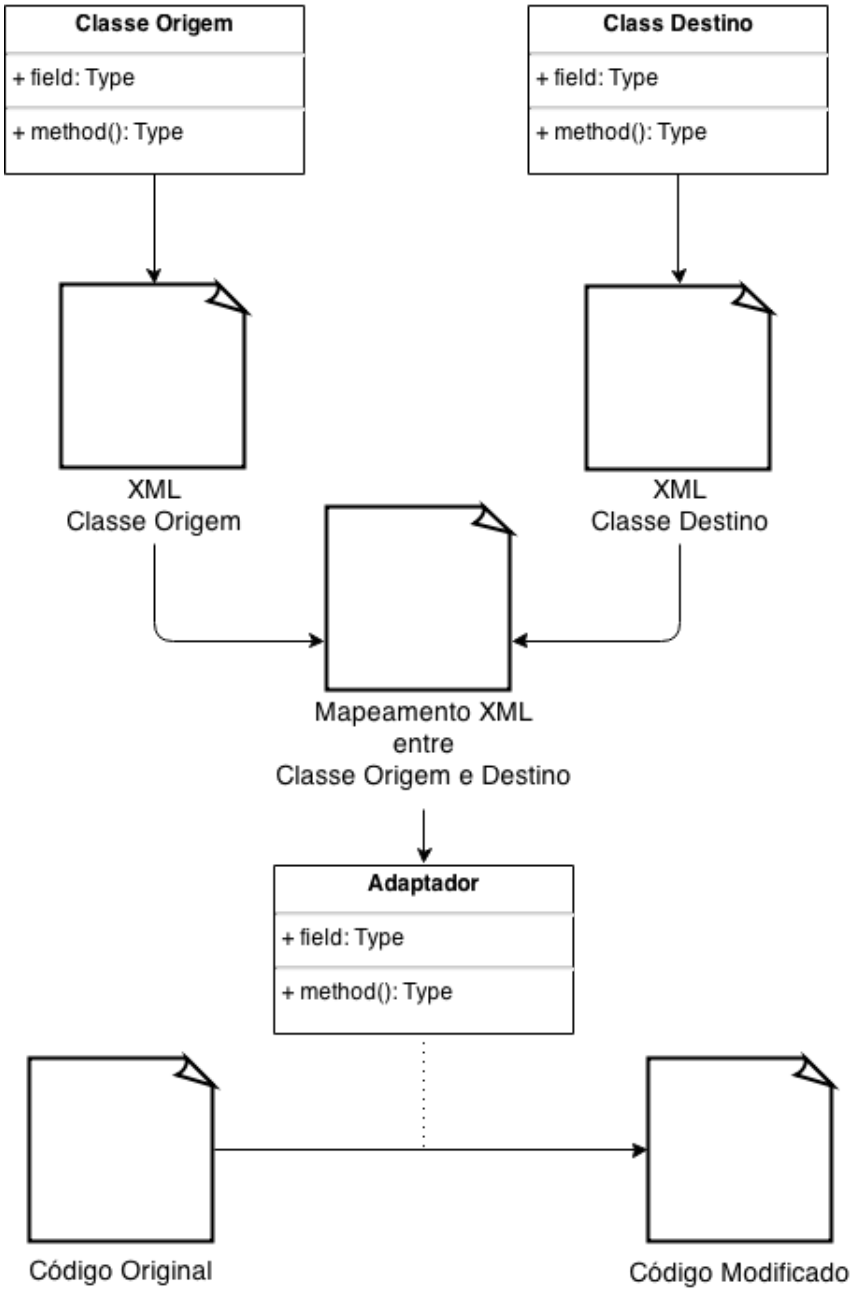
A criação dos adaptadores a partir dos mapeamentos está exemplificada na Figura 4.3 e 4.5. Dependendo do tipo de mapeamento definido no parâmetro tipo do arquivo XML, a criação dos adaptadores possui dois fluxos possíveis.

Seguindo o *Processo de criação dos adaptadores* da Figura 4.3, o primeiro fluxo é destinado para o mapeamento dos métodos e construtores que não possuem alteração semântica, ou seja, os que se classificam dentro do tipo 1 e 2. A adaptação desses dois tipos é considerada mais simples, cria-se um adaptador no qual os métodos classificados como 1 ou 2 mantém a sua sintaxe da coleção de origem, e como a semântica é a mesma, basta seguir o comportamento de uma das duas coleções envolvidas. O segundo fluxo envolve os métodos e construtores que possuirão mudança semântica ao realizar a adaptação. A decisão para a realização da transformação fica a cargo do usuário, é exiba para ele uma janela com as consequências da realização daquela transformação, a partir delas ele decide se continua com a transformação ou não. Caso aceite, a transformação é realizada com ajuda da heurística definida para aquele método e, enfim, o adaptador é criado.

4.5 Efetuar a transformação

Após o mapeamento entre as coleções e a criação do adaptador a partir deles, a próxima etapa é a inclusão do adaptador no código fonte. As únicas alterações necessárias são: inclusão do adaptador no local onde a variável representando a coleção é instanciada e adição da importação relativa à classe que representa o adaptador. A inclusão acontece como mostrado nos Código Fontes 4.13 e 4.14: a lista nomeada como *objects* anteriormente era um `LinkedList` e, a partir da ferramenta implementada, a coleção mais apropriada para as

Figura 4.5: Mapeamento, criação dos adaptadores e transformação do código



características escolhidas foi a `ArrayList`. Sendo assim, a lista foi substituída pelo adaptador `LinkedListToArrayList`.

```
1 private List objects = new LinkedList();
```

Código Fonte 4.13: Instância da coleção `LinkedList` a ser adaptada

```
1 private LinkedListToArrayList objects = new LinkedListToArrayList()
```

Código Fonte 4.14: Adaptação da instância da coleção `LinkedList` para `LinkedListToArrayList`

4.6 Ferramenta de Suporte

Nesta seção, explicamos como se deu a implementação da ferramenta baseada no processo de adaptação explicado nas seções anteriores.

4.6.1 Definir pela árvore de decisão a coleção mais apropriada

A árvore de decisão explicada na Seção 4.1.1 é exibida para o usuário como um conjunto de janelas em níveis disponibilizando as suas características. O usuário marca a características da coleção desejada e passa para o próximo nível da árvore até encontrar a folha que a represente. Na Figura 4.6 é exibido um conjunto possível de janelas representantes da árvore de decisão.

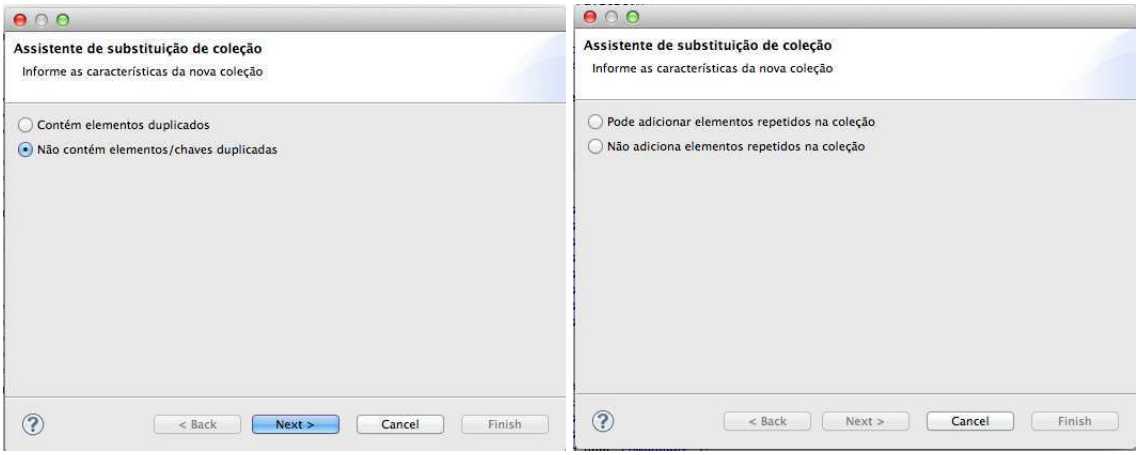


Figura 4.6: Janelas exibindo a implementação da árvore de decisão

4.6.2 Mapeamento entre os métodos da coleção antiga e a nova

A forma como foi implementado na ferramenta o processo de mapeamento explicitado na Seção 4.2 é explicado nos tópicos subsequentes.

Representação da coleção do JCF

A implementação do mapeamento foi feito em XML. Primeiramente, os elementos das classes que representam as coleções do JCF foram divididos em elementos e atributos no formato de arquivo XML, criados a partir de um extrator que, utilizando o código fonte da classe representante da coleção, cria um arquivo XML com tags que representam os seus elementos. Na Figura 4.5, os arquivos seriam o 'XML da Classe Origem' e o 'XML da Classe Destino', criados a partir das representações de coleções "Classe Origem"e "Classe Destino˜. Um arquivo completo representando a coleção `ArrayList` pode ser visto no Apêndice A.

Nos arquivos XML representando as coleções, o elemento `<class>` abrange os atributos `name` que representa o caminho da classe, `parameters` os parâmetros quando se faz o uso de *generics*, `super` que representa a classe que ela estende e `super-parameters` os parâmetros da classe que ela estende. Podemos ver estes elementos no Código Fonte 4.15 representando a assinatura da classe `ArrayList`. Ele ainda abrange os elementos filhos `<interfaces>`, `<constructors>` e `<methods>`.

```
1
2 <class name="java.util.ArrayList" parameters="E"
3    super="java.util.AbstractList" super-parameters="E">
4 ...
```

Código Fonte 4.15: Assinatura da classe `ArrayList`

No elemento `<interfaces>` existem os elementos filhos `<interface>` que representam cada uma das interfaces implementadas por aquela classe, possuindo os atributos `name` com o caminho da interface e `parameter` com o parâmetro no caso de uso de *generics*. Já o elemento filho de `<class>`, o `<constructors>`, contem elementos `<constructor>` que representam cada construtor que a classe possui. Este tem o atributo `id` que serve como índice representante do construtor para ser usado no mapeamento e o atributo `modifiers` que representa a visibilidade do construtor se `public`,

`private` ou `protected`. Dentro do elemento `<construtor>` ainda existem elementos filho `<parameter>`. Eles representam os parâmetros do construtor e tem os atributos `id` que funciona como índice para o mapeamento, `parameters` no caso do uso de *generics* e `type` que é o tipo do parâmetro. O elemento `<methods>` tem os elementos filho `<method>` representando cada método da classe. Cada `<method>` tem os atributos `id` como índice para o mapeamento método a método, `modifiers` representado a visibilidade, `name` representando a assinatura do método e o `return-type` como tipo de retorno do método. Como elementos filho do elemento `<method>` existem os elementos representando os parâmetros de cada método, `<parameter>`, com os atributos `id` para o mapeamento e `type` como tipo do parâmetro. No Código Fonte 4.16 podemos ver o xml com os elementos já descritos representando as interfaces, construtores e métodos da classe `ArrayList`.

Mapeamento entre as classes

Depois da representação das classes, foi feita a representação do mapeamento que serve de base para criação do adaptador. Acompanhando pela Figura 4.5, o mapeamento feito em XML é o 'Mapeamento XML entre Classe Origem e Destino' produto dos arquivos 'XML Classe Origem' e 'XML Classe Destino'. O arquivo completo de um mapeamento entre as coleções `ArrayList` e `LinkedList` pode ser visto no Apêndice B. No arquivo de mapeamento, o elemento `<map>` define as duas classes participantes do mapeamento através dos seus atributos `from` e `to`, no qual `from` representa a coleção que será transformada e o `to` a coleção que substituirá a coleção atual. O elemento `<map>` possui os elementos filhos `<constructors>`, `<methods>` e `<consequences>`. O elemento filho `<constructors>` contem os mapeamentos dos ids dos construtores da classe atual e a que irá substituí-la. Este mapeamento de construtores é representado pelo elemento `<constructor-mapping>` que possui os atributos referentes aos índices dos construtores das classes envolvidas, `fromId` e `toId`. Além desses dois atributos ainda existe o atributo type que representa o tipo de transformação explicado na Seção 4.3. No elemento filho `<methods>` do `<map>`, existem os elementos `<method-mapping>` que são o mapeamento entre os métodos das classes e também possuem os atributos `fromId`, `toId` e `type` com o mesmo significado do mapeamento dos construtores.

```
...
   <interfaces>
      <interface name="java.util.List" parameters="E" />
      <interface name="java.util.RandomAccess" />
      <interface name="java.lang.Cloneable" />
      <interface name="java.io.Serializable" />
   </interfaces>
   <constructors>
      <constructor id="0" modifiers="public">
         <parameter id="0" parameters="? extends E" type="
            java.util.Collection" />
      </constructor>
      <constructor id="1" modifiers="public" />
      <constructor id="2" modifiers="public">
         <parameter id="0" type="int" />
      </constructor>
   </constructors>
   <methods>
      <method id="0" modifiers="public" name="add" return-
         type="void">
         <parameter id="0" type="int" />
         <parameter id="1" type="E" />
      </method>
   ...
   </methods>
...
```

Código Fonte 4.16: Interfaces, construtores e métodos da classe `ArrayList`

Para os métodos com o `type` igual a 3 ou 4, associado a este mapeamento, pode existir uma heurística que ajude a adaptar a transformação entre aquelas coleções. As heurísticas vêm entre os elementos `<constructor-mapping>`, no caso dos construtores e

<method-mapping> no caso dos métodos. Por exemplo, no trecho do adaptador entre as coleções LinkedList e ArrayList visto no Código Fonte 4.17, o método addLast de LinkedList é mapeado para o método add do ArrayList, para isto é necessário a implementação do que definimos como heurística que seriam as linhas de código 2 e 3.

```
1 ...
2       <method-mapping fromId="3" toId="0" type="4">
3          container.add(size(),arg0);
4          return true;
5       </method-mapping>
6 ...
```

Código Fonte 4.17: Trecho do adaptador entre as coleções LinkedList e ArrayList

Em alguns mapeamentos, consideramos o toId como sendo -1 nos casos em que não existe nenhum método da coleção do to que seja mapeável. Por exemplo, em uma adaptação de ArrayList para LinkedList mostrado no trecho de Código Fonte 4.18, nos métodos trimToSize e ensureCapacity do ArrayList não existe método no LinkedList mapeável. Neste caso, acrescentamos um //TODO como forma de alerta ao usuário para posterior análise do uso deste método.

```
1 ...
2       <method-mapping fromId="20" toId="-1" type="4">
3       //TODO
4       </method-mapping>
5 ...
```

Código Fonte 4.18: Trecho do adaptador entre as coleções ArrayList e LinkedList

Como dito anteriormente, após o mapeamento o adaptador é criado e o código original é modificado para agora ter, no lugar da representação da coleção, o adaptador.

Capítulo 5

Estudo Experimental

Realizamos um experimento para comparar a ferramenta criada (*Collection Adapter*), seguindo o processo de seleção e adaptação das instâncias de coleções mostrados no Capítulo 4, com outras ferramentas. Neste capítulo, detalhamos o experimento para avaliação da ferramenta implementada bem como os resultados encontrados e a discussão sobre eles.

5.1 Planejamento do experimento

5.1.1 Seleção do contexto

Definimos o contexto do experimento de acordo com as quatro dimensões definidas por Wholin et al. [6]: *online* ou *offline*, estudante ou profissional, problema *toy* ou real, específico ou geral. Utilizamos, para realizar o experimento, projetos *open source* desenvolvidos em *Java* e que fazem uso do *Java Collections Framework*. O contexto no qual conduzimos o experimento é dito *offline* pois os projetos envolvidos foram configurados localmente para posterior alteração, não afetando sua execução em produção. Trata-se de projetos reais que foram alterados por alunos participantes do programa de monitoria das disciplinas de Estrutura de Dados e Algoritmos do curso de Ciência da Computação da Universidade Federal de Campina Grande. Podemos considerar o experimento como sendo aplicado para um contexto específico, o estudantil, porém, como já dito anteriormente, o problema da seleção incorreta de coleções está em nível geral no desenvolvimento de sistemas, seja os desenvolvedores estudantes ou profissionais.

5.1.2 Perguntas de pesquisa

A execução do experimento foi realizada para responder se a utilização da ferramenta *Collection Adapter* melhora a aplicação das coleções do JCF no código e se, comparada a uma ferramenta similar, a *Collection Adapter* auxilia melhor a substituição da instância da coleção.

Para responder à primeira pergunta, consideramos a melhora na aplicação das implementações de coleções como sendo a diminuição no tempo da análise e substituição de uma implementação por outra, o aumento na qualidade da escolha da implementação de coleção e a diminuição do esforço para realização da substituição. Neste caso, estamos considerando a 'melhora' comparando a realização das atividades de forma manual, sem ajuda de nenhuma ferramenta, com a realização das atividades utilizando a ferramenta *Collection Adapter*.

Para a segunda pergunta, precisávamos da comparação da nossa ferramenta com uma similar. A ferramenta *Collection Adapter* tem três funções principais: selecionar a melhor implementação de coleção a partir de suas características, criar o adaptador e substituir a implementação atual pela selecionada. Durante o planejamento do experimento, não encontramos nenhuma ferramenta que fosse similar à *Collection Adapter* nestas três funções. Optamos por selecionar uma ferramenta que auxilia na análise da implementação de coleção através da busca e, consequentemente, auxilia a localização dos locais onde realizar a substituição da implementação. Esta ferramenta é a *Java Search* [22] que é acoplada a IDE do Eclipse e permite localizar declarações, referências e ocorrências de elementos Java (pacotes, tipos, métodos, campos). Formulamos as seguintes perguntas de pesquisa para serem discutidas com os resultados obtidos:

1. A qualidade da substituição utilizando a ferramenta *Collection Adapter* é superior a encontrada na substituição manual?

2. O esforço da análise e substituição utilizando a ferramenta *Collection Adapter* é inferior ao encontrado na substituição manual?

3. O tempo utilizado na substituição utilizando a ferramenta *Collection Adapter* é inferior ao encontrado no manual?

4. O tempo utilizado na análise com a ferramenta *Collection Adapter* é inferior ao encontrado no manual?

5. A qualidade da substituição utilizando a ferramenta *Collection Adapter* é superior a encontrada na substituição com *Java Search*?

6. O esforço da análise e substituição utilizando a ferramenta *Collection Adapter* é inferior ao encontrado na substituição com *Java Search*?

7. O tempo utilizado na análise utilizando a ferramenta *Collection Adapter* é inferior ao encontrado com *Java Search*?

5.1.3 Seleção de variáveis

Para realização do estudo experimental realizamos a seleção das variáveis independentes e dependentes.

Independentes

As variáveis independentes são as variáveis que podemos controlar e alterar durante o experimento [6]. No experimento elas são as ferramentas *Collection Adapter*, implementada por nós, e *Java Search*. Além das duas ferramentas, incluímos nesse mesmo grupo a forma manual de realizar a análise e de substituir a instância da coleção. Portanto, esta variável possui três valores possíveis: Manual, *Java Search* e *Collection Adapter*.

Dependentes

Os efeitos dos tratamentos das variáveis independentes são medidos nas variáveis dependentes [6]. Tratamento é o conjunto de níveis dos fatores para um ensaio experimental, fatores são as variáveis independentes. No nosso caso as três ferramentas *Collection Adapter*, *Java Search* e o modo manual compõem os fatores do nosso tratamento que é a variável independente ferramentas. Já as variáveis dependentes são as seguintes:

- Qualidade da escolha da implementação de coleção (Q): é a determinação se o aluno selecionou a implementação de coleção que é a mais apropriada para o contexto determinado. Contexto este definido por métricas assim como realizado no Capítulo 3.

No final do experimento, classificamos a substituição realizada pelo aluno como *Interface Errada*, caso o aluno tenha selecionado a implementação que não era a esperada; *Mesma Interface* caso o aluno não tenha selecionado a implementação de coleção esperada porém possuindo a mesma interface da esperada; e *Exato* caso o aluno tenha selecionado exatamente a implementação esperada.

- Esforço (E): é o esforço de realizar a análise e a substituição considerado pelo aluno. O esforço pode ser *Muito Baixo, Baixo, Médio, Alto* ou *Muito Alto* e foi determinado através do preenchimento de um formulário pelos alunos.

- Tempo de análise (Ta): é o tempo que o aluno demorou para realizar a análise do contexto em que a coleção está inserida. Coletamos esse dado a partir de um vídeo gravado com todos os passos realizados pelo aluno durante o experimento. O tempo de análise foi contado a partir do momento que o aluno tem acesso a classe com a instância a ser analisada até o momento que inicia a substituição de uma coleção para outra.

- Tempo de substituição (Ts): é o tempo que o aluno demorou para realizar a substituição de uma coleção por outra. Também coletamos esse dado a partir de um vídeo gravado com todos os passos realizados pelo aluno durante o experimento. O tempo de substituição foi contado a partir do momento que o aluno inicia a substituição de uma coleção para outra até o momento que finaliza a substituição de forma a não conter erros de compilação no código.

5.1.4 Unidades experimentais e Sujeitos

Os tratamentos são aplicados às unidades experimentais que, no nosso experimento, são quatro classes do projeto **Argo UML**, ferramenta *open source* em Java para modelagem UML. Selecionamos as classes a partir dos resultados levantados no Estudo Exploratório do Capítulo 3, analisando a existência de uma variável instanciando uma coleção do *Java Collections Framework* que não seria a mais apropriada para o contexto que ela se encontra, como no exemplo da Figura 1.1.

Os sujeitos do nosso experimento são 12 alunos participantes do programa de monitoria

das disciplinas de Estrutura de Dados e Algoritmos do curso de Ciência da Computação da Universidade Federal de Campina Grande. Assumimos que as experiências dos alunos são iguais devido ao período que se encontram no curso.

5.1.5 Instrumentação

Para a execução e análise do nosso experimento foi planejada a necessidade das seguintes ferramentas:

- Eclipse: Utilizada para exibição das unidades experimentais, bem como para utilização das ferramentas *Java Search* e a *Collection Adapter* que foi implementada como *plugin* da IDE;
- JUnit: *Framework* utilizado para o aluno executar testes de unidade após finalizar os passos do experimento;
- *Microsoft Expression Encoder*: Utilizada para gravar a tela do computador com os movimentos que o aluno realizar;
- *Google Sites*: Utilizado para construir a página com o passo a passo para realização do experimento;
- *Google Docs*: Utilizado para construir as apresentações explicando as instruções para utilização das ferramentas envolvidas no experimento;
- *Google Forms*: Utilizado para construir o formulário a ser preenchido no final do experimento por cada aluno;
- *R Studio*: Utilizado para realizar a análise estatística dos dados.

5.1.6 *Design* de experimento

Dividimos o experimento em duas fases. Na primeira, os alunos têm a opção de não realizar a substituição caso achem que a coleção selecionada já está adequada ao contexto. Agrupamos os doze alunos em quatro grupos de três pessoas. Cada aluno presente em um grupo executou a substituição da coleção com uma ferramenta distinta: *Manual, Java Search* e

Collection Adapter. As três formas de substituição foram aplicadas para o mesmo código fonte conforme exibido na Tabela 5.1.

Na primeira fase, todos os alunos têm acesso a uma página exibindo os passos a serem seguidos no experimento. Para os alunos que utilizaram as ferramentas de *Collection Adapter* e *Java Search* exibiu-se uma apresentação sobre o funcionamento da ferramenta. Logo depois, foi apresentada ao aluno a classe e a instância de coleção que ele deveria analisar e substituir ou não por outra coleção. Neste momento, é iniciada a gravação da tela do aluno para capturar todas as alterações realizadas. Ao finalizar a análise, com ou sem substituição, o aluno preencheu um formulário no qual definiu o esforço realizado no processo.

Tabela 5.1: *Design* de um fator com mais de um tratamento para a fase 1 do experimento

Código fonte	**Alunos**	**Manual**	***Java Search***	***Collection Adapter***
C1	Grupo 1(A1, A2 e A3)	A1	A2	A3
C2	Grupo 2(A4, A5 e A6)	A4	A5	A6
C3	Grupo 3(A7, A8 e A9)	A7	A8	A9
C4	Grupo 4(A10, A11 e A12)	A10	A11	A12

Na segunda fase, os alunos que optaram por não realizar a substituição, por achar a coleção instanciada adequada ao contexto, não tem mais a opção de não transformar. Com um intervalo de dois meses entre as fases, os alunos entraram em contato com o mesmo código e ferramenta que utilizaram na primeira fase e substituíram a coleção por alguma outra. Nesta fase, os alunos podiam acessar as apresentações já exibidas na primeira fase, mas só se necessário, e precisavam preencher o formulário com o esforço considerado no final do processo.

5.2 Resultados

Nesta seção serão exibidos os resultados das duas fases do experimento exibindo os valores obtidos para cada variável independente.

5.2.1 Primeira fase do experimento

Estão exibidos na Figura 5.1 os resultados que mostram a porcentagem de alunos que decidiram alterar a implementação de coleção, afirmando que aquela não é apropriada à situação, e os que decidiram não alterar achando que é apropriada para o contexto encontrado.

Figura 5.1: Porcentagem de alunos que alteraram a coleção

Qualidade da alteração

Na Figura 5.2 estão exibidos os resultados referentes à porcentagem de alunos que selecionou a coleção que consideramos exata, a mesma interface da coleção que consideramos exata ou selecionou a coleção e interface consideradas erradas. A Tabela 5.2 exibe os alunos que efetuaram a substituição e a classificação da qualidade da substituição.

Figura 5.2: Qualidade da alteração das coleções pelos alunos

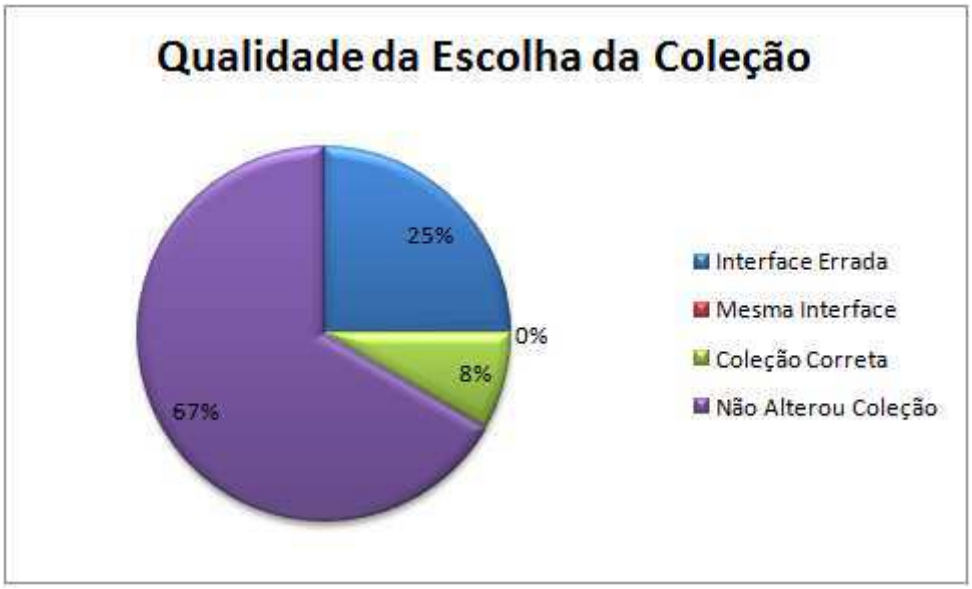


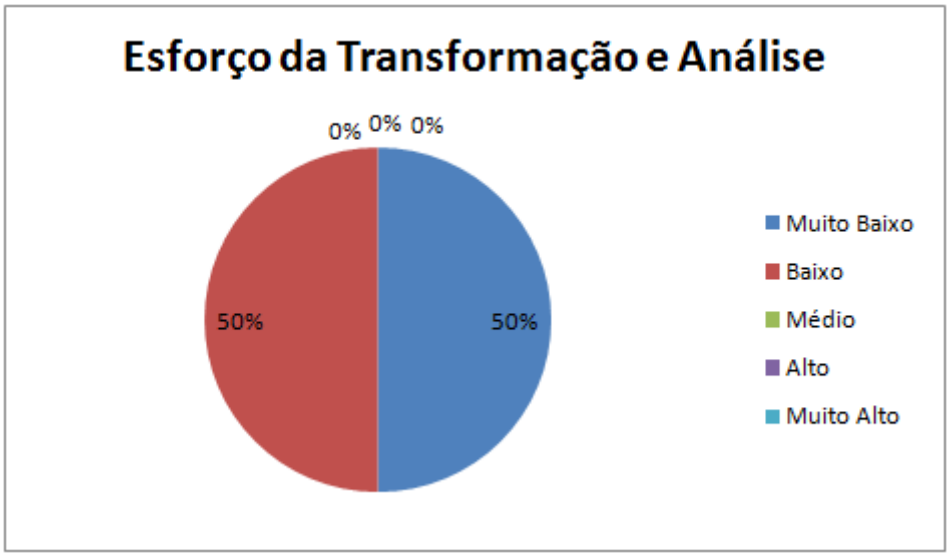
Tabela 5.2: Qualidade da alteração das coleções pelos alunos

Aluno	**Ferramenta**	**Qualidade da Escolha (3 - Exato; 2 - Mesma interface; 1 - Interface errada)**
A2	*Java Search*	1
A3	*Collection Adapter*	1
A6	*Collection Adapter*	3
A12	*Collection Adapter*	1

Esforço da análise e substituição

A Figura 5.3 exibe a classificação em porcentagem do esforço de análise e substituição dos alunos que efetuaram a substituição da coleção em: Muito Baixo, Baixo, Médio, Alto e Muito Alto.

Figura 5.3: Classificação do esforço da análise e substituição das coleções pelos alunos



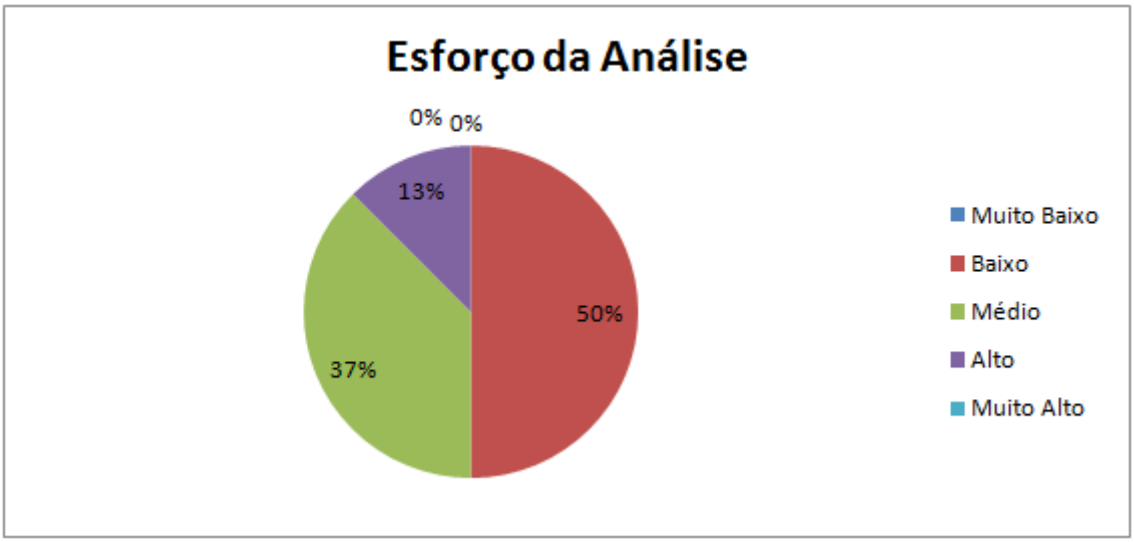
A Tabela 5.3 mostra a classificação do esforço dos alunos que efetuaram a substituição, ordenados pela ferramenta utilizada.

Tabela 5.3: Classificação do esforço da análise e substituição das coleções pelos alunos

Aluno	**Ferramenta**	**Esforço**
A2	*Java Search*	Baixo
A3	*Collection Adapter*	Baixo
A6	*Collection Adapter*	Muito Baixo
A12	*Collection Adapter*	Muito Baixo

A Figura 5.4 exibe a classificação em porcentagem do esforço de análise dos alunos que não efetuaram a substituição da coleção em: Muito Baixo, Baixo, Médio, Alto e Muito Alto.

Figura 5.4: Classificação do esforço da análise das coleções pelos alunos



A Tabela 5.4 mostra a classificação do esforço dos alunos que não efetuaram a substituição, ordenados pela ferramenta utilizada.

Tabela 5.4: Classificação do esforço da análise e substituição das coleções pelos alunos

Aluno	**Ferramenta**	**Esforço**
A9	*Collection Adapter*	Médio
A5	*Java Search*	Médio
A8	*Java Search*	Baixo
A11	*Java Search*	Baixo
A1	Manual	Baixo
A4	Manual	Baixo
A7	Manual	Médio
A10	Manual	Alto

Tempo de análise e substituição

As Tabelas 5.5 e 5.6 mostram o tempo utilizado pelos alunos para realizar a análise do contexto em que a coleção está inserida. A Tabela 5.5 exibe o tempo de análise dos alunos que não realizaram a substituição e a Tabela 5.6 o tempo dos que realizaram. Adicionalmente, a

Tabela 5.6 também exibe o tempo utilizado pelos alunos para realizar a substituição de uma coleção para outra.

Tabela 5.5: Tempo de análise dos alunos que não realizaram a substituição

Aluno	**Ferramenta**	**Tempo de análise**
A9	*Collection Adapter*	0:05:17
A5	*Java Search*	0:24:00
A8	*Java Search*	0:08:58
A11	*Java Search*	0:13:29
A1	Manual	0:23:58
A4	Manual	0:07:04
A7	Manual	0:10:26
A10	Manual	0:06:17

Tabela 5.6: Tempo de análise e substituição dos alunos que realizaram a substituição

Aluno	**Ferramenta**	**Tempo de análise**	**Tempo de substituição**
A3	*Collection Adapter*	0:03:09	0:02:11
A6	*Collection Adapter*	0:07:50	0:02:53
A12	*Collection Adapter*	0:13:12	0:01:07
A2	*Java Search*	0:13:06	0:00:15

5.2.2 Segunda fase do experimento

Os resultados dos alunos da primeira fase que realizaram a substituição foram agrupados com os resultados dos alunos da segunda fase. Desta forma, podemos realizar a comparação das substituições pelos grupos descritos na Tabela 5.1.

Qualidade da alteração

A Figura 5.5 exibe a classificação em porcentagem da qualidade da alteração das coleções pelos alunos que efetuaram a substituição da coleção na primeira e segunda fase e a Tabela 5.7

Figura 5.5: Qualidade da alteração das coleções pelos alunos em porcentagem

e a Figura 5.6 mostram a qualidade da alteração das coleções pelos alunos classificada por ferramentas.

Realizamos teste estatístico de comparação entre os valores encontrados para cada ferramenta, como a qualidade da escolha da implementação de coleção se trata de uma variável categórica ordinal utilizamos o teste de *Kruskal-Wallis*. Tínhamos as seguintes hipóteses nula e alternativa:

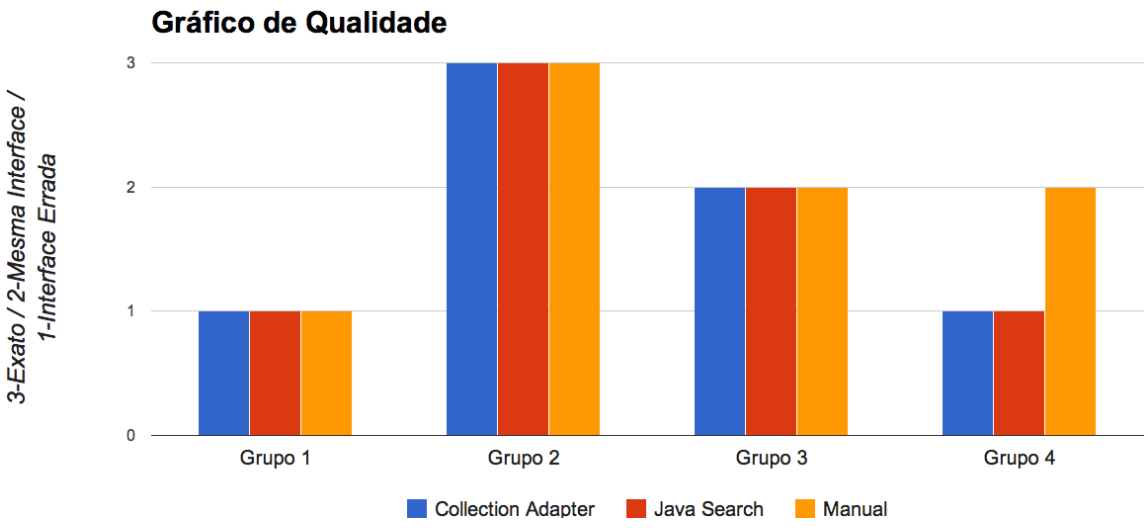
- H_0: Os valores de qualidade encontrados com a ferramenta *Collection Adapter* não diferem dos encontrados no *Java Search* e Manual.
- H_A: Os valores de qualidade encontrados com a ferramenta *Collection Adapter* diferem dos encontrados no *Java Search* e Manual.

Não rejeitamos a hipótese nula, como pode ser visto no teste exibido no Código Fonte 5.1 executados no R.

Tabela 5.7: Qualidade da alteração das coleções pelos alunos classificadas por ferramentas

Aluno	**Ferramenta**	**Qualidade da Escolha (3 - Exato; 2 - Mesma interface; 1 - Interface errada)**	**Grupo**
A3	*Collection Adapter*	1	Grupo 1
A6	*Collection Adapter*	3	Grupo 2
A9	*Collection Adapter*	2	Grupo 3
A12	*Collection Adapter*	1	Grupo 4
A2	*Java Search*	1	Grupo 1
A5	*Java Search*	3	Grupo 2
A8	*Java Search*	2	Grupo 3
A11	*Java Search*	1	Grupo 4
A1	Manual	1	Grupo 1
A4	Manual	3	Grupo 2
A7	Manual	2	Grupo 3
A10	Manual	2	Grupo 4

Figura 5.6: Qualidade da alteração das coleções pelos alunos



```
> collectionAdapter <- c(1,3,2,1)
> javaSearch <- c(1,3,2,1)
> manual <- c(1,3,2,2)
> qualidade <- data.frame(collectionAdapter, javaSearch, manual)
> qualidade <- stack(qualidade)
> kruskal.test(qualidade[,"values"]~qualidade[,"ind"], data=
   qualidade)

   Kruskal-Wallis rank sum test

data:  qualidade[, "values"] by qualidade[, "ind"]
Kruskal-Wallis chi-squared = 0.2946, df = 2,
p-value = 0.863
```

Código Fonte 5.1: Teste de Kruskal-Wallis para a variável de qualidade

Esforço da substituição

A Figura 5.7 exibe a classificação em porcentagem do esforço de análise e substituição dos alunos que efetuaram a substituição da coleção na primeira e segunda fase em: *Muito Baixo, Baixo, Médio, Alto* e *Muito Alto*.

A Tabela 5.8 e a Figura 5.8 mostram a classificação do esforço dos alunos que efetuaram a substituição, ordenados pela ferramenta utilizada nas duas fases do experimento.

Realizamos teste estatístico de comparação entre os valores encontrados para cada ferramenta, como o esforço de coleção se trata de uma variável categórica ordinal utilizamos o teste de *Kruskal-Wallis*. Tínhamos as seguintes hipóteses nula e alternativa:

- H_0: Os valores de esforço encontrados com a ferramenta *Collection Adapter* não difere dos encontrados no *Java Search* e Manual.
- H_A: Os valores de esforço encontrados com a ferramenta *Collection Adapter* difere dos encontrados no *Java Search* e Manual.

Não rejeitamos a hipótese nula, como pode ser visto no teste exibido no Código Fonte 5.2 executados no R.

Figura 5.7: Classificação do esforço da análise e substituição das coleções pelos alunos em porcentagem

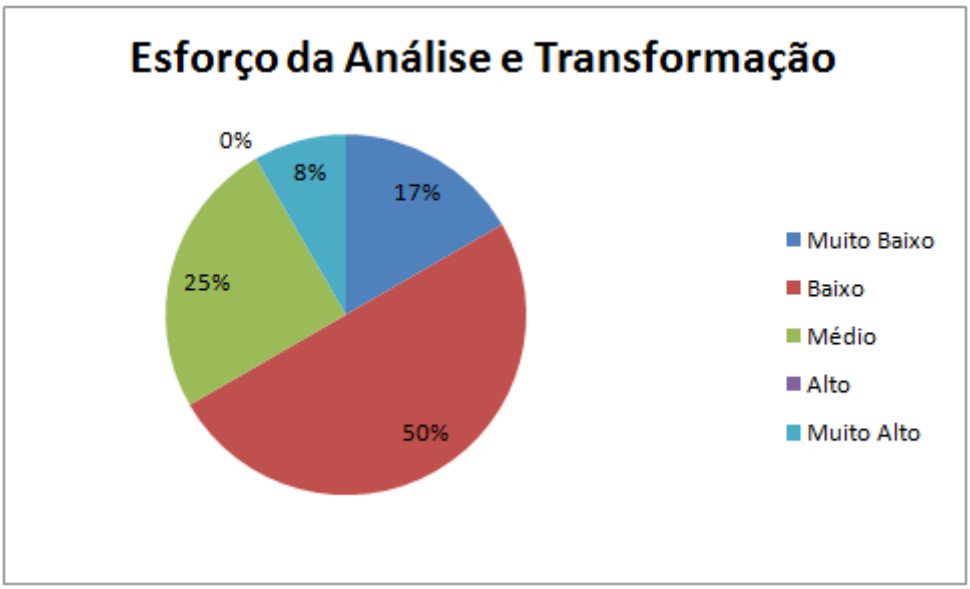


Figura 5.8: Classificação do esforço da análise e substituição das coleções pelos alunos

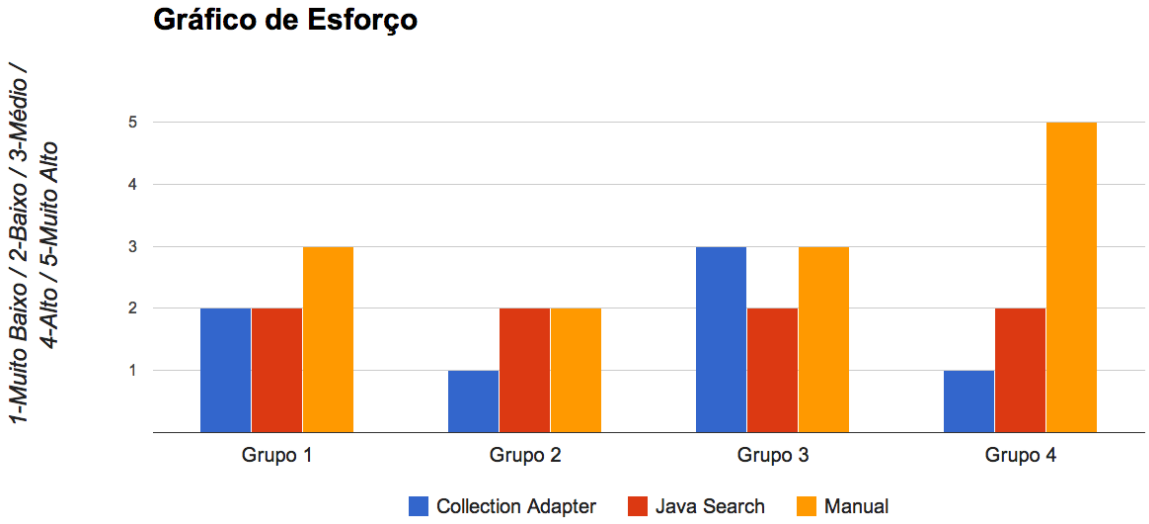


Tabela 5.8: Classificação do esforço da análise e substituição das coleções pelos alunos

Aluno	**Ferramenta**	**Esforço**	**Grupo**
A3	*Collection Adapter*	Baixo	Grupo 1
A6	*Collection Adapter*	Muito Baixo	Grupo 2
A9	*Collection Adapter*	Médio	Grupo 3
A12	*Collection Adapter*	Muito Baixo	Grupo 4
A2	*Java Search*	Baixo	Grupo 1
A5	*Java Search*	Baixo	Grupo 2
A8	*Java Search*	Baixo	Grupo 3
A11	*Java Search*	Baixo	Grupo 4
A1	Manual	Médio	Grupo 1
A4	Manual	Baixo	Grupo 2
A7	Manual	Médio	Grupo 3
A10	Manual	Muito Alto	Grupo 4

```
1  > collectionAdapter <- c(2,1,3,1)
2  > javaSearch <- c(2,2,2,2)
3  > manual <- c(3,2,3,5)
4  > esforco <- data.frame(collectionAdapter, javaSearch, manual)
5  > esforco <- stack(esforco)
6  > kruskal.test(esforco[,"values"]~esforco[,"ind"], data=esforco)
7
8     Kruskal-Wallis rank sum test
9
10 data:  esforco[, "values"] by esforco[, "ind"]
11 Kruskal-Wallis chi-squared = 4.5722, df = 2,
12 p-value = 0.1017
```

Código Fonte 5.2: Teste de Kruskal-Wallis para a variável de esforço

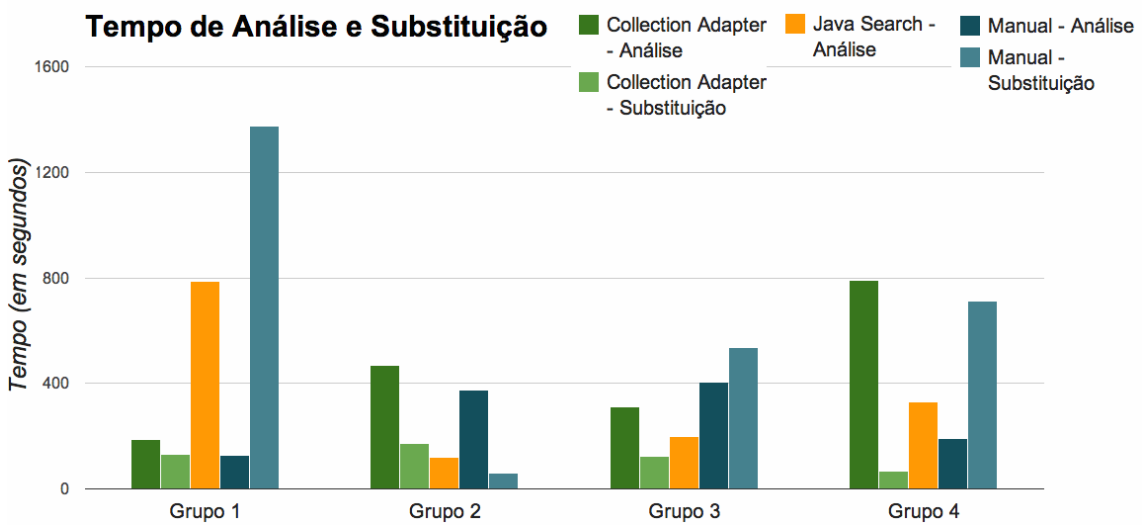
Tempo de análise e substituição

A Tabela 5.9 e a Figura 5.9 mostram o tempo utilizado pelos alunos para realizar a análise do contexto em que a coleção está inserida e também exibe o tempo utilizado pelos alunos para realizar a substituição de uma coleção para outra nas duas fases do experimento.

Tabela 5.9: Tempo de análise e substituição dos alunos

Aluno	**Ferramenta**	**Tempo de análise**	**Tempo de substituição**	**Grupo**
A3	*Collection Adapter*	00:03:09	00:02:11	Grupo 1
A6	*Collection Adapter*	00:07:50	00:02:53	Grupo 2
A9	*Collection Adapter*	00:05:10	00:02:02	Grupo 3
A12	*Collection Adapter*	00:13:12	00:01:07	Grupo 4
A2	*Java Search*	00:13:06	00:00:15	Grupo 1
A5	*Java Search*	00:02:00	00:04:00	Grupo 2
A8	*Java Search*	00:03:19	00:02:57	Grupo 3
A11	*Java Search*	00:05:30	00:10:48	Grupo 4
A1	Manual	00:02:09	00:22:56	Grupo 1
A4	Manual	00:06:15	00:01:00	Grupo 2
A7	Manual	00:06:45	00:08:56	Grupo 3
A10	Manual	00:03:12	00:11:53	Grupo 4

Figura 5.9: Classificação do tempo da análise e substituição das coleções pelos alunos



Realizamos testes estatísticos de normalidade, homocedasticidade e de comparação entre os valores de tempo encontrados para cada ferramenta. Sendo estes os testes de *Shapiro-Wilk*, *Levene* e *One-way Anova*, respectivamente.

Tanto para os tempos de análise como substituição, os dados se comportaram de forma normal e homogênea. Com relação ao teste de comparação entre os grupos, tínhamos as seguintes hipóteses nula e alternativa:

- H_0: A média de tempo encontrada com a ferramenta *Collection Adapter* não difere das médias dos tempos encontradas no *Java Search* e Manual.
- H_A: A média de tempo encontrada com a ferramenta *Collection Adapter* difere das médias dos tempos encontradas no *Java Search* e Manual.

Nos tempos de análise e substituição não rejeitamos a hipótese nula, como pode ser visto nos testes exibidos no Código Fonte 5.3 executados no R.

```
1  oneway.test(values~ind, data=temposDeAnalise, var.equal=F)
2
3     One-way analysis of means (not assuming equal variances)
4
5  data:  values and ind
6  F = 0.5989, num df = 2.000, denom df = 5.292, p-value = 0.5828
7
8  oneway.test(values~ind, data=temposDeSubstituicao, var.equal=F)
9
10    One-way analysis of means (not assuming equal variances)
11
12 data:  values and ind
13 F = 2.3951, num df = 2.000, denom df = 4.141, p-value = 0.2036
```

Código Fonte 5.3: Teste Anova para os tempos de análise e substituição

5.3 Discussão do Estudo Experimental

Neste seção, apresentamos uma discussão geral sobre os resultados obtidos neste estudo, bem como sugestões de trabalhos futuros que podem ser desenvolvidos a partir deste. Devido a pequena população dos dados, não realizamos teste estatístico, apenas discutimos os resultados obtidos de forma qualitativa.

5.3.1 Primeira fase do experimento

Durante o experimento podemos observar na Figura 5.1 que, na primeira fase, a maior proporção de alunos preferiu não alterar a coleção para outra. Levantamos possíveis motivos para essa decisão: um seria o costume dos desenvolvedores de não analisar as estruturas antes de usá-las, por comodidade ou falta de tempo se considerarmos o mundo real do desenvolvimento de *software*; outro motivo seria o pouco conhecimento sobre as estruturas que usam, tendo sempre uma que considera a aplicável a várias situações; adicionalmente, um motivo, constatado durante os vídeos gravados no experimento, foi que alunos evitaram a alteração da coleção devido as consequências da alteração no código, como os erros de

compilação que surgiram, mostrando que alguns evitam alterações mesmo que sejam para a melhora do código devido ao trabalho adicional. Inclusive, verificamos que alguns alunos utilizaram a documentação do JCF para ler sobre a estrutura da coleção e mesmo assim não quiseram realizar a alteração. Para nós, esse fato mostra a dificuldade dos alunos na interpretação da documentação e traz mais evidências que o aumento do nível de abstração na exibição das características pode ajudar os alunos a melhor realizar a aplicação de coleções.

Com relação ao uso do *Java Search* constatamos que nenhum aluno havia utilizado a ferramenta anteriormente. Apesar de ser exibida uma apresentação breve antes do experimento, explicando como realizar as buscas, muitos alunos não obtiveram sucesso nos resultados das buscas configuradas por eles. Podemos atribuir o motivo dessa constatação à falta de intimidade com a ferramenta ou até mesmo à dificuldade de manuseamento da mesma, por não ser tão intuitiva para os usuários no primeiro contato. Devido à dificuldade com a ferramenta, alguns alunos preferiram realizar a busca manual a utilizar a ferramenta.

Já com relação ao uso da ferramenta *Collection Adapter*, implementada por nós, constatamos que alguns alunos tiveram dificuldade na interpretação de algumas das características exibidas para as coleções. Características como *"Possui manipulação concorrente dos elementos da coleção"* e *"Possui um elemento chave que servirá para mapear outro elemento"* não foram entendidas pelos alunos. A primeira, relacionada ao acesso *thread-safe* da coleção, pode ter gerado dificuldade devido a pouca experiência dos alunos com *threads* que só será vista por eles em disciplinas posteriores. Com relação à segunda característica, associada à definição se a coleção faz parte de uma estrutura de mapa, podemos procurar uma forma de melhor abordar essa característica em uma nova versão da ferramenta.

Qualidade da alteração

Sobre a qualidade da escolha da coleção mostrada na Tabela 5.2, dentre os alunos que realizaram a alteração da coleção, apenas um realizou a alteração para a coleção que considerávamos a melhor aplicável para o contexto. Os outros alteraram para uma coleção que não fazia parte nem da mesma interface que considerávamos melhor. O aluno que alterou para a melhor coleção, utilizou a nossa ferramenta sem ter dificuldades na interpretação das características exibidas. Entendemos que os alunos que utilizaram a nossa ferramenta e não selecionaram a coleção que achávamos ser a mais apropriada a partir das características exi-

bidas tiveram dificuldades na interpretação das mesmas.

Consideramos que o aluno que utilizou a ferramenta *Java Search* falhou na interpretação do contexto em que a coleção se encontrava e por isso houve a seleção de um tipo de coleção diferente do esperado. Este aluno interpretou que adicionar elementos diferentes a uma coleção implica dizer que a intenção da coleção é não ter elementos repetidos, caracterizando assim um comportamento propício a utilização de um conjunto, como o `HashSet`. Nós havíamos interpretado de forma diferente: para caracterizar a alteração da coleção para um conjunto deve haver, antes da adição de um elemento, a verificação de se aquele elemento já existe na coleção e só adicioná-lo caso ainda não exista, caso mostrado na Figura 1.1.

Esforço da análise e substituição

Na Figura 5.3 podemos verificar que, dentre os alunos que realizaram a substituição, todos consideraram o esforço da análise e substituição de baixo a muito baixo. Juntamente com os resultados exibidos na Tabela 5.3, apesar da pouca quantidade de alunos que realizaram a substituição nessa primeira fase do experimento, podemos ver que os alunos que consideraram o esforço como muito baixo utilizaram a ferramenta implementada *Collection Adapter*.

Para os alunos que não realizaram a substituição, consideramos o esforço como sendo o esforço da análise do contexto da coleção e não o esforço da realização da substituição. Os resultados exibidos na Figura 5.4 nos mostram que a metade dos alunos consideraram o esforço como sendo baixo. Podemos associar esse resultado ao costume de não dedicar um tempo considerável na programação para a análise da coleção que ele irá aplicar, considerando a ação de análise insignificante na programação e consequentemente de esforço desprezível. A Tabela 5.4 exibe o esforço associado à ferramenta utilizada, o interessante é verificar que o esforço de que utilizou a ferramenta *Java Search*, mesmo sem ter o domínio da mesma, foi considerado pelos alunos menor em média do que o esforço considerado pelos alunos que realizaram a análise manual. Desconsideramos o esforço para a nossa ferramenta neste caso por ela não ter sido utilizada pelo aluno.

Tempo de análise e substituição

Para a primeira fase do experimento realizamos a discussão sobre o tempo de análise, na segunda fase realizaremos a análise dos dois tempos: análise e substituição. Como pode ser

visto no esquema do *design* do experimento presente na Tabela 5.1, o grupo 1 dos alunos A1, A2 e A3 receberam o mesmo código fonte para realizar a análise. Realizamos a análise de tempo por grupo.

Analisando as Tabelas 5.5 e 5.6, no caso do grupo 1, os alunos que utilizaram o *Java search* ou não utilizaram ferramenta tiveram tempos bem superiores ao que utilizou o *Collection Adapter*. No segundo grupo, o aluno que usou a *Colletion Adapter* e o que fez manualmente tiveram tempos equivalentes e inferiores ao que fez a análise com ajuda do *Java Search*. No terceiro e quarto grupo os alunos que realizaram a análise com o *Collection Adapter* obtiveram tempo inferior às outras ferramentas. Em todos os quatro grupos a ferramenta *Collection Adapter* teve resultado satisfatório para análise das coleções, acreditamos que por já exibir as características a ferramenta auxilie com facilidade a análise. Na metade dos grupos o *Java Search* obteve tempo superior comparado à análise manual. Como já citado anteriormente, isto pode ser devido a dificuldade do usuário ao manipular a ferramenta já que foi o primeiro contato com a mesma.

5.3.2 Segunda fase do experimento

A segunda fase do experimento foi analisada juntamente com os dados da primeira fase dos alunos que realizaram a substituição. Foi realizada análise estatística dos dados para as variáveis dependentes encontradas, porém o baixo número de sujeitos em cada grupo diminuiu a significância dos testes e, por isso, optamos por fazer uma análise qualitativa dos dados encontrados.

Qualidade da alteração

A porcentagem da qualidade da escolha da coleção exibida na Figura 5.5 nos mostra que a maioria dos alunos fizeram a escolha não sugerida da coleção. Se analisarmos os alunos por grupo, como exibido na Tabela 5.1, a ferramenta *Collection Adapter* obteve o mesmo resultado na qualidade de escolha que a ferramenta *Java Search*. Os alunos que realizaram a substituição manualmente obtiveram resultado superior às outras ferramentas no grupo 4. Neste caso, o aluno apenas escolheu a interface adequada, utilizando uma coleção com ordenação quando não se fazia necessário, dizendo ser a única coleção da qual ele lembrava

que implementava a interface `Set`. Isto nos mostra a falta de familiaridade dos alunos com outros tipos de coleção diferentes de `ArrayList` e `LinkedList`. Considerando as perguntas definidas na Seção 5.1:

A qualidade da substituição utilizando a ferramenta *Collection Adapter* é superior a encontrada na substituição manual?

Acreditamos que podemos considerar que a qualidade da alteração da coleção com a utilização de ferramentas se mostrou inferior em apenas um dos casos comparadas a substituição sem a ferramenta. Porém, se considerada a justificativa do usuário, podemos perceber que a seleção da coleção foi feita sem um fundamento específico, confirmando ter sido uma seleção aleatória. Seguindo o teste estatístico realizado no Código Fonte 5.1, os valores encontrados nas duas ferramentas não diferem estatisticamente.

A qualidade da substituição utilizando a ferramenta *Collection Adapter* é superior a encontrada na substituição com *Java Search*?

No caso da ferramenta *Collection Adapter*, o auxílio se dá na seleção das características da coleção em questão. Já na ferramenta *Java Search*, o auxílio se dá na busca por essas mesmas características. Comparando as duas ferramentas, consideramos que a ferramenta *Collection Adapter* não possui qualidade inferior e sim igual à ferramenta *Java Search*. Seguindo o teste estatístico realizado no Código Fonte 5.1, os valores encontrados nas duas ferramentas não diferem estatisticamente.

Esforço da substituição

Na Figura 5.7, exibe-se a porcentagem de esforço da análise e substituição considerado pelos alunos durante o experimento. Metade dos alunos consideraram o esforço da substituição como baixo mesmo a maioria deles tendo selecionado a coleção incorreta. O que pode nos mostrar que eles não são cientes do empenho que se deve considerar ao selecionar ou alterar uma coleção.

Na Tabela 5.8, podemos verificar que apenas um aluno considerou a atividade de substituição *Muito Alto*. Isto se deu por durante o experimento o aluno ter tido dificuldades em instanciar uma coleção da interface *Set* por não ser utilizada comumente por ele. Este fato

nos mostra que os alunos estão acostumados a apenas utilizar a interface *List* e suas implementações. O aluno que considerou o esforço médio para a utilização do *Collection Adapter* afirmou que sentiu dificuldade por não poder olhar o código enquanto utilizava a ferramenta, pois a janela da ferramenta se sobrepunha ao código. Essa é uma melhora que pretendemos implementar na ferramenta.

Considerando as perguntas de pesquisa sobre o esforço da substituição, temos:

O esforço da análise e substituição utilizando a ferramenta *Collection Adapter* é inferior ao encontrado na substituição manual?

Considerando a comparação do esforço por grupos, a ferramenta *Collection Adapter* possui esforço inferior ou igual à realização da substituição manual. Seguindo o teste realizado no Código Fonte 5.2, os valores encontrados nas duas ferramentas não diferem estatisticamente.

O esforço da análise e substituição utilizando a ferramenta *Collection Adapter* é inferior ao encontrado na substituição com *Java Search*?

Com relação a comparação do esforço entre as ferramentas *Collection Adapter* e *Java Search*, a nossa ferramenta possui esforço inferior ou igual na maioria dos grupos e possuiu desempenho superior em um grupo devido ao fato de o aluno ter tido dificuldade por não poder olhar o código enquanto utilizava a ferramenta. Podemos considerar o experimento inconclusivo para esta pergunta. Seguindo o teste realizado no Código Fonte 5.2, os valores encontrados nas duas ferramentas não diferem estatisticamente.

Tempo de análise e substituição

Para a discussão sobre o tempo exibida na Tabela 5.9, analisaremos o tempo de substituição e análise entre as transformações com a ferramenta *Collection Adapter* e manual. Apenas o tempo de análise entre as ferramentas *Collection Adapter* e *Java Search* será analisado pois a substituição com essa ferramenta é também realizada de forma manual. Consideramos as perguntas de pesquisa levantadas no experimento, teremos:

O tempo utilizado na substituição utilizando a ferramenta *Collection Adapter* é inferior ao encontrado no manual?

O tempo de substituição da ferramenta *Collection Adapter* se mostrou inferior em 3 dos 4 casos mostrados na Tabela 5.9. No caso em que se mostrou superior, a classe utilizada para a execução do experimento pelo Grupo 2 necessitava de uma substituição de uma instância de coleção do tipo `LinkedList` para uma instância do tipo `ArrayList`. Dado esse cenário era esperado que a substituição manual fosse mais rápida, por as coleções terem a mesma interface e apenas ser necessária a mudança no código no local onde acontece a instanciação da coleção. Seguindo o teste realizado no Código Fonte 5.3, os valores encontrados nas duas ferramentas não diferem estatisticamente.

O tempo utilizado na análise com a ferramenta *Collection Adapter* é inferior ao encontrado no manual?

O tempo de análise da ferramenta *Collection Adapter* comparado à análise manual foi superior na maioria dos casos. Porém se analisarmos também o tempo de substituição a nossa ferramenta demostrou um melhor comportamento. Acreditamos que por o usuário ainda não ter familiaridade suficiente com a ferramenta para perceber que as características da coleção a ser utilizadas já são exibidas para ele, sem precisar gastar tanto tempo em análise, ele demorou mais tempo que o necessário. Seguindo o teste realizado no Código Fonte 5.3, os valores encontrados nas duas ferramentas não diferem estatisticamente.

O tempo utilizado na análise utilizando a ferramenta *Collection Adapter* é inferior ao encontrado com *Java Search*?

O tempo de análise da ferramenta *Collection Adapter* comparado à ferramenta *Java Search* foi superior na maioria dos casos. Esse dado era esperado por a ferramenta *Java Search* dar auxílio na localização do comportamento do elemento analisado. Seguindo o teste realizado no Código Fonte 5.3, os valores encontrados nas duas ferramentas não diferem estatisticamente.

5.4 Ameaças à validade

Dentre os tipos de ameaças à validade descritos em [6], abaixo selecionamos algumas ameaças à validade que podem acontecer no nosso estudo experimental.

1. Ameaça à validade interna: se as características das coleções desenvolvidas na árvore de decisão não forem representativas do comportamento da coleção em questão, podemos ter uma ameaça à validade interna; o *software Collection Adapter* criado, assim como qualquer *software*, está suscetível a erros, o que pode vir a invalidar a coleta de dados realizada. Testes de unidade foram implementados para o sistema, visando garantir que a sintaxe do código antes e depois da modificação não tenha sido alterada, sem gerar erros de compilação, por exemplo. Manter a semântica não foi considerado nos testes pois o usuário pode desejar que haja alteração semântica.

2. Ameaça à validade externa: selecionamos 12 alunos, que podem não ser representativos para generalização da população de desenvolvedores em Java. Buscamos selecionar alunos participantes do programa de monitoria das disciplinas de estrutura de dados e algoritmos, por terem tido um desempenho satisfatório na área do nosso estudo, para conseguir amenizar esta ameaça.

Capítulo 6

Trabalhos Relacionados

Durante o nosso levantamento bibliográfico, selecionamos trabalhos que estão relacionados à seleção e aplicação de estrutura de dados e à migração de APIs. Neste capítulo, apresentamos os trabalhos selecionados que são mais relevantes ao nosso e fazemos uma análise comparativa entre eles e o estudo apresentado neste documento.

6.1 Seleção e aplicação de estrutura de dados

Na literatura não encontramos, até o momento de finalização deste trabalho, resultados que sugerem melhoras na aplicação de implementações de coleções do JCF. Existem estudos para análise do comportamento de algumas estruturas de dados, como trabalho de Göβner et al. [3], que realizou um estudo para verificação de como as interfaces em *Java* são utilizadas na prática. Segundo seu estudo, dentre outros resultados obtidos, enquanto a utilização geral de interfaces não parece muito grande, há alguns pontos no código fonte em que as interfaces são introduzidas e utilizadas em grande escala, como na utilização de coleções. O artigo compreende a formalização de métricas aplicando-as no *Java Development Kit*(JDK) apenas para análise da sua utilização. O autor não considera a indicação de qual seria a melhor interface para cada caso. Além disto, como são utilizadas métricas formais, como a exibida na Figura 6.1, comparada a linguagem natural feita em nosso estudo, tem uma interpretação mais difícil.

Kawrykow e Robillard [23] realizou um estudo para analisar casos em que uma determinada API não foi utilizada da forma mais efetiva. O estudo foi feito a partir do desen-

$$PU_C = \frac{\sum_{i=1}^{n} \begin{cases} 1 & \Leftrightarrow V_i : I \wedge C \leq I \\ 0 & else \end{cases}}{\sum_{i=1}^{n} \begin{cases} 1 & \Leftrightarrow V_i : I \wedge C \leq I \vee V_i : C \\ 0 & else \end{cases}}$$

Figura 6.1: Fórmula do uso de polimorfismo do trabalho de Göβner et al. [3]

volvimento de uma técnica para detectar automaticamente tais padrões de mau uso de API em projetos de *software*. A técnica principal de detecção do mau uso foi a verificação se o código cliente da API simula o comportamento de um método da API ao invés de chamá-lo. O trabalho do autor poderia ser associado mais ao nosso trabalho na etapa de análise do contexto em que as coleções foram inseridas. Por exemplo, seguindo a técnica principal, verificar se o aluno está criando uma coleção em vez de utilizar uma já existente no JCF. Caracteriza-se um trabalho futuro a utilização da análise do mau uso da coleção de forma automática aumentando a eficiência da nossa ferramenta, assim como foi realizado no trabalho de Kawrykow e Robillard.

O trabalho de Dekel e Herbsleb [24] adota uma abordagem associada a documentação de uma API. Eles afirmam que a documentação contém diretrizes de uso, como regras ou ressalvas, dos quais autores que usam o código devem estar cientes para que sejam lidas, evitando erros e ineficiências no código. Então criaram um *plug-in* do Eclipse que busca invocações de método os quais têm associado diretivas na documentação. Com isso, ele buscou levar os leitores a investigar mais profundamente destacando as diretrizes marcadas no JavaDoc. Enquanto Dekel tem como objetivo melhorar a exibição da documentação da API para o usuário para que quando ele for buscar na documentação as diretivas fiquem destacadas, nós por outro lado visamos a simplificação das características exibidas na documentação da JCF, colocando-as com linguagem mais natural para o usuário sem que haja a necessidade de buscar e estudar a documentação.

6.2 Migração de APIs

Como visto nas seções anteriores, nos baseamos para realizar as transformações das instâncias de coleções em trabalhos relacionados à migração de APIs. O processo de migração consiste na adaptação da aplicação cliente para a nova versão da API que possui as alterações incorporadas. Verificamos a existência de diversos métodos para realizar a adaptação ou transformação da aplicação cliente para a nova API. Dentre os métodos estudados, selecionamos a utilização dos adaptadores para realização da migração de uma coleção para outra.

No trabalho de Balaban et al. [10], foi elaborada uma técnica para migrar de forma automática as aplicações que utilizam classes legadas de uma API. Um das diferenças iniciais com o nosso trabalho é que no de Balaban todas as transformações da migração são consideradas refatoramento, entretanto no nosso trabalho não são todas transformações que preservam o comportamento do programa, então não consideramos as transformações como sendo refatoramentos. Como se trata de refatoramento, os autores tentam preservar o mesmo comportamento que a aplicação possuía antes da transformação e este esforço torna o processo mais complicado por envolver algoritmos de garantia de correture, por exemplo. O nosso trabalho inclui apenas coleções presentes no JCF; o trabalho de Balaban, apesar de utilizar como exemplos classes presentes no JCF, afirma servir para qualquer API. Porém, se o usuário da nossa ferramenta desejar utilizar a transformação com ajudar de adaptadores para outras APIs, ele precisará implementar os mapeamentos através do XML. Em ambos os trabalhos as técnicas foram implementadas como um *puglin* do Eclipse, apesar de no nosso trabalho não ter sido associada ao menu de refatoramento. No trabalho de Balaban, o usuário tem que fazer o especificação da migração, questão que não foi explicada como foi realizada, e no nosso foi feito por nós em XML, no qual o usuário não se envolve.

O trabalho de Nitas e Notkin [2] aborda uma técnica, denominada *Twinning*, com uma forma de mapeamento denominada *Deep-Adaptation*. Este mapeamento modifica a aplicação cliente para usar uma **API C** que tem um conjunto de interfaces com as duas implementações: a da **API A** que é a API usada pela aplicação cliente e da **API B** que é a API que a aplicação cliente quer passar a utilizar. O autor considera que o mapeamento é custoso de implementar porque o desenvolvedor tem que criar a **API C**, porém mais fácil de manter

porque diferencia o que é implementação da API e o que é implementação da aplicação cliente. Nosso trabalho se assemelha ao de Nitas e Notkin por também apresentar a criação de uma "nova API". Porém, o que nos diferencia é o fato de só fazemos substituição no código do cliente no local da instanciação da variável do elemento e não em todos os locais em que usava-se a antiga API. Também existe um diferencia por Nitas e Notkin abordarem uma migração genérica de qualquer API, ficando a cargo do usuário que vai efetuar a transformação a definição do mapeamento entre o código antigo que será substituído e o código novo que irá substituí-lo. No nosso caso, como estamos tratando de um conjunto finito que é o conjunto de coleções do JCF, não deixamos a cargo do usuário a definição do mapeamento da transformação. Criamos o mapeamento prévio das duas coleções envolvidas na transformação: a coleção que está em uso e a coleção selecionada pelo usuário para substituí-la.

Capítulo 7

Conclusão

Neste trabalho, propusemos uma abordagem semi-automática para seleção de interfaces e implementações existentes no *Java Collections Framework* e a alteração de clientes do *framework* para utilizarem os componentes selecionados, a partir de uma técnica baseada na migração de APIs.

Inicialmente, realizamos um estudo do comportamento dos desenvolvedores quanto a utilização de um subconjunto de coleções do JCF, em cinco projetos *open source* em *Java* selecionados. Implementamos um programa que possui métricas para avaliar o contexto em que as coleções foram inseridas. Através da determinação do contexto, conseguimos determinar qual implementação de coleção é a mais apropriada para a situação encontrada. Os resultados do estudo nos mostraram que a existência de projetos que possuem formas inadequadas de aplicação de coleções, passíveis de transformação para uma melhor seleção das mesmas.

Criamos uma abordagem, baseada em árvores de decisão, para selecionar a mais apropriada interface ou implementação de um elemento do *Java Collections Framework* a partir de características de alto nível de abstração. Levantamos as características das implementações de coleções presentes no *Java Collections* e montamos uma **árvore de decisão**(AD). Elevando o nível de abstração das características de cada coleção para o usuário e agregando-as em uma forma rápida de definição das características desejadas, o usuário não precisa se preocupar com a documentação do JCF nem com as especificações associadas a cada tipo de estrutura de dados.

Desenvolvemos uma técnica, baseada em adaptadores e migração de APIs, para realizar

uma modificação semi-automática em um cliente do JCF selecionando a implementação de coleção mais apropriada. Primeiro, realizamos a associação entre os elementos das coleções envolvidas para criação dos adaptadores, realizando a classificação das associações, por ordem de complexidade, considerando a sintaxe e a semântica dos métodos a serem adaptados. Depois criamos os adaptadores, a partir da associação definida, e realizamos a substituição no código fonte do adaptador criado. A técnica foi implementada no formato de uma ferramenta *plugin* do Eclipse.

As técnicas de migração de APIs existentes visam ser o mais genéricas possíveis, abordando a migração entre quaisquer API. Optamos por trabalhar apenas no âmbito do *Java Collections Framework* objetivando aumentar o nível de automatização no processo. A redução do escopo nos permitiu, além de maior automação, diminuir o trabalho do usuário/desenvolvedor já que os mapeamentos foram previamente elaborados.

Realizamos a avaliação da técnica criada e nela os resultados evidenciaram alguns dos benefícios desta abordagem, juntamente com uma série de limitações, principalmente na definição de requisitos de coleta. O uso de tais requisitos, ajudando a seleção de implementações destas estruturas, em conjunto com a substituição da implementação existente, são os primeiros passos para alertar os desenvolvedores de boas práticas no uso de estruturas de dados e, eventualmente, realizar o alerta e permitir a substituição automática.

7.1 Trabalhos Futuros

Em trabalhos futuros, pretendemos realizar um estudo para definir melhor os requisitos de coleta já que alguns usuários relataram ter dificuldades com as características selecionadas para representar cada coleção na árvore de decisão. Este estudo visará aumentar a performance e aceitação da ferramenta implementada. Além disso, pretendemos aumentar o escopo de coleções de nossa ferramenta de apoio para torná-lo mais útil para o propósito geral de desenvolvimento Java e também pretendemos adaptar o processo para a realização da migração para APIs genéricas sem diminuir o grau de automação já conquistado. Após estas melhoras, também pretendemos como trabalho futuro realizar o experimento com um número maior de amostras aumentando o grau de significância dos resultados encontrados. Com as novidades que o Java 8 trouxe, como o *Lambda*, pretendemos adaptar nossa aborda-

gem para também realizar a migração nos casos que o utilizam.

7.2 Considerações Finais

Sabemos que existem diversas pesquisas realizadas e em andamento na área de migração de APIs, porém todas as pesquisas encontradas abordam a migração como solução para o problema de evolução e versionamento de APIs. Aplicamos a abordagem de migração em uma área ainda não utilizada, visando a melhor utilização do *Java Collections Framework*. Desta forma, contribuímos para o crescimento da aplicação da migração de APIs em outras áreas do desenvolvimento, bem como nossa técnica é um passo inicial para ser um auxílio ao desenvolvedor ter sua programação facilitada na utilização das coleções, partindo de definições de um maior nível abstração que a documentação da API.

Referências Bibliográficas

[1] ARGOUML. Tigris org. `http://argouml.tigris.org/`, Junho 2012.

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

[3] GößNER, J.; MAYER, P.; STEIMANN, F. Interface utilization in the java development kit. In: . Editors HADDAD, H.; OMICINI, A.; WAINWRIGHT, R. L.; LIEBROCK, L. M. ACM, c2004. p. 1310–1315.

[4] ORACLE. Java se documentation. `http://docs.oracle.com/javase/7/docs/technotes/guides/collections/`, Janeiro 2014.

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

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

[7] DIG, D.; NEGARA, S.; MOHINDRA, V.; JOHNSON, R. Reba: refactoring-aware binary adaptation of evolving libraries. In: . ICSE '08. ACM, c2008.

[8] KAPUR, P.; COSSETTE, B.; WALKER, R. J. Refactoring references for library migration. *SIGPLAN Not.*, v. 45, n. 10, Oct. 2010.

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

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

[11] FOWLER, M. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[12] DIG, D.; JOHNSON, R. The role of refactorings in api evolution. In: . ICSM '05. IEEE Computer Society, c2005.

[13] SAVGA, I.; RUDOLF, M.; GÖTZ, S.; ASSMANN, U. Practical refactoring-based framework upgrade. In: . GPCE '08. ACM, c2008.

[14] TANEJA, K.; DIG, D.; XIE, T. Automated detection of api refactorings in libraries. In: . ASE '07. ACM, c2007.

[15] MURPHY-HILL, E.; PARNIN, C.; BLACK, A. P. How we refactor, and how we know it. In: . ICSE '09. IEEE Computer Society, c2009.

[16] SCHÄFER, T.; JONAS, J.; MEZINI, M. Mining framework usage changes from instantiation code. In: . ICSE '08. ACM, c2008.

[17] 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.

[18] APACHE. The apache software foundation. `http://ant.apache.org/`, Junho 2012.

[19] ASPECTJ. The eclipse foundation. `http://www.eclipse.org/aspectj/`, Junho 2012.

[20] FINDBUGS. The university of maryland. `http://findbugs.sourceforge.net/`, Junho 2012.

[21] JEDIT. Jedit org. `http://www.jedit.org/`, Junho 2012.

[22] JAVASEARCH. The eclipse foundation. `http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Fconcepts%2Fconcept-java-search.htm`, Janeiro 2014.

[23] KAWRYKOW, D.; ROBILLARD, M. P. Improving api usage through automatic detection of redundant code. In: . ASE '09. IEEE Computer Society, c2009.

[24] DEKEL, U.; HERBSLEB, J. D. Improving API documentation usability with knowledge pushing. In: . c2009.

Apêndice A

Arquivo XML representando a coleção `ArrayList`

```
<class name="java.util.ArrayList" parameters="E"
   super="java.util.AbstractList" super-parameters="E">
   <interfaces>
      <interface name="java.util.List" parameters="E" />
      <interface name="java.util.RandomAccess" />
      <interface name="java.lang.Cloneable" />
      <interface name="java.io.Serializable" />
   </interfaces>
   <constructors>
      <constructor id="0" modifiers="public">
         <parameter id="0" parameters="? extends E" type="
            java.util.Collection" />
      </constructor>
      <constructor id="1" modifiers="public" />
      <constructor id="2" modifiers="public">
         <parameter id="0" type="int" />
      </constructor>
   </constructors>
```

```
18    <methods>
19       <method id="0" modifiers="public" name="add" return-
            type="void">
20          <parameter id="0" type="int" />
21          <parameter id="1" type="E" />
22       </method>
23       <method id="1" modifiers="public" name="add" return-
            type="boolean">
24          <parameter id="0" type="E" />
25       </method>
26       <method id="2" modifiers="public" name="remove" return-
            type="E">
27          <parameter id="0" type="int" />
28       </method>
29       <method id="3" modifiers="public" name="remove" return-
            type="boolean">
30          <parameter id="0" type="java.lang.Object" />
31       </method>
32       <method id="4" modifiers="public" name="get" return-
            type="E">
33          <parameter id="0" type="int" />
34       </method>
35       <method id="5" modifiers="public" name="clone" return-
            type="java.lang.Object" />
36       <method id="6" modifiers="public" name="indexOf" return
            -type="int">
37          <parameter id="0" type="java.lang.Object" />
38       </method>
39       <method id="7" modifiers="public" name="clear" return-
            type="void" />
```

```
40      <method id="8" modifiers="public" name="isEmpty" return
           -type="boolean" />
41      <method id="9" modifiers="public" name="lastIndexOf"
42         return-type="int">
43         <parameter id="0" type="java.lang.Object" />
44      </method>
45      <method id="10" modifiers="public" name="contains"
           return-type="boolean">
46         <parameter id="0" type="java.lang.Object" />
47      </method>
48      <method id="11" modifiers="public" name="size" return-
           type="int" />
49      <method id="12" modifiers="public" name="subList"
           return-type="java.util.List"
50         return-type-parameters="E">
51         <parameter id="0" type="int" />
52         <parameter id="1" type="int" />
53      </method>
54      <method id="13" modifiers="public" name="toArray"
           return-type="java.lang.Object[]" />
55      <method id="14" modifiers="public" name="toArray"
           parameters="T"
56         return-type="T[]">
57         <parameter id="0" type="T[]" />
58      </method>
59      <method id="15" modifiers="public" name="addAll" return
           -type="boolean">
60         <parameter id="0" type="int" />
61         <parameter id="1" parameters="? extends E" type="
              java.util.Collection" />
62      </method>
```

```
63      <method id="16" modifiers="public" name="addAll" return
           -type="boolean">
64         <parameter id="0" parameters="? extends E" type="
              java.util.Collection" />
65      </method>
66      <method id="17" modifiers="public" name="iterator"
           return-type="java.util.Iterator"
67         return-type-parameters="E" />
68      <method id="18" modifiers="public" name="set" return-
           type="E">
69         <parameter id="0" type="int" />
70         <parameter id="1" type="E" />
71      </method>
72      <method id="19" modifiers="public" name="ensureCapacity
           "
73         return-type="void">
74         <parameter id="0" type="int" />
75      </method>
76      <method id="20" modifiers="public" name="trimToSize"
77         return-type="void" />
78      <method id="21" modifiers="public" name="removeAll"
79         return-type="boolean">
80         <parameter id="0" parameters="?" type="java.util.
              Collection" />
81      </method>
82      <method id="22" modifiers="public" name="retainAll"
83         return-type="boolean">
84         <parameter id="0" parameters="?" type="java.util.
              Collection" />
85      </method>
86      <method id="23" modifiers="public" name="listIterator"
```

```
87          return-type="java.util.ListIterator" return-type-
               parameters="E" />
88       <method id="24" modifiers="public" name="listIterator"
89          return-type="java.util.ListIterator" return-type-
               parameters="E">
90          <parameter id="0" type="int" />
91       </method>
92       <method id="25" modifiers="public" name="equals" return
            -type="boolean">
93          <parameter id="0" type="java.lang.Object" />
94       </method>
95       <method id="26" modifiers="public" name="hashCode"
            return-type="int" />
96       <method id="27" modifiers="public" name="toString"
            return-type="java.lang.String" />
97       <method id="28" modifiers="public" name="containsAll"
98          return-type="boolean">
99          <parameter id="0" parameters="?" type="java.util.
               Collection" />
100      </method>
101   </methods>
102 </class>
```

Apêndice B

Arquivo XML representando o mapeamento entre as coleções `ArrayList` e `LinkedList`

```
<map from="java.util.ArrayList" to="java.util.LinkedList">
   <constructors>
      <constructor-mapping fromId="0" toId="0" type="1"/>
      <constructor-mapping fromId="1" toId="1" type="1"/>
      <constructor-mapping fromId="2" toId="-1" type="4">
         container = new java.util.LinkedList&lt;E&gt;();
      </constructor-mapping>
   </constructors>
   <methods>
      <method-mapping fromId="0" toId="18" type="1"/>
      <method-mapping fromId="1" toId="17" type="1"/>
      <method-mapping fromId="2" toId="20" type="1"/>
      <method-mapping fromId="3" toId="21" type="1"/>
      <method-mapping fromId="4" toId="22" type="1"/>
      <method-mapping fromId="5" toId="23" type="1"/>
      <method-mapping fromId="6" toId="24" type="1"/>
```

```
17        <method-mapping fromId="7" toId="25" type="1"/>
18        <method-mapping fromId="8" toId="45" type="1"/>
19        <method-mapping fromId="9" toId="26" type="1"/>
20        <method-mapping fromId="10" toId="27" type="1"/>
21        <method-mapping fromId="11" toId="28" type="1"/>
22        <method-mapping fromId="12" toId="42" type="1"/>
23        <method-mapping fromId="13" toId="30" type="1"/>
24        <method-mapping fromId="14" toId="29" type="1"/>
25        <method-mapping fromId="15" toId="31" type="1"/>
26        <method-mapping fromId="16" toId="32" type="1"/>
27        <method-mapping fromId="17" toId="39" type="1"/>
28        <method-mapping fromId="18" toId="36" type="1"/>
29        <method-mapping fromId="19" toId="-1" type="4">
30        //TODO
31        </method-mapping>
32        <method-mapping fromId="20" toId="-1" type="4">
33        //TODO
34        </method-mapping>
35        <method-mapping fromId="21" toId="47" type="1"/>
36        <method-mapping fromId="22" toId="48" type="1"/>
37        <method-mapping fromId="23" toId="43" type="1"/>
38        <method-mapping fromId="24" toId="38" type="1"/>
39        <method-mapping fromId="25" toId="40" type="1"/>
40        <method-mapping fromId="26" toId="41" type="1"/>
41        <method-mapping fromId="27" toId="44" type="1"/>
42        <method-mapping fromId="28" toId="46" type="1"/>
43    </methods>
44 </map>
```

Apêndice C

Arquivo XML representando o mapeamento entre as coleções `LinkedList` e `ArrayList`

```
<map from="java.util.LinkedList" to="java.util.ArrayList">
   <constructors>
      <constructor-mapping fromId="0" toId="0" type="1"/>
      <constructor-mapping fromId="1" toId="1" type="1"/>
   </constructors>
   <methods>
      <method-mapping fromId="0" toId="0" type="4">
         container.add(0,arg0);
      </method-mapping>
      <method-mapping fromId="1" toId="0" type="4">
         container.add(arg0);
      </method-mapping>
      <method-mapping fromId="2" toId="0" type="4">
         container.add(0,arg0);
         return true;
      </method-mapping>
```

```
17      <method-mapping fromId="3" toId="0" type="4">
18         container.add(size(),arg0);
19         return true;
20      </method-mapping>
21      <method-mapping fromId="4" toId="2" type="4">
22         return container.remove(0);
23      </method-mapping>
24      <method-mapping fromId="5" toId="2" type="4">
25         return container.remove(size());
26      </method-mapping>
27      <method-mapping fromId="6" toId="2" type="4">
28         return container.isEmpty()?null:container.remove(0);
29      </method-mapping>
30      <method-mapping fromId="7" toId="2" type="4">
31         return container.isEmpty()?null:container.remove(
              size()-1);
32      </method-mapping>
33      <method-mapping fromId="8" toId="4" type="4">
34         return container.get(0);
35      </method-mapping>
36      <method-mapping fromId="9" toId="4" type="4">
37         return container.get(size()-1);
38      </method-mapping>
39      <method-mapping fromId="10" toId="4" type="4">
40         return container.isEmpty()? null: container.get(0);
41      </method-mapping>
42      <method-mapping fromId="11" toId="4" type="4">
43         return container.isEmpty()? null: container.get(size
              ()-1);
44      </method-mapping>
45      <method-mapping fromId="12" toId="2" type="4">
```

```
46          container.remove(indexOf(arg0));
47          return true;
48      </method-mapping>
49      <method-mapping fromId="13" toId="2" type="4">
50          container.remove(lastIndexOf(arg0));
51          return true;
52      </method-mapping>
53      <method-mapping fromId="14" toId="0" type="4">
54          container.add(size(),arg0);
55          return true;
56      </method-mapping>
57      <method-mapping fromId="15" toId="4" type="4">
58          return get(0);
59      </method-mapping>
60      <method-mapping fromId="16" toId="-1" type="4">
61          //TODO
62          return null;
63      </method-mapping>
64      <method-mapping fromId="17" toId="1" type="1"/>
65      <method-mapping fromId="18" toId="0" type="1"/>
66      <method-mapping fromId="19" toId="2" type="4">
67          return container.remove(0);
68      </method-mapping>
69      <method-mapping fromId="20" toId="2" type="1"/>
70      <method-mapping fromId="21" toId="3" type="1"/>
71      <method-mapping fromId="22" toId="4" type="1"/>
72      <method-mapping fromId="23" toId="5" type="1"/>
73      <method-mapping fromId="24" toId="6" type="1"/>
74      <method-mapping fromId="25" toId="7" type="1"/>
75      <method-mapping fromId="26" toId="9" type="1"/>
76      <method-mapping fromId="27" toId="10" type="1"/>
```

```
77        <method-mapping fromId="28" toId="11" type="1"/>
78        <method-mapping fromId="29" toId="14" type="1"/>
79        <method-mapping fromId="30" toId="13" type="1"/>
80        <method-mapping fromId="31" toId="15" type="1"/>
81        <method-mapping fromId="32" toId="16" type="1"/>
82        <method-mapping fromId="33" toId="0" type="4">
83           container.add(size(),arg0);
84        </method-mapping>
85        <method-mapping fromId="34" toId="2" type="4">
86           return container.remove(size()-1);
87        </method-mapping>
88        <method-mapping fromId="35" toId="2" type="4">
89           return container.remove(0);
90        </method-mapping>
91        <method-mapping fromId="36" toId="18" type="1"/>
92        <method-mapping fromId="37" toId="18" type="4">
93           return container.get(0);
94        </method-mapping>
95        <method-mapping fromId="38" toId="24" type="1"/>
96        <method-mapping fromId="39" toId="17" type="1"/>
97        <method-mapping fromId="40" toId="25" type="1"/>
98        <method-mapping fromId="41" toId="26" type="1"/>
99        <method-mapping fromId="42" toId="12" type="1"/>
100       <method-mapping fromId="43" toId="23" type="1"/>
101       <method-mapping fromId="44" toId="27" type="1"/>
102       <method-mapping fromId="45" toId="8" type="1"/>
103       <method-mapping fromId="46" toId="28" type="1"/>
104       <method-mapping fromId="47" toId="21" type="1"/>
105       <method-mapping fromId="48" toId="22" type="1"/>
106    </methods>
107 </map>
```

Apêndice D

Arquivo Java representando o adaptador entre as coleções `LinkedList` e `ArrayList`

```
1
2  package br.edu.ufcg.splab.colladapt;
3
4  public class LinkedListToArrayList<E> extends java.util.LinkedList<
       E> {
5      private java.util.ArrayList<E> container;
6      public LinkedListToArrayList(java.util.Collection<? extends E>
           arg0){
7          container=new java.util.ArrayList<E>(arg0);
8      }
9      public LinkedListToArrayList(){
10         container=new java.util.ArrayList<E>();
11     }
12     @java.lang.Override
13     public void addFirst(E arg0) {
14         container.add(0, arg0);
15     }
16     @java.lang.Override
```

```
17    public void addLast(E arg0) {
18       container.add(arg0);
19    }
20    @java.lang.Override
21    public boolean offerFirst(E arg0) {
22       container.add(0, arg0);
23       return true;
24    }
25    @java.lang.Override
26    public boolean offerLast(E arg0) {
27       container.add(size(), arg0);
28       return true;
29    }
30    @java.lang.Override
31    public E removeFirst() {
32       return container.remove(0);
33    }
34    @java.lang.Override
35    public E removeLast() {
36       return container.remove(size());
37    }
38    @java.lang.Override
39    public E pollFirst() {
40       return container.isEmpty() ? null : container.remove(0);
41    }
42    @java.lang.Override
43    public E pollLast() {
44       return container.isEmpty() ? null : container.remove(size() -
             1);
45    }
46    @java.lang.Override
47    public E getFirst() {
48       return container.get(0);
```

```
49    }
50    @java.lang.Override
51    public E getLast() {
52       return container.get(size() - 1);
53    }
54    @java.lang.Override
55    public E peekFirst() {
56       return container.isEmpty() ? null : container.get(0);
57    }
58    @java.lang.Override
59    public E peekLast() {
60       return container.isEmpty() ? null : container.get(size() - 1)
            ;
61    }
62    @java.lang.Override
63    public boolean removeFirstOccurrence(java.lang.Object arg0) {
64       container.remove(indexOf(arg0));
65       return true;
66    }
67    @java.lang.Override
68    public boolean removeLastOccurrence(java.lang.Object arg0) {
69       container.remove(lastIndexOf(arg0));
70       return true;
71    }
72    @java.lang.Override
73    public boolean offer(E arg0) {
74       container.add(size(), arg0);
75       return true;
76    }
77    @java.lang.Override
78    public E element() {
79       return get(0);
80    }
```

```
81    @java.lang.Override
82    public java.util.Iterator<E> descendingIterator() {
83       return null;
84    }
85    @java.lang.Override
86    public boolean add(E arg0) {
87       return container.add(arg0);
88    }
89    @java.lang.Override
90    public void add(int arg0, E arg1) {
91       container.add(arg0, arg1);
92    }
93    @java.lang.Override
94    public E remove() {
95       return container.remove(0);
96    }
97    @java.lang.Override
98    public E remove(int arg0) {
99       return container.remove(arg0);
100   }
101   @java.lang.Override
102   public boolean remove(java.lang.Object arg0) {
103      return container.remove(arg0);
104   }
105   @java.lang.Override
106   public E get(int arg0) {
107      return container.get(arg0);
108   }
109   @java.lang.Override
110   public java.lang.Object clone() {
111      return container.clone();
112   }
113   @java.lang.Override
```

```
114    public int indexOf(java.lang.Object arg0) {
115       return container.indexOf(arg0);
116    }
117    @java.lang.Override
118    public void clear() {
119       container.clear();
120    }
121    @java.lang.Override
122    public int lastIndexOf(java.lang.Object arg0) {
123       return container.lastIndexOf(arg0);
124    }
125    @java.lang.Override
126    public boolean contains(java.lang.Object arg0) {
127       return container.contains(arg0);
128    }
129    @java.lang.Override
130    public int size() {
131       return container.size();
132    }
133    @java.lang.Override
134    public <T> T[] toArray(T[] arg0) {
135       return container.toArray(arg0);
136    }
137    @java.lang.Override
138    public java.lang.Object[] toArray() {
139       return container.toArray();
140    }
141    @java.lang.Override
142    public boolean addAll(int arg0, java.util.Collection<? extends E
       > arg1) {
143       return container.addAll(arg0, arg1);
144    }
145    @java.lang.Override
```

```
146    public boolean addAll(java.util.Collection<? extends E> arg0) {
147       return container.addAll(arg0);
148    }
149    @java.lang.Override
150    public void push(E arg0) {
151       container.add(size(), arg0);
152    }
153    @java.lang.Override
154    public E pop() {
155       return container.remove(size() - 1);
156    }
157    @java.lang.Override
158    public E poll() {
159       return container.remove(0);
160    }
161    @java.lang.Override
162    public E set(int arg0, E arg1) {
163       return container.set(arg0, arg1);
164    }
165    @java.lang.Override
166    public E peek() {
167       return container.get(0);
168    }
169    @java.lang.Override
170    public java.util.ListIterator<E> listIterator(int arg0) {
171       return container.listIterator(arg0);
172    }
173    @java.lang.Override
174    public java.util.Iterator<E> iterator() {
175       return container.iterator();
176    }
177
178    @java.lang.Override
```

```
179    public boolean equals(java.lang.Object arg0) {
180       return container.equals(arg0);
181    }
182    @java.lang.Override
183    public int hashCode() {
184       return container.hashCode();
185    }
186    @java.lang.Override
187    public java.util.List<E> subList(int arg0, int arg1) {
188       return container.subList(arg0, arg1);
189    }
190    @java.lang.Override
191    public java.util.ListIterator<E> listIterator() {
192       return container.listIterator();
193    }
194    @java.lang.Override
195    public java.lang.String toString() {
196       return container.toString();
197    }
198    @java.lang.Override
199    public boolean isEmpty() {
200       return container.isEmpty();
201    }
202    @java.lang.Override
203    public boolean containsAll(java.util.Collection<?> arg0) {
204       return container.containsAll(arg0);
205    }
206    @java.lang.Override
207    public boolean removeAll(java.util.Collection<?> arg0) {
208       return container.removeAll(arg0);
209    }
210    @java.lang.Override
211    public boolean retainAll(java.util.Collection<?> arg0) {
```

```
212         return container.retainAll(arg0);
213     }
214 }
```