Trabalho de Conclusão de Curso

Modelagem de um Assistente Inteligente para Auxiliar a Manutenção de Linhas de Produto de Software

Jean Carlos de Carvalho Melo
jean.melo@ic.ufal.br

Orientadores:

Márcio de Medeiros Ribeiro
Evandro de Barros Costa

Maceió, 24 de Fevereiro de 2012

Jean Carlos de Carvalho Melo

Modelagem de um Assistente Inteligente para Auxiliar a Manutenção de Linhas de Produto de Software

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação do Instituto de Computação da Universidade Federal de Alagoas.

Orientadores:

Márcio de Medeiros Ribeiro

Evandro de Barros Costa

Maceió, 24 de Fevereiro de 2012

Monografia apresentada pelo aluno Jean Carlos de Carvalho Melo como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação do Instituto de Computação da Universidade Federal de Alagoas, aprovada pela comissão examinadora que abaixo assina.

Márcio de Medeiros Ribeiro - Orientador
Instituto de Computação
Universidade Federal de Alagoas

Evandro de Barros Costa - Orientador
Instituto de Computação
Universidade Federal de Alagoas

Patrick Henrique Brito - Examinador
Instituto de Computação
Universidade Federal de Alagoas

Examinador2 - Examinador
Instituto de Computação
Universidade Federal de Alagoas

Maceió, 24 de Fevereiro de 2012

Agradecimentos

Em primeiro lugar, louvo a Deus por essa importante conquista. Só o SENHOR conhece todas as provações que enfrentei e, certamente, Ele tem me auxiliado em todos os desafios da minha vida profissional, pessoal e espiritual.

Segundo, a minha linda família, em especial minha mãe, meu pai, minha irmã, meus avós e meus tios, que me apoiaram desde o príncipio e continuam batalhando para oferecer boas condições de estudo para mim. Agradeço também a minha amada, Edinez, pelo ombro amigo e companheiro que me ajudou bastante em todos os momentos.

Durante esses quatro anos de faculdade, conheci pessoas maravilhosas que participaram dessa realização. Sou grato aos membros do GrOW, liderado pelos professores Pacca, Evandro, Ig e Alan, com quem aprendi bastantes lições referente à todos os aspectos da vida. Não poderia deixar de mencionar meus colegas de turma porque juntos atravessamos diversas dificuldades ao longo do curso, mas, por fim, somos vencedores. São eles: Ana, Endhe, Helynne, Márcio, Olavo, Vitor.

Aos meus orientadores Evandro Barros e Márcio Ribeiro pelos incentivos, contribuições e sugestões que proporcionaram a execução deste trabalho.

A todos os professores do Instituto de Computação pela dedicação ao trabalho elevando assim o nome da instituição tão importante para o nosso Estado.

Aos meus amigos de infância e da igreja que sempre acreditaram em mim.

Finalmente, gostaria de ressaltar que este trabalho é um grande marco em minha jornada rumo à excelência profissional. Assim sendo, dedico-o a todos que contribuiram para tanto reconhecendo que é o mínino que posso fazer.

Resumo

Geralmente, Linhas de Produto de Software (LPS) são implementadas utilizando compilação condicional e pré-processadores. As diretivas de pré-processadores, como `#ifdef` e `#endif`, servem para separar as funcionalidades, envolvidas em *features*, da família de *software*. Porém, o uso destas diretivas possuem algumas notáveis desvantagens, por exemplo, ofuscamento, poluição do código e dificulta a separação de interesses. Assim, *Virtual Separation of Concerns* (VSoC), ou Separação Virtual de Interesses, é uma abordagem ferramental que atenua alguns dos problemas que norteiam o desenvolvimento de LPS com compilação condicional. VSoC permite ao programador esconder o código fonte de *features* que não lhe interesse a fim de proporcionar um foco e, consequentemente, maior produtividade em seu trabalho. Contudo, as *features* escondidas possibilitam que uma manutenção no código produza alguns *bugs* visto que estas *features* não serão visualizadas pelos desenvolvedores. A fim de remediar esse problema, *Emergent Interfaces*, ou Interfaces Emergentes, objetiva estabelecer contratos entre *features* evitando que uma manutenção no código possa quebrar outras funcionalidades. Entretanto, esta abordagem ainda é insuficiente atualmente, uma vez que a mesma só consegue capturar dependências simples. Portanto, este trabalho propõe a modelagem de um Assistente Inteligente que funcionará como uma extensão de interfaces emergentes para capturar algumas dependências que podem produzir erros de comportamentos no sistema ou na LPS por algum descuido na manutenção do código. Para isso, será apresentado um conjunto de regras que permite detectar essas dependências. Também, será mostrado como a nossa abordagem funcionará em cenários de manutenção de sistemas reais.

Conteúdo

1 Introdução **1**
1.1 Objetivos . . . 2
1.2 Organização . . . 3

2 Embasamento Teórico **4**
2.1 Linhas de Produto de Software . . . 4
2.1.1 Benefícios . . . 6
2.2 Mecanismos para Implementação de *Features* . . . 6
2.2.1 Herança . . . 7
2.2.2 Compilação Condicional . . . 7
2.2.3 Programação Orientada a Aspectos . . . 8
2.3 Separação Virtual de Interesses . . . 9
2.4 Interfaces Emergentes . . . 10

3 Problemática: Exemplos de Motivação **13**
3.1 Variável não declarada . . . 15
3.2 Variável não usada . . . 16
3.3 Atribuição possivelmente errada . . . 18
3.4 Comportamento modificado . . . 19

4 Solução Proposta **20**
4.1 Regras . . . 23
4.1.1 Variável não declarada . . . 23
4.1.2 Variável não usada . . . 26
4.1.3 Atribuição possivelmente errada . . . 28
4.1.4 Comportamento modificado . . . 29
4.2 Funcionamento Geral . . . 33

5 Trabalhos Relacionados **36**

6 Conclusão **38**

Lista de Figuras

2.1 Variabilidades de um carro. 5
2.2 *Feature model* de uma linha de produtos de carros. 5
2.3 Aceleração de veículos: variabilidades implementadas nas subclasses. 7
2.4 Exemplo de Aspecto. 8
2.5 CIDE no IDE Eclipse. 10
2.6 Ferramenta Emergo em execução. 11

3.1 Variável usada em diferentes *features* 14
3.2 *Bug* no produto *Libxml2* 15
3.3 Correção do *bug* no produto *Libxml2* 16
3.4 *Bug* no editor *Vim* 16
3.5 Correção do *bug* no editor *Vim* 17
3.6 Variável *did_star* sem uso 17
3.7 Atribuição possivelmente errada no editor *Vim* 18
3.8 *Bug* no produto *GTK+* 19

4.1 Fluxo Representativo de Cenários de Manutenção 22
4.2 Fluxo Alternativo com o Assistente Inteligente 22
4.3 *Warning* baseado em Interfaces Emergentes. 24
4.4 Declaração e uso em *features* distintas 25
4.5 Mensagem do Assistente Inteligente. 26
4.6 *Warning* para variável sem uso. 27
4.7 Variável *i* pode não ser usada. 27
4.8 *Bug* no editor *Vim*. 28
4.9 Mensagem para atribuição possivelmente errada. 29
4.10 Manutenção no método *computeLevel*. 30
4.11 Método *setScore* da Classe *NetworkFacade*. 30
4.12 *Bug* no produto *Empathy*. 31
4.13 Superclasse A. 32
4.14 Subclasse B. 32
4.15 Manutenção utilizando *arrays*. 33
4.16 Visão de alto nível do funcionamento do assistente inteligente. 35

Lista de Códigos

2.1 Trecho de código da LPS Lampiro implementado com pré-processadores. . . . 8

Lista de Tabelas

4.1 Sintaxe das Regras 23
4.2 Tabela verdade das funcionalidades A, B e R. 25

Capítulo 1

Introdução

Uma Linha de Produto de Software (LPS) consiste de uma família de sistemas de software que compartilham um conjunto de características comuns e gerenciadas satisfazendo as necessidades de um segmento particular de mercado ou missão, e que são desenvolvidos a partir de artefatos reusáveis e de uma forma preestabelecida [5]. Também pode ser vista como um paradigma para desenvolver aplicações de software utilizando plataformas e customização em massa [17]. Através da reutilização desses artefatos, é possível construir produtos selecionando as funcionalidades específicas de cada sistema de acordo com os requisitos do cliente [17]. Desta forma, as *features* representam as diversas funcionalidades da LPS. Ou seja, *features* são unidades semânticas que podem distinguir cada produto da linha [27].

LPS são frequentemente implementadas utilizando pré-processadores [9] e compilação condicional associando as diretivas como `#ifdef` e `#endif` para envolver o código de cada *feature*. Embora os pré-processadores sejam largamente utilizados, eles possuem várias desvantagens incluindo a falta de separação de interesses [25]. Além disso, eles poluem o código por causa das inúmeras diretivas que entrelaçam variáveis, métodos e etc. A fim de superar isso, pesquisadores propuseram a Separação Virtual de Interesses, do inglês *Virtual Separation of Concerns* (VSoC) [9] como uma forma de permitir os desenvolvedores a esconder o código das *features* que não sejam relevantes para a atividade que o mesmo irá executar. Isto é, o desenvolvedor só precisará se concentrar na(s) *feature(s)* que lhe interesse sem se distrair com as demais [3]. Entretanto, por mais benéfica que seja essa abordagem, ela não é suficiente para prover modularização de *features*, que tem por objetivo compreensibilidade independente de outras *features*, capacidade de mudanças drásticas em determinada *feature* sem afetar as outras e desenvolvimento paralelo [15]. Com a VSoC o desenvolvedor pode, ao prestar alguma manutenção na LPS, introduzir um *bug* em uma certa *feature* que não esteja sendo visualizada por ele. Isso acontece porque as *features* possuem elementos de códigos compartilhados como, por exemplo, variáveis e métodos. E ainda, um *bug* pode só ser detectado após um tempo considerável (em versões posteriores) considerando que este erro apenas esteja presente para uma específica configuração de *features*.

Para solucionar esse problema de modularização de *features*, Interfaces Emergentes (ou *Emergent Interfaces*) [21] dispõe do estabelecimento de contratos entre os elementos de código que compõem as *features* com a finalidade de capturar as dependências existentes entre as mesmas. Interfaces Emergentes porque os contratos entre as *features* são calculados sob demanda e exibidos na forma de interfaces. Assim, essa abordagem carrega consigo as vantagens da VSoC e ainda proporciona que as dependências sejam calculadas sob demanda para o desenvolvedor. Assim, ao vizualizar tais interfaces, o desenvolvedor pode evitar que suas manutenções afetem outras *features* que não são de seu interesse no momento. Contudo, esta abordagem de interfaces emergentes é insuficiente atualmente, uma vez que a mesma ainda deixa de capturar algumas dependências entre *features* porque não desfruta de diferentes tipos de regras (sintática, de tipo, semântica), principalmente semântica. Consequentemente, deixará escapar *bugs* no sistema ou na LPS que poderiam ter sido corrigidos. Assim sendo, o desenvolvedor ainda terá problemas no que diz respeito à produtividade.

Existe uma outra abordagem que verifica se todas as variantes de uma LPS estão funcionando corretamente. Essa abordagem é conhecida como *Safe Composition* [26], ela parte do pressuposto que nem todas as combinações de *features* produz produtos de software corretos. Assim sendo, o *Safe Composition* faz uma análise em toda a linha de produtos utilizando o *feature model* juntamente com suas restrições para garantir que todos os programas que são membros de uma LPS sejam realmente corretos. Porém, isso só acontece após a linha de produto ser codificada ou depois de efetuada a manutenção. Logo, o desenvolvedor pode precisar refazer todas as modificações que fizera anteriormente, perdendo tempo e, consequentemente, diminuindo a produtividade.

Portanto, este trabalho apresenta a modelagem de um Assistente Inteligente baseado em regras a fim de maximizar a produtividade do desenvolvedor oferecendo interfaces mais expressivas à medida que o desenvolvedor for alterando o código. Isto é, o assistente inteligente executará as regras durante a manutenção (nem antes como apresentado por [22], nem tampouco depois por [26]). O assistente funcionará como uma extensão do conceito de Interfaces Emergentes detectando as dependências entre *features*.

1.1 Objetivos

Este trabalho possui os seguintes objetivos:

- Modelar um Assistente Inteligente a fim de auxiliar o desenvolvedor no processo de manutenção;
- Apresentar as regras e o funcionamento geral do assistente;
- Por fim, aumentar a produtividade do desenvolvedor.

1.2 Organização

O trabalho está está organizado da seguinte maneira:

- **Capítulo 2:** Apresenta a fundamentação teórica necessária para permitir o entendimento do contexto deste trabalho.
- **Capítulo 3:** Mostra alguns cenários, extraídos de sistemas reais, que ocorreram no processo de manutenção em *features*. Assim, será esclarecido a importância do tema.
- **Capítulo 4:** Neste capítulo, são apresentadas as propostas do trabalho, as regras concebidas para capturar as dependências entre *features* e o funcionamento geral do assistente inteligente.
- **Capítulo 5:** São apresentados alguns trabalhos relacionados à abordagem proposta.
- **Capítulo 6:** Por fim, as considerações finais são expostas bem como os trabalhos futuros.

Capítulo 2

Embasamento Teórico

O objetivo deste capítulo é apresentar os conceitos necessários para a compreensão deste trabalho. O capítulo divide-se em quatro seções: a Seção 2.1 descreve o conceito de Linha de Produto de Software bem como seus benefícios. Na Seção 2.2 são apresentados alguns mecanismos usados para implementar variabilidades em LPS; são eles: Herança, Compilação Condicional, Programação Orientada a Aspectos. A Seção 2.3 aborda o conceito de Separação Virtual de Interesses mostrando uma ferramenta de suporte (CIDE) para implementação de LPS. Por fim, a Seção 2.4 descreve uma recente abordagem que remedia o problema de modularidade com Separação Virtual de Interesses chamada Interfaces Emergentes.

2.1 Linhas de Produto de Software

Uma Linha de Produto de Software (LPS) consiste em uma família de sistemas [16] que compartilham um conjunto gerenciado de funcionalidades e são desenvolvidos a partir de um núcleo comum de artefatos [5]. Também pode ser vista como um paradigma para desenvolver aplicações de software usando plataformas e customização em massa [17]. As plataformas representam o núcleo de artefatos utilizados na construção de produtos da linha (são reusados em toda a LPS), ao passo que a customização em massa se caracteriza pela produção em larga escala no que tange a variabilidade dos produtos de acordo com os requisitos dos usuários. Desta forma, os produtos derivados de uma LPS possui elementos de variabilidade que os diferenciam entre si. As variabilidades encontradas em cada produto da linha pode ser mapeadas em *features*, pois estas representam as diversas funcionalidades da LPS. Para exemplificar, a Figura 2.1 ilustra algumas variabilidades encontradas no contexto de carro.

Assim sendo, (Kang et al. 1990) [7] propôs um método de análise de domínio, conhecido como FODA (*Feature Oriented Domain Analysis*), que pode ser usado para especificar as partes comuns e as partes variáveis dos sistemas de software que compõe uma família de sistemas. Esta notação traz uma representação semântica que descreve o relacionamento estrutural entre as *features*. O conjunto de *features* é geralmente representado por um *feature model*, que denota

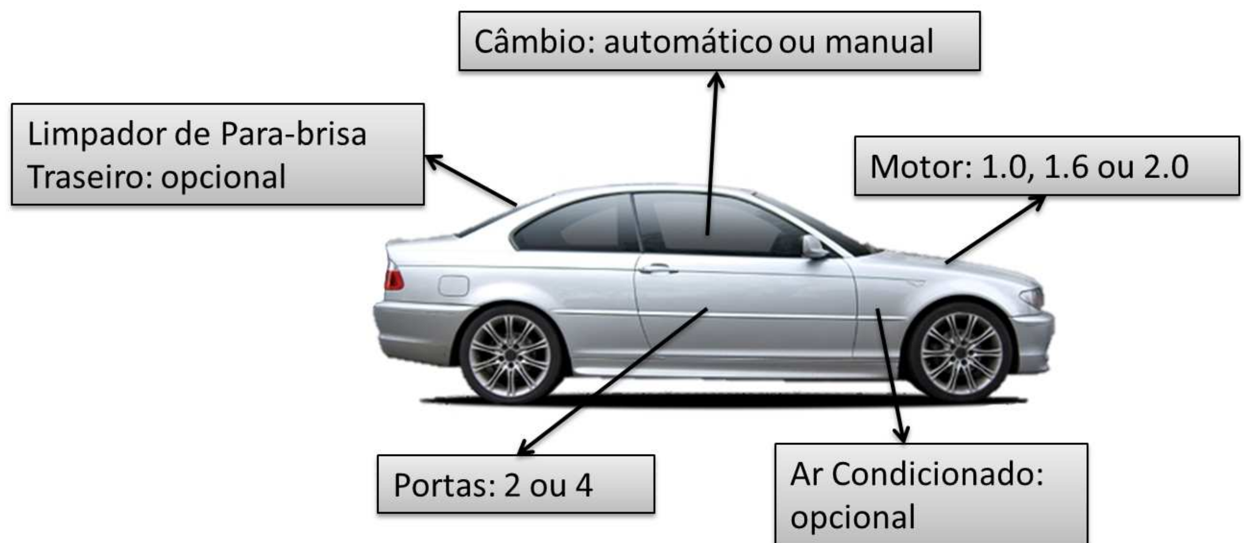

Figura 2.1: Variabilidades de um carro.

o escopo da LPS, restrições e a quantidade de produtos. As *features* possuem diferentes tipos de associação, são eles:

- **Obrigatória:** a *feature* deve estar sempre presente nos produtos da linha.
- **Opcional:** *feature* que pode ou não ser escolhida para compor determinado produto.
- **Alternativa:** só uma *feature* pode existir, ou seja, a escolha de uma causa automaticamente a exclusão de outra(s) alternativa(s) a ela.

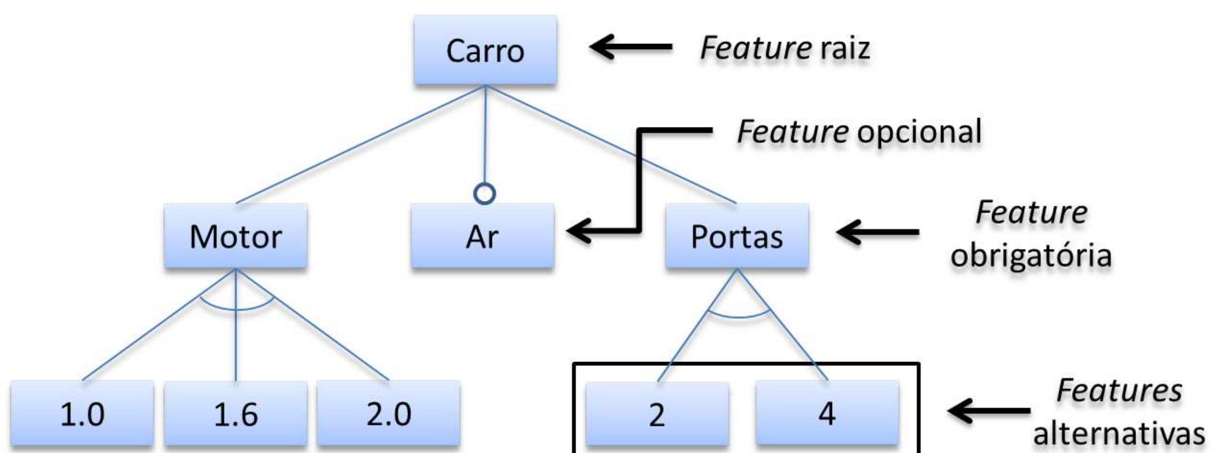

Figura 2.2: *Feature model* de uma linha de produtos de carros.

Além disso, a notação FODA permite modelar regras de dependência entre *features* de duas maneiras: (i) *requires* - significa que se uma *feature* estiver definida, então outra será seleci-

onada também e (ii) *excludes* - significa que a escolha de uma *feature* implica na exclusão de outra.

As variabilidades de um carro exibidas na Figura 2.1 foram simplificadas e mapeadas em um *feature model* baseado na notação FODA. A Figura 2.2 apresenta esse *feature model* de uma LPS de carros.

O elemento `Carro` representa a LPS sendo a *feature* raiz. As *features* `Motor e Portas` são funcionalidades *obrigatórias* da linha de produtos. Ou seja, todos os produtos da linha devem possuir motor e portas. O `Motor` pode ser composto por apenas uma das *features*: `1.0`, `1.6` ou `2.0`. Da mesma forma, o carro só pode ter duas ou quatro portas. Já as *features* denotadas por um arco (sem preenchimento) são mutualmente exclusivas (*alternativas*), como os nós `2` e `4`. Note-se ainda que a *feature* `Ar` é *opcional*, e sua notação é simbolizada por um círculo não preenchido.

2.1.1 Benefícios

A seguir são descritas as principais vantagens do uso de linhas de produtos de software no desenvolvimento de software [17]:

- *Redução do custo de desenvolvimento* - o reuso de artefatos entre produtos distintos promove redução no custo do desenvolvimento.
- *Redução no tempo de entrega (time-to-market)* - inicialmente, o tempo de produção dos artefatos e dos primeiros produtos pode ser alto, porém após a produção o tempo de entrega é reduzido significativamente em comparação com produtos individuais, não pertencendo a família alguma.
- *Maior qualidade* - uma vez produzido o conjunto de artefatos que serão reutilizados, testados e revisados entre os diferentes produtos pertencentes a Linha aumenta acentuadamente as possibilidades de detecção e correção de erros.
- *Redução no esforço de manutenção* - correções ou modificações em um artefato podem se propagar entre os diversos produtos da família de sistemas.
- *Evoluções gerenciáveis* - adaptações/extensões efetuadas em um núcleo de artefatos se refletem automaticamente em todos os produtos da família de sistemas.

2.2 Mecanismos para Implementação de *Features*

Esta seção descreve algumas abordagens usadas para implementar variabilidades em LPS. Primeiro, na Seção 2.2.1, mostra-se o conceito de Herança e como a mesma pode ser utilizada para implementar variações na LPS. A Seção 2.2.2 considera o mecanismo de Compilação Condicional. Por fim, a Seção 2.2.3 detalha sobre Programação Orientada a Aspectos.

2.2.1 Herança

Herança é um relacionamento para representar classes hierarquicamente. Classes podem herdar estado e comportamento de outras classes. A herança pode ser usada como um mecanismo de implementação de varibilidades, uma vez que as funcionalidades básicas estejam contidas nas superclasses e as variabilidades nas subclasses. Dessa forma, a superclasse pode ser reutilizada. Além disso, para cada classe pai pode-se definir diversas variações adicionando *n* classes filhas. As subclasses podem adicionar novos atributos e métodos bem como sobreescrever métodos.

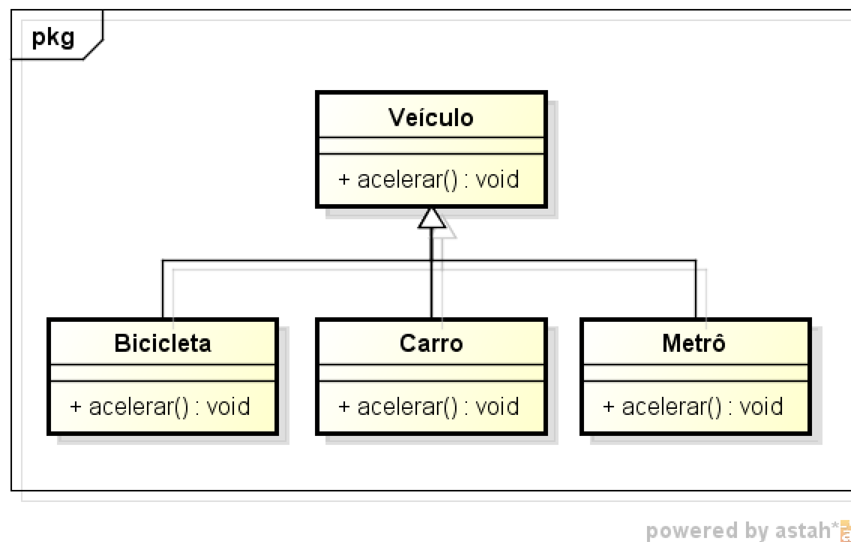

Figura 2.3: Aceleração de veículos: variabilidades implementadas nas subclasses.

A Figura 2.3 ilustra três tipos de veículos com diferentes métodos de aceleração. Por exemplo, a rapidez com a qual a velocidade de uma bicicleta varia é inferior a dos demais veículos em questão. Esta variabilidade pode ser implementada através de subclasses.

Devido ao fato de que a herança pode ter um alto acomplamento, uma mudança na superclasse pode exigir alterações em todas as suas subclasses. E, quanto mais variabilidades implica em uma árvore de herança complexa.

2.2.2 Compilação Condicional

Compilação condicional é um mecanismo bem conhecido usado para implementar LPS levando em consideração as variabilidades em linguagens como C e C++. Compilação condicional habilita trechos de código para serem incluídos ou não no processo de compilação. Basicamente, o pré-processador analisa o código que deve ser compilado ou não baseado nas variáveis. Desta forma, se a variável é verdadeira, então o código será compilado. Caso contrário, o pré-processador irá ignorar o trecho de código e o mesmo não será compilado. Este mecanismo usa um símbolo especial (//#) indicando que a linha de código será pré-processada.

O Código 2.1 ilustra um exemplo comum de técnica anotativa usando diretivas similares ao pré-processador C/C++ para delimitar o código fonte, como o `#ifdef` (linha 3) e o `#endif` (linha 7). Esse trecho de código foi extraído de uma LPS chamada *Lampiro*[1], a mesma implementa o protocolo XMPP de troca de mensagens em J2ME. Neste caso, o método `startCompression` (linha 4) está comentado porque o pré-processador não encontrou a definição da *feature* `COMPRESSION`. Se `COMPRESSION` estivesse sido selecionada, então o código não seria comentado ou removido.

```
1 private SocketChannel channel = null;
2 ...
3 // #ifdef COMPRESSION
4 //@ protected void startCompression() {
5 //@     channel.startCompression();
6 //@ }
7 // #endif
```

Código 2.1: Trecho de código da LPS Lampiro implementado com pré-processadores.

2.2.3 Programação Orientada a Aspectos

Programação Orientada a Aspectos (do inglês, *Aspect-Oriented Programming* (AOP)) é uma abordagem bastante conhecida que tenta solucionar dois problemas clássicos no que tange separação de interesses (ou, *separation of concerns*): **a.** espalhamento (*scattering*) e **b.** entrelaçamento (*tangling*) [11]. Este entrelaçamento e espalhamento tornam o desenvolvimento e a manutenção de sistemas mais difícil.

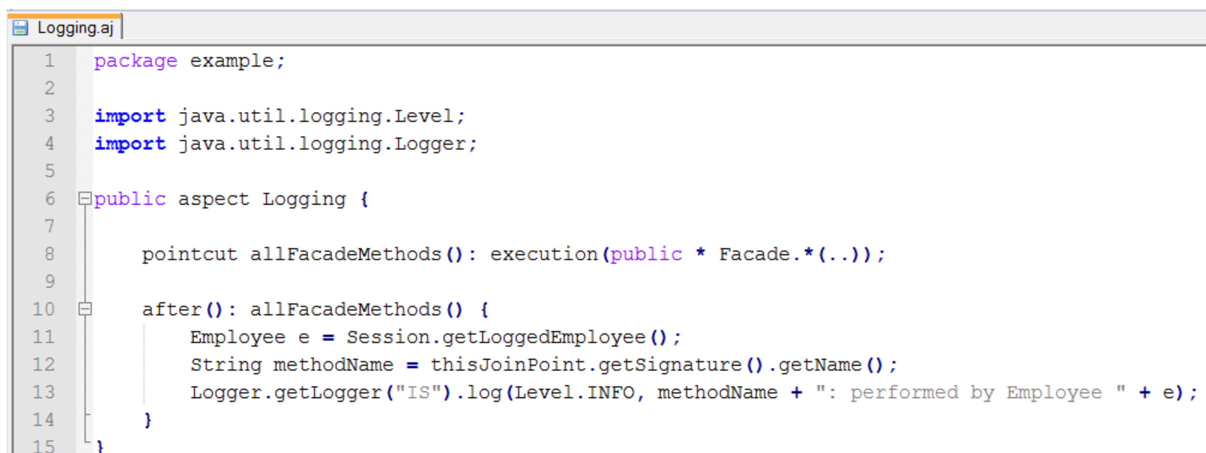

Figura 2.4: Exemplo de Aspecto.

[1] http://www.lampiro.bluendo.com

Em outras palavras, AOP é uma extensão de orientação a objetos que lida com *crosscutting concerns*. *Crosscutting concerns* representam os requisitos não-funcionais do sistema, que normalmente não são encapsulados numa classe ou objeto. Exemplos de *crosscutting concerns* são: transação, logging, tracing, tratamento de exceções, entre outros. A Figura 2.4 mostra um exemplo bem tradicional de aspecto escrito na linguagem AspectJ [10] (uma extensão orientada a aspectos para Java). Esse aspecto *Logging* é executado em todos os métodos da classe *Facade*. O ponto de junção (*join point*) **execution** (linha 8) define onde os aspectos podem executar: durante a execução de métodos. O *pointcut* **allFacadeMethods** captura todas as execuções dos métodos da classe *Facade*. Já o *advice* **after** indica o que será introduzido depois de cada *join point*. A porção de código que deve ser acrescida no fim de cada execução de todos os métodos de *Facade* está definida no *advice* (linhas 11–13).

Assim, AOP aumenta a modularidade separando os *crosscutting concerns* em unidades de modularização chamadas aspectos. Em seguida, o costurador (*weaver*), como o próprio nome já diz, é responsável por unir os aspectos às classes.

2.3 Separação Virtual de Interesses

Separação Virtual de Interesses originário de *Virtual Separation of Concerns* (VSoC) [9] consiste em uma fragmentação simbólica de preocupações uma vez que não há separação física do trecho de código relacionado à(s) *feature(s)* de interesse do desenvolvedor com o restante do código [8]. Todo o código fonte permanece no programa, porém os códigos são separados virtualmente com o auxílio de alguma ferramenta de apoio. A ideia fundamental da VSoC é permitir que o desenvolvedor pode se concentrar apenas nas *features* relevantes a sua tarefa. Assim, os códigos das demais *features* são escondidos visto que não são relevantes para a atual tarefa. Vale salientar que a VSoC é bem parecida com a compilação condicional (ver Seção 2.2.2). Mas, ao invés de utilizar diretivas como `#ifdef` e `#endif`, os trechos de código são destacados de alguma maneira que sinalize as *features*.

O CIDE[2], *Colored Integrated Development Environment* ou Ambiente Colorido Integrado de Desenvolvimento, é uma ferramenta de suporte ao desenvolvedor que provê a separação virtual de interesses através de cores de fundo (do inglês, *background colors*) nas *features* [13]. CIDE é um *plug-in* para o IDE Eclipse que faz a associação de cada *feature* com alguma cor de fundo. Assim, cada *feature* recebe uma cor única. Se um segmento de código possuir mais de uma *feature*, então o CIDE colore o trecho de código com uma mistura de cores das presentes *features*.

A Figura 2.5 extraída do próprio site da ferramenta CIDE ilustra o *plug-in* em ação. Nota-se que há três *features* coloridas com as seguintes cores: amarela, vermelha e azul. Observa-se ainda que a *feature* amarela tem seu código oculto (*hidden code*) e que existe uma sobreposição

[2] `http://www.fosd.de/cide/`

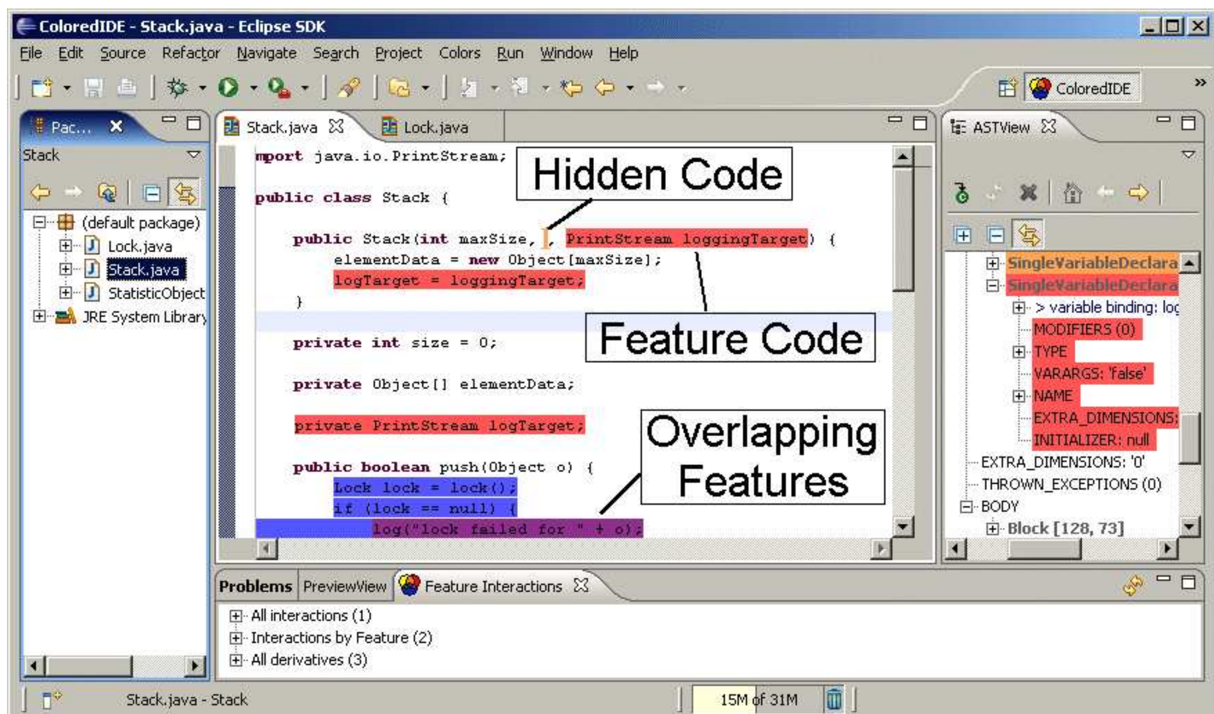

Figura 2.5: CIDE no IDE Eclipse.

de *features* mostrada de roxo - combinação de azul com vermelho.

Entretanto, ao esconder trechos de código que possuem variáveis ou métodos compartilhados entre *features*, o desenvolvedor pode introduzir erros.

2.4 Interfaces Emergentes

Para atacar o problema de modularidade com VSoC (vide Seção 2.3), pesquisadores recentemente propuseram o conceito de Interfaces Emergentes (ou Emergent Interfaces) [22]. A abordagem de modularização emergente de *features* tem como objetivo estabelecer contratos entre os elementos de código que compõem as *features*. A abordagem é emergente porque os contratos entre as *features* são calculados a medida que o desenvolvedor presta manutenção no código e exibidos na forma de interfaces. Interfaces Emergentes mantém os benefícios envolvidos em esconder códigos de *features*, ou seja, as vantagens da VSoC e também permitem estabelecer contratos entre trechos de código de diferentes *features* sob demanda, sem uma estrutura rígida pré-definida. Assim, ao vizualizar tais interfaces, o desenvolvedor pode evitar que sua manutenção afete as demais *features* que não são de seu interesse no momento.

A fim de utilizar essa abordagem ferramental, o desenvolvedor, em primeiro lugar, seleciona um segmento de código que pretende manter. Após a seleção, uma ferramenta como Emergo [3] realiza algumas verificações com o objetivo de identificar as dependências existentes entre o código selecionado e o restante. Em seguida, encontradas as dependências em relação às outras

[3]http://twiki.cin.ufpe.br/twiki/bin/view/SPG/Emergo

features a interface emerge para o desenvolvedor alertando-o de alguma possível inserção de defeito no sistema ou LPS. O Emergo é um *plug-in* do Eclipse que automatiza o cálculo de dependências de *features* em linhas de produto baseadas em pré-processador.

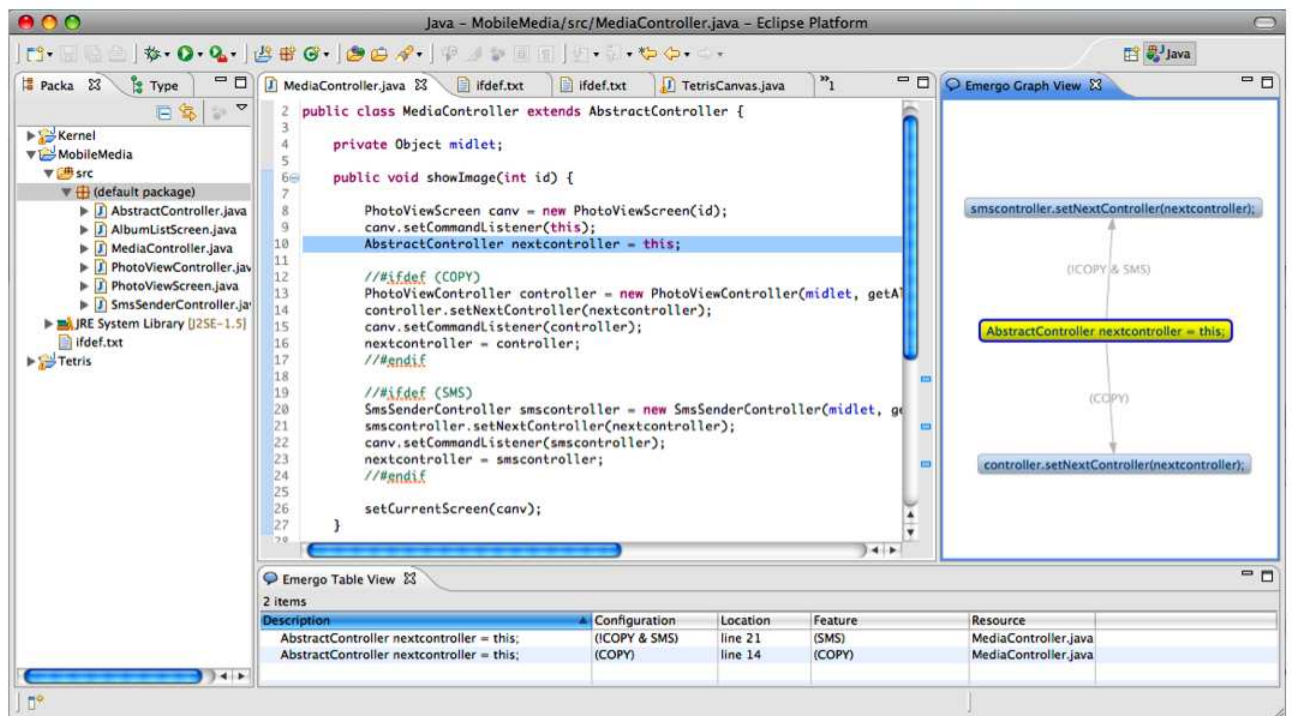

Figura 2.6: Ferramenta Emergo em execução.

Considere, por exemplo, a Figura 2.6, retirada do site da ferramenta Emergo, que apresenta uma utilização do *plug-in*. Percebe-se que o desenvolvedor selecionou o código (linha 10) que contém uma atribuição a variável *nextcontroller*. A figura também mostra duas *features* (COPY e SMS) da linha de produtos *MobileMedia*. A propósito, a *MobileMedia* é uma LPS para aplicações que manipulam fotos, músicas e vídeos em dispositivos móveis. Foi desenvolvida baseada em uma LPS anterior chamada *MobilePhoto* [29]. Após o desenvolvedor ter selecionado o trecho de código que fará manutenção, o Emergo realiza uma computação automática a fim de identificar as dependências entre *features* e provê dois tipos de visualizações para exibir as interfaces emergentes: uma baseada em tabela e outra em grafo. A mensagem também pode aparecer em forma de um *pop-up*.

No entanto, esta abordagem ainda é insuficiente atualmente, uma vez que a mesma ainda deixa de capturar algumas dependências entre *features* porque não desfruta de diferentes tipos de regras (sintática, de tipo, semântica), principalmente semântica. Consequentemente, deixará escapar *bugs* no sistema ou na LPS que poderiam ter sido corrigidos. Assim sendo, o desenvolvedor ainda terá problemas no que diz respeito à produtividade na fase de manutenção do código. Esta pesquisa propõe maximizar a produtividade do desenvolvedor, deixar a interface mais expressiva e concisa. Para isso, será construído um conjunto de regras, especialmente

semânticas, a fim de capturar as dependências entre as *features*. E em seguida, as dependências serão filtradas com o objetivo de expor os acoplamentos mais críticos que o desenvolvedor, eventualmente, não tinha percebido.

Capítulo 3

Problemática: Exemplos de Motivação

A compilação condicional pode reduzir a legibilidade do código, pois, quanto maior for o número de linhas de código mais ilegível o mesmo se tornará. Assim, por ele ficar confuso, dificulta o entendimento devido ao emaranhado de `#ifdefs` entrecortando o código. Além disso, o risco de manutenção aumenta podendo acarretar novos erros e trabalho dobrado. O tempo para realizar a manutenção será afetado causando, consequentemente, uma diminuição na produtividade uma vez que os desenvolvedores precisarão gastar mais tempo para compreender o programa propriamente dito para, depois, fazer as alterações necessárias. Inclusive, correções de erros, implementações de novas funcionalidades, entre outros, cooperam para a improdutividade. Essa poluição de diretivas no programa atrapalha no processo de identificação de *bugs* tanto de natureza sintática quanto, especialmente, semântica onde estes necessitam de atenção diferenciada por serem mais complexos. Isso porque a compilação condicional estar sujeita à penetração de erros sutis que, geralmente, só são descobertos quando é compilado um produto específico da linha de produtos. Vale ressaltar que uma linha de produtos pode possui até 2^n combinações de produtos distintos, sendo n o número de *features* da linha, ou seja, encontrar uma combinação particular pode levar um tempo considerável. Em função destas limitações, a VSoC, como discutido na Seção 2.3, atenua alguns dos problemas da compilação condicional com diretivas. Suas principais características são: (i) uso de *features* escondidas e (ii) o uso de cores de fundo (*background colors*) para cada *feature*. Contudo, as *features* ocultas possibilitam que uma manutenção no código produza alguns *bugs* ao passo que estas *features* não serão vizualizadas pelos desenvolvedores.

Há um aspecto importante para ser analisado a respeito de compilação condicional é, justamente, a implementação e/ou manutenção de *features* de forma paralela pela equipe de desenvolvimento. Nesse tipo de abordagem, a equipe seria dividida em grupos menores, cada qual responsável por manter algumas funcionalidades. Assim sendo, cada desenvolvedor prestaria serviços (desenvolvimento e/ou manutenção) em *features* específicas. Desse modo, ele apenas precisaria conhecer as *features* que lhe interesse sem necessitar entender as demais uma vez que elas estão sob responsabilidade de outros membros da equipe. Porém, caso essas *features*

compartilhem variáveis, métodos ou outras partes de código, aumenta-se o risco de uma simples mudança em certa *feature* afetar várias outras. Aliás, esse tipo de acoplamento é trivialmente encontrado em sistemas **reais**. Para exemplificar, a Figura 3.1 mostra a variável inteira *html* sendo usada nas *features LIBXML_HTML_ENABLED*, *LIBXML_VALID_ENABLED* e *HAVE_-SYS_MMAN_H*. Esse código foi extraído do produto *Libxml2 - The XML C parser and toolkit of Gnome*[1] na *release* 2.7.6.

Libxml2 – Versão 2.7.6

Linhas 160-163

```
160  #if defined(LIBXML_HTML_ENABLED) || defined(LIBXML_VALID_ENABLED)
161  static int html = 0;
162  static int xmlout = 0;
163  #endif
```

Linhas 2178-2187

```
2178  #ifdef LIBXML_HTML_ENABLED
2179  #ifdef LIBXML_PUSH_ENABLED
2180      else if ((html) && (push)) {
2181          FILE *f;
2182
2183  #if defined(_WIN32) || defined (__DJGPP__) && !defined (__CYGWIN__)
2184      f = fopen(filename, "rb");
2185  #else
2186      f = fopen(filename, "r");
2187  #endif
```

Linhas 2209-2214

```
2209  #endif /* LIBXML_PUSH_ENABLED */
2210  #ifdef HAVE_SYS_MMAN_H
2211      else if ((html) && (memory)) {
2212      int fd;
2213      struct stat info;
2214      const char *base;
```

Figura 3.1: Variável usada em diferentes *features*

As seções subsequentes apresentam alguns cenários **reais** que ocorreram devido a falta de modularidade no que tange à compilação condicional. Sem muito esforço, esses tipos de *bugs* são localizados em sistemas como *Vim - the text editor*[2], *GNOME*[3], *The Linux Kernel*[4], *GTK+*[5], entre outros. Assim, os exemplos a seguir foram padronizados, no sentido de exibição, a fim de explicitar melhor as peculiaridades de cada cenário. Desse modo, cada cenário será apresentado ao longo de três versões/*releases* que destacarão o sistema antes, durante e depois do *bug*. Também será discorrido como uma determinada manutenção no código de alguma *feature* afeta as demais e, consequentemente, o sistema como um todo. Assume-se que o desenvolvedor está trabalhando com VSoC - *features* que não são de interesse da manutenção estariam escondidas.

[1] http://xmlsoft.org/
[2] http://www.vim.org/
[3] http://www.gnome.org/
[4] http://www.kernel.org/
[5] http://www.gtk.org/

3.1 Variável não declarada

Este tipo de problema ocorre quando uma variável é declarada em uma determinada *feature*, por exemplo, *feature A* (conquanto ela não seja *mandatory*) e a mesma variável também é usada em outra *feature B*. Isso porque se a *feature A* não for selecionada para um determinado produto da linha e a *feature B* esteja, acontecerá um erro de compilação.

Libxml2 – Versão 2.7.3	Libxml2 – Versão 2.7.6
Linhas 159-162	**Linhas 160-163**
159 `#if defined(LIBXML_HTML_ENABLED) || defined(LIBXML_VALID_ENABLED)` 160 `static int html = 0;` 161 `static int xmlout = 0;` 162 `#endif`	160 `#if defined(LIBXML_HTML_ENABLED) || defined(LIBXML_VALID_ENABLED)` 161 `static int html = 0;` 162 `static int xmlout = 0;` 163 `#endif`
Linhas 2550-2556	**Linhas 2550-2558**
2550 `int saveOpts = 0;` 2551 2552 `if (format)` 2553 `saveOpts |= XML_SAVE_FORMAT;` 2554 2555 `if (output == NULL)` 2556 `ctxt = xmlSaveToFd(1, encoding, saveOpts);`	2550 `int saveOpts = 0;` 2551 2552 `if (format)` 2553 `saveOpts |= XML_SAVE_FORMAT;` 2554 `if (xmlout)` 2555 `saveOpts |= XML_SAVE_AS_XML;` 2556 2557 `if (output == NULL)` 2558 `ctxt = xmlSaveToFd(1, encoding, saveOpts);`

Figura 3.2: *Bug* no produto *Libxml2*

A Figura 3.2 mostra uma manutenção no código do projeto *GNOME*, mais especificamente, no produto *Libxml2* durante a versão 2.7.6 que acarretou na criação de um *bug*. Anteriormente, a variável estática e inteira *xmlout* foi declarada sob as *features LIBXML_HTML_ENABLED* e *LIBXML_VALID_ENABLED*. Em seguida, acrescentou-se, dentre outras funcionalidades, um *if (xmlout)* dentro de uma diferente *feature* como indica a figura abaixo. Ou seja, devido à desatenção do desenvolvedor ao manter o código, causou-se um problema para outra *feature* e, consequentemente, para o produto. Ele não observou que a variável *xmlout* está circundada por *LIBXML_HTML_ENABLED* e *LIBXML_VALID_ENABLED* em sua declaração.

Desse modo, a variável em questão só será declarada e compilada se a *feature LIBXML_-HTML_ENABLED* ou a *LIBXML_VALID_ENABLED* for selecionada. Porém, a mesma variável também é referenciada em outra *feature*. Assim sendo, se a *feature* que é referenciada estiver definida e as *features LIBXML_HTML_ENABLED* e *LIBXML_VALID_ENABLED* não estiverem, ocorrerá um erro de compilação, uma vez que a variável *xmlout* seria referenciada sem ter sido declarada. A fim de solucionar o problema, abriu-se um *bug report* no *bugzilla*[6] do *GNOME*. Esse problema foi consertado na *release* posterior, utilizou-se `#ifdef` *LIBXML_HTML_ENABLED* || *LIBXML_VALID_ENABLED* circundando o *if (xmlout){...}* como pode ser visto na Figura 3.3.

[6]http://bugzilla.gnome.org/

Libxml2 – Versão 2.7.6

Linhas 2552-2555

```
2552                 if (format)
2553                     saveOpts |= XML_SAVE_FORMAT;

2554                 if (xmlout)
2555                     saveOpts |= XML_SAVE_AS_XML;
```

Libxml2 – Versão 2.7.7

Linhas 2656-2662

```
2656                 if (format)
2657                     saveOpts |= XML_SAVE_FORMAT;
2658
2659 #if defined(LIBXML_HTML_ENABLED) || defined(LIBXML_VALID_ENABLED)
2660                 if (xmlout)
2661                     saveOpts |= XML_SAVE_AS_XML;
2662 #endif
```

Figura 3.3: Correção do *bug* no produto *Libxml2*

3.2 Variável não usada

Depois de algumas buscas por casos em que variáveis são declaradas, mas não são utilizadas em sequer uma só vez no código, percebeu-se que erros dessa classe ocorrem frequentemente em sistemas que são implementados com compilação condicional.

Gnome-vfs – Versão cvs (head)

Linhas 321-324

```
321     ProxyHostAddr *elt;
322

323     input = (gchar *) data;
324     elt = g_new0(ProxyHostAddr, 1);
```

Linhas 348-350

```
348
349     if (ip_addr) {
350         if (!has_error) {
```

Gnome-vfs – Versão cvs (head)

Linhas 321-325

```
321     ProxyHostAddr *elt;
322     gint i;
323
324     input = (gchar *) data;
325     elt = g_new0(ProxyHostAddr, 1);
```

Linhas 349-380

```
349 #ifdef ENABLE_IPV6
350     else if (have_ipv6() && inet_pton(AF_INET6, hostname, &host6) > 0) {
351         ip_addr = TRUE;
352         elt->type = PROXY_IPv6;
353         for (i = 0; i < 16; ++i) {
354             elt->addr6.s6_addr[i] = host6.s6_addr[i];
355         }
356         if (netmask) {
357             gchar *endptr;
358             gint width = strtol(netmask, &endptr, 10);
359
360             if (*endptr != '\0' || width < 0 || width > 128) {
361                 has_error = TRUE;
362             }
363             for (i = 0; i < 16; ++i) {
364                 elt->mask6.s6_addr[i] = 0;
365             }
366             for (i = 0; i < width / 8; i++) {
367                 elt->mask6.s6_addr[i] = 0xff;
368             }
369             elt->mask6.s6_addr[i] = (0xff << (8 - width % 8)) & 0xff;
370             ipv6_network_addr(&elt->addr6, &mask6, &elt->addr6);
371         } else {
372             for (i = 0; i < 16; ++i) {
373                 elt->mask6.s6_addr[i] = 0xff;
374             }
375         }
376     }
377 #endif
378
379     if (ip_addr) {
380         if (!has_error) {
```

Figura 3.4: *Bug* no editor *Vim*

Serão apresentados dois casos para mostrar como esse tipo de problema ocorre na prática. Os exemplos a seguir foram retirados dos *bug reports* do projeto *GNOME* e do editor de texto *Vim*, respectivamente. Os *bugs* reportados estão disponíveis na internet e qualquer pessoa pode

visualizá-los pelo site *bugzilla*[7] e/ou *google code*[8].

Gnome-vfs – Versão cvs (head)

Linhas 318-325

```
318 #ifdef ENABLE_IPV6
319     struct in6_addr host6, mask6;

320 #endif
321     ProxyHostAddr *elt;
322     gint i;
323
324     input = (gchar *) data;
325     elt = g_new0(ProxyHostAddr, 1);
```

Gnome-vfs – Versão cvs (head)

Linhas 318-325

```
318 #ifdef ENABLE_IPV6
319     struct in6_addr host6, mask6;
320     gint i;
321 #endif
322     ProxyHostAddr *elt;

323
324     input = (gchar *) data;
325     elt = g_new0(ProxyHostAddr, 1);
```

Figura 3.5: Correção do *bug* no editor *Vim*

A Figura 3.4 apresenta uma variável inteira *i* que foi introduzida pelo programador do produto *gnome-vfs*. O desenvolvedor não atentou que a variável *i* só é usada se a *feature ENABLE_IPV6* for selecionada para o produto. Outro desenvolvedor, que contribui para o projeto *GNOME*, percebeu que a variável estava sendo declarada, mas não usada. Isso porque o mesmo compilou o sistema sem que a *feature ENABLE_IPV6* estivesse definida. O problema foi corrigido simplesmente modificando o lugar onde a variável *i* estava sendo declarada. Agora, a declaração da mesma se encontra dentro de `#ifdef ENABLE_IPV6` como mostra a Figura 3.5.

VIM – Revision 2b475ed86e64

Linhas 2840-2847

```
2840    long            j;
2841    int             yanktype = oap->motion_type;
2842    long            yanklines = oap->line_count;
2843    linenr_T        yankendlnum = oap->end.lnum;
2844    char_u          *p;
2845    char_u          *pnew;
2846    struct block_def    bd;

2847    int             did_star = FALSE;

```

Linhas 3113-3130

```
3113 #ifdef FEAT_CLIPBOARD
3114     /*
3115      * If we were yanking to the '*' register, send result to clipbc
3116      * If no register was specified, and "unnamed" in 'clipboard', n
3117      * to the '*' register.
3118      */
3119     if (clip_star.available
3120             && (curr == &(y_regs[STAR_REGISTER])
3121                 || (!deleting && oap->regname == 0
3122                                 && (clip_unnamed & CLIP_U
3123     {
3124         if (curr != &(y_regs[STAR_REGISTER]))
3125             /* Copy the text from register 0 to the clipboard regist
3126             copy_yank_reg(&(y_regs[STAR_REGISTER]));
3127
3128         clip_own_selection(&clip_star);
3129         clip_gen_set_selection(&clip_star);

3130         did_star = TRUE;

```

VIM – Revision b93a3fc3897b

Linhas 2840-2849

```
2840    long            j;
2841    int             yanktype = oap->motion_type;
2842    long            yanklines = oap->line_count;
2843    linenr_T        yankendlnum = oap->end.lnum;
2844    char_u          *p;
2845    char_u          *pnew;
2846    struct block_def    bd;
2847 #if defined(FEAT_CLIPBOARD) && defined(FEAT_X11)
2848    int             did_star = FALSE;
2849 #endif
```

Linhas 3115-3134

```
3115 #ifdef FEAT_CLIPBOARD
3116     /*
3117      * If we were yanking to the '*' register, send result to clipb
3118      * If no register was specified, and "unnamed" in 'clipboard', 
3119      * to the '*' register.
3120      */
3121     if (clip_star.available
3122             && (curr == &(y_regs[STAR_REGISTER])
3123                 || (!deleting && oap->regname == 0
3124                                 && (clip_unnamed & CLIP_
3125     {
3126         if (curr != &(y_regs[STAR_REGISTER]))
3127             /* Copy the text from register 0 to the clipboard regis
3128             copy_yank_reg(&(y_regs[STAR_REGISTER]));
3129
3130         clip_own_selection(&clip_star);
3131         clip_gen_set_selection(&clip_star);
3132 # ifdef FEAT_X11
3133         did_star = TRUE;
3134 # endif
```

Figura 3.6: Variável *did_star* sem uso

Um aspecto importante a destacar é que trivialmente podem-se encontrar dependências entre *features*, isto é, compartilhamento de variáveis, de métodos, entre outros. Por isso, é necessário

[7] http://bugzilla.gnome.org/

[8] http://code.google.com/p/vim/

ter convicção de que a variável *i*, neste caso, é apenas utilizada na *feature ENABLE_IPV6* a fim de que a solução deste problema não acarrete outros fatalmente. Essa certeza de que a variável só se encontra em uma *feature* específica pode demandar tempo e esforço por parte do desenvolvedor dependendo do número de linhas de código do sistema, dado que haja vários outros `#ifdefs`.

A Figura 3.6 mostra outro cenário com variável sem uso no sistema. Ele foi extraído do editor de texto *Vim*. Na *revision 2b475ed86e64*, a variável *did_star* é usada apenas se a *feature FEAT_CLIPBOARD* estiver definida. A execução do sistema sem a *feature* em questão causou um aviso (mais conhecido como *warning*). Assim, foi solicitado sua correção e a solução foi ajustar os `#ifdefs`.

3.3 Atribuição possivelmente errada

O objetivo para essa classe de erros é exibir como uma manutenção de código pode quebrar certa(s) *feature(s)* devido alguma atribuição incorreta.

VIM – Revision 89dc68c0ab6f
Linhas 1279-1298

```
1279 # ifdef HAVE_GTK2
1280     title = CONVERT_TO_UTF8(title);
1281 # endif
1282
1283     /* GTK has a bug, it only works with an absolute path. */
1284     if (initdir == NULL || *initdir == NUL)
1285         mch_dirname(dirbuf, MAXPATHL);
1286     else if (vim_FullName(initdir, dirbuf, MAXPATHL - 2, FALSE) == F
1287         dirbuf[0] = NUL;
1288     /* Always need a trailing slash for a directory. */
1289     add_pathsep(dirbuf);
1290
1291     /* If our pointer is currently hidden, then we should show it. *
1292     gui_mch_mousehide(FALSE);
1293
1294 #ifdef USE_FILE_CHOOSER
1295     /* We create the dialog each time, so that the button text can b
1296      * or "Save" according to the action. */
1297     fc = gtk_file_chooser_dialog_new((const gchar *)title,
1298             GTK_WINDOW(gui.mainwin),
```

VIM – Revision e4d849f4df03
Linhas 843-860

```
843     title = CONVERT_TO_UTF8(title);

844
845     /* GTK has a bug, it only works with an absolute path. */
846     if (initdir == NULL || *initdir == NUL)
847         mch_dirname(dirbuf, MAXPATHL);
848     else if (vim_FullName(initdir, dirbuf, MAXPATHL - 2, FALSE) == F
849         dirbuf[0] = NUL;
850     /* Always need a trailing slash for a directory. */
851     add_pathsep(dirbuf);
852
853     /* If our pointer is currently hidden, then we should show it. *
854     gui_mch_mousehide(FALSE);
855
856 #ifdef USE_FILE_CHOOSER
857     /* We create the dialog each time, so that the button text can b
858      * or "Save" according to the action. */
859     fc = gtk_file_chooser_dialog_new((const gchar *)title,
860             GTK_WINDOW(gui.mainwin),
```

Figura 3.7: Atribuição possivelmente errada no editor *Vim*

Desse modo, a Figura 3.7 apresenta um cenário de manutenção retirado do projeto *Vim - the text editor* onde o desenvolvedor modificou fragmentos do código fonte. Uma dessas alterações foi a remoção do `#ifdef` da *feature HAVE_GTK2* que poderia provocar outro *bug*, uma vez que com essa modificação a variável *title* que é passada como parâmetro no método *gtk_file_chooser_dialog_new* (linha 859 da segunda coluna), agora, sempre estará sendo atribuída (linha 843 da segunda coluna). Vale ressaltar que o método em questão está sob outra *feature USE_FILE_CHOOSER* e, pensando na técnica de separação de interesses, o desenvolvedor não estaria visualizando-a. Porém, de qualquer forma, o programador responsável precisa estar ciente de que a variável *title* é utilizada não só na *feature HAVE_GTK2*. Essa atribuição possivelmente errada não causou maiores prejuízos porque a variável *title* passou a ser apenas

convertida para a codificação UTF8 não alterando seu valor propriamente dito.

3.4 Comportamento modificado

Essa Seção mostra que as tarefas de manutenção podem causar problemas comportamentais no sistema ou na linha de produto. Conforme visto no exemplo da Seção 3.3, o comportamento do sistema poderia ser fatalmente afetado por causa de uma simples atribuição possivelmente equivocada.

O cenário apresentado a seguir exibe como o desenvolvedor pode fazer poucas alterações no código e, em consequência disso, provocar efeito drástico no que diz respeito a semântica do sistema ou LPS. Ele foi extraído do projeto *GTK+*[9], parte integrante do *GNOME*.

GTK+ Versão 2.3.x
Linhas 125-134

```
125 #ifndef G_OS_WIN32
126         c = getc_unlocked (stream);
127 #else
128         c = getc (stream);
129 #endif
130
131         if (c == EOF)
132           goto done;
133         else
134           n_read++;
```

GTK+ Versão 2.3.x
Linhas 125-134

```
125 #ifndef HAVE_FLOCKFILE
126         c = getc_unlocked (stream);
127 #else
128         c = getc (stream);
129 #endif
130
131         if (c == EOF)
132           goto done;
133         else
134           n_read++;
```

Figura 3.8: *Bug* no produto *GTK+*

O programador atualizou o código fonte e introduziu um *bug* semântico sutil no sistema. A Figura 3.8 apresenta a simples mudança que foi realizada pelo desenvolvedor. Percebe-se que na versão anterior ao problema, havia um `#ifndef G_OS_WIN32` que foi modificado por `#ifndef HAVE_FLOCKFILE`. Parece correta e trivial esta mudança, mas por causa dela foi aberto um *bug report* no *bugzilla* do *GNOME* para solucionar um problema sério. A pessoa que reportou o *bug* disse[10] que o `#ifndef HAVE_FLOCKFILE` está inverso causando falha nos produtos que não possui a *feature HAVE_FLOCKFILE*. Isso aconteceu porque o desenvolvedor responsável pela atualização estava desapercebido quanto ao comportamento do sistema. Para encontrar esse *bug*, possivelmente, o desenvolvedor/reportador precisou analisar o fluxo de dados dos produtos gerados com e sem a *feature HAVE_FLOCKFILE* diminuindo assim a produtividade devido ao tempo e esforço empenhado em verificar diversas linhas de código associadas ao `#ifndef HAVE_FLOCKFILE`. A correção foi basicamente trocar de `#ifndef` para `#ifdef HAVE_FLOCKFILE`.

[9] http://www.gtk.org/
[10] https://bugzilla.gnome.org/show_bug.cgi?id=135642

Capítulo 4

Solução Proposta

Este capítulo descreve as regras geradas através da observação de cenários de manutenção reais as quais trazem uma inestimável contribuição ao trabalho. Também, será apresentado o funcionamento geral do assistente inteligente.

Os projetos de software tornam-se cada vez mais amplo e complexo, por isso fica difícil de verificar se o código está funcionando corretamente antes de submetê-lo (*commit*). Pensando em LPS, torna-se ainda mais complicado visto que uma linha de produtos pode possui até 2^n combinações de produtos distintos, sendo n o número de *features*. Linhas de produtos de software são inerentemente complexas e propensas a todos os tipos de erros [8]. Assim sendo, encontrar um defeito pode levar um tempo considerável de testes pois necessitará averiguar todas as configurações possíveis da linha. Por isso que as atividades de manutenção [18] são responsáveis por mais de dois terços do custo do ciclo de vida de um sistema de software [4]. A manutenção de software é essencial para a confiabilidade do software, e reportar defeitos é, igualmente, fundamental para a manutenção do mesmo.

Muitos projetos de sistemas ou linhas de produto de software, principalmente os grandes projetos, contam com relatórios de *bugs* (do inglês, *bug reports*) para dirigir a atividade de manutenção de forma corretiva [1]. Especialmente em projetos de software *open source*, os *bugs* são reportados na maioria das vezes pelos usuários ou desenvolvedores do sistema. Esses *bugs* são armazenados em um banco de dados através de alguma ferramenta de acompanhamento de *bugs*, mais conhecidas como *bug tracking systems*. Essa ideia de permitir aos usuários reportar e, consequentemente, ajudar a resolver *bugs* é assumida como uma função importante para melhorar a qualidade do software como um todo [19]. Os sistemas de rastreamento de *bugs* permitem aos usuários reportar, descrever, classificar, acompanhar e comentar nos *bug reports*. Por exemplo, o *Bugzilla* é um dos sistemas de *bug tracking, open source,* mais popular [20], sendo utilizado em grandes projetos como Mozilla, Kernel do Linux, Eclipse, entre outros. A maioria dos *bug reports*, inclusive do *Bugzilla*, vem com uma série de campos pré-definidos (e.g., versão e sistema operacional) e também com campos de textos livres que podem ser preenchidos (e.g., título e descrição do *bug*). Além disso, usuários e desenvolvedores podem

enviar anexos.

A quantidade de *bugs* reportados normalmente excede o número de recursos disponíveis (por exemplo, pessoal e financeiro) para lidar com todos os defeitos a fim de resolvê-los [2]. Isso causa uma sobrecarga nos desenvolvedores que gastam tempo e esforço analisando os *bug reports* [20] e precisam ser ainda mais eficientes na correção dos erros. Por conta disso, os desenvolvedores podem ao solucionar um *bug* inserir outros mais sem perceber. Existem vários fatores que podem influenciar o desenvolvedor a inserir problemas no sistema ou na linha de produto de software. Pode ser desatenção, tempo, problema familiar e etc.

Conforme mostrado no Capítulo 3, sabe-se que a compilação condicional reduz a legibilidade do código devido à poluição de `#ifdefs` entrelaçando o mesmo. Isso propicia o aumento do risco nos procedimentos de manutençao podendo causar novos defeitos. Os erros sutis inseridos durante a manutenção, geralmente, só são detectados quando uma *feature* específica ou uma combinação de *feature* são selecionadas. E, esses problemas ocasionados por modificações no código são facilmente localizados em sistemas ou LPS reais, como *Vim - the text editor*[1] e *GNOME*[2], por exemplo. Além do mais, erros podem permanecer escondidos na implementação até o cliente, eventualmente, requisitar um produto defeituoso [8].

Desta forma, percebeu-se que diversos sistemas de software espalhados pelo mundo (como os supracitados) possui a seguinte característica no que tange o processo de manutenção. Após finalizada uma versão estável do software ou família de software, este é disponibilizado aos usuários. Nesse meio tempo, a medida com que os defeitos vão sendo reportados junto ao sistema de rastreamento de *bugs* os desenvolvedores vão trabalhando a fim de solucioná-los. Porém, como já foi discutido, nem sempre essas manutenções são corretivas como esperado, pois em muitos casos acarretam outros problemas, inclusive, maiores - os que afetam o comportamento do sistema ou LPS. Logo, só em *releases* posteriores que esses problemas são descobertos por causa de um certo *commit*, o qual ao tentar corrigir um determinado *bug* causou outro no sistema. Em seguida, o progamador, responsável pelo módulo do sistema que está incorreto, corrige o erro e submete a versão correta.

Os desenvolvedores já tem um tempo escasso para analisar e solucionar os problemas que estão nos *bug reports* e ainda por cima perdem mais tempo corrigindo erros uns dos outros. A Figura 4.1 exibe o que acontece normalmente nos sistemas ou linhas de produto de software. A **Versão 1** representa uma versão ligeiramente anterior a versão com *bug*, não obrigatoriamente a primeira. Entretanto, por alguma ocasião, precisa-se mexer no código a fim de solucionar algum defeito ou acrescentar uma funcionalidade. Feito isso, o desenvolvedor termina sua manutenção e dá *commit*. Porém, ao invés do sistema migrar para uma versão estável, ele adquire versões intermediárias indesejadas (como **Versão 2**). E estas possuem erros causados pela própria manutenção por parte dos desenvolvedores. Depois de encontradas as falhas e, obviamente, solucionadas é que o software passa para o estado esperado (**Versão 3**).

[1] http://www.vim.org/
[2] http://www.gnome.org/

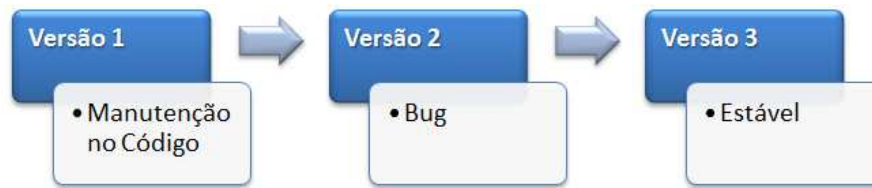

Figura 4.1: Fluxo Representativo de Cenários de Manutenção

Sob a perspectiva da programação utilizando pré-processadores, isto é, usando diretivas como `#ifdef` e `#endif`, existe uma série de cuidados que são imprescindíveis a fim de que os produtos de uma LPS funcionem de maneira satisfatória. Ou seja, tais produtos não podem conter erros, pois poderia comprometer os ddemais produtos. Pensando nisso, surge uma pergunta: É possível minimizar o aparecimento dessas versões intermediárias contidas de *bugs*? Se sim, como? Sim. A partir de observações em diversos *bug reports* espalhados na web, constatou-se a necessidade de definir um conjunto de regras a fim de prevenir a existência de erros. Por isso que a solução proposta serve para detectar o máximo de defeitos possíveis inferindo se alguma variante da linha de produto de software tenderá a se comportar erroneamente de acordo com as funcionalidades previstas na documentação.

A propósito, a Figura 4.2 mostra como os sistemas e LPS devem se comportar quando uma atividade de manutenção acontecer a partir de então. Após a **Versão 1** ser modificada, ela passará por uma bateria de verificações para apontar a possível existência de defeito. Esses sucessivos testes nada mais são do que um conjunto de regras, as quais serão executadas em cada manutenção efetuada no código. Com isso, espera-se que o sistema ou LPS migre diretamente para uma versão sem *bugs*, como mostra a Figura 4.2 (seta superior). É lógico que o sistema proposto não garante que todas as modificações feitas no sistema não gere versões com *bugs*.

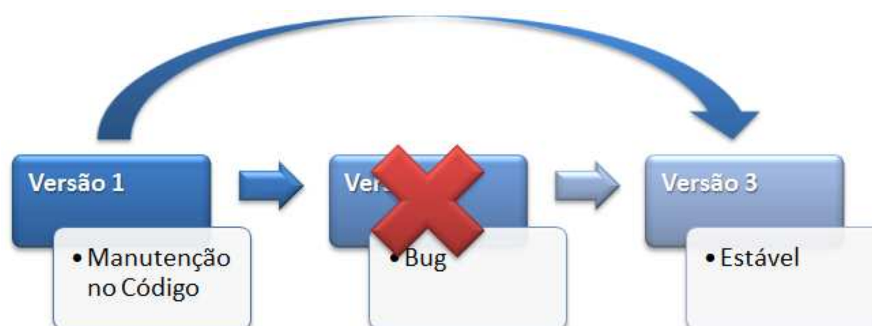

Figura 4.2: Fluxo Alternativo com o Assistente Inteligente

Diante do exposto, grande parte dos erros inseridos durante o processo de manutenção foram categorizados segundo consta no Capítulo 3, onde se tem: (i) Variável não declarada (Seção 3.1); (ii) Variável não usada (Seção 3.2); (iii) Atribuição possivelmente errada (Seção 3.3); (iv) Comportamento modificado (Seção 3.4). As regras foram igualmente catalogadas para atender a essas classes de problemas.

Já que as regras serão apresentadas de acordo com cada categoria de problema, faz-se necessário expor a sintaxe referente as regras para melhor entendimento. A Tabela 4.1 traz a representação da sintaxe.

F	Feature
C	Classe
D extends C	Subclasse
m	Método
v	Variável
a[]	Array
d.v	Declaração de v
u.v	Uso de v
unique(*)	Retorna se (*) é único
¬	Operador de negação
exists(*)	Retorna 'true' ou 'false'
remove(*)	Remove o parâmetro
add(*)	Adiciona o parâmetro
in	Denota-se 'dentro de'
get(*)	Retorna o elemento relacionado ao parâmetro
getLine(*)	Retorna a linha relacionada ao parâmetro
getFeatures(*)	Retorna as features associadas ao parâmetro
getExpression(*)	Retorna o parâmetro em lógica proposicional
isOverride(*)	Retorna 'true' se sobrescrito
isCalled(*)	Retorna 'true' se (*) é chamado

Tabela 4.1: Sintaxe das Regras

4.1 Regras

Esta seção descreve as regras concebidas para capturar as dependências entre *features*. Antes de apresentar cada regra será dada uma explicação de alto nível do contexto.

4.1.1 Variável não declarada

Quando existir alguma declaração de variável dentro de uma *feature* que não seja obrigatória pode acontecer de a variável não ser declarada para algum determinado produto. Supondo que exista no código pelo menos uma referência à variável em questão dentro de outra *feature*, então o erro de variável não declarada ocorreria sempre que a *feature* detentora da declaração não estiver definida.

A regra 4.1 evita que defeitos de variáveis não declaradas ocorram. Lembrando que as funções *getExpression* e *exists* retornam a expressão do `#ifdef` em lógica proposicional e *true* se existir o parâmetro, respectivamente.

$$\frac{\neg\, getExpression(d.v) \,\wedge\, exists(u.v)}{warning} \tag{4.1}$$

Em primeiro lugar, obtém-se a expressão em lógica proposicional da(s) *feature(s)* que circundam a declaração da variável através de `getExpression(d.v)` e, em seguida, verifica-se o valor da expressão. Se o resultado da expressão (`getExpression(d.v)`) for falso significa dizer que a variável não será declarada. Mas, só isso não basta porque não necessariamente uma variável que não seja declarada estará sendo usada. Por isso, tem-se a segunda premissa da regra: *exists(u.v)*. As *features* que usam a variável servem para descobrir se determinada manutenção na declaração vai impactar a linha de produtos. Portanto, sempre que uma variável não for declarada e existir algum uso dela ocorrerá um erro de tipo [8]. Pode ser detectado em tempo de compilação. Em outras palavras, caso aconteça esse problema o sistema ou a LPS simplesmente não roda.

Frente a isso, alguma ferramenta de suporte que faça uso da regra 4.1 informaria ao desenvolvedor sobre a possibilidade da variável não ser declarada deixando-o ciente da situação. Seguindo a ideia de interfaces emergentes, o aviso (do inglês, *warning*) a ser exibido seria simples e conciso, como mostrado na Figura 4.3. De acordo com a mensagem, a variável não declarada (*<v>*) está sendo usada em outras *features* (*<getFeatures(u.v)>*).

Provides *<v>* **to** *<getFeatures(u.v)>*

Figura 4.3: *Warning* baseado em Interfaces Emergentes.

Com a leitura da mensagem acima, o programador estará a par da circunstância antes de finalizar sua manutenção por completa. Desta forma, ele saberá que ao alterar a declaração da variável pode acarretar em problemas para as demais *features*.

Agora, caso a expressão (`getExpression(d.v)`) seja verdadeira, a variável será declarada de modo que a regra não constará problema algum e, assim, a ferramenta de apoio não alertará ao programador.

A demonstração da regra será detalhada por meio de um cenário real, como mostra a Figura 4.4. As letras em negrito denotam o que foi acrescido de uma versão para outra.

O desenvolvedor não fez nenhuma remoção, pelo contrário, acrescentou um comando *if* que utiliza a variável *xmlout* como expressão booleana da estrutura de controle. Porém, ele não observou que a variável em questão é inicializada (declaração + atribuição) nas *features* *LIBXML_HTML_ENABLED*, *LIBXML_VALID_ENABLED* diferentemente da *feature* mandatória onde se encontra o uso. Em outras palavras, a declaração de *xmlout* está relacionada com apenas essas duas *features*, ao contrário de *if (xmlout)* que está contido em outra. Sempre que as *features* detentoras da inicialização de *xmlout* não forem selecionadas para uma determinada variante da linha de produtos de software, ocorrerá um erro do tipo “Variável não declarada”.

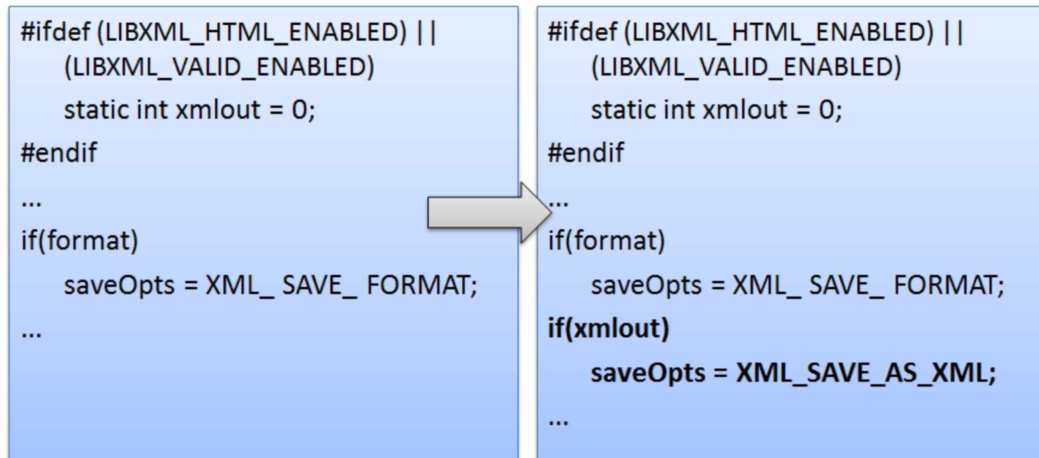

Figura 4.4: Declaração e uso em *features* distintas

A partir de agora, as *features LIBXML_HTML_ENABLED e LIBXML_VALID_ENABLED* serão chamadas de A e B, respectivamente, e a *feature* raiz obrigatória que possui o uso de *xmlout* de R para ajudar na compreensão. A tabela verdade 4.2 apresenta a execução da regra para expressão do `#ifdef` (A ∨ B) e a *feature* R. Observa-se que a *feature* R sempre estará definida para quaisquer produtos, pois se trata de uma funcionalidade mandatória. O resultado da expressão como um todo só é verdade quando as *features* A e B são falsas, isto é, não estão selecionadas.

A	B	R	A ∨ B	¬ (A ∨ B)	¬ (A ∨ B) ∧ R
F	F	V	F	V	V
F	V	V	V	F	F
V	F	V	V	F	F
V	V	V	V	F	F

Tabela 4.2: Tabela verdade das funcionalidades A, B e R.

Neste exemplo, sabe-se que *static int xmlout = 0;* e *if (xmlout)* representam *d.v* e *u.v* na regra, respectivamente. Analisando passo a passo, fica:

$$\neg\ getExpression(d.v)\ \wedge\ exists(u.v)\ = \tag{4.2}$$
$$\neg\ getExpression(static\ int\ xmlout = 0)\ \wedge\ exists(if\ (xmlout))\ = \tag{4.3}$$
$$\neg\ (A\ \vee\ B)\ \wedge\ R\ = \tag{4.4}$$
$$\neg\ (A\ \vee\ B) \tag{4.5}$$

Portanto, o resultado final depende apenas das *features* A e B, já que a *feature* R sempre será selecionada. Esse desfecho evidencia que a variável *xmlout* pode está sendo usada em uma *feature* que o desenvolvedor não está mantendo enquanto que a variável pode não ser declarada.

Por este motivo que a regra apresentaria esse caso ao desenvolvedor de forma semelhante à Figura 4.5. Considere que variáveis globais bem como qualquer elemento de código que não pertence explicitamente a uma *feature* faz parte de uma *feature* raiz obrigatória (do inglês, *root feature*).

Figura 4.5: Mensagem do Assistente Inteligente.

4.1.2 Variável não usada

No processo de manutenção, os programadores podem facilmente poluir o código removendo as linhas que contém os usos das variáveis e deixando suas declarações. Sabendo que uma variável pode ter um único uso. Assim, removida essa última utilização implica num problema conhecido como "Variável sem uso" ou "Variável não usada". Não faz sentido deixar essas declarações sem utilidade no código.

Para isso, foi construída uma regra que objetiva eliminar essas declarações sem uso do sistema ou LPS. A regra 4.6 mostra a condição que precisa ser satisfeita para a regra detectar o problema. A leitura da regra é simples e bem parecida a da Seção 4.1.1.

$$\frac{getExpression(d.v) \wedge \neg\, exists(u.v)}{warning} \tag{4.6}$$

Caso a expressão proposicional da declaração de uma variável qualquer, aqui chamada de *'v'*, seja verdadeira para uma determinada variante da linha de produtos ou para uma simples aplicação; então, a variável será declarada. Porém, para satisfazer a regra, além da variável ser declarada, ela não deve ser usada em lugar algum. Logo, toda vez que a variável for declarada e não existir uso significa que a declaração não importa para o produto. Não considera-se como um erro em si, mas como um descuido que provoca poluição no programa.

Diante do exposto, a regra proporcionaria ao desenvolvedor um auxílio lhe dizendo que a variável *'v'* não está sendo usada. Seria algo semelhante aos IDEs, por exemplo, o Eclipse deixa a declaração da variável sublinhada de amarelo mostrando que está sem uso. A Figuras 4.6 exibe a mensagem genérica para o caso de variável sem uso e como apareceria para o desenvolvedor, respectivamente. É óbvio que se a variável está sem uso significa dizer que todas as *features* selecionadas não referenciam a variável em curso.

Suponha agora que o desenvolvedor removeu propositalmente o único uso de *'v'*. A regra supracitada detectaria, mas como não há mais nenhuma funcionalidade que utilize *'v'*, isto é, todas as *features* definidas ou não são desprovidas de referência à variável. Então, seria mais sensato pensar em remover também a declaração sem incomodar o desenvolvedor. Depois dessa

Provides <d.v> **to** <getFeatures(u.v)>

Figura 4.6: *Warning* para variável sem uso.

ação, apenas notificaria o programador da decisão tomada, deixando-o livre se quiser reverter a modificação. É exatamente isso que a regra 4.7 sugere.

$$\frac{remove(unique(u.v))}{remove(d.v)} \tag{4.7}$$

Considere a Figura 4.7 para exemplificar a execução das regras. Esse trecho de código foi retirado do *gnome-vfs*, módulo *http*. Vale ressaltar que o código em negrito representa o que foi adicionado na *release* posterior.

```
#ifdef ENABLE_IPV6
    struct in6_addr host6, mask6;
#endif
ProxyHostAddr *elt;
...
```

```
#ifdef ENABLE_IPV6
    struct in6_addr host6, mask6;
#endif
ProxyHostAddr *elt;
gint i;
...
#ifdef ENABLE_IPV6
    else if(have_ipv6() && inet_pton
    (AF_INET6, hostname, &host6) > 0)
       ip_addr = TRUE;
       elt->type = PROXY_IPv6;
       for(i = 0; i < 16; ++i)
          elt->addr6.s6_addr[i] =
             host6.s6_addr6[i];
...
#endif
...
```

Figura 4.7: Variável *i* pode não ser usada.

Percebe-se que a regra 4.7 não faz sentido nesse cenário, uma vez que o último uso de *i* não foi excluído. Neste caso, o programador inseriu um trecho de código dentro da funcionalidade *ENABLE_IPV6*. Nesse meio tempo, ele também declarou uma variável global *i* – fora do `#ifdef ENABLE_IPV6`. Mas, nota-se que a variável *i* só é usada se o IPV6 for habilitado. Acontece que roteadores, servidores, sistemas operacionais, entre outros precisam estar plenamente compatíveis com o IPv6, mas a internet ainda está baseada no IPv4. Desta forma, pode acontecer facilmente da *feature ENABLE_IPV6* não ser definida, e assim *i* não será utilizada. Assim aconteceu, o defeito foi reportado[3] e, consequentemente, corrigido. A regra 4.6 seria útil já que ela detectaria na hora. Veja de maneira detalhada:

[3] https://bugzilla.gnome.org/show_bug.cgi?id=167715

$$getExpression(d.v) \wedge \neg exists(u.v) = \quad (4.8)$$
$$getExpression(gint\ i) \wedge \neg exists(for(i=0; i<16; ++i)\{...\}) = \quad (4.9)$$
$$V \wedge \neg ENABLE_IPV6 = \quad (4.10)$$
$$\neg ENABLE_IPV6 \quad (4.11)$$

Fazendo ENABLE_IPV6 igual a falso implica ¬ENABLE_IPV6 verdadeiro. Daí, a regra alertaria da existência de *i* sem uso, caso a *feature ENABLE_IPV6* não esteja definida.

4.1.3 Atribuição possivelmente errada

Sob a perspectiva de VSoC, o desenvolvedor, eventualmente, pode alterar os valores de variáveis globais ou até mesmo de variáveis compartilhadas por diferentes *features* para fazer funcionar sua manutenção. Assim, as variáveis podem ser usadas em outras *features* de responsabilidade de outros membros da equipe de desenvolvimento/manutenção. Esses outros desenvolvedores podem estar esperando que as variáveis utilizadas em suas *features* possuam valores específicos (que estavam sendo usados antes da última modificação) para a execução correta de alguma funcionalidade. Desse modo, uma simplória mudança na atribuição de alguma variável que esteja sendo usada nas demais *features* pode acarretar *bugs* de *runtime* no sistema.

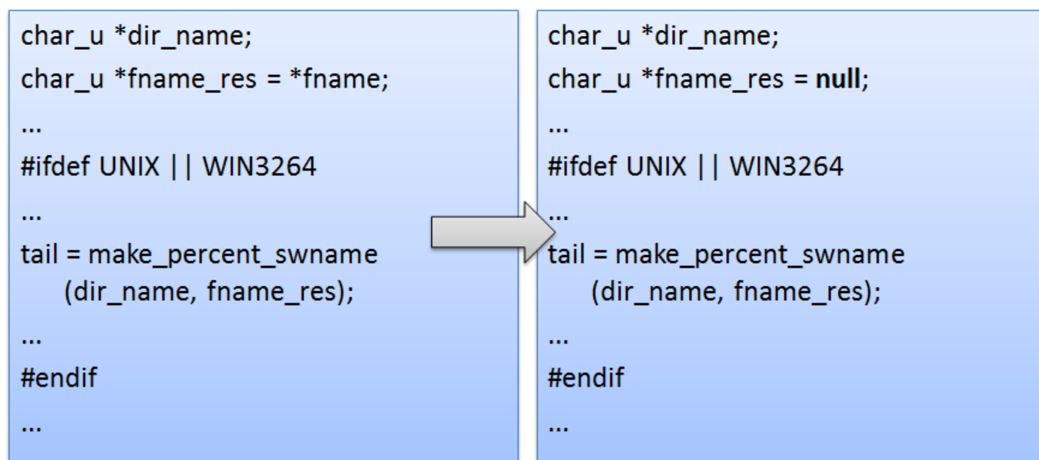

Figura 4.8: *Bug* no editor *Vim*.

A Figura 4.8 mostra uma correção de *bug* extraída do projeto *Vim*. Observa-se que o ponteiro *fname_res* é passado como parâmetro de função dentro do `#ifdef UNIX || WIN3264`. Não consta na imagem, mas antes do uso de *fname_res* há aproximadamente 100 linhas de código e vários `#ifdefs` espalhados. Para alterar alguma linha de código, o programador deveria analisar diversos `#ifdefs` para garantir que outras funcionalidades não sejam quebradas. Entretanto,

ao tentar corrigir um erro, a variável *fname_res* passou a receber o valor **null** na atribuição. O desenvolvedor não atentou que o ponteiro era usado em outras *features* e estas não se comportam adequadamente quando diferentes valores são assumidos para algumas de suas variáveis, como o caso de *fname_res*.

A fim de evitar esse tipo de transtorno, a regra 4.12 busca detectar e, em seguida, possibilitar que alguma ferramenta alerte o programador.

$$\frac{exists(u.v)\ \wedge\ add(m.v)\ \wedge\ (getLine(m.v)\ <\ getLine(u.v))}{warning} \tag{4.12}$$

No processo de manutenção, o desenvolvedor pode inserir/alterar alguma escrita (do inglês, *write*) de qualquer variável. Se já existia algum uso da mesma e a linha que o desenvolvedor modificou for anterior a esse uso, então pode acontecer da funcionalidade detentora do uso não executar corretamente. Isso porque o valor da variável pode alcançar o uso com o valor alterado. É importante destacar que as *features* em questão não podem ser mutualmente exclusivas. Para este cenário, o *pop-up* que apareceria para o programador poderia ser algo como mostra a Figura 4.9.

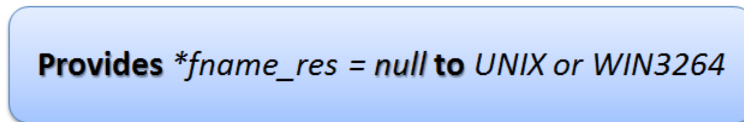

Figura 4.9: Mensagem para atribuição possivelmente errada.

4.1.4 Comportamento modificado

De acordo com [28], *bugs* semânticos são difícies de serem detectados pois possuem causas muito diversas. Por isso, as formas de resolvê-los também são diversificadas. Não obstante, os autores observaram um padrão de correção de *bugs* que causam defeitos de comportamentos: condições (por exemplo, condição *if*) são difíceis de corrigir corretamente.

Os autores de [23] já tinham alertado quanto à esse tipo de problema. O exemplo que eles usaram foi de um jogo chamado Best Lap[4]. Best lap é um jogo de corrida onde o usuário tenta realizar a melhor volta. A Figura 4.10 mostra o corpo do método *computeLevel* responsável por calcular a pontuação do jogador. A *feature* opcional `ARENA` possui uma chamada a *setScore* da classe *NetworkFacade*, esse método publica a pontuação do jogador na rede.

Suponha agora que o jogo também permita pontuações negativas. Assim sendo, o desenvolvedor responsável por implementar essa funcionalidade simplesmente alterou a atribuição da variável *totalScore* (veja a linha em negrito da Figura 4.10). Essa solução parece viável, mas a Figura 4.11 desmente. Ela exibe a implementação do método estático *setScore*. Nota-se que o

[4]Best lap é um produto comercial desenvolvido pela Meantime Mobile Creations.

```
public void computeLevel() {
    ...
    int totalScore = perfectCurvesCounter  * PERFECT_CURVE_BONUS
                   + perfectStraightCounter  * PERFECT_STRAIGHT_BONUS
                   + gc_levelManager.getCurrentCountryId()
                   - totalLapTime * SRC_TIME_MULTIPLIER;
    ...
    #ifdef ARENA
    NetworkFacade.setScore(totalScore);
    #endif
}
```

Figura 4.10: Manutenção no método *computeLevel*.

menor valor que pode ser assumido é zero (*score = 0*) por causa do operador condicional ternário. Ou seja, o desenvolvedor finaliza sua manutenção pensando que está tudo bem. Contudo, selecionando a *feature* `ARENA` para um produto da linha, o resultado do jogo pode aparentar que está funcionando corretamente mostrando tanto valores positivos quanto negativos. O problema é: quando o usuário estiver jogando em rede, ele notará que a pontuação de todos os jogadores tão somente irá variar de zero em diante.

```
public class NetworkFacade {
    ...
    public static void setScore(int s) {
        score = (s < 0) ? 0 : s;
    }
    ...
}
```

Figura 4.11: Método *setScore* da Classe *NetworkFacade*.

Logo, a manutenção impactou no funcionamento correto do sistema. Toda vez que a funcionalidade `ARENA` estiver definida o sistema se comportará de maneira inesperada quando em rede.

A Figura 4.12 mostra como o código foi modificado a fim de permitir chamada VoIP[5] no produto. Esse trecho de código foi extraído do *Bugzilla*[6] do produto *Empathy*[7], parte integrante do projeto *GNOME*. *Empathy* é um programa de mensagens que suporta conversas baseada em texto, voz, e vídeo bem como transferência de arquivos sobre diferentes protocolos. Observa-se

[5]VoIP (abreviatura de *Voice over Internet Protocol*) é uma tecnologia que permite o uso da internet para transmissão de voz.
[6]https://bugzilla.gnome.org/show_bug.cgi?id=482190
[7]http://live.gnome.org/Empathy

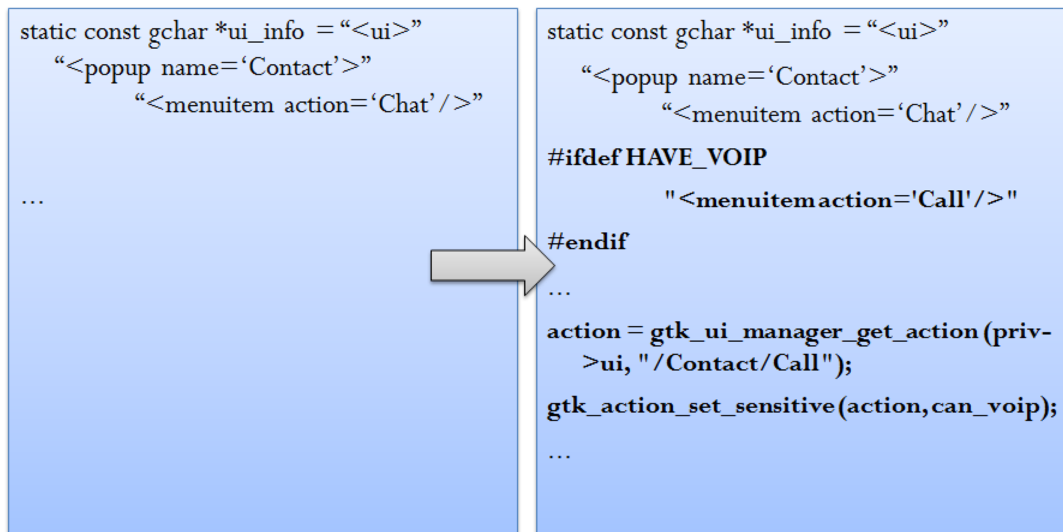

Figura 4.12: *Bug* no produto *Empathy*.

que a *feature* `HAVE_VOIP` possui uma ação igual a *Call*. Porém, mais adiante, o desenvolvedor passa como parâmetro diretamente o caminho "/Contact/Call" sem circundar o trecho de código com `#ifdef HAVE_VOIP`. Desse modo, o método *gtk_action_set_sensitive* é chamado com a variável $action = null$ se a funcionalidade VoIP for disabilitada. Logo, o programa *Empathy* não compila sem a funcionalidade VoIP definida. Os exemplos supracitados ilustram como o processo de manutenção em código pode causar problemas no funcionamento de algumas *features* e, até mesmo, em variantes da linha.

Outro problema surge quando se trata do paradigma de orientação a objetos, devido ao impecilho de assimilar o conceito de forma a utilizá-lo adequadamente [12]. Neste contexto, a regra 4.13 propõe resolver a questão de método sobrescrito.

$$\frac{(isOverride(m) \wedge isCalled(m)) \; in \; D \; extends \; C}{warning} \tag{4.13}$$

As Figuras 4.13 e 4.14 ilustram como a regra supracitada pode ser útil. Neste cenário, note que já existia uma chamada ao método *m* da classe A (superclasse) através de *instance.m();*. Desta forma, o fluxo de dados transcorria do método *main* da classe B até o método *m* da classe A. Porém, o desenvolvedor por algum motivo sobrecreveu o método *m* da classe pai (vide a linha em negrito da classe B) inserindo um possível *bug* no sistema. A partir de então, o método da classe A não será mais invocado, uma vez que *instance.m();* sempre invocará o método *m* da própria classe B. Embora o comportamento do sistema foi alterado, pode parecer que o mesmo está funcionando corretamente. Contudo, ao longo do tempo, ele irá apresentar falhas que dificilmente serão solucionadas em pouco tempo. A fim de evitar esse tipo de problema, a regra 4.13 verifica se algum método da superclasse já era chamado dentro de uma subclasse e, em seguida, esse método foi sobrescrito na subclasse. Com isso, a regra se adianta a solucionar possíveis problemas futuros.

```
public class A {
    ...
    public void m() { ... }
     ...
}
```

Figura 4.13: Superclasse A.

```
public class B extends A {

   public void m() { ... }

   public static void main(String[] args) {
      B instance = new B();

      instance.m();
   }
}
```

Figura 4.14: Subclasse B.

Manipular *arrays* pode ser trivial, mas os desenvolvedores precisam prestar mais atenção em *arrays* quando estão programando usando as diretivas de pré-processadores, como `#ifdefs` e `#endifs`. Um exemplo bem simples que consta a falta de atenção por parte do programador é apresentado na Figura 4.15. Nas linhas de código subsequentes (não mostrado na figura) existe uma estrutura de repetição (*for*) que percorre todos os itens do *array*. Na primeira versão, o programa estava funcionando normalmente sem apresentar falhas. Entretanto, na versão posterior, modificada pelo programador a fim de extrair uma linha de produto a partir de uma única aplicação, há um erro gritante a nível de compilação condicional. Caso a *feature* `HAVE_UNFAIR` não seja selecionada para alguma variante da linha de produto e, do contrário, a *feature* `HAVE_EXTREME` esteja definida; então, pode ocorrer um erro conhecido como *IndexOutOfBounds*. Isso porque o comando *for* tentará percorrer todos os elementos do *array*, mas não conseguirá devido a indefinição do terceiro elemento (`level[3] = UNFAIR`).

A regra 4.14 tem como objetivo de resolver esse tipo de problema mencionado. Na verdade, essa regra verifica se existe algum item do *array* que não terá atribuição por causa de alguma *feature* não definida, conquanto que ele não seja o último elemento. Sempre que existir um elemento desse tipo acarretará no problema de *IndexOutOfBounds*.

$$\frac{[(exists(getExpression(get(a[0,\ 1,..,n-1]))\ =\ FALSE))\ \wedge\ (a[i]\ \neq\ a[n-1])\ in\ C]}{warning} \quad (4.14)$$

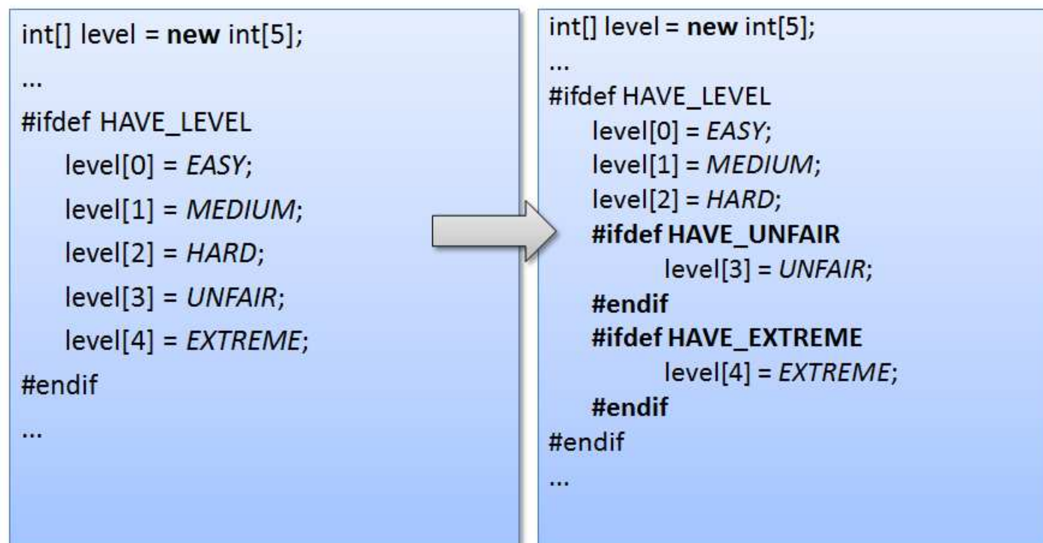

Figura 4.15: Manutenção utilizando *arrays*.

4.2 Funcionamento Geral

Antes de tudo, é importante enfatizar que a soluçao proposta deve funcionar como uma extensão de interfaces emergentes. Via de regra, o usuário do emergo (sistema que implementa a ideia de interfaces emergentes) posiciona o *mouse* sobre o trecho de código que irá prestar manutenção e, então, seleciona-o. O código selecionado passa a ser interpretado como uma lista de elementos de seleção (declarações de variáveis/métodos, inicializações, atribuições, entre outros) que é conhecida como SEL (do inglês, *Selection Elements List*). Imediatamente após a seleção do código, são feitas análises no código baseadas nas análises de fluxo de dados [22] a fim de capturar as dependências entre *features*. Em seguida, uma interface emerge comunicando ao programador sobre os contratos existentes relacionados ao ponto de manutenção. Ou seja, o desenvolvedor precisa selecionar o trecho de código que irá modificar antes de iniciar de fato a manutenção. Feito isso, o emergo faz algumas verificações para alertar o desenvolvedor de uma possível inserção de defeito na LPS ou em uma variante. É importante deixar claro que tudo isso ocorre antes do programador alterar o código. Mas, **a)** se o programador ignorar as interfaces? Ou, **b)** se ele nunca selecionar um ponto de manutenção?

- **a)** caso o desenvolvedor venha a ignorar as interfaces emergentes, o emergo teoricamente cumpriu seu papel, pois, por vontade própria, o desenvolvedor desconheceu as informações contidas nas interfaces. Porém, o emergo não atuará efetivamente junto a manutenção deste desenvolvedor.

- **b)** já se o programador não selecionar o local onde prestará manutenção, então o sistema não acompanhará o processo de manutenção visto que o evento só é disparado após a seleção do código.

Desse modo, para a primeira questão (**a**), realmente, o problema não está na ferramenta e, sim, no desenvolvedor que quis realizar a manutenção sem auxílio. Entretanto, para a segunda questão (**b**), sempre que o programador quiser alterar o código, antes ele precisa selecionar o mesmo. Isso é complicado porque os desenvolvedores não possuem o hábito de selecionar o código para depois alterar. O desenvolvedor normalmente já vai alterando o código e resolvendo os problemas existentes. Além disso, os desenvolvedores não possuem tempo suficiente para pensar cautelosamente na manutenção, tipicamente com prazos em dias ou mesmo em horas [28]. Logo, se o programador não selecionar o código, então o sistema não estará sendo usado.

Sem falar que existem alguns problemas com a ferramenta de interfaces emergentes. Os próprios autores [23] admitem que a ferramenta só computa dependências simples (como mostrado na Figura 4.10). Assim sendo, o emergo ainda deixa escapar *bugs* que poderiam ter sido alertados ao desenvolvedor. Portanto, o desenvolvedor ainda terá problemas no que diz respeito à produtividade.

Assim sendo, a solução proposta é uma extensão do conceito de interfaces emergentes, mas não funciona apenas antes da manutenção. A ideia é continuar com os benefícios de interfaces emergentes e, ao mesmo tempo, acompanhar todo o processo de manutenção do código. Já foi discutido que fica difícil para o programador estar sempre selecionando um ponto de manutenção antes de efetuar às devidas mudanças. Então, devido a forma padrão de salvar em arquivo (usando as teclas de atalho *Ctrl + S*) nos IDEs Eclipse[8], NetBeans[9], Notepad++[10] e Dev-C++[11], por exemplo, a ferramenta que implemente a ideia proposta estará sempre rodando em *background* a medida que o desenvolvedor for salvando o código. Por sinal, os desenvolvedores já estão acostumados com esse famoso atalho para salvar suas modificações no arquivo de programa. Desse modo, os desenvolvedores não precisarão selecionar o código antes de mantê-lo. A partir do momento que o programador dá um *Ctrl + S* é disparado um evento que executa todas as regras a fim de detectar possíveis erros inseridos. Além do mais, se o desenvolvedor estiver alterando o código sem inserir *bugs*, então o mesmo não será incomodado com mensagens desnecessárias.

A solução deve funcionar da seguinte maneira (ver Figura 4.16). Após cada *Ctrl + S*, as regras serão executadass para todas as possibilidades das *features* a fim de capturar as dependências existentes entre o código modificado e o restante. Essas possibilidades são assumidas de acordo com o *feature model* da linha de produto ou com a presença/ausência da funcionalidade em aplicações simples. À medida que as regras vão encontrando problemas, começa-se a ser construída uma mensagem que alertará o desenvolvedor exibindo o(s) problema(s).

Perceba que as regras serão executadas durante toda a manutenção na proporção que o desenvolvedor efetua a ação ‘Salvar’ ou *Ctrl + S*. Ao passo que as interfaces emergentes funcio-

[8]http://eclipse.org/
[9]http://netbeans.org/
[10]http://notepad-plus-plus.org/
[11]http://www.bloodshed.net/devcpp.html

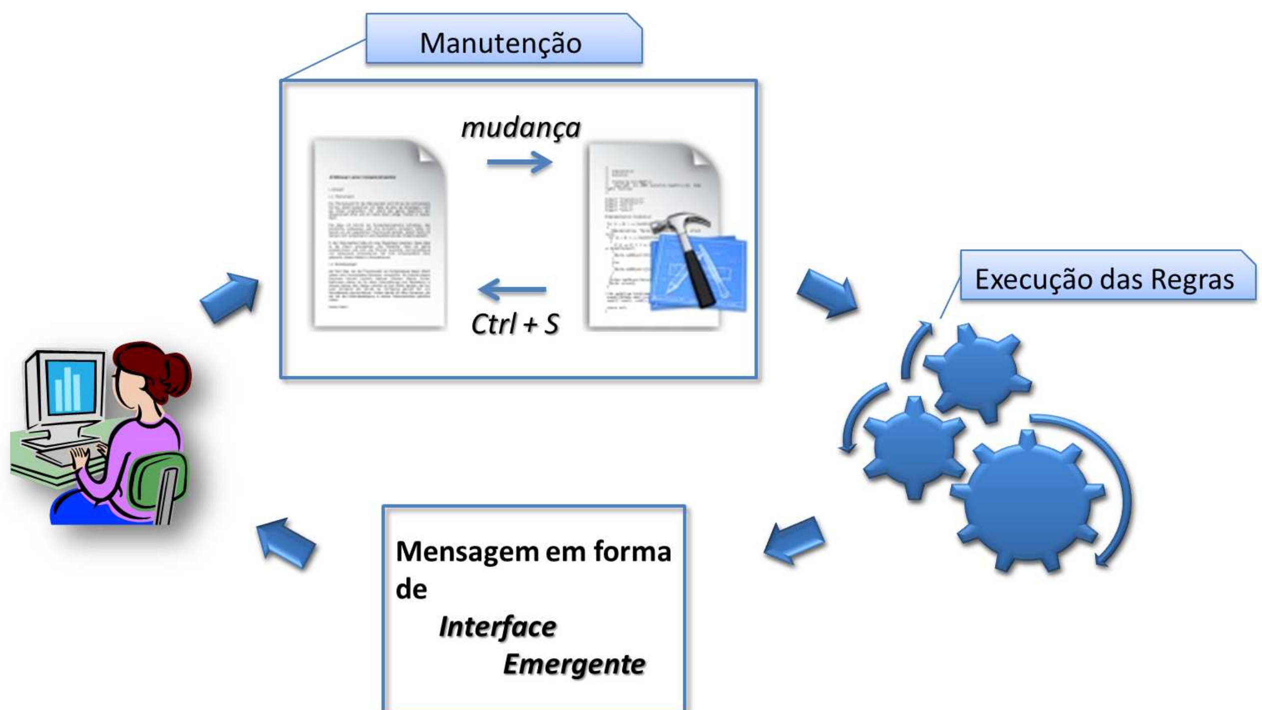

Figura 4.16: Visão de alto nível do funcionamento do assistente inteligente.

nam antes do desenvolvedor alterar o código. Ainda existe uma outra abordagem que verifica se todas as possíveis combinações de uma LPS estão funcionando corretamente. Essa abordagem é conhecida como *Safe Composition* [26], ela parte do pressuposto que nem todas as combinações de *features* produz produtos de software corretos. Assim sendo, o *Safe Composition* faz uma análise em toda a linha de produto utilizando o *feature model* juntamente com suas restrições para garantir que todos os programas que são membros de uma linha de produtos são realmente corretos. Porém, isso acontece após a linha de produto ser codificada ou depois de efetuada a manutenção.

Capítulo 5

Trabalhos Relacionados

Existem diversos trabalhos, como [6], [28] e [24], que propõem investigar manutenções incorretas. Sliwerski *et al.* [24] propuseram uma maneira de localizar indícios de *bugs* usando mineração de repositórios. Eles encontraram que são mais fáceis para os desenvolvedores realizarem mudanças incorretas na sexta-feira. Em [2], foi aplicado aprendizagem de máquina com o objetivo de encontrar programadores ideais para cada correção de defeitos. Já Gu *et al.* [6] estudaram a taxa de manutenções incorretas em três projetos do Apache. O trabalho proposto é complementar aos demais, pois analisou diversos sistemas e linhas de produtos de *software* catalogando os *bugs* encontrados após manutenções. Além disso, nossa solução detecta erros antes mesmo do desenvolvedor realizar o *commit* no código.

Alguns pesquisadores [14] estudaram 30 milhões de linhas de código, escrito na linguagem C, de sistemas que utilizam as diretivas de pré-processadores. Eles descobriram que diretivas como `#ifdefs` são importantes para diversas maneiras de anotar o código. Desse modo, os desenvolvedores podem facilmente introduzir erros no programa. Por exemplo, abrir um parêntese sem fechá-lo ou até mesmo anotar uma variável e depois usá-la globalmente. Nós também analisamos diversos sistemas de *software* implementado com pré-processadores catalogando as dependências entre *features* a fim de propor uma representação (em regras) para solucioná-las.

Com o objetivo de alcançar modularidade com VSoC, interfaces emergentes [22] permitem estabelecer contratos entre trechos de código de *features* distintas. Porém, até então, esta abordagem só captura dependências simples entre *features* [23]. Além disso, a mesma apenas funciona antes da manutenção ocorrer. Nossa proposta é complementar a esta, uma vez que as análises de quebras de dependências serão detectadas conforme o desenvolvedor for alterando o código.

Por fim, devido ao *feature model*, sabe-se que nem todas as combinações de *features* de uma LPS produz produtos de *software* corretos. Dependendo do domínio do problema, a seleção de uma funcionalidade pode exigir a seleção de outras, ou o contrário, selecionando uma *feature* pode excluir ou impedir a seleção de outras (por exemplo, *features* alternativas). Modelos de *features* são mecanismos para modelar as partes comuns e variáveis de LPS. Composição se-

gura (*safe composition*) é usado para garantir que todos os produtos derivados de uma linha de produtos sejam realmente corretos [26]. Thaker et al. [26] determinou restrições de composição para módulos de *feature* (do inglês, *feature modules*) e usou essas restrições para assegurar a composição segura. Contudo, a composição ocorre somente após a manutenção ser completamente finalizada. O desenvolvedor só saberá se existe algum erro em sua manutenção depois de realizar o *commit*. A nossa abordagem difere de *safe composition* já que está inserida junto ao processo de manutenção. Ou seja, a medida que o desenvolvedor for salvando as mudanças no código as regras vão sendo executadas a fim de detectar *bugs* e avisá-lo imediatamente.

Capítulo 6

Conclusão

Este trabalho apresentou uma solução para o problema de manutenções incorretas tanto em simples aplicações quanto em LPS, *Vim* e *Lampiro*, por exemplo.

Em primeiro lugar, os *bug reports* foram analisados a fim de identificar problemas reais concernentes à manutenção de *software* enfrentados por desenvolvedores de diferentes equipes. Como consequência, percebeu-se que as manutenções incorretas se enquadram num relevante problema que merece ser investigado com afinco. Constatou-se que diversos problemas encontrados nos sistemas de armazenamento de *bugs* possuem algumas características comuns. Desse modo, alguns problemas foram categorizados devido a semelhança de erros cometidos no processo de manutenção do código e, consequentemente, foram usados para derivar as regras propostas a fim de alertar os desenvolvedores. Essas regras, inclusive, são bastantes úteis para projetar ferramentas de detecção de *bugs*.

Como trabalho futuro pretende-se implementar o Assistente Inteligente, aqui modelado, para detectar defeitos no processo de modificações no código baseada nas regras apresentadas neste trabalho. Incrementar o número de regras. Depois de construído o assistente, disponibilizá-lo para ser testado pelos programadores. Por fim, planeja-se realizar um estudo mais profundo no que diz respeito aos erros semânticos a fim de adquirir conhecimento e criar novas regras.

Bibliografia

[1] John Anvik, Lyndon Hiew, and Gail C. Murphy. Coping with an open bug repository. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, eclipse '05, pages 35–39, New York, NY, USA, 2005. ACM.

[2] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 361–370, New York, NY, USA, 2006. ACM.

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

[4] B. Boehm and V. R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, 2001.

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

[6] Zhongxian Gu, Earl T. Barr, David J. Hamilton, and Zhendong Su. Has the bug really been fixed? In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 55–64, New York, NY, USA, 2010. ACM.

[7] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[8] Christian Kästner. *Virtual Separation of Concerns: Toward Preprocessors 2.0*. Dissertation, University of Magdeburg, Germany, May 2010.

[9] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 311–320, New York, NY, USA, 2008. ACM.

[10] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting started with aspectj. *Commun. ACM*, 44:59–65, October 2001.

[11] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[12] M. Kolling. The problem of teaching object-oriented programming. *Journal of Object Oriented Programming*, 11(8), 1999.

[13] Christian Kästner. Cide: Decomposing legacy applications into features. In *Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proceedings. Second Volume (Workshops)*, pages 149–150. Kindai Kagaku Sha Co. Ltd., Tokyo, Japan, 2007.

[14] Jörg Liebig, Christian Kästner, and Sven Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of c code. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 191–202, New York, NY, USA, 2011. ACM.

[15] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15:1053–1058, December 1972.

[16] D. L. Parnas. On the design and development of program families. *IEEE Trans. Softw. Eng.*, 2:1–9, January 1976.

[17] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[18] C. V. Ramamoorthy and Wei-Tek Tsai. Advances in software engineering. *Computer*, 29:47–58, October 1996.

[19] Eric S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.

[20] Christian Robottom Reis, Renata Pontin de Mattos Fortes, Renata Pontin, and Mattos Fortes. An overview of the software engineering process and tools in the mozilla project, 2002.

[21] Márcio Ribeiro and Paulo Borba. Towards feature modularization. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, pages 225–226, New York, NY, USA, 2010. ACM.

[22] Márcio Ribeiro, Humberto Pacheco, Leopoldo Teixeira, and Paulo Borba. Emergent feature modularization. In *Proceedings of the ACM international conference companion on*

Object oriented programming systems languages and applications companion, SPLASH '10, pages 11–18, New York, NY, USA, 2010. ACM.

[23] Márcio Ribeiro, Felipe Queiroz, Paulo Borba, Társis Tolêdo, Claus Brabrand, and Sérgio Soares. On the impact of feature dependencies when maintaining preprocessor-based software product lines. In *Proceedings of the 10th ACM international conference on Generative programming and component engineering*, GPCE '11, pages 23–32, New York, NY, USA, 2011. ACM.

[24] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *SIGSOFT Softw. Eng. Notes*, 30:1–5, May 2005.

[25] Henry Spencer. ifdef considered harmful, or portability experience with c news. In *In Proc. Summer'92 USENIX Conference*, pages 185–197, 1992.

[26] Sahil Thaker, Don Batory, David Kitchin, and William Cook. Safe composition of product lines. In *Proceedings of the 6th international conference on Generative programming and component engineering*, GPCE '07, pages 95–104, New York, NY, USA, 2007. ACM.

[27] Salvador Trujillo, Don Batory, and Oscar Diaz. Feature refactoring a multi-representation program into a product line. In *Proceedings of the 5th international conference on Generative programming and component engineering*, GPCE '06, pages 191–200, New York, NY, USA, 2006. ACM.

[28] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 26–36, New York, NY, USA, 2011. ACM.

[29] Trevor J. Young and Trevor J. Young. Using aspectj to build a software product line for mobile devices. msc dissertation. In *University of British Columbia, Department of Computer Science*, pages 1–6, 2005.