Virgílio Borges de Oliveira

Extração Semi-automática de Linhas de Produtos de Software

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica de Minas Gerais, como requisito parcial para a obtenção do grau de Mestre em Informática.

Belo Horizonte

Dezembro de 2010

Resumo

A extração de uma linha de produtos de softwares (LPS) não trivial de um sistema legado é uma tarefa que consome tempo. Primeiro, os desenvolvedores precisam identificar os componentes responsáveis pela implementação de cada *feature* do software. Em seguida, devem localizar as linhas de código que referenciam os componentes descobertos no passo anterior. Finalmente, devem extrair essas linhas para módulos independentes ou devem anotar as linhas de alguma forma. A fim de acelerar a extração de linhas de produtos, esta dissertação descreve uma abordagem semi-automática para anotar o código de *features* opcionais em uma LPS. A abordagem proposta baseia-se em uma ferramenta existente para desenvolvimento de linhas de produtos chamada CIDE, que aprimora a IDE Eclipse permitindo associar cores de fundo às linhas de código que implementam uma *feature*. Esta abordagem foi aplicada com sucesso na extração de *features* opcionais de três sistemas não triviais: Prevayler (um banco de dados em memória primária), JFreeChart (uma biblioteca para criação de gráficos) e ArgoUML (uma ferramenta para modelagem UML).

Abstract

The extraction of non-trivial software product lines (SPL) from a legacy application is a time-consuming task. First, developers need to identify the components responsible for the implementation of each program feature. Next, they must locate the lines of code that reference the components discovered in the previous step. Finally, they must extract such lines to independent modules or they must annotate the lines in some way. In order to speed up product line extraction, this dissertation describes a semi-automatic approach to annotate the code of optional features in SPL. The proposed approach is based on an existing tool to product line development, called CIDE, that enhances standard IDE with the ability to associate background colors to the lines of code that implement a feature. This approach was successfully applied in the extraction of optional features from three non-trivial systems: Prevayler (an in-memory database system), JFreeChart (a chart library), and ArgoUML (a UML modelling tool).

Dedico este trabalho à minha mãe Maria Alice e à minha irmã Letícia pelo constante carinho e incentivo; à minha esposa Luciana, amiga e companheira de todas as horas, e à minha avó Hebe, hoje com 91 anos, cuja alegria de viver sempre foi fonte de inspiração em minha vida.

Agradecimentos

A família e os amigos são primordiais para nos ajudar a vencer os nossos desafios, com apoio e carinho. A todos, sinceros agradecimentos.

Agradeço ao meu orientador, o ilustre Professor Marco Túlio Valente, cuja orientação firme, prestimosa e atenta me manteve focado e deu-me o estímulo e a segurança necessária para que eu desenvolvesse este trabalho da melhor maneira. Além de uma pessoa incrível e inteligente, é um verdadeiro exemplo que pretendo seguir na minha vida profissional e pessoal. Você, Professor, é meu ídolo!

Agradeço ao meu grande amigo Ricardo Terra, sem dúvida o principal incentivador para o meu ingresso neste curso de mestrado. Muito obrigado, amigo.

Agradeço aos meus colegas da PUC Minas, em especial à Geovália Coelho, minha companheira de muitas manhãs e tardes durante todo o curso.

Agradeço aos meus amigos Gerson Borges, Osmar Ventura, Carla Cardoso, Ralph Liebessohn, Márcio Amaral e a todos os caros colegas de trabalho que torceram e torcem por mim.

Agradeço aos colegas Rógel Garcia e Leonardo Passos, pelas sugestões e críticas construtivas durante o desenvolvimento da ferramenta proposta neste trabalho. Meus agradecimentos também ao Christian Kästner e Sven Apel, da Universidade de Magdeburg, pela cessão do código fonte da ferramenta CIDE.

Por fim, agradeço aos professores da PUC Minas e especialmente à secretária Giovana Silva, pela enorme paciência e atenção a mim dedicadas.

Conteúdo

Lista de Figuras **vi**

Lista de Tabelas **vii**

1 Introdução **1**

1.1 Motivação 1

1.2 Objetivos 2

1.3 Visão Geral da Solução Proposta 2

1.4 Estrutura da Dissertação 3

2 Linhas de Produtos de Software **4**

2.1 Conceitos Básicos 4

2.2 Técnicas baseadas em Composição 6

2.2.1 Programação Orientada por Aspectos 6

2.2.2 Programação Orientada por *Features* 9

2.3 Técnicas baseadas em Anotações 11

2.3.1 Diretivas de Pré-processamento 11

2.3.2 CIDE (Colored IDE) 12

2.4 Comparação das Abordagens 13

2.5 Trabalhos Relacionados 14

2.6 Comentários Finais 16

3 Abordagem Proposta **17**

3.1 Introdução 17

3.2 Algoritmo de Coloração 20

3.3 Processo de Extração 28

3.4 Implementação 29
3.5 Comentários Finais 35

4 Avaliação **37**
4.1 Introdução 37
4.2 Resultados do Prevayler 40
4.3 Resultados do JFreeChart 41
4.4 Resultados do ArgoUML 42
4.5 Discussão 45
4.6 Riscos à Validade do Estudo Realizado 47
4.7 Comentários Finais 48

5 Conclusões **49**
5.1 Contribuições 49
5.2 Trabalhos Futuros 50

Bibliografia **52**

Lista de Figuras

2.1 Modelo de Produção de uma LPS [Bos01] 5
2.2 Modularização de features via AOP [ZJ04] 7
2.3 Código da classe *Buffer* orientado por objetos [Ape07] 8
2.4 Aspecto *BufferSync*: adiciona sincronização à classe *Buffer* [Ape07] 8
2.5 Classe *Stack* original [KAK08] 10
2.6 Refatoração da classe *Stack* e seus refinamentos usando AHEAD [KAK08] 10
2.7 Exemplo de código anotado com diretivas de pré-processamento 12
2.8 Ferramenta CIDE (*Colored IDE*) 13

3.1 Método `push` com as *features* anotadas pelo algoritmo proposto 20
3.2 Rotina principal 21
3.3 Propagação de cores 23
3.4 Expansão de cores 25
3.5 Regras para expansão de cores 26
3.6 Exemplos de fragmentos de programas onde anotações podem gerar erros sintáticos (a), erros de tipo (b) ou quando não é seguro aplicar uma expansão de cor (c). Os retângulos indicam código anotado pelas fases de propagação e expansão. 27
3.7 Fluxograma do Processo Iterativo de Extração 28
3.8 Métodos da classe FeatureExtractionAction 30
3.9 Primeira página da interface gráfica da ferramenta 31
3.10 Segunda página da interface gráfica da ferramenta 32
3.11 AST da classe `Stack` 33

Lista de Tabelas

2.1 Exemplo de um LPS 12
2.2 Comparação da ferramenta CIDE com compilação condicional (CC), programação orientada por aspectos (AOP) e programação orientada por *features* (FOP) 14

3.1 *Features* e sementes da classe `Stack` 20
3.2 Funções de consulta a AST 21
3.3 Funções que retornam nodos da AST 22
3.4 Expansões Semi-automáticas 27

4.1 Bytes anotados pela extração manual 39
4.2 Bytes anotados no estudo de caso do Prevayler (M= extração manual; A= algoritmo de anotação) 41
4.3 Expansões Semi-automáticas para o JFreeChart 42
4.4 Bytes anotados no estudo de caso do JFreeChart (M= extração manual; A= algoritmo de anotação) 42
4.5 Sementes das *features* e número de iterações para o estudo de caso do ArgoUML 43
4.6 Expansões Semi-automáticas para o ArgoUML (ações padrão não aceitas estão indicadas entre parênteses) 43
4.7 Bytes anotados no estudo de caso do ArgoUML (M= extração manual; A= algoritmo de anotação) 44

Capítulo 1

Introdução

1.1 Motivação

Linhas de Produtos de Softwares (LPS) é uma metodologia de desenvolvimento que propõe o reúso sistemático de componentes e artefatos de software na geração de uma família de sistemas. A adoção dessa técnica na construção de produtos de software pode levar a um aumento significativo na produtividade e na qualidade dos sistemas, proporcionando maior satisfação aos clientes que poderão adquirir produtos que atendam especificamente às suas necessidades. No entanto, em sistemas que foram implementados sem a preocupação de separar as funcionalidades básicas das funcionalidades particulares de um domínio de uso, a extração de uma LPS tem se revelado uma tarefa trabalhosa, que requer dos desenvolvedores um vasto conhecimento do código fonte e uma reengenharia elaborada [KAB07].

Técnicas composicionais, como Programação Orientada a Aspectos [KLM+97] e Programação Orientada a *Features* [Pre97], possibilitam a separação física de cada *feature* do sistema em módulos distintos. No entanto, para *features* cujo código se encontra entrelaçado ao núcleo, diversas refatorações podem ser necessárias para viabilizar o uso de linguagens composicionais [NV09, NOV09]. Como alternativa, técnicas baseadas em anotação, como diretivas de pré-processamento [KR88], possibilitam que qualquer trecho do código fonte seja associado a uma determinada *feature*. Apesar de não possuir as mesmas limitações que as técnicas composicionais para tratar *features* de granularidade fina, anotações tendem a se misturar ao código, tornando-o de difícil entendimento e manutenção [KAK08, LAL+10].

Uma solução para eliminar o ofuscamento causado pelas diretivas de pré-processamento tradicionais é a chamada de Separação Virtual de Interesses [KA09]. A técnica consiste em substituir as diretivas por cores de fundo, que serão associadas a cada trecho do

código anotado. Uma implementação dessa técnica é a ferramenta CIDE [Käs07], uma extensão da IDE Eclipse, utilizada com sucesso na extração de LPS em diversos estudos de caso [KAK08, KTA08]. A ferramenta permite que o usuário navegue pelo código do sistema marcando manualmente os trechos pertencentes a cada *feature* com uma determinada cor. No entanto, em sistemas mais complexos, a marcação manual mostra-se uma tarefa tediosa e sujeita a erros, exigindo tempo, atenção e profundo conhecimento do sistema em questão.

1.2 Objetivos

Esta dissertação tem três objetivos principais:

- Projetar, implementar e avaliar um algoritmo para extração semi-automática de LPS utilizando técnicas baseadas em anotação de código, mais especificamente usando princípios de separação virtual de interesses tais como implementados pela ferramenta CIDE. A proposta visa reduzir o trabalho demandado pela marcação manual do código associado a determinada *feature*, o qual, na maioria das vezes, encontra-se espalhado pelo código fonte do sistema.

- Propor um processo iterativo para extração de LPS com o objetivo de orientar os desenvolvedores na aplicação do algoritmo proposto. O processo inicia-se com a escolha de elementos do programa que serão usados como entrada para o algoritmo. Posteriormente, o usuário é questionado quanto a anotação de determinados trechos onde a marcação automática necessita de uma análise mais criteriosa. Finalmente, avalia-se a cobertura alcançada e, caso necessário, realiza-se uma nova iteração refinando os elementos de entrada.

- Avaliar o processo de extração proposto em sistemas não triviais de grande porte, comparando os resultados obtidos pela aplicação da solução semi-automática com extrações realizadas manualmente.

1.3 Visão Geral da Solução Proposta

O algoritmo proposto neste trabalho foi implementado como uma extensão da ferramenta CIDE [KTA08]. Deve-se fornecer como entrada um conjunto de elementos do programa, chamados de sementes, e a cor que será usada na anotação.

O algoritmo foi dividido em duas fases. Na primeira fase, chamada de propagação de cores, todos os elementos do código fonte ligados direta ou indiretamente às sementes

fornecidas são anotados. Na segunda fase, chamada de expansão, o contexto dos trechos anotados na fase anterior são verificados a fim de evitar blocos vazios, expressões inconsistentes, erros de sintaxe e erros de tipo. Em determinadas situações, onde não é possível garantir uma expansão segura, uma intervenção do usuário se faz necessária para evitar que sejam coloridos trechos de código que não dizem respeito à *feature* a ser extraída. Essas expansões são chamadas de semi-automáticas. O algoritmo proposto é um algoritmo de ponto fixo, isto é, suas fases são repetidas sucessivamente, até que nenhum novo trecho seja colorido.

1.4 Estrutura da Dissertação

O restante deste trabalho está organizado da seguinte maneira:

- Capítulo 2: Este capítulo apresenta uma revisão da literatura dos temas relacionados ao trabalho, tais como: linhas de produtos de software, programação orientada por aspectos, programação orientada por *features*, compilação condicional e a ferramenta CIDE. Realiza-se também um estudo comparativo entre abordagens composicionais e abordagens baseadas em anotação;

- Capítulo 3: Neste capítulo, descreve-se o algoritmo de coloração semi-automática e o processo de extração de *features*, fornecendo-se detalhes de sua implementação e funcionamento;

- Capítulo 4: Neste capítulo, descreve-se a aplicação da abordagem proposta em três estudos de caso, explicando a metodologia usada e analisando os resultados obtidos em cada experiência;

- Capítulo 5: Este capítulo apresenta as conclusões desta dissertação, apontando suas principais contribuições, as limitações da solução e propostas para trabalhos futuros.

Capítulo 2

Linhas de Produtos de Software

2.1 Conceitos Básicos

No domínio da engenharia de software, a indústria de desenvolvimento de sistemas tem passado pelos mesmos desafios dos demais setores no tocante a necessidade de reduzir custos, aumentar a produtividade e a qualidade dos produtos, sem deixar de lado as necessidades particulares dos consumidores. À medida que o tamanho e a complexidade dos sistemas aumentam, surge a necessidade da adoção de um paradigma de desenvolvimento baseado em linhas de produtos. Similar ao conceito de linha de montagem de automóveis introduzido por Henry Ford no princípio do século XX, uma linha de produtos de software (LPS) consiste em um paradigma para desenvolvimento de aplicativos (sistemas ou produtos de software) que combina uma plataforma comum de desenvolvimento de software com princípios e idéias de customização em massa [PBvdL05].

Por plataforma de desenvolvimento entende-se uma base tecnológica na qual outras tecnologias ou processos são construídos. No contexto de software, uma plataforma é composta por um conjunto de interfaces e subsistemas de software que disponibilizam uma estrutura comum na qual subprodutos podem ser desenvolvidos e produzidos de forma eficiente. Por customização em massa entende-se a produção em larga escala de bens adaptados às necessidades individuais dos clientes [PBvdL05]. Os subprodutos gerados a partir de uma linha de produtos compõem uma família de sistemas. O conceito de família de sistemas foi introduzido por Parnas em 1976, como sendo um conjunto de programas nos quais as propriedades comuns são tão relevantes que torna-se vantajoso estudá-las em primeiro lugar para posteriormente determinar as características particulares de cada membro da família [Par76]. Dessa forma, pode-se explorar as similaridades compartilhadas pelos sistemas a fim de alcançar economias no processo de desenvolvimento, permitindo que produtos de software sejam sintetizados a partir de artefatos comuns [CN02].

A incorporação dos princípios de linhas de produtos no processo de desenvolvimento de sistemas pode promover a reutilização sistemática de componentes de software. Além do aumento da produtividade e da qualidade dos produtos pode-se atingir outros benefícios, entre eles: diminuição do esforço com a manutenção, evolução organizada, ciclos de desenvolvimento mais curtos, simplificação do cálculo para estimativa de custos e diversos produtos compartilhando uma mesma interface. Tais benefícios têm impacto direto na satisfação dos consumidores, que passam a obter produtos adaptados às suas necessidades. Contudo, a adoção desse modelo no processo de desenvolvimento requer um alto investimento inicial, que pode ser considerado de risco se os requisitos principais não forem levantados corretamente, além de exigir um alto nível de gerenciamento técnico e organizacional [Nor06].

O processo de desenvolvimento de uma LPS envolve três atividades que se relacionam entre si. Os ativos base (*core assets*) formam a base da linha de produção, incluindo todos os elementos envolvidos no ciclo de desenvolvimento, tais como: levantamento de requisitos, modelo de domínio, arquitetura de software, documentação, planos de teste, processos e métodos, ferramentas, cronogramas e o código propriamente dito. Em alguns casos, os produtos de software já existentes podem também fazer parte dos ativos da base. A partir dos ativos base, novos produtos são desenvolvidos (*product development*). Esses produtos irão realimentar o processo, gerando novos ativos base. O modelo de produção de uma LPS é mostrado na Figura 2.1.

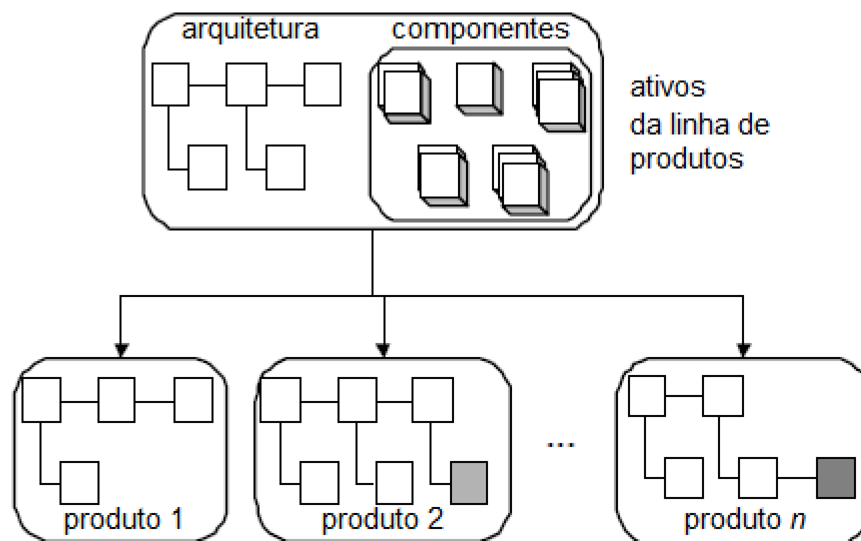

Figura 2.1: Modelo de Produção de uma LPS [Bos01]

O gerenciamento (*management*) em múltiplos níveis organiza os processos, sendo de fundamental importância para a perfeita interação entre as três atividades. A gerência é

responsável por coordenar, motivar e preparar a equipe de desenvolvimento, oferecendo treinamentos apropriados, alocando recursos necessários, gerenciando interfaces externas, criando e implementando um plano de adoção de linhas de produtos.

Dentre os diferentes modelos para implementação de uma LPS, destaca-se o modelo extrativo [Kru01]. Esse modelo propõe a extração de uma LPS a partir de um sistema já existente, implementado sem a preocupação de separar funcionalidades (*features*) comuns de funcionalidades particulares de um domínio de uso. Dentro do modelo extrativo, as soluções para extração de *features* se dividem em dois grupos: baseadas em composição e baseadas em anotações.

Soluções baseadas em composição – tais como programação orientada por aspectos – propõem que as *features* de uma família de produtos sejam lexicamente implementadas em módulos distintos. Com isso, na fase de composição de um sistema, desenvolvedores podem escolher os módulos que serão incluídos em sua compilação. Por outro lado, soluções baseadas em anotações permitem marcar e associar trechos arbitrários de código como responsáveis pela implementação de uma determinada *feature*, o que caracteriza um modelo de granularidade mais fina (*fine-grained*) para extensão de componentes [KAK08].

2.2 Técnicas baseadas em Composição

2.2.1 Programação Orientada por Aspectos

Programação Orientada por Aspectos (AOP) é um paradigma de programação emergente para a modularização de interesses transversais. Neste contexto, os interesses transversais constituem os aspectos, que são encapsulados e implementados em módulos separados [KLM+97].

AspectJ, uma extensão da linguagem Java orientada por aspectos, é uma das implementações mais conhecidas desse paradigma [KHH+01]. Ela define um conjunto de construções que dão suporte a dois tipos de mecanismos para modularização de requisitos transversais: dinâmicos e estáticos. Requisitos transversais dinâmicos são implementados a partir da definição de pontos de junção (*join points*), isto é, pontos pré-determinados na execução de um programa em Java. A união de diversos pontos de junção e seus respectivos parâmetros é chamada de conjuntos de junção (*pointcuts*). Instruções conhecidas como adendos (*advices*) determinam se o código de um aspecto irá executar antes (*before*), depois (*after*) ou no lugar (*around*) da execução de um ponto de junção. Para a implementação de requisitos transversais estáticos são usadas declarações chamadas de inter-tipos (*inter-type declarations*), que permitem ao programador incluir novos atribu-

tos e métodos em classes e interfaces, a fim de modificar a hierarquia de tipos do sistema base [GJ05a].

Programação orientada por aspectos pode ser usada para a extração de uma LPS, bastando para isso que cada *feature* seja modularizada como um aspecto, conforme mostra a Figura 2.2. A partir daí, na fase de composição do software, o desenvolvedor poderá escolher os módulos que deseja incluir na compilação. No entanto, esta técnica é considerada de granularidade grossa, uma vez que AspectJ permite instrumentar somente pontos pré-definidos da execução do sistema base [KAK08]. Desta forma, caso o trecho de código associado a uma *feature* esteja em uma parte do software que não corresponde a um ponto de junção, será necessário que o desenvolvedor refatore o código base, a fim de permitir sua implementação por meio de aspectos [NOV09, NV09].

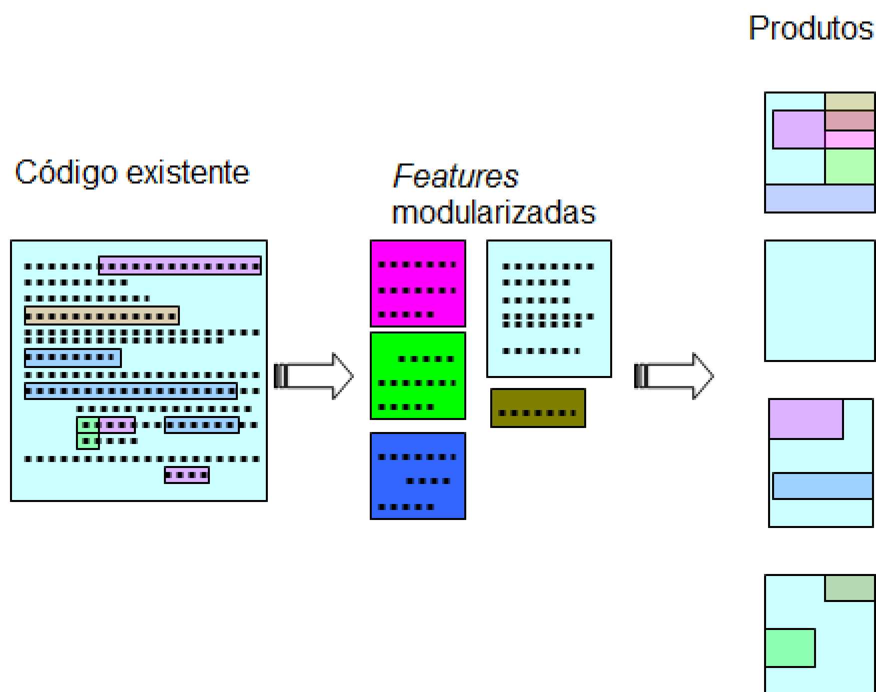

Figura 2.2: Modularização de features via AOP [ZJ04]

Um exemplo da aplicação de aspectos é mostrado nas Figuras 2.3 e 2.4. Nesse exemplo, a classe `Buffer` (Figura 2.3) permite o armazenamento de um conjunto de itens de dados. A inclusão de novos itens é realizada via método `put` (linha 3), enquanto o acesso a um determinado item da coleção é realizado via método `get` (linha 4). O aspecto `BufferSync` (Figura 2.4) tem o objetivo de adicionar sincronização no acesso aos métodos da classe `Buffer`. Primeiro, declara-se um *pointcut* chamado `SyncPC` (linhas 2-4), com o objetivo de interceptar a execução dos métodos `Buffer.get` e `Buffer.put`. Em seguida, declara-se um *advice* do tipo *around* que faz chamadas aos métodos `lock` e `unlock` imediatamente

antes e depois de executar o método interceptado pelo *pointcut* `SyncPC` (linhas 6-8).

```
1: class Buffer {
2:    Vector<Item> buf = new Vector<Item>();
3:    void put(Item e) { buf.add(e); }
4:    Item get(int i) { return buf.get(i); }
5: }
```

Figura 2.3: Código da classe *Buffer* orientado por objetos [Ape07]

```
 1: abstract aspect BufferSync {
 2:    pointcut syncPC():
 3:       execution(Item Buffer.get(int)) ||
 4:       execution(void Buffer.put(Item));
 5:    Object around():syncPC() {
 6:       lock();
 7:       Object res = proceed();
 8:       unlock();
 9:       return res;
10:    }
11: }
```

Figura 2.4: Aspecto *BufferSync*: adiciona sincronização à classe *Buffer* [Ape07]

Estudos recentes demonstram como pode ser feita a decomposição de um software legado, visando a extração de variabilidades via AOP. Por exemplo, já foram estabelecidas regras que definem as refatorações necessárias no código durante o processo de modularização via aspectos [BCH+05, BCH+06, NOV09, NV09]. Os resultados mostram que essa tarefa não pode ser automatizada facilmente, uma vez que requer dos desenvolvedores um profundo conhecimento estrutural do sistema base, do seu fluxo de dados e de suas dependências. Em sistemas de grande porte, pode-se observar diversas dificuldades no processo de extração de linhas de produtos, principalmente no tocante às limitações de AspectJ em tratar variabilidades de granularidade fina [KAB07, KAK08]. Por exemplo, em situações onde o código se encontra no interior de métodos, pode ser necessário extrair métodos vazios (*hook methods*), solução que acaba por deixar o código base semanticamente confuso.

2.2.2 Programação Orientada por *Features*

Programação orientada por *features* (FOP) é uma extensão do modelo orientado por objetos que flexibiliza a composição dos objetos a partir de um conjunto de *features* [Pre97]. Esse paradigma permite que os desenvolvedores criem programas baseados em variabilidades modularizadas, isto é, cada *feature* é implementada em um módulo distinto e representa um incremento na funcionalidade do sistema base [BSR03].

No contexto de linhas de produtos, *features* são usadas para distinguir os sistemas de uma mesma família de produtos. Assim, as classes implementam as funcionalidades básicas da LPS (núcleo), enquanto as funcionalidades opcionais são mantidas em módulos sintaticamente independentes do núcleo base, de acordo com os princípios de separação de interesses. *Features* também podem refinar outras *features* de maneira incremental, com o objetivo de encapsular fragmentos de diversas classes, inserindo ou alterando seus membros (atributos ou métodos) e modificando a hierarquia de tipos. Para derivar um produto de uma LPS, o desenvolvedor deve selecionar as *features* necessárias para sua composição. Os módulos correspondentes são então combinados com o núcleo base do software para gerar o produto desejado.

Um dos ambientes para desenvolvimento baseados em FOP mais representativos é conhecido como *Algebraic Hierarchical Equations for Application Design* (AHEAD). Para a linguagem Java, essas ferramentas estão disponíveis em um pacote chamado *Jakarta Tool Suite* (JTS), que estende a linguagem com um novo conjunto de comandos denominados Jak (abreviação de Jakarta). Quando um arquivo Jak é compilado, ferramentas específicas do pacote JTS são invocadas para compor o sistema.

Assim como em AOP, soluções baseadas em FOP são consideradas de granularidade grossa e, eventualmente, transformações no código fonte são necessárias durante a modelagem das *features*. Para ilustrar essa limitação, a Figura 2.5 mostra o código Java de uma classe `Stack` com um método `push`. Pode se observar que os trechos de código responsáveis por implementar sincronização de acesso aos elementos empilhados estão misturados ao código base (linhas 4 e 6).

A Figura 2.6 mostra as refatorações e refinamentos necessários para extrair a *feature* Multithread do código apresentado na Figura 2.5, usando AHEAD. Na refatoração, foi necessário remover todo o código referente a Multithread, uma vez que ele não faz parte do núcleo da aplicação. A primeira modificação ocorreu na assinatura do método `push` (linha 2), onde o objeto responsável pelo controle de transações não é mais passado como parâmetro. Além disso, a linha de comando responsável pelo empilhamento dos dados (linha 5 da Figura 2.5) foi removida do método `push` para um novo método `h2` (linha 9 da Figura 2.6). Finalmente, a expressão do `if` responsável por verificar se o objeto de

```
1: class Stack {
2:    void push(Object o, Transaction txn) {
3:       if (o == null || txn == null) return ;
4:       Lock l = txn.lock(o);
5:       elementData[size++] = o;
6:       l.unlock();
7:       fireStackChanged();
8:    }
9: }
```

Figura 2.5: Classe *Stack* original [KAK08]

```
 1: class Stack { // núcleo base da aplicação
 2:    void push(Object o) {
 3:       if (o == null || h1()) return;
 4:       h2(o);
 5:       fireStackChanged();
 6:    }
 7:    boolean h1() { return false; }
 8:    void h2(Object o) {
 9:       elementData[size++] = o;
10:    }
11: }
12: refines class Stack { // modularização da feature multithread
13:    ThreadLocal<Transaction> pushTxn = new ThreadLocal<Transaction>();
14:    void push(Object o, Transaction txn) {
15:       pushTxn.set(txn);
16:       Super.push(o);
17:    }
18:    void push(Object o) {
19:       throw new UnsupportedOperationException (
20:          "Call push(Object, Transaction) instead");
21:    }
22:    boolean h1() {
23:       return pushTxn.get() == null;
24:    }
25:    void h2(Object o) {
26:       Lock l = pushTxn.get().lock(o);
27:       Super.h2(o);
28:       l.unlock();
29:    }
30: }
```

Figura 2.6: Refatoração da classe *Stack* e seus refinamentos usando AHEAD [KAK08]

transações é nulo (linha 3 da Figura 2.5) foi removida e em seu lugar adicionou-se uma chamada a um novo método `h1` (linha 3 da Figura 2.6), que sempre retorna falso (linha 7 da Figura 2.6).

Uma vez concluídas as refatorações, a classe `Stack` é então refinada (linhas 12-30 da Figura 2.6). O refinamento consiste na definição de um novo módulo, para onde foi movido todo o código referente à *feature* Multithread. O método `push` é sobrecarregado para receber novamente o objeto de transações como parâmetro (linha 14). O método `h1` é refinado para verificar se o objeto de transações é nulo (linhas 22-24). Finalmente, o método `h2` é refinado, adicionando as chamadas aos métodos `lock` e `unlock` (linhas 26-28) a fim de sincronizar o acesso ao método `h2` original da classe.

2.3 Técnicas baseadas em Anotações

2.3.1 Diretivas de Pré-processamento

O uso de diretivas de pré-processamento na implementação de uma LPS consiste na anotação de trechos de código que serão associados a uma determinada *feature* do sistema. Algumas linguagens de programação, tais como C e C++, possuem suporte nativo a essas diretivas, que são avaliadas por um pré-processador antes do início da compilação [KR88]. Outras linguagens, como Java, necessitam de módulos adicionais (*plug-ins*) para dar suporte a diretivas de pré-processamento.

No contexto de gerência de variabilidades, as diretivas mais comuns para anotação do código são: `#if` ou `#ifdef`, `#else` e `#endif`. Tais diretivas definem blocos de comandos que poderão ou não fazer parte do código fonte do software, dependendo do valor de uma constante simbólica definida pelo programador, antes da compilação. As diversas combinações possíveis para os valores das constantes resultarão em diferentes produtos de software.

A Figura 2.7 mostra o código correspondente ao método `push()` de uma classe `Pilha`, a fim de ilustrar o funcionamento dessa técnica. Os trechos delimitados pela diretiva `#ifdef LOCK` correspondem à *feature* que habilita o suporte a *threads* e os delimitados pela diretiva `#ifdef LOGGING` correspondem à *feature* de *logging*. A definição das constantes `LOCK` e `LOGGING` irá determinar a inclusão dos respectivos trechos, possibilitando a geração de quatro produtos diferentes para essa linha de produtos trivial, conforme ilustrado na Tabela 2.1.

Abordagens baseadas em diretivas de pré-processamento são consideradas de granularidade fina, pois permitem a anotação de qualquer trecho de código. Consequentemente,

```
 1: public boolean push(Object o)
 2:    #ifdef LOCK
 3:    Lock lock = new lock();
 4:    if(lock == null) {
 5:       #ifdef LOGGING
 6:       log("lock failed for " + o);
 7:       #endif
 8:       return false;
 9:    }
10:    #endif
11:    elementDatasize++ = o;
12:    #ifdef LOGGING
13:    log("pushed " + o + ", new size: " + size);
14:    #endif
15:    #ifdef LOCK
16:    unlock(lock);
17:    #endif
18:    return true;
19: }
```

Figura 2.7: Exemplo de código anotado com diretivas de pré-processamento

	Lock	**Logging**
Produto 1	Sim	Sim
Produto 2	Sim	Não
Produto 3	Não	Sim
Produto 4	Não	Não

Tabela 2.1: Exemplo de um LPS

programadores inexperientes podem anotar trechos vitais para a estrutura do programa, que se removidos levarão a erros de sintaxe. Além disso, tais diretivas dificultam a legibilidade e a manutenibilidade do sistema, uma vez que ficam entrelaçadas ao código fonte [Spe92].

2.3.2 CIDE (Colored IDE)

CIDE (*Colored IDE*) é uma ferramenta para gerenciamento de *features*, proposta por Christian Kästner e Sven Apel [Käs07]. A abordagem consiste em substituir as diretivas de pré-processamento textuais por anotações visuais, associando cores de fundo aos trechos anotados [KTA08]. A ferramenta foi implementada como uma extensão (*plug-in*) para a IDE Eclipse, que adiciona novas funcionalidades ao editor, permitindo que o programador

marque os trechos de código referentes a uma *feature* com uma determinada cor de fundo. A cada *feature* estará associada uma cor. Desta forma, é possível esconder os trechos marcados com a cor desejada e gerar projeções do sistema excluindo trechos coloridos.

A Figura 2.8 mostra um exemplo de como as anotações visuais são exibidas no editor da plataforma Eclipse. O código marcado com a cor amarela corresponde à *feature* de *Logging* e o código em azul corresponde à *feature* de sincronização em ambientes *multi-thread* (*Lock*). Os trechos que pertencem simultaneamente às duas variabilidades são exibidos com a combinação das duas cores. Essa solução simplifica a visualização dos trechos anotados, uma vez que não faz uso de diretivas textuais que tendem a se misturar ao código, dificultando sua legibilidade.

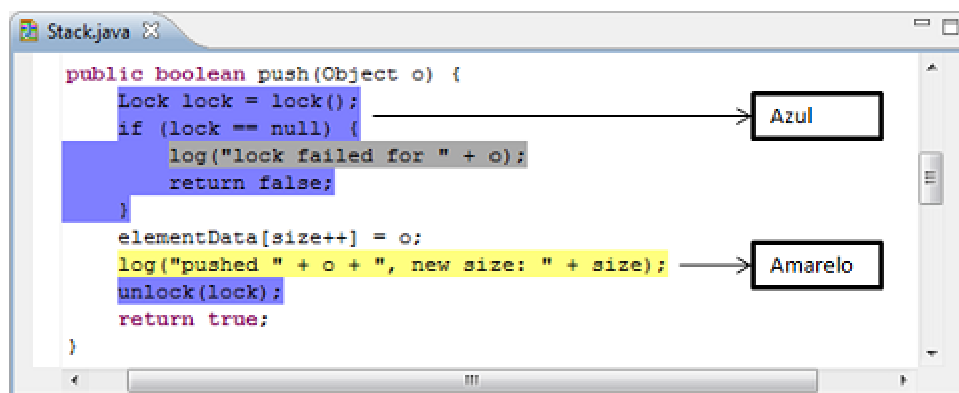

Figura 2.8: Ferramenta CIDE (*Colored IDE*)

Diferente de soluções baseadas em compilação condicional, a ferramenta CIDE não permite que trechos arbitrários do código fonte sejam anotados. Para cada arquivo-fonte é gerada uma AST (*Abstract Syntax Tree*), isto é, uma árvore de sintaxe abstrata onde os nodos representam a estrutura sintática (*tokens*) correspondente ao código fonte do programa. Com isto, a ferramenta restringe a anotação a somente nodos opcionais da AST, isto é, nodos que, se removidos, não geram erros de sintaxe. Desta forma, não é possível marcar qualquer *token* do programa fonte, especificamente aquelas que não carregam informações semânticas, tais como vírgulas, colchetes, parênteses, entre outros. Por esse motivo, se comparada com compilação condicional, essa abordagem possui maior grau de granularidade.

2.4 Comparação das Abordagens

A Tabela 2.2 resume as diferenças existentes entre a ferramenta CIDE e as três principais abordagens para extração de *features*: compilação condicional (CC), programação

orientada por aspectos (AOP) e programação orientada por *features* (FOP). Para fins comparativos, AspectJ [KHH+01] foi escolhida como representante das diversas técnicas de AOP, assim como Jak [Bat04] para FOP. Sobre as propriedades consideradas na Tabela 2.2, coesão de *features* denota a capacidade de manipular *features* a partir de um código com infraestrutura coesa [LHBC05]. Por exemplo, AspectJ não permite que desenvolvedores atribuam um nome a todas as extensões do programa e as manipulem como um único módulo. Por outro lado, Jak oferece mecanismos conhecidos como *mixin-layers* e colaboração, que combinam técnicas de classes parametrizadas e aninhadas para simular um modelo de herança múltipla que suporta *features* coesas. Finalmente, para simular *features* coesas, a ferramenta CIDE permite que o desenvolvedor configure quais *features* deseja visualizar no editor de acordo com suas necessidades, escondendo o código das demais. No entanto, ao realizar atualizações no código, o desenvolvedor deve ficar atento para que as modificações não sofram interferência ou venham a interferir nos trechos de código que foram ocultados.

Propriedades	**CC**	**AOP**	**FOP**	**CIDE**
Extensões	Granularidade extremamente fina	Granularidade grossa	Granularidade grossa	Granularidade fina
Separação de interesses	Não	Sim (físico)	Sim (físico)	Sim (virtual)
Coesão de *features*	Não	Não	Sim (físico)	Sim (virtual)

Tabela 2.2: Comparação da ferramenta CIDE com compilação condicional (CC), programação orientada por aspectos (AOP) e programação orientada por *features* (FOP)

2.5 Trabalhos Relacionados

Os trabalhos relacionados com o algoritmo para extração de linhas de produtos de software a ser descrito nesta dissertação podem ser classificados em dois grupos: relatos de experiências e ferramentas de suporte.

Relatos de Experiências: Lopes-Herrejon, Batory e Cook avaliaram a modularização de *features* usando cinco técnicas: AspectJ, Hyper/J, Jiazzi, Scala e AHEAD [LHBC05]. Como estudo de caso, usou-se um sistema simples para manipulação de expressões matemáticas, que inclui variabilidades como tipos de dados e operações. Devido a simplicidade do sistema, a implementação da LPS não necessitou de extensões de granularidade

fina, suportadas apenas por abordagens baseadas em anotação de código, como diretivas de pré-processamento e a ferramenta CIDE.

Kästner, Apel e Batory documentaram a extração de *features* de uma sistema gerenciador de banco de dados (Berkley DB) usando programação orientada por aspectos (AspectJ) [KAB07]. Uma vez que o Berkley DB é um sistema de tamanho e complexidade consideráveis, foram encontrados diversos problemas durante a extração, a maioria decorrentes das limitações de AspectJ em lidar com variabilidades de granularidade fina. Por exemplo, *features* que necessitam de extensão em blocos aninhados não podem ser extraídas diretamente para aspectos. Nesses casos, foi necessário inserir métodos *hook* no código apenas para permitir a criação de pontos de junção que possam ser instrumentados por AspectJ. A necessidade de se recorrer a métodos *hook* quando se usam técnicas composicionais – como AspectJ – foi relatada também por Murphy et al. em outro estudo sobre extração de features [MLWR01]. Finalmente, as mesmas limitações foram citadas por Adams et al. em seus experimentos na refatoração de compilação condicional para aspectos [AMTH09].

Godil e Jacobsen usaram programação orientada por aspectos para extrair um conjunto de *features* do *framework* Prevayler [GJ05b]. Apesar de não ser detalhado no trabalho, a modularização apresenta os mesmos problemas relatados por Kästner et al. no estudo do Berkley DB, incluindo conjuntos de junção frágeis e complexos, além dos ofuscamentos de código causados por métodos *hook*.

Ferramentas de Suporte: Liu, Batory e Lengauer descrevem uma abordagem formal para refatoração orientada a features [LBL06]. A abordagem proposta, baseada na linguagem composicional AHEAD [Bat04], foi usada para extrair *features* do *framework* Prevayler. Apesar do fato de AHEAD suportar apenas extensões de métodos de granularidade grossa, os autores não detalharam as transformações necessárias no código base que permitiram a extração das *features* para refinamentos (as unidades de modularização suportadas em AHEAD). Foi mencionada apenas a necessidade de reordenar algumas linhas de comando em certas situações.

Binkley et al. desenvolveram a ferramenta AOP-Migrator, um *plugin* para Eclipse que automatiza seis refatorações usadas com maior frequência na migração de sistemas OO para AOP [BCH+05, BCH+06]. Contudo, quando o código orientado por objetos não preenche os pré-requisitos necessários para as refatorações suportadas pela ferramenta, os desenvolvedores devem transformar manualmente o código a fim de possibilitar as refatorações. Essas transformações, denominadas transformações orientadas por objetos, tipicamente incluem reordenação de linhas de comando (ex.: mover um comando para o

início do corpo de um método) ou extração de métodos (a fim de possibilitar a criação de um ponto de junção). FLiP é outra ferramenta para extração de famílias de produtos usando AspectJ [SCN+08]. No entanto, ela apresenta as mesmas limitações descritas na análise da ferramenta AOP-Migrator.

Nassau et al. demonstraram a importância das transformações orientadas por objetos usando quatro sistemas de média complexidade como exemplo [NV09, NOV09]. Foi criada uma ferramenta denominada TransformationMapper, para guiar os desenvolvedores na aplicação dessas transformações. Contudo, a ferramenta apenas indica onde as transformações devem ser aplicadas, cabendo ao desenvolvedor efetuá-las manualmente.

2.6 Comentários Finais

Neste capítulo foram apresentados os principais conceitos relacionados a linhas de produtos de software, bem como as técnicas existentes para sua implementação, incluindo técnicas baseadas em composição (AOP e FOP) e técnicas baseadas em anotações (diretivas de pré-processamento e separação virtual de interesses).

Todas as ferramentas existentes para extração de *features* pesquisadas na revisão de literatura realizada para o desenvolvimento desta dissertação de mestrado baseiam-se em técnicas composicionais, na grande maioria das vezes em aspectos. Assim, pode-se observar nessas ferramentas algumas das limitações inerentes a essa técnica, entre elas o fato de permitir extrair somente *features* de granularidade grossa. Por outro lado, durante o período de realização deste trabalho de dissertação, não se teve conhecimento de ferramentas baseadas em anotações de código que suportem a extração de *features* de sistemas legados.

Capítulo 3

Abordagem Proposta

3.1 Introdução

O principal objetivo da abordagem proposta neste trabalho de mestrado é a extração de *features* opcionais de um sistema existente. Assume-se que um sistema pode ser decomposto em *features* obrigatórias, que constituem o núcleo do sistema, e *features* opcionais, cuja implementação está entrelaçada com o núcleo. Para as *features* opcionais, a ferramenta CIDE será usada para anotar os blocos de código que contribuem para a implementação das mesmas. Por exemplo, suponhamos a existência de uma *feature* opcional F associada a uma cor c. Nesse caso, a anotação de um bloco de código S com a cor c significa que: para construir um produto sem F, deve-se remover S. Em suma, uma vez que o objetivo é desabilitar *features* de um código base pré-existente, a abordagem baseia-se na remoção do código (ou na anotação desse código, pela ferramenta CIDE).

As entradas do algoritmo de anotação são um conjunto S de elementos do programa associados a uma dada *feature* F e uma cor c. Os elementos de S são chamadas de sementes da *feature*, ou simplesmente sementes. Os seguintes elementos de uma programa podem ser usados como sementes: pacotes (exemplo, `mysystem.util`), classes ou interfaces (exemplo, `Logger`), métodos (exemplo, `Logger.log(String)`) e atributos (exemplo, `Logger.filename`). Também é permitido usar uma expressão regular[1] para definir as sementes, (exemplo, `Logger.*` irá selecionar todos os membros da classe `Logger`). Como saída, o algoritmo aplica a cor de fundo c aos fragmentos de código associados à *feature* F.

***Features* Opcionais:** A abordagem proposta nesta dissertação objetiva a extração de *features* opcionais, isto é, *features* que podem ser removidas sem interferir na semântica

[1] A sintaxe é a mesma usada pelas classes do pacote `java.util.regex, disponível na linguagem Java.`

do núcleo base do sistema. Formalizando, suponha o seguinte fluxo de execução de um programa existente:

$$\text{main} \longrightarrow I_1 \longrightarrow F \longrightarrow I_2 \longrightarrow B$$

onde B representa um bloco de código do programa base, F representa um bloco de código marcado como parte de uma *feature* opcional F, I_1 representa o código intermediário executado entre o início do programa e F, e I_2 representa o código intermediário executado entre F e B. Além disso, suponha que $pre(B)$ é uma expressão lógica que denota as pré-condições esperadas em B. Uma *feature* F é opcional se para qualquer fluxo de execução semelhante ao mostrado anteriormente, a seguinte execução é também possível:

$$\text{main} \longrightarrow I_1 \longrightarrow I_2 \longrightarrow \textbf{assert}(pre(B)) \longrightarrow B$$

Em outras palavras, a remoção de F não deve interferir nas pré-condições esperadas em qualquer trecho de código `B` que faz parte do núcleo do sistema.

Interações entre *Features* Uma *feature* F_2 depende de uma *feature* F_1 se existe pelo menos um fluxo de execução:

$$\text{main} \longrightarrow I_1 \longrightarrow F_1 \longrightarrow I_2 \longrightarrow F_2$$

que falha após a remoção de F_1, isto é:

$$\text{main} \longrightarrow I_1 \longrightarrow I_2 \longrightarrow \textbf{assert}(pre(F_2)) \longrightarrow \text{fail}$$

Nesses casos, não é possível ter um produto onde F_2 é selecionada e F_1 não. No entanto, o algoritmo proposto nesta dissertação é incapaz de detectar e prevenir a geração de produtos configurados dessa maneira. Em outras palavras, assume-se que as interações entre as *features* são descobertas, mapeadas e tratadas externamente pelos desenvolvedores, possivelmente por meio de uma ferramenta para modelagem de *features*. Além disso, o algoritmo não lida com o possível impacto das *features* opcionais nos *layouts* das interfaces gráficas (GUI) ou nos recursos externos manipulados pelo sistema (tais como bancos de dados, arquivos de configuração, arquivos *make* e outros).

O problema da *Feature* Opcional: Na extração de uma LPS, o problema da *feature* opcional determina a impossibilidade de se implementar em um único módulo certas *features*

opcionais [LBL06, KAuR+09]. Por exemplo, a implementação de *logging* pode estar entrelaçada ou aninhada com a implementação de outras *features* opcionais, como replicação. Desta forma, o código relacionado a *logging* deve ser quebrado em módulos tais como `Core-Logging` (que fornece *logging* para o núcleo do sistema) e `Replication-Logging` (que fornece *logging* para os módulos de replicação). Quando se utiliza técnicas baseadas em composição, esses pequenos módulos são chamados de derivativos [LBL06]. No algoritmo de anotação proposto neste trabalho, o problema da *feature* opcional é solucionado atribuindo duas cores distintas aos trechos de códigos associados a cada *feature*. Contudo, o significado das anotações não muda , isto é, para construir um produto sem F, qualquer código S marcado com a cor de F deve ser removido (independente da existência de outras cores atribuídas a S).

Exemplo: Suponha uma classe `Stack` – adaptada de [KAK08] – com as *features* opcionais e respectivas sementes descritas na Tabela 3.1. A Figura 3.1 mostra o método `push` dessa classe após a anotação realizada pelo algoritmo proposto (as linhas marcadas estão indicadas com comentários). Inicialmente, para *Logging* o algoritmo irá anotar a chamada de método na linha 4. Para as *features Snapshot* e *Replication*, o algoritmo irá propagar as respectivas cores para as chamadas nas linhas 11 e 13. Após essa propagação, em ambos os casos o corpo de um comando `if` será marcado, mas suas expressões não. A seguir, uma expansão de cor será automaticamente aplicada marcando todo o comando `if` (linhas 10-11 e 12-13).

Para *Multithreading*, o algoritmo primeiramente anotará a declaração da variável local `lk` (linha 2) e seus usos na expressão do comando `if` (linha 3). Não é possível inferir com segurança que a anotação deve ser expandida para toda a expressão do comando `if` (uma vez que depende do retorno do método `lock()`). Nesse caso, um expansão semi-automática será disparada, isto é, o algoritmo perguntará ao usuário se autoriza ou não a coloração de toda a expressão do comando `if`. Nesse caso em particular, a expansão sugerida será aceita e toda a expressão do `if` será anotada. Finalmente, essa anotação semi-automática acarretará em uma expansão automática que marcará todo o comando `if` (linhas 3-6), como parte de *Multithreading*. Após essas expansões, a chamada de *logging* na linha 4 será marcada com duas cores de fundo (*Multithreading* and *Logging*). Assim, esta chamada será removida em qualquer produto gerado sem *Multithreading*. Ela também será removida em produtos sem *Logging* (independente de *Multithreading* ser retirado ou não).

Para concluir essa explicação, a *feature Replication* depende da *feature Snapshot* (pois replicação utiliza os dados persistidos pelo snapshot para enviar uma cópia para outro

Feature	Descrição	Semente
Logging	Log dos eventos internos da pilha	Classe `Log`
Snapshot	Grava uma imagem em disco dos elementos da pilha	Método `snapshot`
Replication	Replica a pilha em outro cliente	Método `replicate`
Multithreading	Pilha segura para multithread	Classe `Lock`

Tabela 3.1: *Features* e sementes da classe `Stack`

```
 1: boolean push(Object o) {
 2:   Lock lk = new Lock();                    // Multithreading
 3:   if (lk.lock() == null) {                 // Multithreading
 4:     Log.log("lock failed for " + o);       // Multithreading, Logging
 5:     return false;                          // Multithreading
 6:   }                                        // Multithreading
 7:   elements[top++]= o;
 8:   size++;
 9:   lk.unlock();                             // Multithreading
10:   if ((size % 10) == 0)                    // Snapshot
11:      snapshot("stack.db");                 // Snapshot
12:   if ((size % 100) == 0)                   // Replication
13:      replicate("stack.db", "server2");     // Replication
14:   return true;
15: }
```

Figura 3.1: Método `push` com as *features* anotadas pelo algoritmo proposto

ponto na rede[2]). No entanto, a abordagem proposta não detecta ou trata a geração de produtos com *Replication*, mas sem *Snapshot*. Como mencionado, assume-se que as dependências entre as *features* serão tratadas externamente pelos engenheiros responsáveis pela extração da LPS.

3.2 Algoritmo de Coloração

A Figura 3.2 apresenta a rotina principal do algoritmo para extração de LPS proposto neste trabalho de mestrado. Basicamente, o algoritmo propaga a cor de fundo c pelo programa até que todos os elementos que dependem da semente S estejam marcados. O algoritmo proposto possui de duas fases: propagação de cores e expansão de cores.

[2]Pela definição prévia de dependência de *features*, a chamada `replicate` (linha 13) pressupõe a seguinte pré-condição: `exists("stack.db")`, onde `exists` verifica se um arquivo existe. Esta pré-condição falha se o código anotado para snapshot for removido, pois `stack.db` é um arquivo criado pela função `snapshot`.

Durante a fase de propagação, os nodos C_1 da AST que referem-se a semente S são anotados; posteriormente, os nodos C_2 da AST que referem-se aos elementos definidos em C_1 são também anotados e assim sucessivamente, até que não existam novos nodos para anotar. O propósito da segunda fase, expansão de cores, é verificar o contexto dos nodos vizinhos aos nodos da AST que foram anotados na fase anterior, para estabelecer se eles também devem ser marcados.

$$
\begin{array}{l}
Main(Seed\ S, Color\ c) = \\
\quad \forall s \in S \rightarrow ColorPropagation(s, c); \\
\quad ColorExpansion();
\end{array}
$$

Figura 3.2: Rotina principal

As Tabelas 3.2 e 3.3 apresentam as funções que auxiliam nas fases de propagação e expansão das cores. As funções da Tabela 3.2 retornam informações estruturais a respeito do código fonte, a partir de buscas na AST. Por outro lado, as funções da Tabela 3.3 retornam um conjunto de nodos da AST. Por exemplo, `call(m)` retorna os nodos que contém chamadas ao método `m`. Finalmente, a função *cide(t,c)* é usada para anotar os nodos t da AST com a cor c.

Funções	**Retorno**
$classes(p)$	classes definidas no pacote p
$interfaces(p)$	interfaces definidas no pacote p
$meths(t)$	métodos definidos na classe ou interface t
$fields(t)$	atributos definidos na classe t
$hasType(t)$	variáveis locais e parâmetros formais do tipo t
$hasReturnType(t)$	métodos com tipo de retorno t
$impl(i)$	classes que implementam a interface i (ou uma interface que estende i)
$extends(t)$	classes que herdam da classe t
$formal(p)$	parâmetros formais associados ao parâmetro de chamada p

Tabela 3.2: Funções de consulta a AST

Propagação de cores: A Figura 3.3 apresenta o algoritmo de propagação de cores. Como mencionado anteriormente, este algoritmo marca todos os elementos que dependem, direta ou indiretamente, da semente S. Mais especificamente, os elementos do programa que podem ser afetados pela propagação de cores são: pacotes (regra P1), classes (regra P2), interfaces (regra P3), métodos (regra P4), atributos (regras P5), variáveis locais

Funções	Returno
`call(m)`	chamadas ao método m
`override(m)`	métodos que sobrescrevem m
`access(x)`	leitura/escrita de atributos, variáveis locais ou parâmetros formais x
`actual(p)`	parâmetros de chamada associados ao parâmetro formal p
`declaration(x)`	declaração de um atributo, variável local, parâmetro formal ou interface x
`import(t)`	importações da classe t
`import(p.*)`	importações de todas as classes do pacote p
`new(t)`	instanciações de objetos do tipo t
`impl(m)`	implementações do método m
`package(p)`	implementações do pacote p

Tabela 3.3: Funções que retornam nodos da AST

(regra P6) e parâmetros formais (regra P7). A seguir, segue a descrição de cada uma dessas regras.

- Regra P1: define que para anotar um pacote p deve-se propagar a cor para todas as classes e interfaces declaradas nesse pacote (linhas 2-3). Deve-se também anotar todas as importações (comandos `import`) das classes de p (linha 4).

- Regra P2: define que para anotar uma classe t deve-se propagar a cor para seus métodos e atributos (linhas 6-7), assim como para outros elementos que dependem de t, incluindo: subclasses (linha 8), declarações de variáveis locais (linha 9), métodos que retornam valores desse tipo (linha 10), instanciação de objetos (linha 11) e importações diretas (linha 12).

- Regra P3: define que para anotar uma interface i deve-se propagar a cor para sua definição (linha 14), assim como para outros elementos do programa que se relacionam com i, incluindo: implementação de métodos dessa interface (linha 15), declarações de variáveis locais (linha 16), métodos que retornam valores desse tipo (linha 17) e importações diretas (linha 18).

- Regra P4: define que para anotar um método m deve-se propagar a cor para a sua implementação (linha 20), assim como para todas as chamada de m (linha 21) e para os métodos que sobrescrevem m (linha 22). É importante mencionar que o algoritmo anota apenas chamadas que possam ser inferidas estaticamente. Por exemplo, suponha a chamada `t.m()`, onde o tipo (estático) do objeto `t` é `T`. Essa chamada será anotada em duas situações: quando a implementação de `m` em `T`

$$
\begin{array}{lll}
1: & ColorPropagation(Package\ p, Color\ c) = & (P1) \\
2: & \quad \forall t \in classes(p) \rightarrow ColorPropagation(t, c); & \\
3: & \quad \forall i \in interfaces(p) \rightarrow ColorPropagation(i, c); & \\
4: & \quad \forall p = \texttt{import}(\texttt{p}.*) \rightarrow cide(p, c); & \\
5: & ColorPropagation(Class\ t, Color\ c) = & (P2) \\
6: & \quad \forall m \in meths(t) \rightarrow ColorPropagation(m, c); & \\
7: & \quad \forall f \in fields(t) \rightarrow ColorPropagation(f, c); & \\
8: & \quad \forall s \in extends(t) \rightarrow ColorPropagation(s, c); & \\
9: & \quad \forall v \in hasType(t) \rightarrow ColorPropagation(v, c); & \\
10: & \quad \forall m \in hasReturnType(t) \rightarrow ColorPropagation(m, c); & \\
11: & \quad \forall n = \texttt{new}(\texttt{t}) \rightarrow cide(n, c); & \\
12: & \quad \forall p = \texttt{import}(\texttt{t}) \rightarrow cide(p, c); & \\
13: & ColorPropagation(Interface\ i, Color\ c) = & (P3) \\
14: & \quad p = \texttt{declaration}(\texttt{i}) \rightarrow cide(p, c); & \\
15: & \quad \forall t \in impl(i) \wedge \forall m \in meths(t) \wedge m \in i \rightarrow ColorPropagation(m, c); & \\
16: & \quad \forall t \in hasType(i) \rightarrow ColorPropagation(t, c); & \\
17: & \quad \forall m \in hasReturnType(t) \rightarrow ColorPropagation(m, c); & \\
18: & \quad \forall p = \texttt{import}(\texttt{i}) \rightarrow cide(p, c); & \\
19: & ColorPropagation(Method\ m, Color\ c) = & (P4) \\
20: & \quad p = \texttt{impl}(\texttt{m}) \rightarrow cide(p, c); & \\
21: & \quad \forall s = \texttt{call}(\texttt{m}) \rightarrow cide(s, c); & \\
22: & \quad \forall m' \in \texttt{overrides}(\texttt{m}) \rightarrow ColorPropagation(m', c). & \\
23: & ColorPropagation(Field\ f, Color\ c) = & (P5) \\
24: & \quad d = \texttt{declaration}(\texttt{f}) \rightarrow cide(d, c); & \\
25: & \quad \forall s = \texttt{access}(\texttt{f}) \rightarrow cide(s, c). & \\
26: & ColorPropagation(LocalVariable\ i, Color\ c) = & (P6) \\
27: & \quad d = \texttt{declaration}(\texttt{i}) \rightarrow cide(d, c); & \\
28: & \quad \forall s = \texttt{access}(\texttt{i}) \rightarrow cide(s, c). & \\
29: & ColorPropagation(FormalParam\ p, Color\ c) = & (P7) \\
30: & \quad d = \texttt{declaration}(\texttt{p}) \rightarrow cide(d, c); & \\
31: & \quad \forall s = \texttt{access}(\texttt{p}) \rightarrow cide(s, c); & \\
32: & \quad \forall s = \texttt{actual}(\texttt{p}) \rightarrow cide(s, c). &
\end{array}
$$

Figura 3.3: Propagação de cores

estiver anotada ou quando `m` for um método herdado de uma superclasse `T’` e a implementação de `m` em `T’` estiver anotada.

- Regra P5: define que para anotar um atributo deve-se propagar a cor para a sua declaração (linha 24), assim como para todos os pontos do programa onde esse

atributo é acessado (linha 25).

- Regra P6: define que para anotar uma variável local deve-se propagar a cor para a sua declaração (linha 27), assim como para todos os pontos do programa onde essa variável é lida ou atualizada (linha 28).

- Regra P7: define que para anotar um parâmetro formal deve-se propagar a cor para a sua declaração (linha 30), para todos os pontos do programa onde esse parâmetro é acessado (linha 31) e para os parâmetros de chamada associados a ele (linha 32).

Expansão de Cores: A Figura 3.4 descreve o algoritmo de expansão de cores. Basicamente, o algoritmo consiste de um laço onde as regras de expansão automáticas e semi-automáticas (detalhadas respectivamente na Figura 3.5 e na Tabela 3.4) são chamadas sequencialmente. O laço termina quando um ponto fixo é atingido, isto é, quando as regras chamadas não inserem nenhuma nova anotação em relação a iteração anterior.

Para propor as regras de expansão de cores apresentadas na Figura 3.5 examinou-se as estruturas de programação da linguagem Java compostas por um bloco de código `C` que controla ou cria um contexto para a execução de outro bloco de código `S`. Por exemplo, `S` pode denotar o corpo de um laço e `C` sua expressão. Uma vez identificadas essas estruturas, investigou-se quando as expansões de cores são possíveis se apenas uma de suas partes estiverem anotadas, isto é, uma expansão de `S` (que foi anotado) para `C` (que não foi anotado) e vice-versa. Seguindo esse processo, as seguintes regras para expansão de cores foram propostas:

- Regra E1: define que a cor usada na expressão de uma estrutura de repetição ou de uma estrutura condicional deve ser expandida para todo o corpo da estrutura. A razão é que se a expressão de um `if`, `while`, `switch` etc. não é necessária em um determinado produto, então todo o comando pode ser removido desse produto.

- Regra E2: define que a cor usada no corpo de uma estrutura de repetição ou de uma estrutura condicional deve ser expandida para a expressão dessas estruturas se essa expressão não produz efeitos colaterais. Na especificação de tais regras, a função $free$(`exp`) verifica quando uma expressão `exp` é livre de efeitos colaterais. A razão é que se o corpo de um `if`, `while`, `switch` etc. não é necessário em um determinado produto e a expressão associada a tal comando não apresenta efeitos colaterais, então todo o comando pode ser revido desse produto.

- Regra E3: define que a cor usada no corpo de uma cláusula `else` deve ser expandida para incluir a própria cláusula. Da mesma forma, a cor usada na expressão de um comando `return` deve ser expandida para incluir o próprio comando.

- Regra E4: define que a cor usada no corpo de um método deve ser expandida para a sua assinatura e para suas chamadas, por meio de uma chamada à rotina de propagação de cores. A razão é que se o corpo de um método não é necessário em um determinado produto, então ele pode ser completamente removido.

- Regra E5: define que a cor usada por todos os membros de uma classe deve ser expandida para seu nome e para qualquer uso dessa classe, por meio de uma chamada à rotina de propagação de cores[3].

- Regra E6: define que a cor usada no lado esquerdo de uma atribuição deve ser expandida para incluir seu lado direito. A razão é que se já foi estabelecido ou inferido que a variável `i` não é necessária em um determinado produto, então atribuições para `i` não têm quaisquer consequências e podem ser removidas no contexto desse produto em particular.

```
ColorExpansion() =
  do
    oldcode = code;
    BodyExpansion();
    ExpExpansion();
    StmExpansion();
    MethExpansion();
    ClassExpansion();
    AssignExpansion();
    SemiAutomaticExpansions();
  while code ≠ oldcode
```

Figura 3.4: Expansão de cores

Expansão Semi-automática: O algoritmo proposto pode gerar um código com erros sintáticos ou erros de tipo, conforme mostra a Figura 3.6. Por exemplo, no fragmento de programa apresentado na Figura 3.6(a), as regras propostas para propagação e expansão de cores marcaram apenas partes de uma expressão lógica (linha 1). Consequentemente, uma projeção do sistemas sem o código anotado apresentará erros de sintaxe. No segundo exemplo, o comando `return` de um método `foo` foi anotado (linha 3). Da mesma forma, após a remoção do código anotado, um erro de tipo será relatado (no `return` statement). Finalmente, a Figura 3.6(c) mostra um exemplo onde não é seguro aplicar uma expansão a partir do corpo de uma estrutura `if` (linha 2) para sua expressão (linha 1). A razão

[3]Propagação de cores e expansões não são passos sequenciais. Pelas regras E4 e E5, expansões de cores pode disparar propagações de cores, que consequentemente podem permitir novas expansões e assim sucessivamente

$$BodyExpansion() = \qquad (E1)$$
$$s = [\texttt{if(exp) stm}] \wedge color(\texttt{exp}, c) \rightarrow cide(s, c);$$
$$s = [\texttt{while exp stm}] \wedge color(\texttt{exp}, c) \rightarrow cide(s, c);$$
$$s = [\texttt{do stm while exp}] \wedge color(\texttt{exp}, c) \rightarrow cide(s, c);$$
$$s = [\texttt{case(exp) \{stm\}}] \wedge color(\texttt{exp}, c) \rightarrow cide(s, c);$$
$$s = [\texttt{switch(exp) \{stm\}}] \wedge color(\texttt{exp}, c) \rightarrow cide(s, c);$$
$$s = [\texttt{for(e}_1\texttt{;e}_2\texttt{;e}_3\texttt{) stm}] \wedge color(\texttt{e}_1, c) \wedge color(\texttt{e}_2, c) \wedge color(\texttt{e}_3, c) \rightarrow cide(s, c);$$

$$ExpExpansion() = \qquad (E2)$$
$$s = [\texttt{if(exp) stm}] \wedge color(\texttt{stm}, c) \wedge free(\texttt{exp}) \rightarrow cide(s, c);$$
$$s = [\texttt{while exp stm}] \wedge color(\texttt{stm}, c) \wedge free(\texttt{exp}) \rightarrow cide(s, c);$$
$$s = [\texttt{do stm while exp}] \wedge color(\texttt{stm}, c) \wedge free(\texttt{exp}) \rightarrow cide(s, c);$$
$$s = [\texttt{if(exp) s}_1\texttt{ else s}_2] \wedge color(\texttt{s}_1, c) \wedge color(\texttt{s}_2, c) \wedge free(\texttt{exp}) \rightarrow cide(s, c);$$
$$s = [\texttt{case(exp) \{stm\}}] \wedge color(\texttt{stm}, c) \wedge free(\texttt{exp}) \rightarrow cide(s, c);$$
$$s = [\texttt{switch(exp) \{stm\}}] \wedge color(\texttt{stm}, c) \wedge free(\texttt{exp}) \rightarrow cide(s, c);$$
$$s = [\texttt{for(e}_1\texttt{;e}_2\texttt{;e}_3\texttt{) stm}] \wedge color(\texttt{stm}, c) \wedge free(\texttt{e}_1, \texttt{e}_2, \texttt{e}_3) \rightarrow cide(s, c);$$

$$StmExpansion() = \qquad (E3)$$
$$s = [\texttt{else stm}] \wedge color(\texttt{stm}, c) \rightarrow cide(s, c);$$
$$s = [\texttt{return exp}] \wedge color(\texttt{exp}, c) \rightarrow cide(s, c);$$

$$MethExpansion() = \qquad (E4)$$
$$s = [\texttt{t m(...) \{stm\}}] \wedge color(\texttt{stm}, c) \rightarrow ColorPropagation(m, c);$$

$$ClassExpansion() = \qquad (E5)$$
$$s = [\texttt{class t \{ members \}}] \wedge color(\texttt{members}, c) \rightarrow ColorPropagation(t, c);$$

$$AssignExpansion() = \qquad (E6)$$
$$s = [\texttt{i = exp;}] \wedge color(\texttt{i}, c) \rightarrow cide(s, c);$$

Figura 3.5: Regras para expansão de cores

é que essa expressão pode gerar efeitos colaterais. De fato, no exemplo, a função `bar()` atualiza o valor da variável `y`, que é usada por um trecho de código não anotado após a estrutura `if` (linha 4).

Nas situações mencionadas a aplicação de qualquer tipo de expansão automática é uma tarefa desafiadora, isto é, dependerá do valor da variável `k` (Figura 3.6(a)), da computação realizada pelos demais comandos não anotados além do `return` (Figura 3.6(b)) ou de uma análise complexa para detectar se `bar()` é uma função sem efeitos colaterais (Figura 3.6(c)).

Quando as anotações inseridas levam a erros de sintaxe/tipo (ou quando não é trivial

(a)	(b)	(c)

`1: if ( option == k) {`
`2:    ....`
`3: }`
	`1: T foo() {`
	`2:   ...    // no returns`
	`3:   return t;`
	`4: }`
		`1: if (bar()) {`
		`2:     stm;`
		`3: }`
		`4: x= y; // bar() updates y`

Figura 3.6: Exemplos de fragmentos de programas onde anotações podem gerar erros sintáticos (a), erros de tipo (b) ou quando não é seguro aplicar uma expansão de cor (c). Os retângulos indicam código anotado pelas fases de propagação e expansão.

expandir uma anotação), o algoritmo proposto gera uma mensagem explicando o problema. Além disso, é sugerida uma expansão padrão para tratar o problema detectado, como descrito na Tabela 3.4. Por exemplo, a ferramenta pode sugerir a anotação de toda uma expressão – para evitar um erro de sintaxe como mostrado na Figura 3.6(a) – ou de todo um método – para evitar um erro de tipo como apresentado na Figura 3.6(b). Se o desenvolvedor aceitar as sugestões, a ferramenta automaticamente aplica tais expansões ao código. Finalmente, é importante mencionar que expansões semi-automáticas acontecem somente quando nenhuma expansão automática pode ser aplicada.

	Definição	**Expansão Padrão**
SE1	Apenas partes de uma expressão foram anotadas com a cor c	Anotar toda a expressão com c
SE2	Os comandos `return` de um método foram anotados com a cor c mas o método tem outros comandos que não foram anotados com c	Anotar todo o corpo do método com c
SE3	Em uma chamada do método m, um parâmetro de chamada foi anotado com a cor c mas o respectivo parâmetro formal não foi anotado	Anotar toda a chamada do método com c
SE4	O lado direito de uma expressão foi anotado com a cor c mas o lado esquerdo não foi anotado	Anotar o lado esquerdo e suas referências com a cor c
SE5	A expansão de cores E2 (Figura 3.5) não foi aplicada (usando a cor c) pois não foi possível determinar se a expressão `exp` é livre de efeitos colaterais	Anotar `exp` com c

Tabela 3.4: Expansões Semi-automáticas

Como descrito pela expansão SE5, a decisão de anotar expressões com efeitos colaterais é deixada a cargo do responsável pela extração da linha de produtos. As razões são simples. Primeiramente, a detecção estática de expressões com efeitos colaterais é naturalmente conservadora (na presença de chamada de métodos provenientes de bibliotecas externas,

chamadas dinâmicas de métodos, chamada de métodos nativos etc. [LCC+05])[4]. Segundo, porque efeitos colaterais podem ajudar ou atrapalhar. Por exemplo, podem impactar somente no código da *feature* sem interferir nas demais ou no núcleo do sistema.

3.3 Processo de Extração

O algoritmo proposto foi implementado como uma extensão da ferramenta CIDE. A implementação assume que os desenvolvedores seguirão o processo iterativo apresentado na Figura 3.7 para extração de uma LPS.

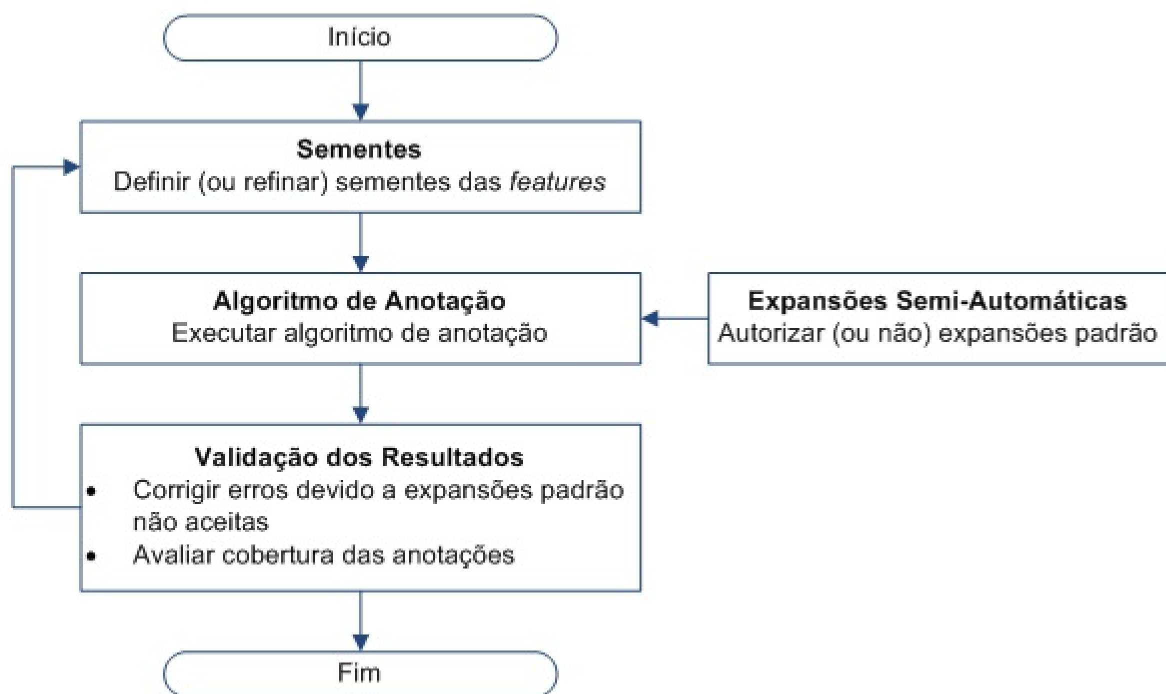

Figura 3.7: Fluxograma do Processo Iterativo de Extração

Primeiro, o desenvolvedor deve definir as sementes da *feature*. Para isso, será necessário, por exemplo, ler a documentação do sistema, entrevistar os desenvolvedores responsáveis pelo código da *feature*, navegar pelo código fonte, estudar as funcionalidades do programa fornecidas pela *feature*, entre outras análises. Após a escolha das sementes – ou pelos menos a seleção das primeiras candidatas – deve-se executar o algoritmo proposto. Durante a execução, o algoritmo poderá perguntar sobre as expansões semi-automáticas

[4]A implementação atual utiliza uma abordagem bastante conservadora para detecção de efeitos colaterais. Basicamente, a função `free(exp)` – chamada pela regra E2 da Figura 3.5 – retorna `true` apenas para expressões que não possuem qualquer chamada de método.

detectadas. Nesse caso, o desenvolvedor deve decidir quando a expansão padrão sugerida será aceita ou não.

Após a execução, a ferramenta apresenta um relatório sobre as expansões padrão que não foram aceitas. Nesse caso, o desenvolvedor deve analisar manualmente o código fonte para corrigir erros de sintaxe ou de tipo provenientes das expansões não aceitas. Por exemplo, no estudo de caso do ArgoUML – a ser descrito com mais detalhes no Capítulo 4 – a ação padrão associada a uma expansão SE4 não foi aceita para o código a seguir:

```
1: cls= org.apache.log4j.Logger.class;
2: ....
3: cls= Class.forName("org.netbeans.api.MDRManager");
```

Nesse exemplo, a projeção do sistema sem o código marcado (dentro do quadro, na linha 1) resultará em um erro de sintaxe. Por outro lado, a ação padrão para essa expansão é "anotar o lado esquerdo e suas referências". Nesse caso em particular, essa ação implicaria na anotação da variável `cls` no lado esquerdo da atribuição (linha 1). No entanto, não se pode propagar a anotação para os usos futuros de `cls`. Por exemplo, na linha 3, `cls` é usada novamente para armazenas outra `Class`. Por essa razão, nesse exemplo, o desenvolvedor não aceitou a ação padrão sugerida pelo algoritmo. Em vez disso, ele anotou somente a atribuição na linha 1.

Finalmente, o desenvolvedor deve avaliar se o código da *feature* foi completamente marcado. O grau de cobertura atingido pelo algoritmo proposto depende exclusivamente das sementes definidas. Basicamente, o algoritmo apenas garante que qualquer código relacionado à *feature* que possa ser alcançado pelas sementes definidas será marcado. No entanto, podem existir fragmentos de código inalcançáveis a partir das sementes inicialmente propostas. Para determinar se isso ocorreu, o desenvolvedor pode, por exemplo, executar o sistema para verificar se todas as funcionalidades associadas à *feature* foram realmente desabilitadas. Se existirem fragmentos de código relevantes que não foram anotados, o desenvolvedor deve redefinir as sementes fornecidas e repetir o processo.

3.4 Implementação

O algoritmo para anotação semi-automática proposto nesta dissertação foi implementado como uma extensão da ferramenta CIDE[5]. Optou-se pela utilização da versão 1.2.0 (*pure-Java version*) como base para a implementação do algoritmo, por ser a versão mais

[5]Disponível em: http://fosd.de/cide

estável da ferramenta durante o período em que se realizou este trabalho. Conforme mencionado na Seção 2.3.2, a ferramenta CIDE é um *plug-in* para a IDE Eclipse versão 3.x, escrito em linguagem Java. A versão adotada suporta somente a anotação de projetos implementados em Java.

Concentrou-se toda a implementação do algoritmo apresentado neste capítulo em um único pacote denominado `coloredide.automation`, adicionado ao projeto da ferramenta CIDE. Esse pacote é composto por quatro classes, descritas a seguir:

- `FeatureExtractionAction`: Classe principal do projeto, que responde ao evento do menu da ferramenta CIDE criado para executar o algoritmo de anotação proposto. Os principais métodos responsáveis pela varredura e anotação do código foram implementados nessa classe, como mostra a Figura 3.8 (esses métodos serão detalhados adiante nesta seção).

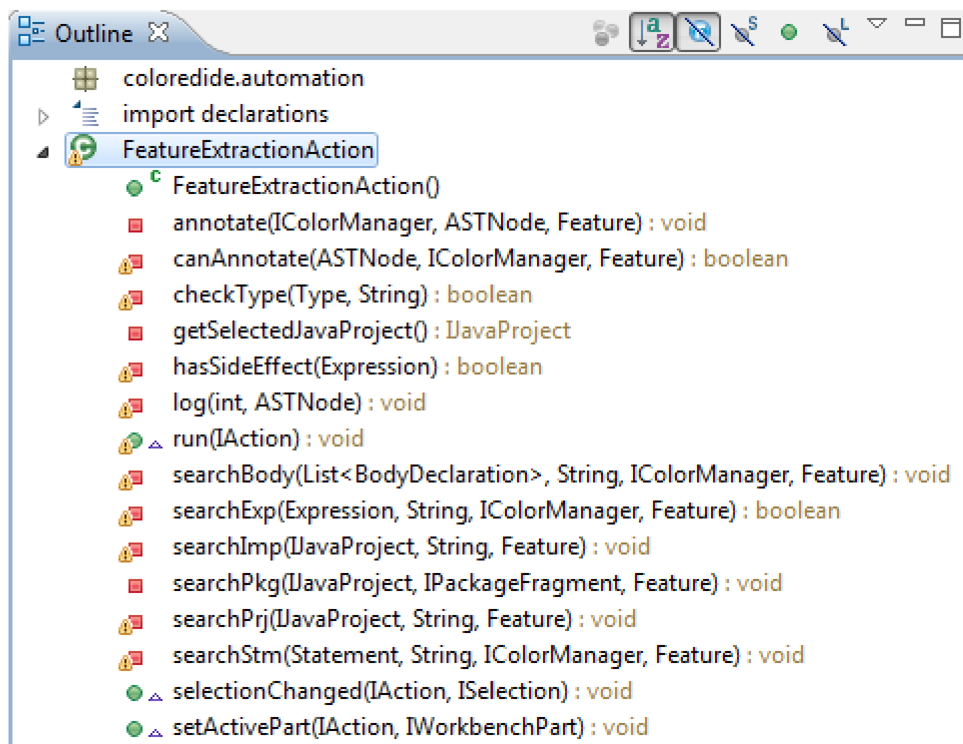

Figura 3.8: Métodos da classe FeatureExtractionAction

- `FeatureExtractionWizard`: Classe que implementa a rotina principal da interface gráfica apresentada para o usuário quando a ferramenta é acionada.

- `FeatureExtractionWizardPage1`: Classe responsável pela implementação da primeira página apresentada na interface gráfica da ferramenta proposta, onde o usuário

informa o nome da *feature* que será extraída e seleciona a cor que será usada na anotação do código, como mostra a Figura 3.9.

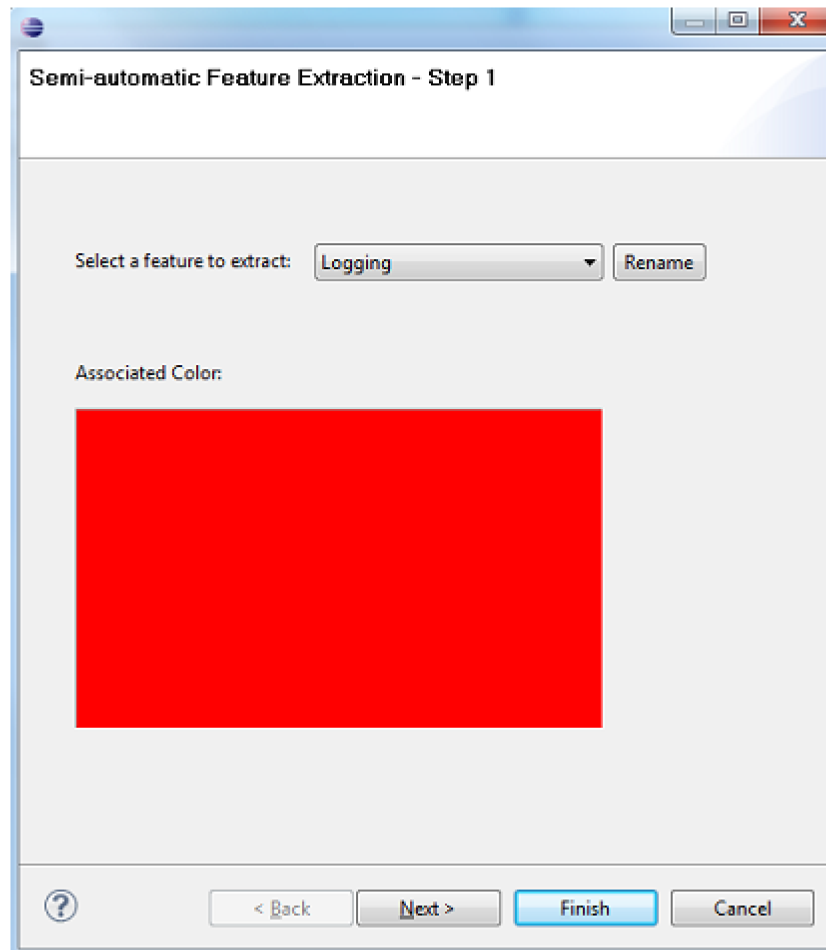

Figura 3.9: Primeira página da interface gráfica da ferramenta

- `FeatureExtractionWizardPage2`: Classe responsável pela implementação da segunda página apresentada na interface gráfica da ferramenta proposta, onde o usuário seleciona as sementes que serão usadas pelo algoritmo de anotação. Todos os membros do projeto são mostrados em forma de árvore, respeitando a seguinte hierarquia: pacotes, classes, atributos e métodos. O usuário pode selecionar um ou mais elementos da árvore, como mostrado na Figura 3.10.

A classe `FeatureExtractionWizard` contém os métodos responsáveis pela implementação das regras propostas na Seção 3.2. A implementação baseia-se essencialmente em uma navegação pela estrutura do programa alvo em busca de referências às sementes selecionadas. A busca é realizada por meio de um caminhamento pela AST (Árvore de Sintaxe Abstrata), gerada a partir do código fonte do sistema alvo. Para gerar a AST, utilizou-se

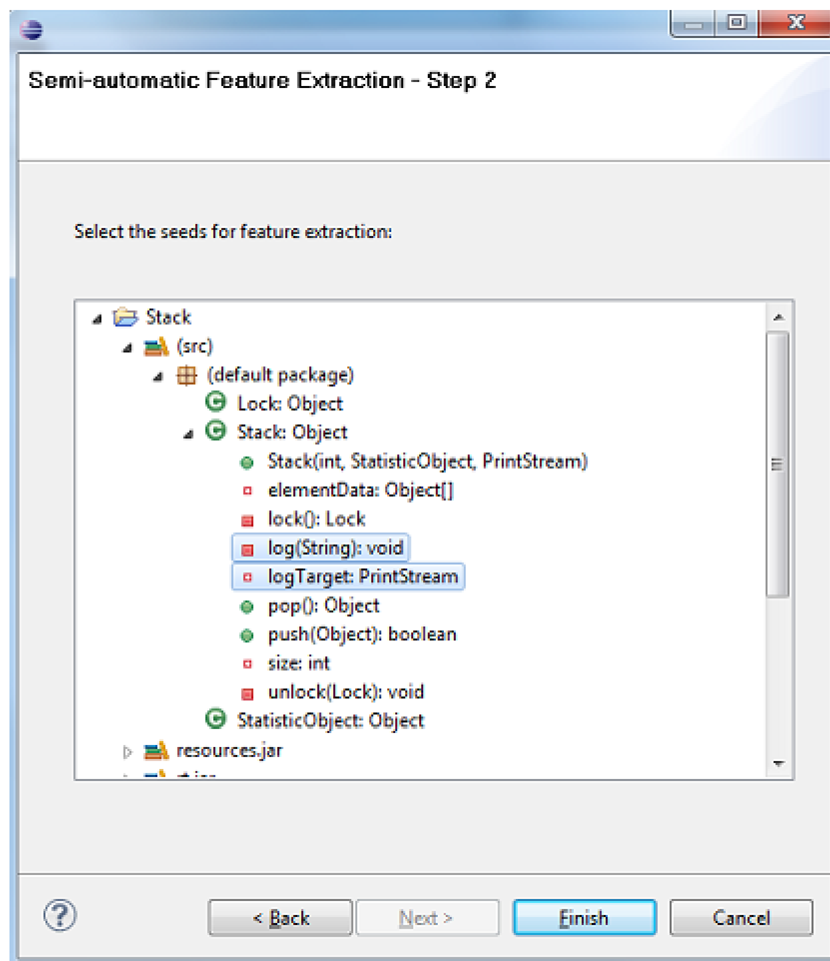

Figura 3.10: Segunda página da interface gráfica da ferramenta

a API de desenvolvimento da IDE Eclipse, que contém classes que implementam *parsers* capazes de gerar uma árvore de sintaxe para arquivos fonte escritos em Java. A ferramenta CIDE implementa a interface `ASTNode` adicionando ao nodo um novo atributo que armazena uma lista de cores. Assim, ao se adicionar uma cor a um determinado nodo da árvore, o fragmento de código correspondente é exibido no editor de textos da ferramenta com uma cor de fundo que é resultado da mistura das cores desse nodo.

A Figura 3.11 mostra a AST estendida pela ferramenta CIDE. A AST mostrada nesse exemplo corresponde ao código da classe `Stack`, apresentado na Figura 2.8. Cada nodo é de uma classe equivalente ao elemento do programa que ele representa. Existem classes de nodos para representar qualquer construção descrita pela gramática da linguagem Java, sendo que todas elas são subtipos derivados direta ou indiretamente da classe `ASTNode`. A raiz da árvore é sempre um nodo da classe *CompilationUnit*, ou seja, todos os elementos do arquivo fonte serão seus descendentes. No exemplo apresentado na figura,

pode-se verificar que o arquivo possui apenas uma declaração de `import` e uma única classe (*TypeDeclaration*). Essa classe possui nove membros declarados em seu corpo (*BODY_DECLARATIONS*), que representam atributos (*FieldDeclaration*) ou métodos (*MethodDeclaration*). Em destaque está um dos nodos da classe *FieldDeclaration*, correspondente ao atributo `logTarget`. Ele foi colorido com a cor vermelha, que foi associada a *feature Logging*. Pode-se observar também outras informações disponíveis nesse nodo, tais como: tipo do atributo, modificadores, inicializadores, dimensões e o nome qualificado. Essas informações podem variar de acordo com a classe do nodo.

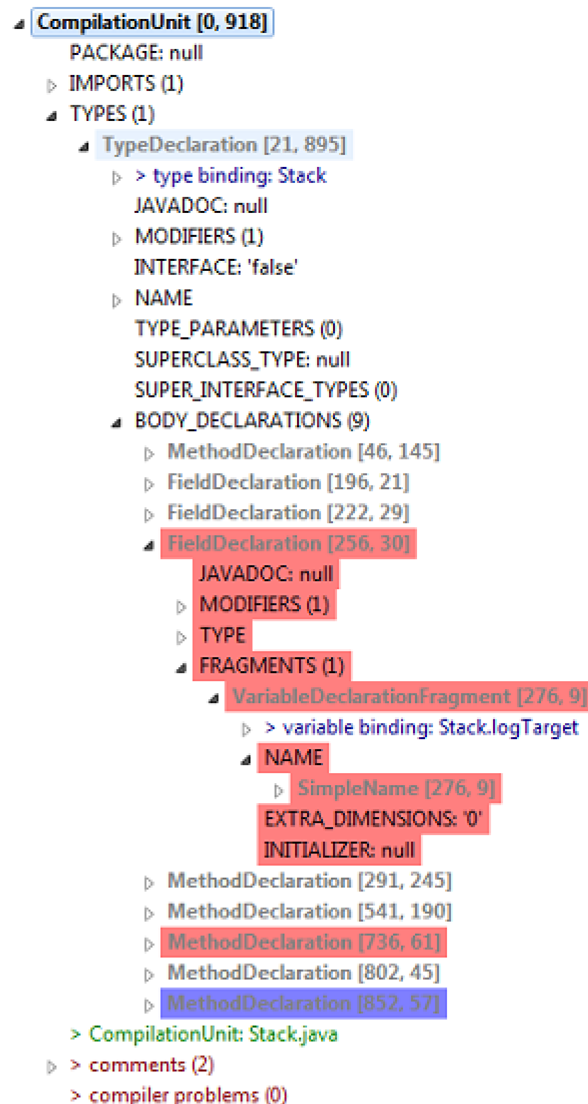

Figura 3.11: AST da classe `Stack`

Uma vez que os atributos de um nodo e o acesso a seus descendentes dependem exclusivamente do seu tipo, é necessário tratar cada tipo de nodo de forma independente.

Para isso, foram criadas funções recursivas que realizam uma busca em profundidade pelos nodos da AST. Cada função é responsável por realizar a busca em determinadas classes de nodos e suas respectivas subárvores, que podem ser: pacotes, elementos do projeto, corpo de comandos, expressões e *imports*. As seguintes funções foram implementadas:

- `searchPrj`: Função chamada pela rotina principal do algoritmo de anotação. Seu objetivo é listar todos os tipos (classe ou enum) declarados em cada pacote do projeto (incluindo pacotes declarados nas bibliotecas importadas pelo projeto). Cada tipo encontrado é passado como parâmetro para a função `searchBody`.

- `searchBody`: Função que realiza a busca pelos elementos que compõem o corpo de uma classe, que podem ser: métodos (*MethodDeclaration*), atributos (*FieldDeclaration*), inicializadores estáticos (*Initializer*) ou outras classes (*TypeDeclaration*). Nodos das classes *MethodDeclaration* e *Initializer* são passados como parâmetro para a função `searchStm`. Nodos da classe *TypeDeclaration* são passados recursivamente para a função `searchBody`. Caso o nodo seja da classe *FieldDeclaration*, a declaração é analisada e as expressões de inicialização dos atributos (caso existam) são passadas para a função `searchExp`.

- `searchStm`: Função que realiza a busca nos comandos encontrados dentro de um bloco, entre eles: atribuições, declarações de variáveis locais, chamadas de métodos, estruturas condicionais (`if`, `switch`), estruturas de repetição (`for`, `enhancedfor`, `while`, `do-while`), tratamento de exceções, comandos de retorno e outros blocos. Quando um comando possui uma ou mais expressões (exemplo: `for`), a função `searchExp` é chamada para cada expressão. O mesmo acontece para os blocos de uma estrutura (exemplo: `if`), quando a função `searchStm` é chamada recursivamente para cada um de seus blocos.

- `searchExp`: Função que realiza a busca pelos termos de uma expressão. Expressões na linguagem Java são estruturas que podem representar desde elementos simples, como identificadores de variáveis, constantes e chamadas de função, até construções mais elaboradas, como expressões pós-fixas, pré-fixas, infixas, condicionais (exemplo: operador ternário), `cast`, `instanceof`, instanciação de classe etc.

- `searchImp`: Função que procura por referências das sementes nas declarações de importação de cada arquivo do projeto.

- `searchPkg`: Função que, a partir dos pacotes fornecidos como semente, adiciona todas as classes e interfaces pertencentes a esses pacotes também como semente.

Toda vez que a busca pela AST encontra um elemento que referencia alguma das sementes fornecidas pelo usuário, o respectivo nodo é anotado, isto é, marcado com a cor correspondente à *feature* que está sendo extraída. A coloração do nodo é realizada por meio da chamada ao método `annotate`. Em certas situações o nodo não precisa ser colorido, pois um de seus parentes na hierarquia da árvore já foi colorido previamente. Essa verificação é realizada pela função `canAnnotate`. A implementação dessas funções foi possível devido à interface disponibilizada pela ferramenta CIDE, que contém os seguintes métodos:

- `boolean IColorManager.addColor(ASTNode n, Feature c)`: Adiciona a cor c à lista de cores do nodo n. Retorna *true* em caso de sucesso; *false* caso contrário.

- `boolean IColorManager.hasColor(ASTNode n, Feature c)`: Retorna *true* caso exista a cor c na lista de cores do nodo n; *false* caso contrário.

- `boolean IColorManager.removeColor(ASTNode n, Feature c)`: Remove a cor c da lista de cores do nodo n. Retorna *true* em caso de sucesso; *false* caso contrário.

Para auxiliar na implementação das regras E2 e SE5, definidas na Seção 3.2 deste capítulo, foi criada uma função denominada `hasSideEffect` cujo objetivo é verificar se uma dada expressão (isto é, um nodo da classe *Expression*) pode gerar algum efeito colateral. A abordagem adotada para realizar essa verificação foi a mais conservadora, isto é, considera-se que uma expressão gera efeitos colaterais se: faz chamadas a métodos, faz uso de algum dos operadores de atribuição (ex.: `=`, `+=`, `-=` etc.), faz uso de operadores de incremento ou decremento (ex.: `++` e `--`).

A ferramenta implementada é capaz de tratar a maioria das construções gramaticais da linguagem Java. No entanto, existem limitações no tratamento de algumas estruturas, tais como: tipos enumerados (`enum`), inicialização de arranjos e sobrecarga de construtores. A versão atual da ferramenta possui cinco classes e um total de 1.562 linhas de código.

3.5 Comentários Finais

Neste capítulo foi apresentado um algoritmo semi-automático para extração de *features* de sistemas legados. Formalizou-se o conceito de *feature* opcional e discutiu-se as limitações da solução proposta em tratar dependências entre *features*. As duas fases em que o algoritmo se divide foram detalhadas: propagação e expansão de cores, assim como as regras e funções utilizadas por cada uma delas. Foram apresentadas também regras que demandam a intervenção do usuário, chamadas de expansões semi-automáticas. Tais

regras são disparadas quando o algoritmo não é capaz de inferir se uma expansão pode ser realizada com segurança.

No final do capítulo, foram apresentados detalhes de uma ferramenta que implementa o algoritmo proposto. Explicou-se o funcionamento da AST implementada pela ferramenta CIDE e as principais funções utilizadas para navegar pelo código fonte do projeto, durante as fases de propagação e expansão das anotações.

Capítulo 4

Avaliação

4.1 Introdução

Investigou-se o uso da abordagem proposta em três sistemas de médio e grande porte. Mais especificamente, a investigação realizada visou responder às seguintes questões:

1. Quantas iterações são necessárias no processo de extração para definir sementes que contemplem a anotação de *features* opcionais nos sistemas avaliados?

2. Qual a quantidade de expansões semi-automáticas solicitadas pelo algoritmo para anotar as *features* opcionais dos sistemas analisados? Dentre elas, as ações padrão sugeridas foram aceitas pelos desenvolvedores?

3. O algoritmo proposto é capaz de anotar com precisão os trechos de código relacionados às *features* que são opcionais nos sistemas analisados? Em outras palavras, espera-se responder se as anotações realizadas pela abordagem proposta correspondem ao código associado à *feature* (precisão) e se elas cobrem todo o código dessa *feature* (cobertura).

Dessa forma, o objetivo principal do estudo de caso realizado consiste em responder se o algoritmo alcança níveis razoáveis de precisão e cobertura, usando sementes simples e um número limitado de expansões semi-automáticas (de preferência, acompanhadas em sua maioria da escolha da expansão padrão, conforme apresentado na Seção 3.2, Tabela 3.4).

Para responder às questões propostas nessa pesquisa, aplicou-se a abordagem para a extração de *features* opcionais nos seguintes sistemas:

- Prevayler, um framework para persistência de objetos em Java[1]. A versão monolítica

[1] `http://www.prevayler.org`, versão 2.3.

do sistema tem 2.974 LOC (excluindo linhas em branco e comentários) e aproximadamente 162 KB de código fonte[2]. O sistema já foi usado em outros estudos de caso para extração de LPS usando AspectJ [GJ05b, KAB07] e AHEAD/Jak [LBL06]. No estudo descrito nessa dissertação, decidiu-se extrair as seguintes *features* opcionais: *Monitor* (responsável pelo logging dos eventos internos), *Replication* (suporta replicação de objetos entre um servidor e múltiplos clientes) e *Censorship* (suporte a *rollbacking* em caso de falhas nas transações).

- JFreeChart, uma biblioteca para construção de diversos tipos de gráficos[3]. A biblioteca tem 91.174 LOC, que correspondem a 7,15 MB de código fonte. No estudo descrito nessa dissertação, foram extraídas duas *features* opcionais da versão monolítica do sistema: *Gráficos de Pizza* (um tipo particular de gráfico) e *Gráficos 3D* (um efeito que se aplica a vários tipos de gráficos, incluindo pizza, barra, linhas e outros).

- ArgoUML, uma ferramenta de código aberto para criação de modelos UML[4]. A ferramenta tem 117.983 LOC, que correspondem a 8,95 MB de código fonte. No estudo, foram extraídas quatro *features* opcionais da versão monolítica do sistema: *Diagrama de Estados*, *Diagrama de Atividades*, *Logging* e *Design Critics* (agentes que executam em paralelo, propondo melhorias nos diagramas UML).

Para aplicar o processo de extração proposto nos sistemas acima, foram seguidos os seguintes passos:

1. Para fornecer uma base de comparação, decidiu-se recorrer a uma anotação manual do código relacionado às *features* desse estudo. Para o Prevayler e o JFreeChart, a extração manual foi realizada pelo próprio autor da dissertação. Para o ArgoUML – o sistema mais complexo considerado – utilizou-se como comparação uma implementação baseada em LPS, que usa diretivas de pré-processamento [CVF11]. Essa implementação foi realizada independentemente por outro desenvolvedor sem conhecimento da abordagem proposta nesse trabalho[5]. Para tornar possível as comparações, implementou-se uma ferramenta que automaticamente importa os códigos

[2]Linhas de código foram calculadas usando o *plug-in* Metrics para IDE Eclipse, versão 1.3.6 (http://sourceforge.net/projects/metrics)

[3]`http://www.jfree.org/jfreechart`, versão 1.0.13.

[4]`http://argouml.tigris.org`, versão 0.28.

[5]A implementação baseada em pré-processador de uma LPS para o ArgoUML – chamada ArgoUML-SPL – está disponível em `http://argouml-spl.tigris.org`.

baseados em pré-processador para a ferramenta CIDE. Essencialmente, essa ferramenta anota os blocos delimitados com `#ifdefs` com cores escolhidas pelo desenvolvedor. A Tabela 4.1 mostra a quantidade de código fonte anotado em cada sistema (em número de bytes). Pode-se observar que as *features* consideradas demandaram anotações em 13,4% do código do Prevayler, 7,4% do JFreeChart e 20,0% do ArgoUML.

2. Em todos os casos, antes de iniciar a extração, foi estudada a arquitetura e a documentação do sistema alvo. Além disso, após finalizada a extração, o sistema foi executado diversas vezes para confirmar se as *features* foram extraídas (isto é, se as funcionalidades foram realmente desabilitadas, sem qualquer impacto no funcionamento do núcleo).

Sistema	Tamanho (MB)	*Features*	% Tam.
Prevayler	0,16	Monitor	4,0
		Replication	6,7
		Censoring	2,7
		Total	13,4
JFreeChart	7,15	Pie Charts	5,1
		3D Charts	2,3
		Total	7,4
ArgoUML	8,95	State Diagrams	3,7
		Activity Diagrams	1,7
		Design Critics	13,3
		Logging	1,3
		Total	20,0

Tabela 4.1: Bytes anotados pela extração manual

3. Aplicou-se o processo de extração descrito na Seção 3.3 para anotar as mesmas *features* consideradas nas extrações manuais.

4. Considerando a extração manual como base comparativa, mediu-se a precisão e a cobertura do algoritmo proposto. O objetivo da precisão é medir se o algoritmo é capaz de anotar códigos relevantes (isto é, código marcado na extração manual). Complementando, a cobertura mede se o algoritmo é capaz de cobrir todo o código relacionado à implementação da *feature*. O cálculo da precisão e da cobertura foi realizado da seguinte maneira:

$$precis\tilde{a}o = \frac{n\acute{u}mero\ de\ bytes\ anotados\ em\ ambas\ as\ extra\c{c}\tilde{o}es}{n\acute{u}mero\ de\ bytes\ anotados\ na\ extra\c{c}\tilde{a}o\ semi\text{-}autom\acute{a}tica}$$

$$cobertura = \frac{número\ de\ bytes\ anotados\ em\ ambas\ as\ extrações}{número\ de\ bytes\ anotados\ na\ extração\ manual}$$

No restante deste capítulo, para cada sistema, serão discutidos detalhes do processo de extração e apresentados os resultados alcançados pela abordagem proposta.

4.2 Resultados do Prevayler

Processo de extração: Após navegar pela API do Prevayler, os seguintes pacotes foram selecionados como semente:

- Monitor: pacote `org.prevayler.foundation.monitor`.
- Censorship: pacote `org.prevayler.implementation.publishing.censorship`.
- Replication: pacote `org.prevayler.implementation.replication`.

Ao executar os sistemas de demonstração distribuídos juntamente com o sistema, o desenvolvedor se certificou que as anotações removeram do sistema as *features* mencionadas. Particularmente, mensagens de log não foram mais geradas. Além disso, os trechos de código responsáveis pelo *rollback* foram anotados. Finalmente, o Prevayler possui um demo que utiliza *Replication* para sincronizar o banco de dados de uma aplicação cliente (chamada `MainReplica`) com o banco de dados de uma aplicação servidora. Ao executar uma projeção desse demo sem *Replication*, foi observado que o serviço de sincronização foi suspenso. Particularmente, no console de `MainReplica` foi impressa a seguinte mensagem: *Trying to find server on localhost*. A próxima mensagem a ser apresentada nesse caso – relatando uma conexão bem sucedida com o servidor – não foi exibida. Assim, após a primeira iteração, considerou-se o processo de extração finalizado.

Precisão/Cobertura: A Tabela 4.2 apresenta o número total de bytes anotados nas extrações manual e automática. Pode-se observar que o algoritmo atingiu 100% de precisão para as três *features*. Quanto a cobertura, o algoritmo atingiu 100% para *Monitor* e *Censorship*. Para *Replication*, no entanto, a cobertura foi de 94%. Nesse caso, o código não anotado inclui quatro atributos da classe `PrevaylerFactory` e seus respectivos métodos acessores. Esses atributos armazenam o endereço IP e o número da porta da aplicação

servidora. Uma vez que o demo considerado nesse estudo assume que o endereço do servidor é sempre *localhost*, tais atributos não foram acessados durante sua execução. Por essa razão, o desenvolvedor não foi capaz de detectar a existência de código não marcado.

feature	**Bytes**			**Precisão**	**Cobertura**
	$M - A$	$M \cap A$	$A - M$		
Monitor	0	6.725	0	1	1
Censorship	0	4.393	0	1	1
Replication	715	11.175	0	1	0,94

Tabela 4.2: Bytes anotados no estudo de caso do Prevayler (M= extração manual; A= algoritmo de anotação)

4.3 Resultados do JFreeChart

Processo de Extração: Após analisar a API do JFreeChart, observou-se que os elementos do programa relacionados aos *Gráficos de Pizza* possuem a substring `Pie` em seus nomes. Assim, decidiu-se usar a seguinte expressão regular como semente para essa *feature*: `*Pie*`. Da mesma forma, para definir a semente dos *Gráficos 3D*, verificou-se que o nome dos elementos que fornecem esse tipo de efeito incluem a substring `3D`. Assim, a expressão regular `*3D*` foi usada como semente para essa *feature*.

Antes de iniciar a anotação do código, a ferramenta mostra os elementos do programa que casam com a expressão regular fornecida. O desenvolvedor pode então conferir se esses elementos realmente contribuem para a implementação da *feature* a ser extraída. No experimento, a expressão regular `*Pie*` encontrou 20 classes, 4 interfaces e 19 métodos. A expressão regular `*3D*` encontrou 7 classes, uma interface e 6 métodos. Nos dois casos, todos os elementos encontrados se relacionavam com as respectivas *features*.

Após a primeira execução do algoritmo, nenhum erro de tipo foi reportado. Além disso, após executar projeções do sistema anotado, confirmou-se que as duas *features* consideradas no estudo foram desabilitadas.

Expansões Semi-automáticas: A Tabela 4.3 mostra o número de expansões semi-automáticas reportadas durante o processo de extração. Para a *feature Gráficos de Pizza*, 21 expansões semi-automáticas foram detectadas pelo algoritmo. Para *Gráficos 3D*, 7 expansões semi-automáticas demandaram autorização. Nos dois casos, todas as ações padrão sugeridas pelo algoritmo foram aceitas.

Regras	Gráficos de Pizza	Gráficos 3D
SE1	10	2
SE2	0	0
SE3	8	4
SE4	3	1
SE5	0	0
Total	21	7

Tabela 4.3: Expansões Semi-automáticas para o JFreeChart

Precisão/Cobertura: A Tabela 4.4 apresenta o número de bytes anotados nas extrações manual e automática. Assim como no experimento com o Prevayler, o algoritmo atingiu 100% de precisão. A cobertura foi de 99% para ambas as *features*, uma vez que para um pequeno número de situações o algoritmo não conseguiu atingir o código marcado manualmente. Por exemplo, o algoritmo foi capaz de anotar todos os comandos de um método chamado `isEmptyOrNull` – que verifica se um conjunto particular de gráficos de pizza está vazio ou nulo – exceto um único comando que retorna `true`.

feature	**Bytes**			**Precisão**	**Cobertura**
	$M - A$	$M \cap A$	$A - M$		
Gráficos de Pizza	1.005	383.165	0	1	0,99
Gráficos 3D	382	172.444	0	1	0,99

Tabela 4.4: Bytes anotados no estudo de caso do JFreeChart (M= extração manual; A= algoritmo de anotação)

4.4 Resultados do ArgoUML

Processo de Extração: A Tabela 4.5 descreve as sementes escolhidas no estudo de caso do ArgoUML. Basicamente, o desenvolvedor selecionou como sementes os pacotes responsáveis pela implementação de cada *feature*. Encontrar tais pacotes não foi difícil, uma vez que seus nomes normalmente incluem o nome da *feature*. Por exemplo, os pacotes selecionados para *Design Critics* tem a substring `cognitive` em seus nomes. Na maioria dos casos, uma segunda iteração foi necessária para incluir nas sementes uma classe relacionada a interface gráfica (ex.: `org.argouml.uml.ui.ActionStateDiagram`).

Nesse terceiro estudo, a *feature ActivityDiagram* depende de *StateDiagram*. De fato, diagramas de atividades são normalmente representados como um tipo especial de diagramas de máquinas de estado. Na implementação do ArgoUML, tal dependência se

manifesta nas classes responsáveis pelo diagrama de atividades, que herdam das classes relacionadas ao diagrama de estados.

feature	**Sementes**	**# Iterações**
State Diagram	Dois pacotes e duas classes	2
Activity Diagram	Três pacotes e um classe	1
Design Critics	Dez pacotes e duas classes	2
Logging	Um pacote	2

Tabela 4.5: Sementes das *features* e número de iterações para o estudo de caso do ArgoUML

Expansões Semi-automáticas: A Tabela 4.6 apresenta o número de expansões semi-automáticas (SE) reportadas durante o processo de extração do ArgoUML. Dois fatores devem ser observados a respeito dos valores apresentados. Primeiro, o número de SEs é significativo, pelo menos em termos absolutos (295 SEs, a maioria delas associada à *feature Design Critics*). No entanto, esses números representam, em média, uma SE reportada para cada 6,18 KB de código fonte anotado na extração. Considerou-se essa média perfeitamente aceitável, principalmente quando comparada a quantidade de trabalho necessário para anotar as mesmas *features* manualmente. Segundo, em apenas três casos o desenvolvedor não aceitou as expansões padrão associadas às expansões semi-automáticas encontradas. O número reduzido de expansões padrão não aceitas sugere que tais ações representam a estratégia mais comum para expandir a cor nos arredores de seu contexto. Apesar de acontecer raramente, os resultados apresentados também demonstram que não seria seguro aplicar automaticamente as expansões padrão.

Rules	**State**	**Activity**	**Critics**	**Logging**	**Total**
SE1	20(1)	18(1)	33	2	73
SE2	0	0	0	0	0
SE3	23	7	35	1	66
SE4	49	14	44	1(1)	108
SE5	8	1	24	15	48
Total	100	40	136	19	295

Tabela 4.6: Expansões Semi-automáticas para o ArgoUML (ações padrão não aceitas estão indicadas entre parênteses)

Precisão/Cobertura: A Tabela 4.7 mostra o total de bytes anotados nas extrações manual e automática. Para *Activity Diagrams*, *Design Critics* e *Logging* tanto a precisão quanto a cobertura foram superiores a 89%, o que pode ser considerado um resultado

positivo para uma abordagem semi-automática. Basicamente, a cobertura não foi 100% devido a situações particulares nas quais as sementes escolhidas não foram capazes de atingir todas as partes do código anotado manualmente. Por exemplo, as sementes escolhidas para *Design Critics* não foram capazes de anotar uma classe que implementa um *parser* para ler uma lista de tarefas armazenada em um arquivo XML (uma vez que, a princípio, o *parser* em questão é capaz de ler qualquer arquivo no formato XML). Por outro lado, a precisão não foi de 100% porque em muitas partes do código o desenvolvedor encarregado da extração manual não expandiu as anotações para os arredores de seu contexto. Por exemplo, em muitas situações, todo o corpo de um método foi anotado, mas a anotação não foi expandida para sua declaração e respectivas chamadas. Para se estabelecer um consenso, discutiu-se as divergências encontradas com o desenvolvedor responsável pela anotação manual, que concordou que a expansão das anotações seria a solução recomendada para esses casos. Em suma, a precisão não foi 100% devido a falhas na anotação decorrentes da extração manual, que podem ser consideradas eventos normais em qualquer processo repetitivo conduzido por humanos.

Feature	**KBytes**			**Precisão**	**Cobertura**
	$M - A$	$M \cap A$	$A - M$		
State Diagram	38,4	290,8	42,1	0,87	0,88
Activity Diagram	6,3	142,3	9,4	0,94	0,96
Design Critics	54,5	1.211,7	8,8	0,99	0,96
Logging	3,7	106,8	12,6	0,89	0,97

Tabela 4.7: Bytes anotados no estudo de caso do ArgoUML (M= extração manual; A= algoritmo de anotação)

Para *State Diagrams*, a cobertura foi de 88%, também devido a ineficiência das sementes escolhidas em atingir algumas partes do código da *feature*. A precisão foi de 87%, mas por uma razão diferente. Um vez que *ActivityDiagram* dependem de *StateDiagram*, o algoritmo propagou a cor usada em *StateDiagram* pelos elementos do programa relacionados a *ActivityDiagram*. Por exemplo, a classe `ActivityDiagramRenderer` foi marcada, uma vez que é uma subclasse de `StateDiagramRenderer`. A justificativa para essa propagação é que o algoritmo assume que não é possível derivar um produto sem *StateDiagram*, mas com *ActivityDiagram*. No entanto, pelo uso de operadores lógicos nas expressões `#ifdef`, o desenvolvedor responsável pela extração manual encontrou uma estratégia que permite um produto configurado dessa forma. Primeiro, ele não expandiu a cor usada em *StateDiagram* para classes como `ActivityDiagramRenderer`, o que explica a existência de 42,1 KB de código fonte anotado apenas pelo algoritmo (e também os 87% da precisão). Segundo, usou-se expressões lógicas como `STATE || ACTIVITY` nas diretivas `#ifdef` que

delimitam classes como `StateDiagramRenderer`. Dessa forma, garantiu-se que tais classes seriam incluídas em produtos onde `STATE=FALSE`, mas `ACTIVITY=TRUE`.

4.5 Discussão

Essa Seção resume os resultados obtidos nos estudos de caso:

- A abordagem presume um conhecimento razoável sobre a estrutura do sistema alvo e particularmente sobre a implementação das *features* que serão extraídas. Basicamente, esse conhecimento é requerido para definir as sementes das *features*, para autorizar as expansões semi-automáticas, para avaliar a necessidade de mais iterações no processo de extração e para avaliar se o funcionamento do núcleo está coerente com a LPS. Em outras palavras, assume-se que os desenvolvedores não serão capazes de executar o algoritmo de anotação sem o mínimo conhecimento interno do programa alvo. De fato, é desejável que o processo de extração proposto seja usado pelos próprios desenvolvedores da versão do sistema não baseada em linhas de produtos.

- Conforme mencionado, a escolha das sementes é um processo crítico para o sucesso do algoritmo. O ideal é que seja possível alcançar qualquer código relacionado às *features* sob extração a partir das sementes selecionadas. Além disso, para detectar códigos que não foram anotados, é importante ter *programas clientes* que acessem todos os serviços fornecidos pelas *features*. Por exemplo, a ausência desses programas foi a principal razão da cobertura não ter atingido 100% do código de replicação no estudo de caso do Prevayler. No entanto, como é comum em testes, programas clientes podem mostrar a existência de código que não foi alcançado, mas não podem provar a ausência de código inalcançável.

- As expansões padrão sugeridas para tratar expansões semi-automáticas provaram ser de grande utilidade. De fato, no estudo de caso do ArgoUML, para as 295 expansões semi-automáticas detectadas, 292 tiveram as expansões padrão aceitas pelos desenvolvedores. A explicação para esse grande número é a seguinte: primeiro, as situações em que as expansões padrão das regras SE1 até SE4 são recusadas acontecem normalmente quando múltiplas cores são misturadas nas mesmas construções abstratas do programa. Por exemplo, a rejeição da ação padrão associada à regra SE1 implica que duas ou mais cores foram usadas em uma expressão. Da maneira semelhante, a não aceitação da expansão padrão da regra SE4 pode ser explicada pelo

reúso de uma variável local para armazenar valores gerados pelo código associado a mais de uma *feature*. Em geral, as situações mencionadas denotam códigos que são mais difíceis de entender e manter (em geral práticas de programação que devem ser evitadas, como o reúso de variáveis). Outra explicação está relacionada ao fato que desenvolvedores – pelo menos a equipe envolvida com os três sistemas avaliados – têm uma tendência a evitar o (re)uso de construções abstratas (ex.: expressões, funções, variáveis locais, etc.) para denotar valores associados a diferentes *features*. Finalmente, para as expansões padrão associadas à regra SE5, a explicação é diferente. Nesse caso particular, as expansões padrão são normalmente aceitas porque o uso de expressões que geram efeitos colaterais em estruturas condicionais ou de repetição é considerado uma má prática de programação.

- Como mencionado no Capítulo 3, a abordagem proposta delega aos desenvolvedores responsáveis pela extração as seguintes tarefas: (a) certificar se as *features* que serão extraídas são opcionais (a remoção do seu código não causará impacto no núcleo), (b) garantir que dependências entre *features* opcionais foram tratadas externamente pelos desenvolvedores. Por exemplo, no estudo do ArgoUML, *ActivityDiagram* depende de *StateDiagram*. O algoritmo proposto assume que essa dependência deve ser descoberta e manipulada pelos desenvolvedores da linha de produtos. Nesse caso, os desenvolvedores devem garantir que nunca se derive um produto com *ActivityDiagram* e sem *StateDiagram*. Contudo, não é impossível obter um produto com essa configuração. De fato, o desenvolvedor a cargo da extração manual foi capaz de gerar tal produto. Para isso, ele recorreu ao uso de expressões lógicas em algumas diretivas `#ifdef`. Porém, a solução proposta – mais especificamente a ferramenta CIDE – não é capaz de associar expressões lógicas aos blocos de código que implementam features opcionais.

- Abordagens baseadas em anotação e composição não são mutuamente exclusivas. Particularmente, apenas mecanismos tradicionais de composição – como métodos e classes – oferecem todos os benefícios da programação modular (como desenvolvimento em paralelo, compilação em separado, reusabilidade etc.). Por outro lado, é importante ressaltar que mecanismos baseados em composição mais sofisticados – incluindo aspectos – também têm sua aplicação, particularmente para tratar requisitos transversais homogêneos, como transações [VCFS10]. Finalmente, é importante mencionar que mesmo quando o uso de aspectos é recomendado, o algoritmo de anotação pode ajudar na identificação dos trechos de código candidatos à refatoração para AOP.

- O processo de extração proposto aplica-se na extração da primeira versão baseada em LPS de um sistema existente. Assim, após a extração da primeira LPS, os desenvolvedores devem manter e evoluir as anotações. No caso de mudanças dentro dos limites do código anotado, a ferramenta CIDE automaticamente anota os elementos inseridos ou alterados. No entanto, no caso de mudanças que afetem as áreas não anotadas do código (por exemplo, inserir uma nova chamada para um método relacionado a uma dada *feature*, em uma parte não marcada do código), os desenvolvedores deve anotar manualmente os novos elementos do programa. O mesmo deve acontecer para códigos que evoluíram devido a operações de refatoração.

4.6 Riscos à Validade do Estudo Realizado

Dois grandes fatores podem impactar nos resultados apresentados nesta Seção. Primeiro, os programas escolhidos para estudo podem não representar todos os cenários para extração de uma linha de produto. Particularmente, foram examinados apenas três sistemas, e assim é possível que tenham sido escolhidas acidentalmente *features* que melhor (ou pior) se aplicam aos requisitos do algoritmo proposto. No entanto, a fim de minimizar esse impacto, buscou-se sistemas bem conhecidos e de código aberto. Além disso, para cada sistema, as *features* foram escolhidas com cautela. Especificamente, foram escolhidas *features* funcionais relacionadas ao domínio do núcleo de cada sistema (ex.: gráficos no JFreeChart e diagramas no ArgoUML) ou requisitos não funcionais típicos (ex.: *logging* no Prevayler e no ArgoUML).

Uma outra ameaça relevante no estudo do Prevayler e do JFreeChart foi o fato de a extração manual ter sido conduzida pelo mesmo desenvolvedor que conduziu o experimento com o algoritmo de anotação semi-automática (isto é, um desenvolvedor com conhecimento total a respeito da abordagem proposta e de suas limitações). Para minimizar o impacto nos resultados, foi realizado um terceiro estudo de caso, envolvendo *features* maiores e mais complexas que nos sistemas anteriores. Mais importante, no terceiro estudo (ArgoUML), a abordagem semi-automática foi comparada com uma extração manual realizada independentemente por um desenvolvedor sem qualquer conhecimento a respeito do algoritmo proposto. Finalmente, o algoritmo proposto foi designado para ser usado pelos próprios desenvolvedores do sistema a que se deseja extrair a linha de produto. Conforme discutido na Seção 4.5, deverá ser simples e fácil para eles definirem sementes expressivas para as *features*. Ainda assim, com um número limitado de iterações do processo de extração, foi possível selecionar sementes expressivas (certamente levou-se um tempo maior até o entendimento do sistema do que seria necessário pelos próprios

desenvolvedores).

4.7 Comentários Finais

Neste capítulo foram apresentados três estudos de caso realizados em sistemas de médio e grande porte: Prevayler, JFreeChart e ArgoUML; com o objetivo de validar o algoritmo proposto nesta dissertação. Em cada estudo, discutiu-se as sementes selecionadas, o número de iterações no processo de extração, o número de expansões semi-automáticas (e o número de expansões padrão rejeitadas), a precisão e a cobertura das anotações realizadas pelo algoritmo. Os resultados obtidos pelo algoritmo foram comparados com extrações manuais, conduzidas separadamente.

Discutiu-se ainda os resultados apresentados, os aspectos positivos e as limitações da solução proposta, e os pré-requisitos para a obtenção de bons resultados na execução do processo de extração de *features* proposto.

Capítulo 5

Conclusões

5.1 Contribuições

As principais contribuições desta dissertação são as seguintes:

- Foi proposta uma abordagem semi-automática para anotação de variabilidades em uma linha de produtos. A abordagem utiliza a ferramenta CIDE para anotar linhas de código encarregadas de implementar as *features* de uma determinada LPS. Inicialmente, desenvolvedores precisam fornecer um conjunto de sementes da *feature* e uma cor. A solução proposta propaga essa cor pelos trechos do código que referenciam as sementes informadas, direta ou indiretamente. Quando possível e seguro, a cor é expandida também para o contexto dos trechos previamente anotados. Quando expansões são necessárias, mas não é possível inferir claramente se são seguras, expansões padrão são sugeridas e estarão sujeitas a aprovação pelos desenvolvedores.

- Foi projetada uma ferramenta que implementa o algoritmo para coloração semi-automática do código de *features* proposto na dissertação. Esta ferramenta – incluindo seu código fonte – está publicamente disponível em: `http://www.dcc.ufmg.br/~mtov/cideplus`.

- O algoritmo proposto foi aplicado com sucesso em três sistemas não triviais. Em todos os casos, os resultados obtidos nos cálculos de precisão e cobertura atingiram valores entre 87% e 100%. Os estudos de caso mostraram a importância da seleção das sementes das *features* e a importância de existirem programas cliente capazes de

testar todos os serviços fornecidos pelas mesmas. Particularmente, esses programas de teste são críticos para a detecção de trechos de código não anotados.

- Concluiu-se que o algoritmo proposto fornece graus de automação bem acima das ferramentas existentes para extração de *features*, como AOP-Migrator [BCH+06], FOR [LBL06] e FLiP [SCN+08]. A razão principal é que tais ferramentas são baseadas em linguagens composicionais, como AspectJ (AOP-Migrator e FLiP) e Jak (FOR). Portanto, são afetadas diretamente pelo modelo de granularidade grossa das extensões suportadas por tais linguagens [KAK08]. A fim de manter esse modelo em conformidade com o código orientado por objetos, abordagens composicionais requerem transformações no código, as quais não podem ser automatizadas facilmente por uma ferramenta [NV09, NOV09]. De acordo com a pesquisa realizada nesse trabalho, o algoritmo apresentado é o primeiro algoritmo para extração de *features* que faz uso de técnicas baseadas em anotação de código.

Publicações: Os resultados desta dissertação deram origem às seguintes publicações:

- Virgílio Borges; Marco Túlio Valente. Coloração Automática de Variabilidades em Linhas de Produtos de Software. III Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software (SBCARS), p. 67-80, 2009.

- Virgílio Borges; Rógel Garcia; Marco Túlio Valente. Uma Ferramenta para Extração Semi-automática de Linhas de Produtos de Software Usando Coloração de Código. I Congresso Brasileiro de Software: Teoria e Prática (Sessão de Ferramentas), p. 73-78, 2010.

5.2 Trabalhos Futuros

A implementação atual do algoritmo de anotação proposto não trata *features* associadas aos seguintes elementos da linguagem Java: *exceptions*, *enums* e tipos genéricos. Propõe-se como trabalho futuro o suporte a esses elementos nas próximas versões da ferramenta. No entanto, é importante mencionar que tais limitações não impactaram nos estudos de caso apresentados. Pretende-se também estender a ferramenta e adaptá-la para suportar outras linguagens de programação além da linguagem Java.

Para tratar melhor os problemas relacionados com a evolução do código, planeja-se estender a ferramenta de maneira que as anotações sejam "contínuas", isto é, sempre que um novo código for alcançável pelas sementes definidas, este será automaticamente

anotado.

Bibliografia

[AMTH09] Bram Adams, Wolfgang De Meuter, Herman Tromp, and Ahmed E. Hassan. Can we refactor conditional compilation into aspects? In *8th ACM International Conference on Aspect-Oriented Software Development (AOSD)*, pages 243–254, 2009.

[Ape07] Sven Apel. *The Role of Features and Aspects in Software Development.* PhD thesis, School of Computer Science – University of Magdeburg, 2007.

[Bat04] Don Batory. Feature-oriented programming and the AHEAD tool suite. In *26th International Conference on Software Engineering (ICSE)*, pages 702–703, 2004.

[BCH+05] David Binkley, Mariano Ceccato, Mark Harman, Filippo Ricca, and Paolo Tonella. Automated refactoring of object oriented code into aspects. In *21st IEEE International Conference on Software Maintenance (ICSM)*, pages 27–36, 2005.

[BCH+06] David Binkley, Mariano Ceccato, Mark Harman, Filippo Ricca, and Paolo Tonella. Tool-supported refactoring of existing object-oriented code into aspects. *IEEE Transactions Software Engineering*, 32(9):698–717, 2006.

[Bos01] Jan Bosch. Software product lines: organizational alternatives. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 91–100, Washington, DC, USA, 2001. IEEE Computer Society.

[BSR03] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. In *25th International Conference on Software Engineering (ICSE)*, pages 187–197, 2003.

[CN02] Paul Clements and Linda M. Northrop. *Software Product Lines: Practices and Patterns.* Addison-Wesley, 2002.

[CVF11] Marcus Vinicius Couto, Marco Tulio Valente, and Euardo Figueiredo. Extracting software product lines: A case study using conditional compilation. In *15th European Conference on Software Maintenance and Reengineering (CSMR)*, 2011.

[GJ05a] Irum Godil and Hans-Arno Jacobsen. Horizontal decomposition of prevayler. In *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 83–100, 2005.

[GJ05b] Irum Godil and Hans-Arno Jacobsen. Horizontal decomposition of Prevayler. In *15th Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 83–100, 2005.

[KA09] Christian Kästner and Sven Apel. Virtual separation of concerns – a second chance for preprocessors. *Journal of Object Technology*, 8(6):59–78, 2009.

[KAB07] Christian Kästner, Sven Apel, and Don Batory. A case study implementing features using AspectJ. In *11th International Software Product Line Conference (SPLC)*, pages 223–232, 2007.

[KAK08] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *30th International Conference on Software Engineering (ICSE)*, pages 311–320, 2008.

[Käs07] Christian Kästner. CIDE: Decomposing legacy applications into features. In *11th International Software Product Line Conference (SPLC) - Demonstration*, pages 149–150, 2007.

[KAuR+09] Christian Kästner, Sven Apel, Syed Saif ur Rahman, Marko Rosenmüller, Don Batory, and Gunter Saake. On the impact of the optional feature problem: analysis and case studies. In *13th International Software Product Line Conference (SPLC)*, pages 181–190, 2009.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *15th European Conference on Object-Oriented Programming (ECOOP)*, volume 2072 of *LNCS*, pages 327–355. Springer Verlag, 2001.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming.

In *11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language.* Prentice Hall, 1988.

[Kru01] Charles W. Krueger. Easing the transition to software mass customization. In *4th International Workshop on Software Product-Family Engineering*, volume 2290 of *Lecture Notes in Computer Science*, pages 282–293. Springer, 2001.

[KTA08] Christian Kästner, Salvador Trujillo, and Sven Apel. Visualizing software product line variabilities in source code. In *2nd International Workshop on Visualisation in Software Product Line Engineering (ViSPLE)*, pages 303–313, 2008.

[LAL+10] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 105–114, New York, NY, USA, 2010. ACM.

[LBL06] Jia Liu, Don Batory, and Christian Lengauer. Feature oriented refactoring of legacy applications. In *28th International Conference on Software Engineering (ICSE)*, pages 112–121, 2006.

[LCC+05] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1-3):185–208, 2005.

[LHBC05] Roberto Lopez-Herrejon, Don Batory, and William R. Cook. Evaluating support for features in advanced modularization technologies. In *19th European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *LNCS*, pages 169–194. Springer-Verlag, 2005.

[MLWR01] Gail C. Murphy, Albert Lai, Robert J. Walker, and Martin P. Robillard. Separating features in source code: an exploratory study. In *23rd International Conference on Software Engineering (ICSE)*, pages 275–284, 2001.

[Nor06] Linda M. Northrop. Software product lines: Reuse that makes business sense. *Software Engineering Conference, Australian*, 0:3, 2006.

[NOV09] Marcelo Nassau, Samuel Oliveira, and Marco Tulio Valente. Guidelines for enabling the extraction of aspects from existing object-oriented code. *Journal of Object Technology*, 8(3):1–19, 2009.

[NV09] Marcelo Nassau and Marco Tulio Valente. Object-oriented transformations for extracting aspects. *Information and Software Technology*, 51(1):138–149, 2009.

[Par76] David Lorge Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2(1):1–9, 1976.

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering - Foundations, Principles, and Techniques*. Springer, 2005.

[Pre97] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, pages 419–443. Springer-Verlag, 1997.

[SCN+08] Sérgio Soares, Fernando Calheiros, Vilmar Nepomuceno, Andrea Menezes, Paulo Borba, and Vander Alves. Supporting software product lines development: FLiP - product line derivation tool. In *23rd Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Tool Demonstrations*, pages 737–738, 2008.

[Spe92] Henry Spencer. #ifdef considered harmful, or portability experience with C News. In *USENIX Conference*, pages 185–197, 1992.

[VCFS10] Marco Tulio Valente, Cesar Couto, Jaqueline Faria, and Sergio Soares. On the benefits of quantification in AspectJ systems. *Journal of the Brazilian Computer Society*, 16(2):133–146, 2010.

[ZJ04] Charles Zhang and Hans-Arno Jacobsen. Resolving feature convolution in middleware systems. In *19th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 188–205. ACM Press, 2004.