

PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Programa de Pós-Graduação em Informática

**EXTRAÇÃO DE LINHAS DE PRODUTOS DE SOFTWARE:
UM ESTUDO DE CASO USANDO COMPILAÇÃO CONDICIONAL**

Marcus Vinícius de Ávila Couto

Belo Horizonte

2010

Marcus Vinícius de Ávila Couto

**EXTRAÇÃO DE LINHAS DE PRODUTOS DE SOFTWARE:
UM ESTUDO DE CASO USANDO COMPILAÇÃO CONDICIONAL**

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

Orientador: Marco Túlio Valente

Belo Horizonte

2010

FICHA CATALOGRÁFICA

Elaborada pela Biblioteca da Pontifícia Universidade Católica de Minas Gerais

C871e Couto, Marcus Vinicius de Ávila
Extração de linhas de produtos de software: um estudo de caso usando
compilação condicional / Marcus Vinicius de Ávila Couto. – Belo
Horizonte, 2010.
72f. : il.

Orientador: Marco Túlio de Oliveira Valente
Dissertação (Mestrado) – Pontifícia Universidade Católica de Minas
Gerais. Programa de Pós-graduação em Informática.
Bibliografia.

1. Engenharia de software – Teses. 2. Software – Desenvolvimento.
3. Software – Reutilização. I. Valente, Marco Túlio de Oliveira.
II. Pontifícia Universidade Católica de Minas Gerais. IV. Título.

CDU: 681.3.03

Bibliotecário: Fernando A. Dias – CRB6/1084



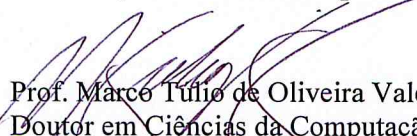
PUC Minas
Programa de Pós-graduação em Informática

FOLHA DE APROVAÇÃO

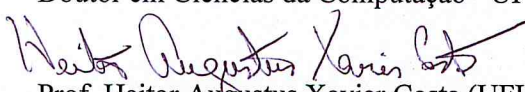
Extração de Linhas de Produtos de Software: Um Estudo de Caso Usando Compilação Condicional

MARCUS VINÍCIUS DE ÁVILA COUTO

Dissertação defendida e aprovada pela seguinte banca examinadora:


Prof. Marco Túlio de Oliveira Valente - Orientador (UFMG)
Doutor em Ciências da Computação - UFMG


Prof. Humberto Torres Marques Neto (PUC Minas)
Doutor em Ciências da Computação - UFMG


Prof. Heitor Augustus Xavier Costa (UFLA)
Doutor em Engenharia Elétrica - Escola Politécnica da USP

Belo Horizonte, 22 de dezembro de 2010.

*Aos meus pais,
por sempre me apoiarem
e estarem ao meu lado.*

AGRADECIMENTOS

Agradeço, primeiramente, aos meus pais, por todo o apoio incondicional que me foi dado em todas as fases da minha vida, e por uma criação que me permitiu sempre dar passos firmes rumo aos meus objetivos. Agradeço também às minhas irmãs, por estarem sempre presentes em minha vida.

Agradeço à minha namorada pelo companheirismo, compreensão e paciência nesses dois anos dedicados intensamente ao mestrado.

Ao professor Marco Túlio de Oliveira Valente, pela confiança, dedicação e disponibilidade. Por todo o direcionamento e incentivo para ir sempre ao próximo passo, sem desanimar. Por me fazer seguir em frente. Sem sua humildade e sabedoria esse trabalho não teria sido possível.

Aos amigos do mestrado, pelos anseios e experiências compartilhados, em especial ao Cristiano Maffort, pelo apoio e incentivo ao meu ingresso na pós-graduação *stricto sensu*.

À secretária acadêmica, Giovanna Silva, pela sua atenção e prestatividade.

Aos professores do PROPPG/Mestrado Informática da PUC Minas, pelo incentivo à busca pelo grau de mestre e pelos conhecimentos transmitidos.

Agradeço à comunidade do ArgoUML pela confiança no trabalho que nos propusemos a desenvolver, e também ao Camilo Ribeiro pelo auxílio no desenvolvimento da linha de produtos ArgoUML-SPL.

Agradeço ao professor Eduardo Magno Figueiredo pelo interesse no trabalho que foi desenvolvido e pelo auxílio na escrita do artigo publicado no CSMR 2011.

Finalmente, agradeço a todos os meus familiares e amigos, que direta ou indiretamente, contribuíram para que eu pudesse chegar a esse momento.

A todos vocês, minha sincera gratidão.

RESUMO

Linhas de Produtos de Software (LPS) é um paradigma de desenvolvimento que visa a criação de sistemas de software personalizáveis. Apesar do crescente interesse em linhas de produtos, pesquisas nessa área geralmente se baseiam em pequenos sistemas sintetizados nos próprios laboratórios dos pesquisadores envolvidos nos trabalhos de investigação. Essa característica dificulta conclusões mais amplas sobre a efetiva aplicação de princípios de desenvolvimento baseado em linhas de produtos de software em sistemas reais. Portanto, a fim de enfrentar a indisponibilidade de linhas de produtos de software públicas e realistas, esta dissertação de mestrado descreve um experimento envolvendo a extração de uma linha de produtos para o ArgoUML, uma ferramenta de código aberto utilizada para projeto de sistemas em UML. Utilizando compilação condicional, foram extraídas oito *features* complexas e relevantes do ArgoUML. Além disso, por meio da disponibilização pública da LPS extraída, acredita-se que a dissertação possa contribuir com outras pesquisas que visem a avaliação de técnicas, ferramentas e linguagens para implementação de linhas de produtos. Por fim, as *features* consideradas no experimento foram caracterizadas por meio de um conjunto de métricas específicas para linhas de produtos. A partir dos resultados dessa caracterização, a dissertação também destaca os principais desafios envolvidos na extração de *features* de sistemas reais e complexos.

Palavras-chave: Linhas de Produtos de Software, LPS, pré-processadores, compilação condicional, ArgoUML, ArgoUML-SPL.

ABSTRACT

Software Product Line (SPL) is a development paradigm that targets the creation of variable software systems. Despite the increasing interest in product lines, research in the area usually relies on small systems implemented in the laboratories of the authors involved in the investigative work. This characteristic hampers broader conclusions about industry-strength product lines. Therefore, in order to address the unavailability of public and realistic product lines, this dissertation describes an experiment involving the extraction of a SPL for ArgoUML, an open source tool widely used for designing systems in UML. Using conditional compilation we have extracted eight complex and relevant features from ArgoUML, resulting in a product line called ArgoUML-SPL. By making the extracted SPL publicly available, we hope it can be used to evaluate the various flavors of techniques, tools, and languages that have been proposed to implement product lines. Moreover, we have characterized the implementation of the features considered in our experiment relying on a set of product-line specific metrics. Using the results of this characterization, it was possible to shed light on the major challenges involved in extracting features from real-world systems.

Keywords: Software Product Lines, SPL, preprocessor, conditional compilation, ArgoUML, ArgoUML-SPL.

LISTA DE FIGURAS

FIGURA 1	Exemplo de código anotado	19
FIGURA 2	Exemplo de problemas decorrentes do uso de pré-processadores	20
FIGURA 3	Exemplo de código colorido usando a ferramenta CIDE	22
FIGURA 4	Logging no Tomcat (exemplo de código espalhado e entrelaçado) (HILSDALE; KERSTEN, 2001)	25
FIGURA 5	AspectJ (exemplo de transversalidade dinâmica)	26
FIGURA 6	AspectJ (exemplo de transversalidade estática)	27
FIGURA 7	Tela principal do ArgoUML	39
FIGURA 8	Estrutura interna de um subsistema no ArgoUML	41
FIGURA 9	Principais subsistemas da arquitetura do ArgoUML	42
FIGURA 10	ArgoUML: Subsistemas Externos	42
FIGURA 11	ArgoUML: Subsistemas de Baixo Nível	43
FIGURA 12	ArgoUML: Subsistemas de Controle e Visão	43

FIGURA 13	Modelo de <i>features</i> do ArgoUML-SPL	45
FIGURA 14	Processo de Extração do ArgoUML-SPL	47
FIGURA 15	Identificação do tipo das anotações	50
FIGURA 16	Exemplo de contagem das métrica SD e TD	54
FIGURA 17	Exemplo de anotação do tipo <code>ClassSignature</code>	57
FIGURA 18	Exemplo de anotação do tipo <code>InterfaceMethod</code>	58
FIGURA 19	Exemplo de anotação do tipo <code>MethodBody</code>	58
FIGURA 20	Exemplo de anotação do tipo <code>Expression</code>	59
FIGURA 21	Exemplo de anotação do tipo <code>BeforeReturn</code> e <code>NestedStatement</code> ...	61

LISTA DE TABELAS

TABELA 1	Comentários sobre os trabalhos apresentados	37
TABELA 2	Constantes utilizadas para anotação de <i>features</i> no ArgoUML-SPL .	48
TABELA 3	Métricas de Tamanho para Produtos	52
TABELA 4	Métricas de Tamanho para <i>Features</i>	52
TABELA 5	<i>Scattering Degree</i> (SD)	55
TABELA 6	<i>Tangling Degree</i> (TD)	55
TABELA 7	Métricas de Granularidade Grossa	60
TABELA 8	Métricas de Granularidade Fina	60
TABELA 9	Métricas de Localização	62

SUMÁRIO

1	INTRODUÇÃO.....	12
1.1	Motivação	12
1.2	Objetivos.....	13
1.3	Visão Geral da Solução Proposta	14
1.4	Estrutura da Dissertação	14
2	LINHAS DE PRODUTOS DE SOFTWARE.....	16
2.1	Introdução	16
2.2	Tecnologias para Implementação de Linhas de Produtos	17
2.2.1	<i>Tecnologias Baseadas em Anotações</i>	17
2.2.1.1	<u>Anotações Textuais</u>	18
2.2.1.2	<u>Anotações Visuais</u>	20
2.2.2	<i>Tecnologias Baseadas em Composição</i>	23
2.2.2.1	<u>Orientação a Aspectos</u>	23
2.2.2.2	<u>Orientação a Features</u>	27
2.3	Experiências de Extração de Linhas de Produtos.....	28
2.3.1	<i>Experiências usando Anotações</i>	28
2.3.2	<i>Experiências usando Composição</i>	30
2.3.3	<i>Outras Experiências</i>	35
2.3.4	<i>Comentários Finais</i>	36
3	ARGOUML-SPL.....	38
3.1	Introdução	38

3.2	Arquitetura do ArgoUML	40
3.3	Features	43
3.4	Processo de Extração	46
3.4.1	<i>Anotação do Código</i>	47
3.4.2	<i>Coleta de Métricas</i>	49
3.5	Comentários Finais	50
4	AVALIAÇÃO.....	51
4.1	Caracterização da LPS Extraída	51
4.1.1	<i>Métricas de Tamanho</i>	51
4.1.2	<i>Métricas de Transversalidade</i>	53
4.1.3	<i>Métricas de Granularidade</i>	56
4.1.4	<i>Métricas de Localização</i>	60
4.2	Discussão	62
4.2.1	<i>Formas Recorrentes de Features</i>	62
4.2.2	<i>Modularização Por Meio de Aspectos: É Viável?</i>	64
4.3	Comentários Finais	65
5	CONCLUSÕES	66
5.1	Contribuições	66
5.2	Trabalhos Futuros	67
	REFERÊNCIAS	69

1 INTRODUÇÃO

1.1 Motivação

Linhas de Produtos de Software (LPS) constituem uma abordagem emergente para projeto e implementação de sistemas que almeja promover o reúso sistemático de componentes de software (CLEMENTS; NORTHROP, 2002; SUGUMARAN; PARK; KANG, 2006). O objetivo final dessa abordagem é migrar para uma cultura de desenvolvimento onde novos sistemas são derivados a partir de um conjunto de componentes e artefatos comuns (os quais constituem o núcleo da linha de produtos).

Os princípios básicos de linhas de produtos de software foram propostos há cerca de dez anos (CLEMENTS; NORTHROP, 2002; SUGUMARAN; PARK; KANG, 2006) (ou até há mais tempo, visto que em essência eles apenas procuram sistematizar estratégias de reúso que norteiam o projeto e implementação de sistemas (PARNAS, 1972, 1976)). No entanto, de um ponto de vista de implementação, ainda não se tem conhecimento de implementações públicas de linhas de produtos de software envolvendo sistemas de médio para grande porte. Normalmente, trabalhos de pesquisa nessa área se valem de sistemas de pequeno porte, integralmente implementados em laboratório pelos próprios autores desses trabalhos. Como exemplo, pode-se citar as seguintes linhas de produtos: *Expression Product Line* (LOPEZ-HERREJON; BATORY; COOK, 2005), *Graph Product Line* (LOPEZ-HERREJON; BATORY, 2001) e *Mobile Media Product Line* (FIGUEIREDO et al., 2008).

Certamente, linhas de produtos sintetizadas em laboratórios são úteis para demonstrar, avaliar e promover os princípios básicos dessa abordagem de desenvolvimento. Adicionalmente, elas tornam mais simples a investigação e a aplicação de novas técnicas, ferramentas e linguagens com suporte a linhas de produtos. Por outro lado, devido a sua própria natureza, sistemas de laboratório não permitem uma avaliação mais precisa sobre a escalabilidade e a aplicabilidade de abordagens de desenvolvimento orientadas por linhas de produtos. Existem diferenças fundamentais entre o projeto de sistemas pequenos (*programming-in-the-small*) e o projeto de sistemas complexos, compostos por vários mó-

dulos (*programming-in-the-large*) (DEREMER; KRON, 1975; BROOKS, 1995). Em resumo, uma vez que linhas de produtos são um recurso para reúso em larga escala, é fundamental avaliar se os resultados de pesquisas recentes nessa área – via de regra obtidos em laboratórios – podem ser extrapolados para ambientes e sistemas reais de programação.

Diante da escassez de linha de produtos de software de tamanho e complexidade consideráveis e publicamente disponíveis para a comunidade acadêmica, a criação de um software dessa natureza se faz necessária para que novos e mais aprofundados estudos possam ser desenvolvidos. Assim, apresenta-se uma linha de produtos extraída a partir de um software maduro e de reconhecida utilidade, o ArgoUML. A linha de produtos extraída, denominada ArgoUML-SPL encontra-se publicamente disponível no sítio <http://argouml-spl.tigris.org>, e com isso espera-se que outros pesquisadores se interessem pelo seu desenvolvimento e evolução, assim como a utilizem para apoiar seus estudos.

1.2 Objetivos

O principal objetivo desta dissertação de mestrado é disponibilizar publicamente uma linha de produtos de software funcional, criada a partir de um sistema real, complexo, maduro e de reconhecida relevância em sua área de atuação.

Os objetivos específicos são:

- Extrair uma LPS de um sistema real, não trivial, maduro e de relevância em sua área de atuação;
- Disponibilizar a LPS extraída para a comunidade de pesquisadores da área;
- Analisar e caracterizar as *features* extraídas segundo as seguintes características: tamanho, transversalidade, granularidade e localização sintática dos trechos de código responsáveis pela sua implementação;
- Classificar as *features* extraídas segundo padrões de codificação utilizados na literatura;
- Avaliar a viabilidade de implementação da linha de produtos extraída por meio de abordagens baseadas na separação física de interesses, tal como aspectos.

1.3 Visão Geral da Solução Proposta

Descreve-se nesta dissertação uma experiência prática de extração de uma linha de produtos de software a partir de um sistema real. O sistema alvo dessa experiência foi o ArgoUML, uma ferramenta de código aberto desenvolvida em Java largamente utilizada para projeto de sistemas em UML. Na versão utilizada, a implementação de diversas *features* opcionais encontra-se entrelaçada com a implementação de funcionalidades mandatórias do sistema. Mais especificamente, descreve-se nesta dissertação uma experiência por meio da qual o código responsável por oito *features* importantes – porém não imprescindíveis ao funcionamento do sistema – foi delimitado usando diretivas de pré-processamento. Com isso, viabilizou-se a geração de produtos do ArgoUML sem uma ou mais dessas *features* opcionais. A versão analisada do ArgoUML possui cerca de 120 KLOC, sendo que cerca de 37 KLOC foram marcadas como pertencentes a pelo menos uma das oito *features* consideradas no estudo. Segundo pesquisas efetuadas na literatura recente sobre linhas de produtos, tais números são bastante superiores a quaisquer outras experiências visando a extração de *features*.

A refatoração do ArgoUML deu origem a uma linha de produtos chamada ArgoUML-SPL. As seguintes *features* opcionais compõem essa linha de produtos: DIAGRAMA DE ESTADOS, DIAGRAMA DE ATIVIDADES, DIAGRAMA DE SEQUÊNCIA, DIAGRAMA DE COLABORAÇÃO, DIAGRAMA DE CASOS DE USO, DIAGRAMA DE IMPLANTAÇÃO, SUPORTE COGNITIVO e LOGGING. Tais *features* foram selecionadas devido ao fato de elas representarem requisitos funcionais relevantes (como no caso dos diagramas UML), opcionais (como no caso do SUPORTE COGNITIVO) ou exemplos clássicos de requisitos não funcionais (como no caso de LOGGING).

Adicionalmente, apresenta-se uma análise e uma caracterização detalhada das *features* da LPS extraída. Basicamente, nessa análise procurou-se fornecer informações sobre o tamanho das *features* extraídas (em linhas de código), grau de transversalidade dessas *features*, granularidade dos componentes responsáveis pela implementação e localização sintática dos trechos de código responsáveis pela implementação dessas *features*. Também apresenta-se uma classificação das *features* extraídas segundo padrões de codificação encontrados na literatura.

1.4 Estrutura da Dissertação

O restante deste texto está organizado conforme descrito a seguir:

- O Capítulo 2 apresenta conceitos básicos sobre linhas de produtos de software e sobre as tecnologias mais utilizadas para sua criação. Ainda nesse capítulo, a Seção 2.3 apresenta algumas experiências de outros autores em trabalhos semelhantes ao proposto nesta dissertação de mestrado;
- No Capítulo 3, apresenta-se uma visão geral da linha de produtos proposta, denominada ArgoUML-SPL, incluindo uma descrição da arquitetura do sistema base (ArgoUML) e das *features* extraídas nessa linha de produtos. A Seção 3.4 descreve o processo de extração do ArgoUML-SPL, detalhando a metodologia usada para delimitação das *features* por meio de diretivas de pré-processamento;
- O Capítulo 4 apresenta uma análise sobre as experiências aprendidas durante o processo de criação do ArgoUML-SPL. Nesse capítulo, realiza-se uma análise quantitativa das *features* extraídas, usando para isso métricas de tamanho, transversalidade, granularidade e localização no código (Seção 4.1). Além disso, a Seção 4.2 discute como as *features* extraídas podem ser classificadas, de acordo com alguns padrões normalmente utilizados na literatura. Essa seção apresenta também alguns desafios e problemas que seriam enfrentados caso se desejasse modularizar as *features* consideradas no trabalho usando orientação a aspectos;
- O Capítulo 5 conclui a dissertação, destacando suas principais contribuições e apresentando possíveis linhas de trabalhos futuros.

2 LINHAS DE PRODUTOS DE SOFTWARE

2.1 Introdução

O principal objetivo do desenvolvimento de sistemas que baseiam-se em linhas de produtos de software é migrar para uma cultura de desenvolvimento onde novos sistemas são sempre derivados a partir de um conjunto de componentes e artefatos comuns, os quais constituem o núcleo da linha de produtos (CLEMENTS; NORTHROP, 2002). Com isso, é possível criar produtos customizáveis segundo as necessidades específicas de cada cliente a um custo mais baixo que o desenvolvimento de software contratado por clientes individuais. Os componentes que adicionam funcionalidades ao código base de uma linha de produtos são denominados *features* (KÄSTNER; APEL; BATORY, 2007). Uma *feature* deve sempre representar incremento de uma funcionalidade relevante aos usuários do sistema. Assim, um produto específico de uma LPS é gerado compondo os componentes do núcleo com componentes que implementam as *features* particulares desse produto. Dessa maneira, além de componentes do núcleo, uma linha de produtos inclui componentes responsáveis pela implementação de *features* que são necessárias em determinados domínios ou ambientes de uso.

Existem três modelos predominantes para criação de linhas de produtos de software: proativo, reativo e extrativo (KRUEGER, 2001; CLEMENTS; KREUGER, 2002). Esses três modelos são detalhados nos próximos parágrafos.

O modelo proativo para linhas de produtos assemelha-se ao modelo de ciclo de vida em cascata, conhecido por ser o primeiro modelo definido em Engenharia de Software e por ser a base de muitos ciclos de vida utilizados hoje em dia. Nesse modelo, cada atividade deve ser finalizada antes que a próxima atividade possa ser iniciada (PRESSMAN, 2005). Dessa maneira, através da abordagem proativa, a organização analisa, projeta e implementa uma linha de produtos para apoiar completamente o escopo dos sistemas necessários no horizonte previsível. É um modelo apropriado quando os requisitos para o conjunto de produtos a serem criados são estáveis e podem ser definidos antecipadamente.

O modelo reativo assemelha-se ao modelo de ciclo de vida em espiral e ao *Extreme Programming* (XP). Ambos os modelos tem por finalidade o desenvolvimento do software através de uma série de iterações, sendo que a cada iteração há um incremento de funcionalidades no software em desenvolvimento (PRESSMAN, 2005). Segundo esse modelo, a organização incrementalmente cresce sua linha de produtos quando há demanda para novos produtos ou quando surgem novos requisitos para os produtos existentes. Essa abordagem é adequada em situações nas quais não é possível prever os requisitos para as variações dos produtos.

A abordagem extrativa reusa um ou mais software existentes para a base inicial da linha de produtos. Para ser considerada uma escolha efetiva, essa abordagem não deve requerer tecnologias complexas para desenvolvimento da linha de produtos e deve possibilitar o reuso do software existente sem a necessidade de um alto grau de reengenharia. É uma técnica apropriada quando existe a disponibilidade de um ou mais sistemas com finalidades semelhantes para reutilização e entre esses sistemas existem diferenças consistentes que caracterizam funcionalidades relevantes aos usuários (*features*) (KRUEGER, 2001; CLEMENTS; KREUGER, 2002). Essa abordagem visa a criação de uma linha de produtos a partir da extração de *features* do código fonte original de um sistema base. É uma técnica também conhecida como refatoração orientada a *features* (BATORY, 2004). Por fim, trata-se de um problema desafiador, porque, geralmente sistemas legados não foram projetadas para serem facilmente refatorados (KÄSTNER; APEL; BATORY, 2007). Essa é a abordagem utilizada nesta dissertação de mestrado.

2.2 Tecnologias para Implementação de Linhas de Produtos

As tecnologias utilizadas para implementação de uma linha de produtos de software podem ser divididas em dois grupos: (i) baseadas em composição e (ii) baseadas em anotações (KÄSTNER; APEL; KUHLEMAN, 2008). Esses dois grupos são detalhados nas próximas subseções.

2.2.1 Tecnologias Baseadas em Anotações

As tecnologias baseadas em anotações propõem que o código fonte das *features* da linha de produtos mantenha-se entrelaçado ao código base do sistema, sendo a identificação dessas *features* feita por meio de anotações explícitas. Anotações explícitas são aquelas em que há a necessidade de acrescentar meta-informações diretamente no código fonte do sistema, a fim de delimitar o código que deverá ser analisado por um pré-processador.

Um pré-processador é um programa automaticamente executado antes do compilador com a finalidade de examinar o código fonte a ser compilado e, de acordo com as meta-informações nele adicionadas, ele deve executar certas modificações antes de repassá-lo ao compilador.

As anotações explícitas podem ser de dois tipos: textuais e visuais. Anotações textuais são aquelas em que o código fonte é anotado por meio da utilização de diretivas especiais entendidas pelo pré-processador, tais como as diretivas de compilação `#ifdef` e `#endif`, utilizadas pelas linguagens C/C++(KERNIGHAN; RITCHIE, 1988). Anotações visuais são aquelas em que há a utilização da camada de visualização de uma IDE para a anotação do código a ser pré-processado. As próximas seções apresentam mais detalhadamente as anotações textuais e uma ferramenta que permite a criação de anotações visuais.

2.2.1.1 Anotações Textuais

A técnica conhecida por pré-processamento baseia-se na utilização de diretivas de compilação para informar a um pré-processador quais trechos de código deverão ser incluídos ou excluídos da compilação do sistema. Diretivas de compilação correspondem a linhas de código que não são compiladas, sendo dirigidas ao pré-processador, que é chamado pelo compilador antes do início do processo de compilação propriamente dito. Portanto, o pré-processador modifica o programa fonte, entregando ao compilador um programa modificado de acordo com as diretivas analisadas nesse processo.

Os exemplos mais conhecidos de diretivas de compilação são as diretivas `#ifdef`, `#ifndef`, `#else`, `#elif` e `#endif` utilizadas pelo pré-processador das linguagens C/C++. As diretivas `#ifdef` e `#ifndef` são utilizadas para marcar o início de um bloco de código que somente será compilado caso as condições condicionais associadas a essas diretivas sejam atendidas. Isso é feito por meio da análise da expressão que segue as diretivas, como no exemplo `#ifdef [expressão]`. As diretivas `#else` e `#elif` demarcam o bloco de código que deverá ser compilado caso o resultado das expressões associadas às diretivas `#ifdef` seja falso. A diretiva `#endif` é utilizada para delimitar o fim do bloco de código anotado(KERNIGHAN; RITCHIE, 1988).

Em Java não há suporte nativo a compilação condicional, portanto não existem diretivas de pré-processamento. No entanto, existem ferramentas de terceiros que acrescentam suporte a essa técnica, tal como a ferramenta `javapp`¹. Essa ferramenta disponibiliza

¹Disponível em <http://www.slashdev.ca/javapp/>

diretivas similares às aquelas existentes em C/C++, incluindo `#ifdef`, `#ifndef` e `#else`. Basicamente, tais diretivas informam ao pré-processador se o código fonte delimitado pela diretiva deve ou não ser compilado. Dessa maneira, no desenvolvimento de uma linha de produtos, é possível selecionar os trechos de código que estarão presentes na versão final de um determinado produto. A Figura 1 apresenta um exemplo de código pertencente a linha de produtos ArgoUML-SPL – a ser detalhada no Capítulo 3 – que foi anotado com a utilização das diretivas providas pela ferramenta **javapp**. Nessa figura, apresenta-se um trecho de código referente à *feature* SUPORTE COGNITIVO. Esse trecho de código tem por finalidade selecionar o tipo de painel de itens-a-fazer que será exibido na tela do sistema para produtos com ou sem essa *feature*. O código da linha 3 foi anotado como pertencente a essa *feature*, ou seja, esse código será incluído apenas em produtos que tenham selecionado tal *feature*. Por outro lado, o código da linha 5 somente será incluído em produtos que não tenham selecionado tal *feature*.

```

1 JPanel todoPanel;
2 // #if defined(COGNITIVE)
3 todoPanel = new ToDoPane(splash);
4 // #else
5 todoPanel = new JPanel();
6 // #endif

```

Figura 1: Exemplo de código anotado

Diretivas de pré-processamento são conhecidas por sua capacidade de poluir o código com anotações extras, tornando-o menos legível e mais difícil de entender, manter e evoluir (SPENCER, 1992; ADAMS et al., 2009; APEL; KÄSTNER, 2009). Essa técnica pode introduzir erros de difícil detecção em uma inspeção manual. A Figura 2 apresenta um bloco de código extraído do ArgoUML-SPL. Esse código pertence a uma classe que implementa uma fábrica para criação de diagramas (Padrão *Factory Method* (GAMMA et al., 1994)). O código anotado tem por finalidade incluir na compilação apenas os trechos de código das *features* DIAGRAMA DE ESTADOS e/ou DIAGRAMA DE ATIVIDADES, caso essas variabilidades tenham sido escolhidas em um determinado produto. É interessante notar que o bloco de código mostrado nessa figura é compartilhado entre essas *features*, conforme pode ser observado nas linhas 2, 7 e 19, em que há expressões booleanas envolvendo as constantes definidas para as anotações dessas *features*. Essa situação demanda maior atenção em eventuais manutenções, uma vez que quaisquer alterações irão ser refletidas não apenas em uma funcionalidade. Observa-se que, devido à granularidade fina do código marcado, operadores booleanos e chaves de fechamento de bloco de código são marcados, como pode ser visto nas linhas 7 a 9 (operador booleano) e 19 a 21 (chave de fe-

chamento de bloco). Marcações de granularidade fina como essas aumentam a propensão a erros (KäSTNER; APEL; KUHLEMANN, 2008).

```

1 ...
2 // #if defined(STATEDIAGRAM) or defined(ACTIVITYDIAGRAM)
3 if ((
4     // #if defined(STATEDIAGRAM)
5     type == DiagramType.State
6     // #endif
7     // #if defined(STATEDIAGRAM) and defined(ACTIVITYDIAGRAM)
8     ||
9     // #endif
10    // #if defined(ACTIVITYDIAGRAM)
11    type == DiagramType.Activity
12    // #endif
13    )
14    && machine == null) {
15    diagram = createDiagram(diagramClasses.get(type), null, namespace);
16 } else {
17 // #endif
18    diagram = createDiagram(diagramClasses.get(type), namespace, machine);
19 // #if defined(STATEDIAGRAM) or defined(ACTIVITYDIAGRAM)
20 }
21 // #endif
22 ...

```

Figura 2: Exemplo de problemas decorrentes do uso de pré-processadores

Por outro lado, pré-processadores também possuem algumas vantagens, incluindo a facilidade de utilização. Mais importante, pré-processadores permitem a anotação de qualquer trecho de código (APEL; KÄSTNER, 2009). Ou seja, eles constituem a tecnologia mais primitiva e ao mesmo tempo mais poderosa para marcação de *features*.

Apesar de ser uma técnica muito criticada devido aos vários efeitos negativos que ela acrescenta à manutenção e à qualidade do código, anotações textuais podem ser adequadas para a criação de linhas de produtos a partir de sistemas legados, desde que haja o apoio de ferramentas facilitadoras. A próxima seção apresenta uma ferramenta que tem por finalidade exatamente minimizar os problemas decorrentes do uso de anotações textuais.

2.2.1.2 Anotações Visuais

Algumas técnicas podem ser utilizadas para minimizar os problemas apresentados por anotações textuais. Uma dessas técnicas é a técnica conhecida como anotações disciplinadas, que é uma abordagem que limita o poder de expressão de anotações para

prevenir erros de sintaxe, sem restringir a aplicabilidade de pré-processadores em problemas práticos (APEL; KÄSTNER, 2009). Basicamente, essa técnica advoga que anotações devem ser inseridas apenas em trechos de código que possuam valor sintático. Para que funcione corretamente, deve haver um mecanismo que permita que os desenvolvedores anotem apenas os elementos dessa estrutura, como classes, métodos ou comandos. Isto requer um esforço extra, uma vez que apenas anotações baseadas na estrutura sintática do programa deverão ser aceitas. Claramente, sem uma ferramenta para controlar as anotações que podem ser efetuadas, essa técnica torna-se inviável.

O paradigma de anotações visuais foi criado com a finalidade de resolver alguns dos problemas inerentes a técnica de anotações textuais por meio do provimento de ferramentas de suporte. Com essa nova técnica, não há a necessidade de adicionar diretivas de compilação ou quaisquer outras formas de anotações textuais no código fonte, o que previne, por exemplo, o ofuscamento do código. Segundo essa técnica, a camada de apresentação das IDEs deve ser utilizada para anotar o código fonte.

A ferramenta CIDE (*Colored IDE*), proposta por Kästner et al., tem por finalidade implementar essa nova abordagem para anotações em código fonte (KÄSTNER; APEL; KUHLEMAN, 2008). Essa ferramenta funciona como um *plugin* para a IDE Eclipse e utiliza a camada de apresentação dessa IDE para efetuar as anotações no código fonte. O CIDE usa a semântica de pré-processadores, isto é, pode ser classificado como uma abordagem anotativa, porém evita a poluição do código fonte. Em outras palavras, essa ferramenta segue o paradigma de *Separação Virtual de Interesses*, ou seja, desenvolvedores não extraem fisicamente o código das *features*, apenas anotam os fragmentos de código no próprio código base original do sistema e utilizam uma ferramenta de suporte para ter diferentes visões do código e para navegar entre as *features* (APEL; KÄSTNER, 2009).

No CIDE, o código responsável pela implementação de cada *feature* é anotado com uma cor de fundo, sendo que fragmentos de código pertencentes a mais de uma *feature* possuem uma cor resultante da mistura das cores das *features* envolvidas. De modo a evitar erros de sintaxe, o CIDE não permite a coloração de trechos arbitrários de código. Apenas elementos estruturais podem ser associados às *features*. Isso é feito através da análise da AST (*Abstract Syntax Tree*) do código fonte, sendo que apenas os nodos dessa árvore podem ser anotados (coloridos). A AST é uma representação em forma de árvore da estrutura sintática abstrata do código fonte. Nessa árvore, cada nó representa uma construção que ocorre no código fonte.

A Figura 3 mostra a utilização do CIDE através da IDE Eclipse. Essa figura mostra

um bloco de código que implementa uma classe denominada **Stack**. O método **push** dessa classe possui trechos de código pertencentes a três diferentes *features*: **SINCRONIZAÇÃO** (vermelho), **PERSISTÊNCIA** (azul) e **LOGGING** (verde). Por meio dessa figura, pode-se observar que trechos de código compartilhados entre duas ou mais *features* apresentam uma cor que resulta da junção das cores dessas *features*. Isso pode ser observado no trecho em amarelo (vermelho + verde), resultante da interseção das *features* **SINCRONIZAÇÃO** e **LOGGING**.

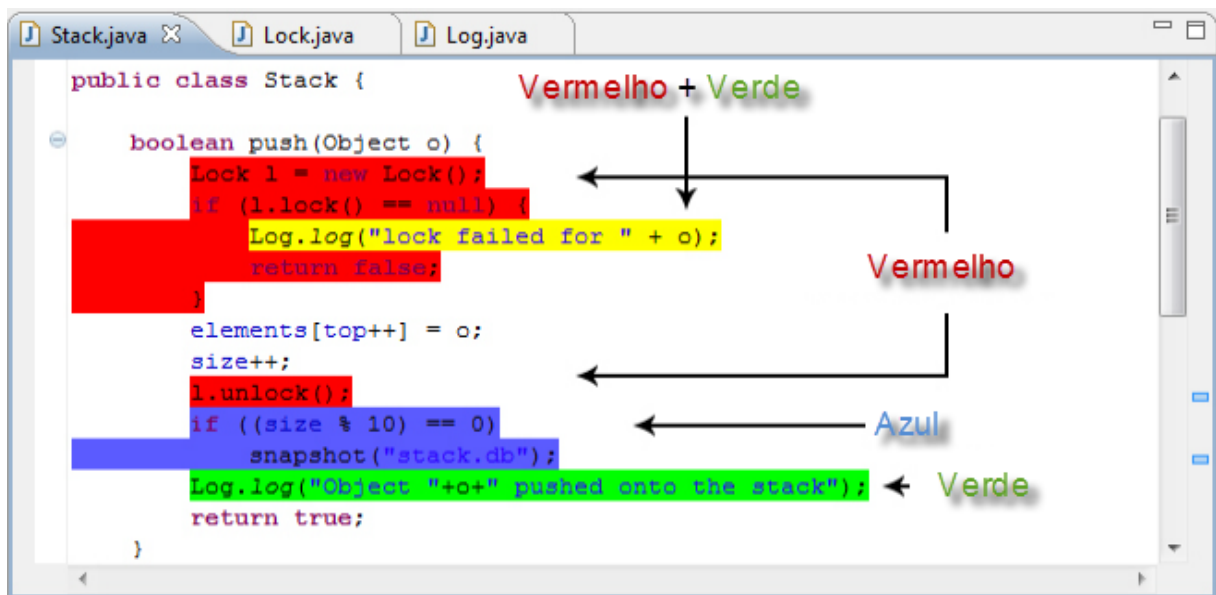


Figura 3: Exemplo de código colorido usando a ferramenta CIDE

O CIDE é uma ferramenta voltada para o desenvolvimento de linhas de produtos de software. Neste sentido, ela possui funcionalidades para gerenciamento das *features*, com a possibilidade de navegação entre diferentes *features* e composição de produtos a partir de uma seleção de *features* realizada pelo usuário. Há também, no CIDE, a possibilidade de fazer projeções dos produtos da LPS. Essa função permite ocultar no editor o código associado a uma *feature*, permitindo que o código restante seja visualizado isoladamente. Com isso, é possível analisar o código fonte resultante de uma determinada seleção de *features*. Finalmente, o CIDE provê um sistema de verificação de tipos que garante que as variantes de uma linha de produtos são bem tipadas. Por exemplo, o sistema de verificação de tipos do CIDE pode detectar situações como métodos que foram removidos de um produto, mas que ainda são chamados em outros pontos do código. Quando um erro de tipos é detectado, a ferramenta sugere uma lista de *quick fixes* que geralmente são restritos à anotação da expressão identificada como incorreta.

As anotações efetuadas com o CIDE são manuais. Por serem manuais, essas ano-

tações tendem a apresentar erros, principalmente em refatorações de aplicações que não são do domínio de conhecimento dos desenvolvedores envolvidos na extração da linha de produtos. Visando melhorar esse cenário, Borges e Valente propuseram um algoritmo para coloração semi-automática do código responsável por implementar as *features* de uma LPS (BORGES; VALENTE, 2009). Esse algoritmo funciona integrado ao CIDE e tem por finalidade auxiliar os desenvolvedores a anotar de forma mais rápida o código das *features* para a extração de uma linha de produtos a partir de um sistema legado.

O algoritmo proposto localiza automaticamente os pontos do código relacionados a uma determinada *feature* e então os marca com uma cor escolhida pelo desenvolvedor. Esse algoritmo recebe como entrada um conjunto de unidades sintáticas responsáveis pela implementação de uma *feature*, denominadas sementes. Basicamente, o algoritmo é dividido em duas fases: (1) fase de propagação, responsável por colorir os trechos de código que diretamente implementam ou usam determinada semente; (2) fase de expansão, responsável por verificar se a vizinhança de um trecho de código colorido deve ser também colorida.

2.2.2 *Tecnologias Baseadas em Composição*

As tecnologias baseadas em composição propõem que cada *feature* seja implementada em um módulo distinto, promovendo a separação física entre o código base do sistema e o código de cada *feature*. Assim, durante a composição do sistema, os desenvolvedores devem escolher os módulos que deverão ser incluídos em um determinado produto a ser gerado. Esse processo geralmente ocorre em tempo de compilação ou em tempo de implantação, o que permite manter o código base separado do código das *features*.

Duas tecnologias baseadas em composição têm sido bastante discutidas recentemente: orientação a aspectos e orientação a *features*. Essas duas tecnologias serão discutidas nas seções seguintes.

2.2.2.1 Orientação a Aspectos

Programação Orientada a Aspectos (ou AOP - *Aspect Oriented Programming*) é uma tecnologia proposta para separação de interesses transversais presentes no desenvolvimento de sistemas. Interesses transversais (ou *crosscutting concerns*) implementam funcionalidades que afetam diferentes partes do sistema. Em outras palavras, para que um determinado interesse transversal seja implementado, é necessário “cortar” (atravessar)

vários módulos do sistema (KICZALES et al., 1997).

A orientação a aspectos tem o objetivo de modularizar a implementação de requisitos transversais. Requisitos transversais geralmente não podem ser adequadamente implementados por meio de programação orientada por objetos tradicional, devido ao fato de que alguns desses requisitos violam a modularização natural do restante da implementação. Usualmente, interesses transversais correspondem a requisitos não funcionais, tais como tempo de resposta, segurança, confiança, persistência, transação, distribuição, tolerância a falhas etc.

Para melhor entender como funciona a orientação a aspectos, é necessária a introdução de alguns conceitos básicos dessa tecnologia. Pontos de junção (ou *join points*) são “pontos bem definidos” da execução de um programa e definem situações em que há a possibilidade de interceptação do fluxo de execução desse programa. Conjuntos de junção (ou *pointcuts*) são conjuntos de pontos de junção. Exemplos de pontos de junção são chamadas e execuções de métodos, chamadas e execuções de construtores, retorno de métodos, retorno de construtores e lançamento e tratamento de exceções. *Advices* são os blocos de código que devem ser executados em um ponto de junção. *Inter-type declarations* permitem a introdução de campos e métodos em classes ou interfaces. Um aspecto caracteriza-se por ser um conjunto de pontos de junção e *advices*, podendo conter ainda *inter-type declarations*.

Na programação orientada por aspectos, os requisitos não-transversais do sistema são modelados por meio de classes e os requisitos transversais são implementados por meio de aspectos. Dessa maneira, uma linguagem orientada por aspectos permite confinar requisitos transversais em módulos fisicamente separados do código base. A orientação a aspectos é, portanto, um complemento à orientação a objetos, não tendo como objetivo a substituição desse paradigma.

A implementação tradicional de interesses transversais – por meio de objetos – apresenta dois principais problemas:

- Código entrelaçado (*code tangling*): quando um requisito transversal encontra-se misturado com código de responsável pela implementação de requisitos não transversais;
- Código espalhado (*code spreading*): quando um requisito transversal encontra-se implementado em diversas classes.

A Figura 4 ilustra a implementação da *feature* LOGGING no Apache Tomcat². Nessa figura, cada barra vertical representa um pacote e as linhas horizontais dentro dessas barras identificam trechos de código que implementam essa *feature*. Essa figura mostra que LOGGING é claramente um requisito transversal cuja implementação está presente de forma entrelaçada e espalhada em diversas classes desse servidor Web.

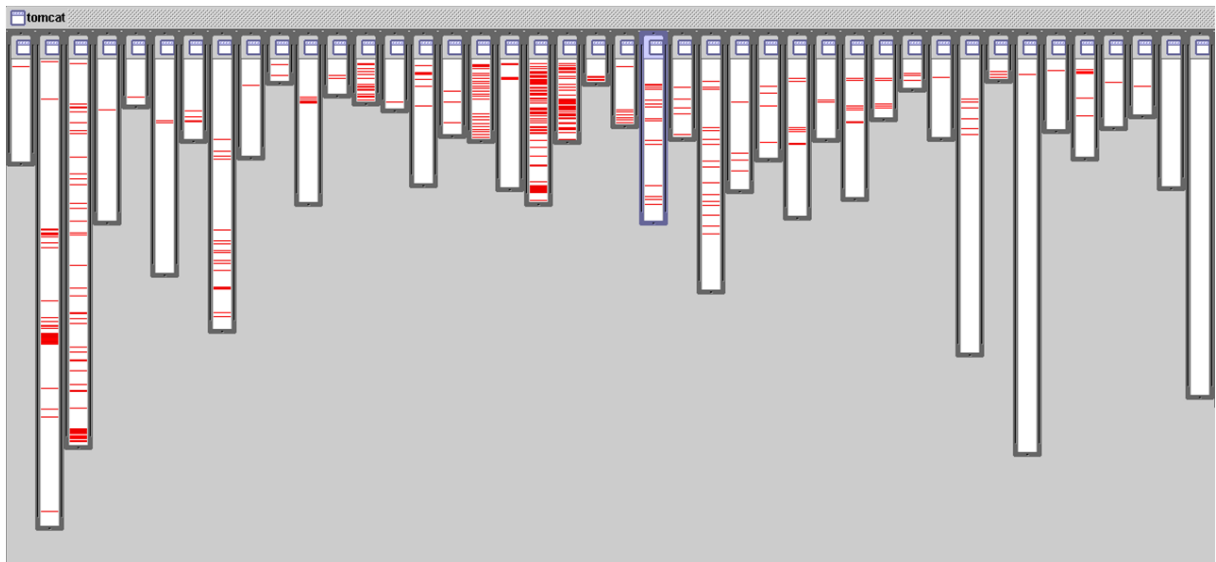


Figura 4: Logging no Tomcat (exemplo de código espalhado e entrelaçado) (HILSDALE; KERSTEN, 2001)

AspectJ: Dentre as linguagens com suporte a AOP, AspectJ é atualmente considerada a mais madura e estável (KICZALES et al., 2001). AspectJ é uma extensão para a linguagem Java com recursos para AOP. AspectJ suporta dois tipos de implementações de requisitos transversais: (i) transversalidade dinâmica, que permite definir implementação adicional em pontos bem definidos da execução de um programa; e (ii) transversalidade estática (*inter-type declarations*), que permite alterar a estrutura estática das classes e interfaces de um programa Java.

A transversalidade dinâmica em AspectJ oferece os seguintes recursos para modularização de interesses transversais: pontos de junção (*join points*), conjuntos de junção (*pointcuts*) e *advices*. Em AspectJ, é possível definir três tipos de *advices*: *advices* executados antes (*before*), após (*after*) ou no lugar de (*around*) pontos de junção. Em resumo, a transversalidade dinâmica permite modificar o comportamento da execução do programa por meio da introdução de *advices* em pontos de junção.

²Disponível em <http://tomcat.apache.org/>

A Figura 5 mostra um aspecto para atualização de um *display* gráfico. Basicamente, esse aspecto faz com que após cada execução de métodos iniciados com a cadeia de caracteres **set** da classe **Figura** ou de suas subclasses, seja chamado o método **refresh** da classe **Display** (linha 4). As linhas 1 a 6 contêm a implementação do aspecto **RefreshingAspect**, as linhas 8 a 17 contêm a implementação da classe **Ponto** e as linhas 19 a 28 contêm a implementação da classe **Linha**. Ambas as classes são subclasses da classe **Figura**. Nesse trecho de código, os pontos de junção (ou *join points*) são representados pela execução dos métodos **set** definidos nas linhas 10, 13, 21 e 24. O conjunto de junção a ser interceptado é representado pela definição do *pointcut* **setPoints** (linha 2). Esse *pointcut* captura a chamada dos métodos iniciados com o conjunto de caracteres **set** em quaisquer subclasses da classe **Figura**. O *advice* definido entre as linhas 3 e 5 faz com que após a execução dos métodos interceptados pelo *pointcut*, o método **refresh** seja executado.

```

1 aspect RefreshingAspect {
2   pointcut setPoints(): execution(void Figura+.set *(..));
3   after(): setPoints() {
4     Display.refresh();
5   }
6 }
7 ...
8 class Ponto extends Figura {
9   ...
10  public void setX(int x) {
11    this.x = x;
12  }
13  public void setY(int y) {
14    this.y = y;
15  }
16  ...
17 }
18 ...
19 class Linha extends Figura {
20   ...
21  public void setP1(Ponto p) {
22    this.p1 = p;
23  }
24  public void setP2(Ponto p) {
25    this.p2 = p;
26  }
27  ...
28 }

```

Figura 5: AspectJ (exemplo de transversalidade dinâmica)

A transversalidade estática por sua vez permite redefinir a estrutura estática dos tipos de um sistema. A transversalidade estática em AspectJ (também conhecida como

inter-type declarations ou *introductions*) permite a introdução de campos e métodos em classes e interfaces, declaração de erros e advertências de compilação e o enfraquecimento de exceções. A Figura 6 apresenta um aspecto que insere dois atributos na classe `Day`, contendo cadeias de caracteres nos idiomas inglês (linha 2) e português (linha 3).

```

1 public aspect Translation {
2     public final static String Day.Description_ENG = "Day ";
3     public final static String Day.Description_PTB = "Dia ";
4     ...
5 }

```

Figura 6: AspectJ (exemplo de transversalidade estática)

A princípio, as tecnologias que podem ser empregadas para extração de linhas de produtos enquadram-se nas abordagens composicional ou anotativa. Entretanto, alguns autores incluem aspectos em ambos os grupos (KÄSTNER; APEL; KUHLEMANN, 2008). O motivo é que embora aspectos encontrem-se fisicamente separados do código fonte do sistema base, eles frequentemente se valem de anotações implícitas para funcionarem corretamente. Anotações implícitas incluem, por exemplo, a introdução de métodos vazios que são usados como “ganchos” para extensões – conhecidos como métodos *hook* – ou o uso de convenções de nomes para simplificar a declaração de conjuntos de junção. No exemplo anterior, a execução de todos os métodos iniciados com a cadeia de caracteres `set` é interceptada por um *pointcut*. Esse exemplo ilustra a utilização de uma convenção de nomes para simplificar a declaração de conjuntos de junção.

2.2.2.2 Orientação a Features

Programação Orientada por *Features* (ou FOP - *Feature Oriented Programming*), assim como orientação por aspectos, é considerada uma técnica moderna para modularização e separação de interesses, particularmente adequada à implementação de requisitos transversais heterogêneos (BATORY, 2004; LIU; BATORY; LENGAUER, 2006). É uma tecnologia criada para síntese de programas em linhas de produtos de software. FOP tem por base a ideia de que sistemas devem ser sistematicamente construídos por meio da definição e composição de *features*, sendo que essas devem ser tratadas como abstrações de primeira classe no projeto de sistemas. Portanto, devem ser implementadas em unidades de modularização sintaticamente independentes. Além disso, deve ser possível combinar módulos que representam *features* de forma flexível, sem que haja perda de recursos de verificação estática de tipos.

Na programação orientada por *features*, assim como na programação orientada por aspectos, classes são usadas para implementar as funcionalidade básicas de um sistema. As extensões, variações e adaptações dessas funcionalidades constituem *features*. Essas *features* são implementadas em módulos sintaticamente independentes das classes do programa. Os módulos criados em FOP podem refinar outros módulos de modo incremental, inserindo ou modificando métodos, atributos ou modificando a hierarquia de tipos.

AHEAD: AHEAD (*Algebraic Hierarchical Equations for Application Design*) é um conjunto de ferramentas que implementa os conceitos básicos de FOP. Seu principal componente, denominado **Jakarta**, é uma linguagem que permite a implementação de *features* em unidades sintaticamente independentes, denominadas refinamentos. Por meio desses refinamentos, é possível adicionar novos campos e métodos em classes do programa base, ou ainda adicionar comportamento extra em métodos existentes. Além da linguagem **Jakarta**, o AHEAD contém um compilador denominado **composer**, responsável por combinar o código das *features* com o código base do sistema, e uma linguagem para descrever as combinações válidas de *features*. Por meio dessa linguagem, o **composer** pode detectar combinações inválidas de *features* ao se tentar gerar um determinado produto da linha de produtos (BATORY, 2004).

2.3 Experiências de Extração de Linhas de Produtos

Nessa seção são apresentadas algumas experiências que envolvem a extração e/ou manutenção de linhas de produtos de software. Na Subseção 2.3.1 é apresentado um estudo que analisou quarenta diferentes linhas de produtos baseadas em técnicas de anotações textuais. A Subseção 2.3.2 apresenta quatro estudos que utilizaram abordagens composicionais para extração de linhas de produtos de software de diferentes tamanhos. A Subseção 2.3.3 apresenta dois estudos recentes para o domínio de dispositivos móveis, um domínio promissor para desenvolvimento de software baseado em linhas de produtos. Por fim, a Subseção 2.3.4 apresenta discussões gerais sobre os resultados desses trabalhos.

2.3.1 Experiências usando Anotações

Liebig et al. realizaram um extenso estudo com o objetivo de avaliar como direti-vas de pré-processamento são usadas para implementar *features* (LIEBIG et al., 2010). O trabalho envolveu a análise de quarenta sistemas (com tamanhos variando entre 10 KLOC até 1 MLOC) de diferentes domínios, todos eles implementados em C. No entanto, como

os sistemas não foram refatorados, as variabilidades consideradas incluem basicamente *features* de baixo nível, normalmente selecionadas por meio de parâmetros de linhas de comando (por exemplo, opções de depuração, otimização ou portabilidade, no caso dos compiladores analisados no trabalho). Via de regra, *features* de baixo nível geralmente não representam requisitos dos *stakeholders* do sistema. Mesmo assim, esses sistemas foram considerados linhas de produtos devido às várias *features* opcionais e alternativas que eles possuem.

Por exemplo, por meio desse estudo, foi possível avaliar a influência que o tamanho de um sistema exerce sobre as variabilidades de uma linha de produtos e qual a complexidade dos mecanismos de compilação condicional utilizados nesses sistemas. Também foi possível mensurar a granularidade e os tipos das expressões condicionais utilizadas para implementação das variabilidades. De posse dessas informações, além da possibilidade de entender melhor o funcionamento desses sistemas, foi possível vislumbrar como melhor empregar técnicas alternativas para implementação de linhas de produtos, tal como aspectos.

Para que fosse possível efetuar esse tipo de análise, foram propostas métricas para inferir e classificar os padrões de uso de compilação condicional nos sistemas avaliados. O conjunto proposto inclui métricas para mensuração da granularidade e espalhamento do código anotado e métricas de tamanho, tais como número de linhas de código (LOC) e linhas de código de *feature* (LOF). Essa última métrica mede o número de linhas de código no interior das diretivas `#ifdef` e `#endif` que delimitam cada *feature*.

Dentre outros, o estudo apresentou os seguintes resultados:

- A variabilidade de um software aumenta com o seu tamanho. Isso pode ser explicado devido ao fato de que sistemas maiores geralmente possuem mais parâmetros de configuração e, conseqüentemente, são mais variáveis. Aproximadamente 23% do código fonte dos sistemas analisados corresponde à implementação de variabilidades. Em alguns sistemas de tamanho médio, tais como o *openvpn* (sistema para criação de redes privadas virtuais), *sqlite* (gerenciador de banco de dados SQL embarcado) e *vim* (editor de texto), mais de 50% do código fonte é utilizado para implementação de variabilidades;
- A complexidade das linhas de produtos baseadas em compilação condicional cresce com o aumento do uso de constantes utilizadas para definição *features* em expressões condicionais (expressões `#ifdef`) e com o aninhamento dessas expressões. Por

outro lado, os autores argumentam que, apesar de ser esperado que sistemas com um alto número de constantes de *features* possuam expressões de *features* mais complexas, não foram encontradas evidências que confirmem essa hipótese. Em muitos dos sistemas avaliados, tais como *freebsd* (sistema operacional), *lynx* (navegador web) e *python* (interpretador), há um elevado número de constantes de *features* e, conseqüentemente, um elevado grau de espalhamento do código dessas *features*. Entretanto, não existe um grande número de expressões condicionais complexas, isto é, expressões que envolvem mais de uma constante de *feature*. Além disso, os dados obtidos revelaram que a maioria das extensões utilizadas são heterogêneas, ou seja, há pouca duplicação de um mesmo trecho de código pelo sistema.

Esse trabalho é limitado pelo fato de utilizar apenas sistemas desenvolvidos em uma única linguagem de programação (C) (LIEBIG et al., 2010). Além disso, foram avaliadas apenas *features* de baixo nível. Outra limitação se refere à classificação das extensões como homogêneas e heterogêneas. Essa classificação foi feita por meio de comparação de sequências de caracteres. Entretanto, por meio desse tipo de comparação fragmentos de código semanticamente equivalentes não são classificados como homogêneos. Por exemplo, um comando $a=b+c$ é semanticamente equivalente ao comando $a=c+b$. Porém, tais atribuições não são classificadas como homogêneas quando apenas suas sequências de caracteres são comparadas.

2.3.2 *Experiências usando Composição*

Muitas das experiências desenvolvidas com a finalidade de extração de linhas de produtos de software a partir de sistemas legados usam abordagens composicionais, na maioria das vezes orientação a aspectos (GODIL; JACOBSEN, 2005; KÄSTNER; APEL; BATTERY, 2007; FIGUEIREDO et al., 2008; ADAMS et al., 2009). Nessa seção, são apresentadas cinco linhas de produtos criadas por autores distintos a fim de avaliar a utilização de aspectos na implementação de linhas de produtos.

Prevayler: Godil e Jacobsen efetuaram a refatoração de um sistema não trivial de código aberto, denominado Prevayler (GODIL; JACOBSEN, 2005). Esse sistema implementa um sistema gerenciador de banco de dados em memória principal, por meio do qual objetos de negócios podem ser persistidos. Nesse trabalho, foram refatoradas as seguintes *features*:

- GERENCIAMENTO DE SNAPSHOT: suporte ao armazenamento em disco de objetos.

Sem esta variabilidade, o armazenamento é feito apenas em memória primária;

- **CENSURA:** recuperação de dados em caso de falha na execução de uma transação. É dependente da variabilidade GERENCIAMENTO DE SNAPSHOT (isto é, sempre que CENSURA for incluída na derivação de um produto, GERENCIAMENTO DE SNAPSHOT também deverá ser incluída);
- **SUORTE A REPLICAÇÃO:** permite a replicação de dados entre um servidor e diversos clientes;
- **LOGGING PERSISTENTE:** suporte ao *logging* de transações realizadas no sistema de arquivos.
- **RELÓGIO:** permite restaurar transações em caso de falhas;
- **MULTI-THREADING:** provê suporte a sistemas com múltiplas *threads*.

Godil e Jacobsen realizaram uma avaliação empírica sobre o código do Prevayler antes e após a refatoração. Essa avaliação foi baseada em dois conjuntos de métricas: “Separação de Interesses” e “Acoplamento, Coesão e Tamanho”. Em princípio, os autores imaginaram que após a refatoração haveria uma redução do número apresentado por essas métricas. Entretanto, ficou mostrado que a redução desses números depende diretamente da natureza do código transversal. Algumas das métricas apresentaram números otimistas em relação a refatoração. Por exemplo, para o código base, houve uma redução de 43% no acoplamento e um aumento de 71% na coesão da versão refatorada para aspectos. Ou seja, através da refatoração do sistema foi possível criar um código base reduzido, mais coeso e menos acoplado. Esses números estão em conformidade com a ideia de que a remoção de funcionalidades transversais resulta em um código menos acoplado e mais coeso. No entanto, para outras métricas, tal como LOC, os números não se mostraram tão otimistas. O resultado dessa métrica mostrou que o tamanho total do sistema na versão refatorada para aspectos aumentou em relação a versão original. Isso ocorreu devido a natureza heterogênea do código transversal refatorado e ao acréscimo de linhas de código para definição de *pointcuts*, *advices* e outras informações necessárias à implementação de aspectos.

Oracle Berkeley DB: O Oracle Berkeley DB, assim como o Prevayler, é um sistema gerenciador de banco de dados que pode ser embarcado em outras aplicações. Ele foi escolhido por Kästner, Apel e Kuhleemann a fim de avaliar a viabilidade de extração de

uma linha de produtos a partir de um software de um domínio bem conhecido e não trivial (o Berkeley DB possui aproximadamente 84 KLOC) (KÄSTNER; APEL; BATORY, 2007). Para essa extração, foi utilizada orientação a aspectos, com a linguagem AspectJ. Foram identificadas um total de 38 *features* nesse sistema, incluindo *features* relacionadas a persistência, transações, cache, *logging*, estatísticas, sincronização de *threads* etc.

A refatoração do Oracle Berkeley DB foi efetuada manualmente nesse estudo, pois não foram encontradas ferramentas que pudessem auxiliar essa atividade de maneira produtiva. Entretanto, mesmo antes que o processo de refatoração chegasse ao fim, o trabalho foi interrompido devido ao fato de ele ter se tornado repetitivo, não sendo mais esperado que novos conhecimentos fossem gerados. Algumas *features* que demandam a anotação de um grande número de linhas de código, tal como o sistema de persistência, não puderam ser refatoradas devido a limitações técnicas da linguagem AspectJ. As *features* efetivamente extraídas correspondem a cerca de 10% do tamanho do sistema.

Os aspectos criados nessa refatoração eram frágeis e difíceis de manter, devido ao fato de estarem intimamente ligados ao código base do sistema. Devido a esse fato, as *features* refatoradas dependiam de detalhes de implementação e estavam fortemente acopladas ao código base. Esta união forte e implícita torna a evolução e a manutenção do código refatorado mais difícil. Por exemplo, não é possível fazer qualquer mudança local sem entender os aspectos ou sem uma ferramenta que produza informações sobre as extensões de uma determinada parte do código. Mudanças no código fonte podem alterar os *join points* e, quando os *pointcuts* não são atualizados de acordo, o conjunto de *join points* definidos pode mudar o comportamento do sistema, comprometendo sua funcionalidade. Segundo os autores, entender e manter o código orientado a aspectos sem uma ferramenta de apoio tornou-se bastante complexo devido à implícita união e fragilidade dos aspectos criados.

Apenas itens básicos de AspectJ foram utilizados na refatoração do Berkeley DB. Mecanismos avançados tais como *if*, *this* ou *cflow* foram usados apenas em casos raros. Extensões homogêneas também foram raramente usadas. Outro problema encontrado se refere ao acesso a variáveis locais, pois extensões criadas com AspectJ não podem acessar variáveis locais de um método. Como AspectJ não pode alterar a assinatura dos métodos existentes, também não foi possível adicionar novas exceções a esses métodos. Por fim, houve problemas com os modificadores de visibilidade (*private*, *protected* e *public*), uma vez que frequentemente foi necessário mudar a visibilidade de algumas dessas variáveis.

Kästner, Apel e Kuhlemann ainda apresentaram outros problemas em relação aos

aspectos que foram criados, tais como dificuldade de leitura e aumento do tamanho do código fonte devido à declaração de *pointcuts* e *advices* (KÄSTNER; APEL; BATORY, 2007). Segundo os autores, AspectJ adiciona diversas complexidades que não são necessariamente inerentes às refatorações de aplicações legadas em LPS. Assim, contrário às expectativas iniciais, os autores depararam-se com situações complexas e inesperadas, o que os levou a declarar que aspectos escritos com a linguagem AspectJ não são adequados para implementação de *features* em refatorações de sistemas legados.

MobileMedia e BestLap: Figueiredo et al. apresentaram um estudo que avaliou quantitativa e qualitativamente os impactos positivos e negativos de orientação a aspectos em várias mudanças aplicadas no código base e no código das *features* de duas linhas de produtos de software para dispositivos móveis (FIGUEIREDO et al., 2008). Nesse estudo, vários cenários de evolução das linhas de produtos foram avaliados com base em métricas para medida de impacto de mudança, modularidade e dependência entre as *features*.

As duas linhas de produtos escolhidas foram MobileMedia e BestLap. MobileMedia é uma linha de produtos para dispositivos móveis com aproximadamente 3 KLOC, cujo objetivo é implementar aplicações para gerência de documentos multimídia, incluindo fotos, vídeo e músicas. BestLap é um projeto comercial com aproximadamente 10 KLOC e que pode ser utilizado em 65 dispositivos móveis diferentes. Esse software é um jogo em que os jogadores devem tentar alcançar a *pole position* em uma pista de corrida. Ambos os sistemas foram implementados em Java e AspectJ. As versões Java foram utilizadas como suporte para análise das versões que utilizavam aspectos. Nas versões Java, compilação condicional foi o mecanismo utilizado para implementação de variabilidades. As versões iniciais dos sistemas, tanto em Java como em AspectJ, foram obtidas com parceiros. A partir dessas versões, foram efetuadas diversas evoluções, que foram analisadas pelos autores. Por exemplo, para o MobileMedia, foram criadas sete novas versões para linguagem Java com equivalentes em AspectJ.

Esse estudo foi dividido em três grandes fases: (1) projeto e compreensão dos cenários de mudanças das linhas de produtos; (2) preparação das versões das linhas de produtos; e (3) avaliação quantitativa e qualitativa das versões das linhas de produtos. Na primeira fase, cinco estudantes de pós-graduação foram responsáveis pela criação de oito versões da linha de produtos MobileMedia e cinco versões do BestLap. Essas versões foram desenvolvidas tanto em Java quanto em AspectJ. Na segunda fase, dois pesquisadores independentes efetuaram a validação das versões que foram geradas para as linhas de

produtos. O objetivo da terceira fase foi comparar a estabilidade dos projetos orientados a aspectos com os baseados em compilação condicional.

Nesse estudo, as implementações que utilizaram aspectos deram origem a projetos mais estáveis, particularmente quando o propósito das alterações introduzidas nos sistemas consistiam em *features* alternativas e opcionais. Entretanto, os mecanismos de orientação a aspectos não lidaram bem com a introdução de *features* mandatórias, tais como tratamento de exceção e classificação de mídias na linha de produtos MobileMedia. Para esses casos, os componentes adicionados na versão Java tiveram que ser também adicionados na versão em aspectos. Porém nessa última, aspectos adicionais tiveram que ser criados para tratamento das exceções incluídas. Alterações de *features* mandatórias para alternativas também não foram bem suportadas por aspectos. Nesses casos, houve a necessidade de adicionar e alterar muitos componentes e operações devido ao fato dos aspectos contarem com pontos de junção (*join points*) providos pelo código base.

Parrot VM: Adams et al. realizaram um estudo sobre a viabilidade de se refatorar software que utiliza compilação condicional para implementação de variabilidades, de forma que essas variabilidades passem a ser implementadas por meio de aspectos (ADAMS et al., 2009). Esse tipo de refatoração é proposto por muitos pesquisadores devido ao fato de aspectos modularizarem fisicamente o código das variabilidades. Como ainda não está claro se orientação a aspectos pode implementar os padrões de uso de compilação condicional, isso tornou-se a motivação principal para o desenvolvimento desse trabalho.

São apresentados nesse trabalho alguns padrões de compilação condicional com a finalidade de se avaliar a possibilidade de refatoração desses padrões para aspectos. O estudo desses padrões também forneceu requisitos para a criação de um modelo de interação sintática entre o código envolvido na compilação condicional e o código base do sistema. Para prospectar os padrões de compilação condicional analisados em sistemas existentes, foi criada uma ferramenta denominada **R3V3RS3**.

O software utilizado como estudo de caso neste trabalho foi o Parrot VM. Parrot VM é uma máquina virtual de código aberto que utiliza técnicas de pré-processamento em sua implementação. O uso de pré-processadores na implementação desse sistema é baseado nas melhores práticas dessa abordagem. Devido a esse fato, os autores consideram esse sistema um bom caso para uma refatoração para aspectos. Foram avaliadas várias versões do Parrot VM, o que permitiu obter informações sobre a importância de determinados padrões de compilação condicional. A estabilidade dos padrões ao longo do

tempo destacou quais padrões permaneceram importantes e quais foram usados apenas temporariamente. Os padrões que se mantiveram em várias versões no decorrer do tempo foram caracterizadas como importantes para serem refatorados para aspectos.

Os padrões de compilação condicional estudados foram divididos em dois grandes grupos, nos moldes da classificação proposta por Kästner, Apel e Kuhlemann (KÄSTNER; APEL; KUHLEMANN, 2008): granularidade grossa e granularidade fina. No estudo temporal efetuado no Parrot VM, foi mostrado que um grande número de condicionais de granularidade fina era usado, o que não é bom para refatorações para aspectos devido a dificuldade de se tratar esse tipo de granularidade com essa tecnologia. Uma vez que o Parrot VM foi considerado um estudo de caso interessante e diante dos resultados obtidos na pesquisa, os autores afirmam que compilação condicional é ainda a técnica mais indicada para implementação de variabilidades com características transversais em sistemas desenvolvidos em C/C++.

2.3.3 *Outras Experiências*

Acredita-se que o domínio de dispositivos móveis possuem diversas peculiaridades que podem ser melhor exploradas por meio de abordagens de desenvolvimento baseadas em linhas de produtos (MARINHO et al., 2010; FURTADO; SANTOS; RAMALHO, 2010). Nessa seção, são apresentadas duas linhas de produtos propostas recentemente para esse domínio.

MobiLine: Marinho et al. propuseram o uso de princípios de linhas de produtos para a criação de sistemas sensíveis ao contexto embarcados em dispositivos móveis (MARINHO et al., 2010). Segundo os autores, nesse tipo de software, o reúso não é sistematicamente utilizado e, com a criação de uma linha de produtos, eles querem mostrar que é possível reutilizar uma base de software comum para criação de diversos produtos nesse domínio. A principal contribuição desse trabalho é a criação de uma linha de produtos para sistemas sensíveis ao contexto embarcado em dispositivos móveis, juntamente com a descrição do processo de criação. Essa linha de produtos foi chamada de MobiLine.

A criação da linha de produtos foi dividida em três ciclos: (1) identificação das semelhanças e variações entre diversos software sensíveis ao contexto para dispositivos móveis. Esse ciclo foi responsável pela geração da base da linha de produtos; (2) identificação das *features* de um sub-domínio específico; e (3) configuração de um produto da LPS proposta. Alguns problemas identificados durante o processo, tais como problemas de modelagem e problemas com as ferramentas existentes, são discutidos juntamente com

possíveis soluções, de modo a guiar outros pesquisadores interessados nesse assunto.

ArcadEX Game SPL: Furtado et al. apresentaram um conjunto de diretrizes com a finalidade de simplificar as tarefas de análise de domínio durante a criação de linhas de produtos de software voltadas para jogos digitais (FURTADO; SANTOS; RAMALHO, 2010). O domínio dos jogos digitais apresenta diversas peculiaridades e esse estudo enfrenta esses desafios por meio de uma análise global do domínio, a fim de identificar características que possam servir como referência para a criação de um guia para essa atividade.

A fim de validar as diretrizes propostas, apresentou-se um estudo de caso no qual uma linha de produtos denominada ArcadEX Game SPL é concebida. Nesta LPS, cerca de trinta jogos digitais foram analisados segundo o guia proposto, tendo em vista incluí-los como candidatos a serem utilizados na implementação dessa linha de produtos. No entanto, não apresenta-se uma implementação da linha de produtos, sendo mostrado apenas o diagrama de *features* criado após a fase de análise do domínio. Em resumo, a principal contribuição desse trabalho é a apresentação de técnicas para enriquecer as tarefas de análise de domínio quando da criação de linhas de produtos de software voltadas para jogos digitais.

2.3.4 *Comentários Finais*

Grande parte das linhas de produtos de domínio livre se baseiam em sistemas de demonstração, construídos em laboratório. Por exemplo, a *MobileMedia Product Line* (MMPL) completa possui apenas 3 KLOC (FIGUEIREDO et al., 2008). Além dela, existem outras linhas de produtos menores tais como a *Expression Product Line* (EPL) (LOPEZ-HERREJON; BATORY; COOK, 2005) e a *Graph Product Line* (GPL) (LOPEZ-HERREJON; BATORY, 2001), ambas com 2 KLOC. A EPL consiste em uma gramática de expressões cujas variabilidades incluem tipos de dados (literais), operadores (negação, adição etc) e operações (impressão e avaliação). A GPL é uma linha de produtos na área de grafos, cujas variabilidades incluem características das arestas (direcionadas ou não direcionadas, com peso ou sem peso etc), métodos de busca (em profundidade ou em largura) e algoritmos clássicos de grafos (verificação de *loops*, caminho mais curto, árvore geradora mínima etc). O Prevayler, utilizado por Godil e Jacobsen, também é um sistema relativamente pequeno e possui aproximadamente 3 KLOC (GODIL; JACOBSEN, 2005).

Por outro lado, no trabalho apresentado por Kästner, Apel e Batory não foi possível efetuar a refatoração das *features* propostas inicialmente, tendo sido refatorados apenas

aproximadamente 8 KLOC do Oracle Berkeley DB (KÄSTNER; APEL; BATORY, 2007). Em Liebig et al. os sistemas estudados não foram refatorados e as variabilidades consideradas incluem basicamente *features* de nível muito baixo, normalmente selecionadas por meio de parâmetros de linhas de comando (por exemplo, opções de depuração, otimização ou portabilidade, no caso dos compiladores analisados no trabalho) (LIEBIG et al., 2010).

Adams et al. apresentam suas conclusões baseados em análises efetuadas em apenas um sistema (Parrot VM) (ADAMS et al., 2009). Em Marinho et al., propõe-se que os produtos de uma LPS sejam reconfigurados e adaptados em tempo de execução, o que foge ao que é tipicamente proposto na literatura para composição de produtos (MARINHO et al., 2010). Convencionalmente, produtos de uma LPS são compostos em tempo de compilação ou de implantação. Por fim, a LPS proposta por Furtado et al. não foi implementada, sendo apresentado apenas um modelo conceitual (FURTADO; SANTOS; RAMALHO, 2010).

A Tabela 1 apresenta um resumo sobre os principais pontos dos trabalhos apresentados.

Tabela 1: Comentários sobre os trabalhos apresentados

Linhas de Produtos	Comentários
40 sistemas (Liebig et al.)	Linhas de produtos com apenas <i>features</i> de baixo nível. Não houve extração de uma LPS.
Prevayler (Godil e Jacobsen)	Sistema relativamente pequeno (3 KLOC)
Oracle Berkeley DB (Kästner, Apel e Kuhlemann)	Extração não foi concluída devido a limitações de AspectJ. <i>Features</i> correspondem a aproximadamente 10% do código.
MMPL / EPL / GPL	Sistemas de demonstração, sintetizados em laboratório, com tamanho reduzido (2-3 KLOC)
Parrot VM (Adams et al.)	Conclusões baseadas em análises de apenas um sistema (Parrot VM)
MobiLine (Marinho et al.)	Composição de produtos de modo não convencional
ArcadEX Game SPL (Furtado et al.)	Linha de produtos proposta não foi implementada

A fim de disponibilizar uma linha de produtos real, criada a partir de um software relevante, maduro e de relativa complexidade, no próximo capítulo é apresentada a linha de produtos ArgoUML-SPL.

3 ARGOUML-SPL

3.1 Introdução

O ArgoUML¹ é uma ferramenta de código fonte aberto desenvolvida em Java que possui aproximadamente 120 KLOC e que tem por finalidade a modelagem de sistemas em UML². O sistema permite a criação dos seguintes diagramas: diagrama de classes, diagrama de estados, diagrama de atividades, diagrama de casos de uso, diagrama de colaboração, diagrama de implantação e diagrama de sequência. Nesta dissertação de mestrado, foi avaliada a versão 0.28.1 do sistema. Essa versão é compatível com o UML 1.4 e oferece suporte a *profiles*, sendo distribuído com *profiles* para Java e UML 1.4. Um *profile* é um mecanismo de extensão para customização de modelos UML para domínios e plataformas específicas (FOWLER; SCOTT, 2000). A Figura 7 apresenta a tela principal da ferramenta ArgoUML. Nessa figura é exibida a área de edição dos digramas (acima, à direita), a seção que contém os componentes do diagrama (acima, à esquerda) e uma área destinada a mensagens de itens a fazer (abaixo).

A integração entre o ArgoUML e outras ferramentas UML é garantida pelo suporte nativo a XMI³, um formato de arquivos baseado em XML⁴ largamente usado para troca de dados entre ferramentas dessa natureza. É também possível exportar os diagramas criados para formatos de imagens, tais como GIF, PNG, PostScript, EPS, PGML e SVG. Essa funcionalidade facilita a distribuição de diagramas entre as equipes de desenvolvimento de um projeto de software.

O sistema possui suporte a internacionalização, sendo que na versão avaliada estão disponíveis dez idiomas diferentes. Há ainda suporte a linguagem para especificação formal de restrições (OCL), geração de código a partir dos diagramas de classes para as linguagens Java, C++, C#, PHP4 e PHP5, havendo ainda a possibilidade de se adicionar outras

¹Disponível em <http://argouml.tigris.org/>

²Unified Modeling Language.

³XML Metadata Interchange.

⁴eXtensible Markup Language.

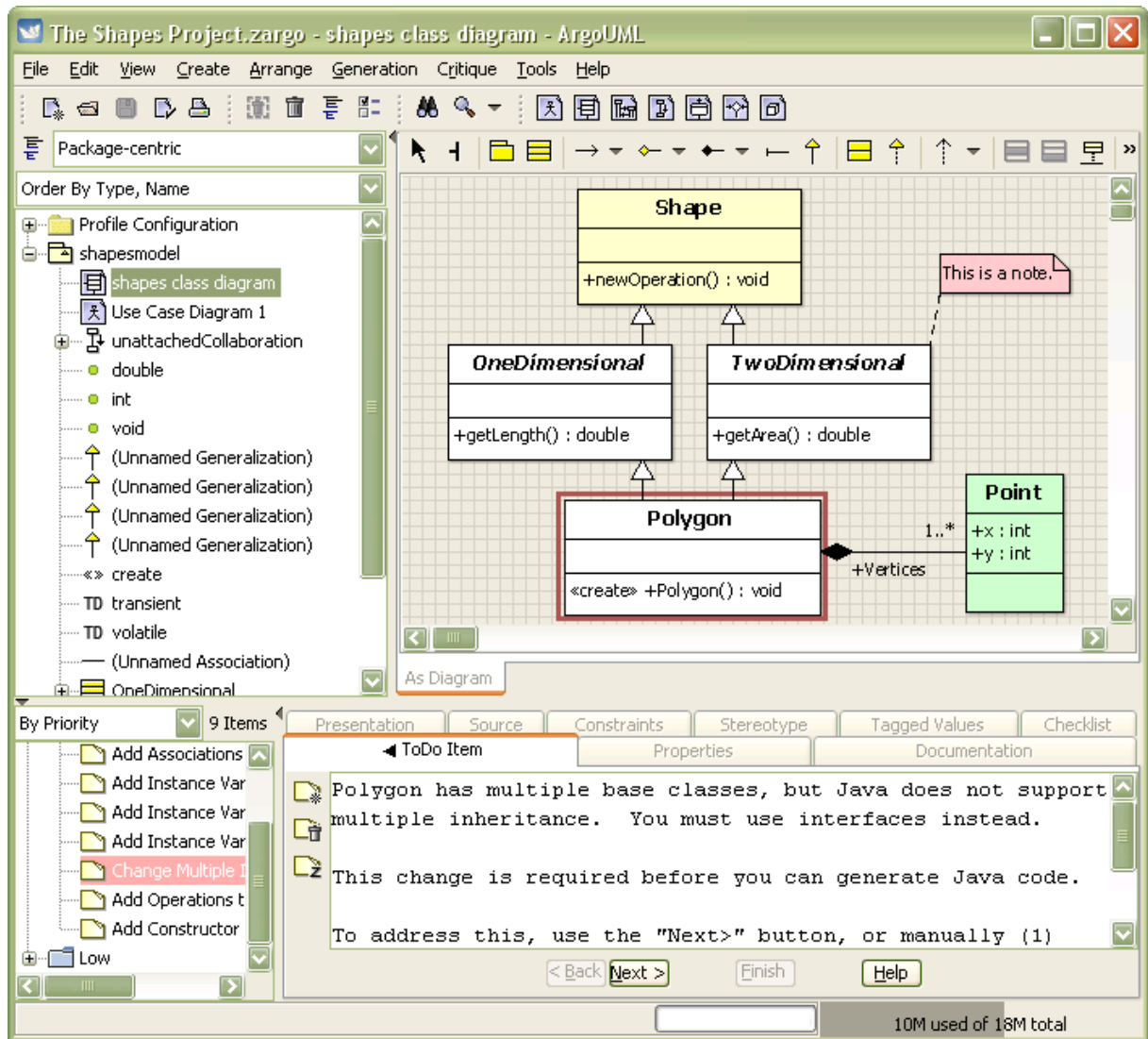


Figura 7: Tela principal do ArgoUML

linguagens através da instalação de *plugins*. O ArgoUML provê suporte para que seja feita engenharia reversa em código fonte Java, ou seja, dado um conjunto de classes de um projeto, ele automaticamente cria os artefatos UML correspondentes. O sistema disponibiliza ainda diversas funcionalidades de apoio aos projetistas tais como lista de itens a fazer (*ToDo Lists*), *checklists* para verificação da evolução do desenvolvimento de um projeto e exibição de múltiplas visões e perspectivas dos diagramas. Um dos diferenciais do ArgoUML é o conceito de *Suporte Cognitivo*, que será explorado na Seção 3.3.

A extração de uma LPS a partir de uma versão monolítica da ferramenta ArgoUML teve como objetivo principal disponibilizar à comunidade de pesquisadores da área uma linha de produtos real, criada a partir de um software relevante, maduro e de relativa complexidade. A partir dessa ideia foi criada a linha de produtos ArgoUML-SPL, dispo-

nível publicamente a qualquer pesquisador ou desenvolvedor de software interessado em estudá-la ou utilizá-la, no sítio <http://argouml-spl.tigris.org>.

O restante deste capítulo está organizado conforme descrito a seguir. Na Seção 3.2, apresenta-se uma visão geral sobre a arquitetura do ArgoUML. A Seção 3.3 apresenta as *features* extraídas nesta dissertação e por fim a Seção 3.4 descreve os procedimentos utilizados para a extração dessas *features* e para a coleta de informações para as análises efetuadas no Capítulo 4.

3.2 Arquitetura do ArgoUML

O ArgoUML possui uma arquitetura organizada em subsistemas. Cada subsistema possui suas próprias responsabilidades, não interferindo diretamente no funcionamento dos demais. Cada subsistema corresponde a um pacote Java. Além disso, subsistemas podem oferecer uma classe de fachada (Padrão *Facade* (GAMMA et al., 1994)) para facilitar a sua interação com outros subsistemas. Há ainda a possibilidade de utilização de interfaces para prover conexões com *plugins*. O uso de *plugins* permite adicionar recursos à ferramenta, tais como geração de código e engenharia reversa para outras linguagens.

Um *plugin* (ou módulo) é uma coleção de classes que podem ser habilitadas e desabilitadas no ArgoUML. Tais módulos são carregados automaticamente na inicialização do sistema. A interação desses módulos com o núcleo do sistema é feita através da implementação de interfaces providas por uma API específica do sistema. Isto é, os desenvolvedores que desejarem criar tais módulos deverão criar classes que implementem essas interfaces, e o seu código fonte será chamado nos pontos onde tais interfaces são utilizadas no sistema. Esse tipo de interação é o mesmo utilizado em *frameworks* modernos (JOHNSON, 1997). Módulos que representam *plugins* podem ser internos ou externos, sendo que a única diferença entre eles é que os internos são integrados ao sistema, fazendo parte de seu pacote principal (`argouml.jar`) e os externos são pacotes separados, inclusive podendo ser criados e distribuídos por terceiros. Para que um módulo seja conectado ao ArgoUML, ele necessita implementar a interface `ModuleInterface`. Essa interface possui métodos para habilitação, desabilitação e identificação do módulo. Ela também é responsável por registrar as classes do ArgoUML que serão afetadas por esse módulo.

A Figura 8 apresenta os principais componentes de um subsistema genérico e descreve como ele interage com outros subsistemas (TOLKE; KLINK; WULP, 2010). Nessa figura, a classe de fachada é representada pela classe `ComponentFacade`. A comunicação

com outro subsistema é realizada através da ligação entre essa classe e a classe `ClassFromOtherComponent`. A conexão com *plugins* é demonstrada através das classes `ComponentPlugin1` e `ComponentPlugin2`, que fornecem um ponto de acesso às classes `PluggableClass1` e `PluggableClass2`, que representam *plugins*. As interfaces `ComponentInterface1` e `ComponentInterface2` representam as interfaces que devem ser implementadas pelos subsistemas ou *plugins* que desejarem utilizar esse subsistema específico.

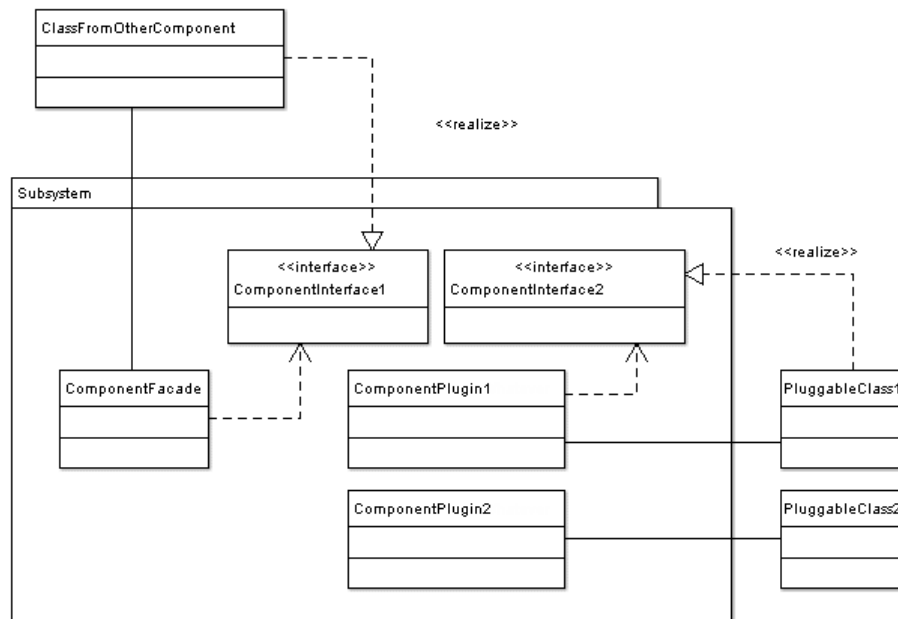


Figura 8: Estrutura interna de um subsistema no ArgoUML

A Figura 9 apresenta os principais subsistemas da arquitetura do ArgoUML, apresentando seus componentes e as suas dependências. Tais subsistemas são descritos a seguir:

- Externos: são bibliotecas de terceiros utilizadas no ArgoUML e que não são mantidas pelo projeto, tais como bibliotecas para edição gráfica (GEF) e para especificação de restrições em OCL. Os outros subsistemas podem ter dependências em relação a tais subsistemas. A Figura 10 apresenta os subsistemas externos;
- Baixo Nível: são subsistemas de infraestrutura que só existem para dar suporte a outros subsistemas. São responsáveis por tarefas como configuração, internacionalização, gerenciamento de tarefas, gerenciamento de modelos e para implementação de interfaces gráficas. O subsistema de modelo tem como propósito abstrair dos outros subsistemas qual o repositório de modelos (MDR, EMF, NSUML) está em uso, além de prover uma interface consistente para manipulação de dados nesses re-

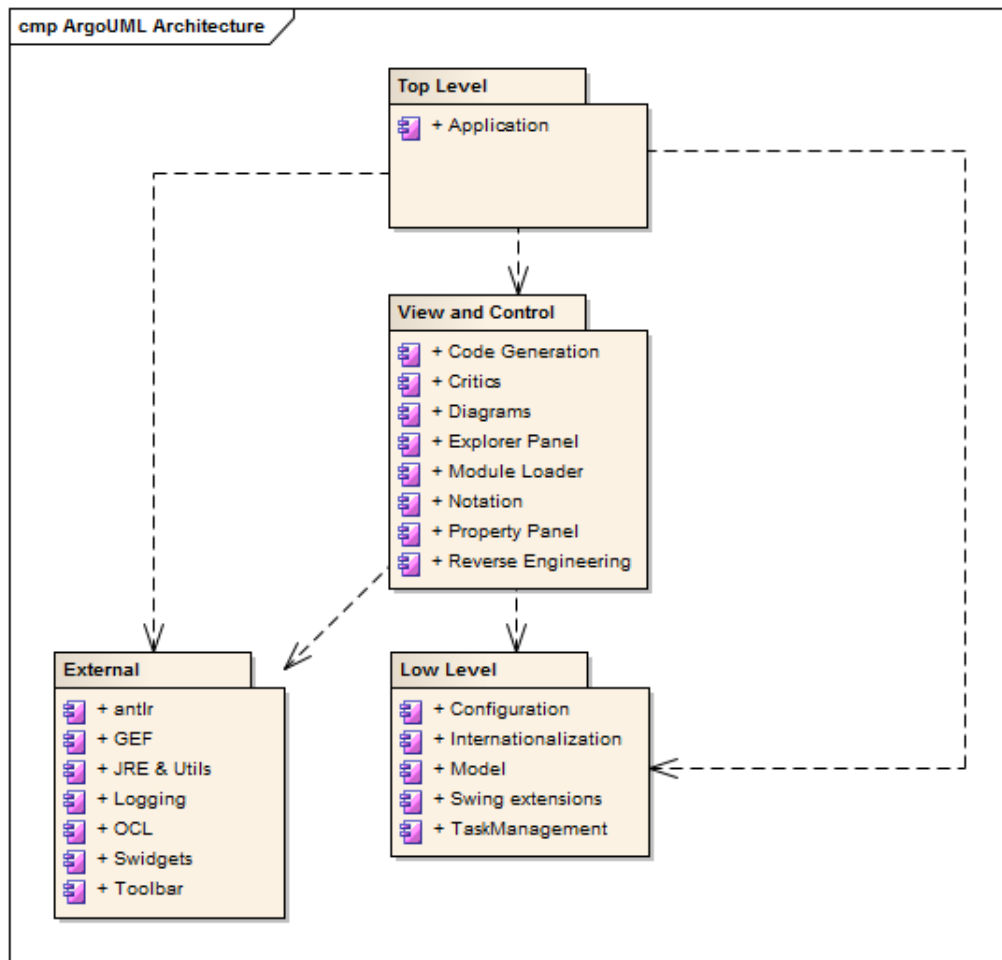


Figura 9: Principais subsistemas da arquitetura do ArgoUML

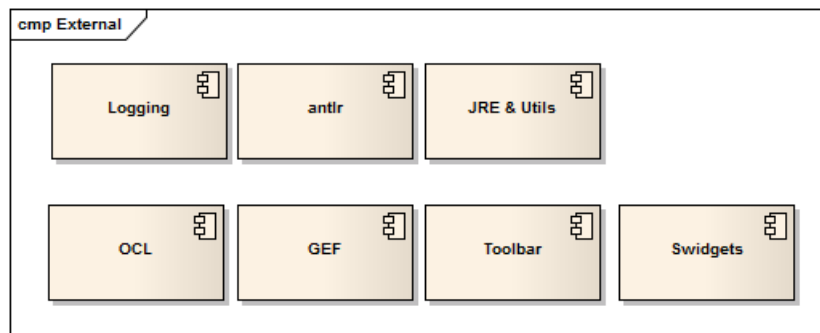


Figura 10: ArgoUML: Subsistemas Externos

positórios. Esse subsistema segue o padrão MVC (*Model-View-Control*) (KRASNER; POPE, 1988). A Figura 11 apresenta os subsistemas de baixo nível;

- **Controle e Visão:** são subsistemas responsáveis pela criação e manipulação dos diagramas, geração de código, engenharia reversa, gerenciamento da interface gráfica, dentre outros. Dependem diretamente do subsistema de modelo (que pertence ao

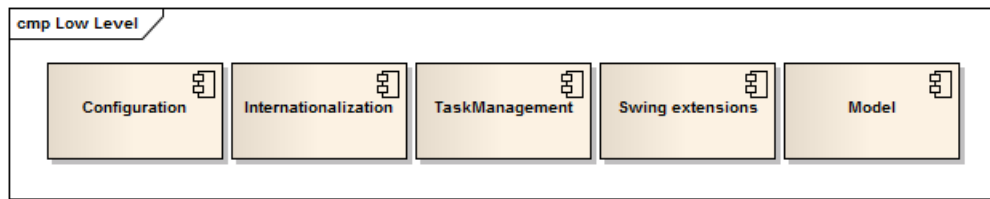


Figura 11: ArgoUML: Subsistemas de Baixo Nível

grupo de subsistemas de baixo nível) para manipular corretamente os diagramas gerados. A Figura 12 apresenta os subsistemas de controle e visão e suas dependências;

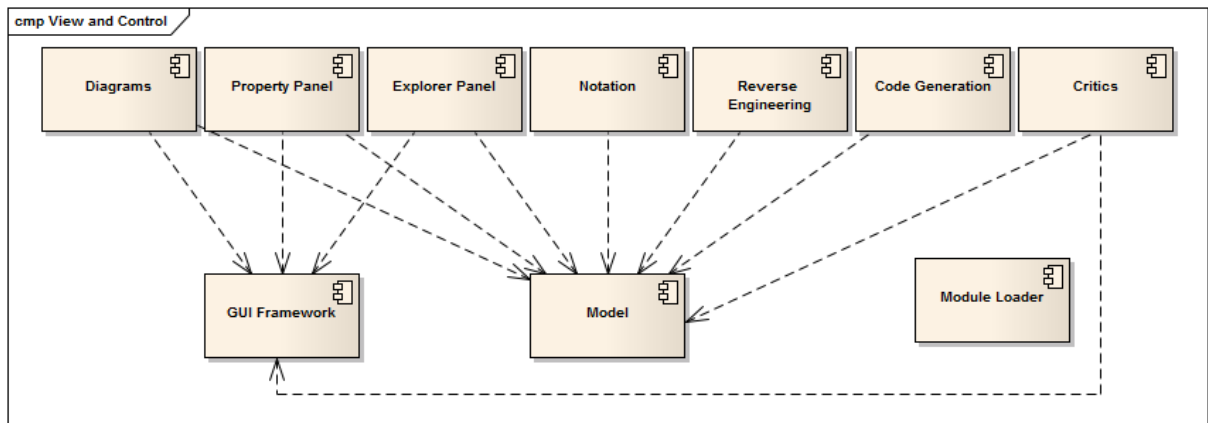


Figura 12: ArgoUML: Subsistemas de Controle e Visão

- Alto Nível: é o ponto de partida do ArgoUML, sendo responsável por inicializar os demais subsistemas, ou seja, contém o método `main()` da aplicação. Ele depende dos outros subsistemas do ArgoUML, porém outros subsistemas não possuem dependências em relação a ele.

3.3 Features

Para criação do ArgoUML-SPL, foram selecionadas oito *features*, representando requisitos funcionais e não funcionais do sistema. A *feature* LOGGING é um representante dos requisitos não funcionais do sistema. Essa *feature* tem por finalidade registrar eventos que representam exceções e informações para *debug*, sendo que a sua escolha deveu-se à sua natureza espalhada e entrelaçada pelo código base do sistema. As outras sete *features* escolhidas representam requisitos funcionais, são elas: SUPORTE COGNITIVO, DIAGRAMA DE ESTADOS, DIAGRAMA DE ATIVIDADES, DIAGRAMA DE SEQUÊNCIA, DIAGRAMA DE COLABORAÇÃO, DIAGRAMA DE CASOS DE USO e DIAGRAMA DE IMPLANTAÇÃO.

A *feature* SUPORTE COGNITIVO tem por finalidade prover informações que auxiliem os projetistas a detectar e resolver problemas em seus projetos (ROBBINS; REDMILES, 1999). Essa *feature* é implementada por agentes de software que ficam em execução contínua em segundo plano efetuando análises sobre os diagramas criados, com a finalidade de detectar eventuais problemas nos mesmos. Esses agentes podem recomendar boas práticas a serem seguidas na construção do modelo, fornecer lembretes sobre partes do projeto que ainda não foram finalizadas ou sobre a presença de erros de sintaxe, dentre outras tarefas.

As *features* DIAGRAMA DE ESTADOS, DIAGRAMA DE ATIVIDADES, DIAGRAMA DE SEQUÊNCIA, DIAGRAMA DE COLABORAÇÃO, DIAGRAMA DE CASOS DE USO e DIAGRAMA DE IMPLANTAÇÃO proveem suporte para a construção dos respectivos diagramas UML.

Diagramas de estados descrevem o ciclo de vida de um objeto em termos dos eventos que mudam o seu estado. Um diagrama de atividades é um tipo especial de diagrama de estado em que são representados os estados de uma atividade, em vez de estados de um objeto. Ao contrário dos diagramas de estado, que são orientados a eventos, diagramas de atividades são orientados a fluxos de controle (BEZERRA, 2001; PENDER, 2003; GROUP, 2010).

Um diagrama de sequência tem o objetivo de mostrar como mensagens são trocadas entre os objetos no decorrer do tempo. Um diagrama de colaboração mostra as interações entre objetos. A diferença entre diagramas de sequência e diagramas de colaboração está na ênfase dada às interações entre os objetos. Nos diagramas de sequência, a ênfase está na ordem temporal das mensagens trocadas entre os objetos. Os diagramas de colaboração enfatizam os relacionamentos existentes entre os objetos que participam da realização de um cenário (BEZERRA, 2001; PENDER, 2003; GROUP, 2010).

Um diagrama de casos de uso é uma representação de funcionalidades externamente observáveis e de elementos externos ao sistema que interagem com ele (BEZERRA, 2001). Esse diagrama descreve os usuários relevantes do sistema, quais são os serviços de que eles necessitam e quais são os serviços que eles devem prover ao sistema (PENDER, 2003).

Um diagrama de implantação descreve uma arquitetura de execução, cuja configuração de hardware e software que define como o sistema será configurado e como ele deverá operar. O propósito do diagrama de implantação é apresentar uma visão estática do ambiente de instalação do sistema (PENDER, 2003; GROUP, 2010).

As *features* DIAGRAMA DE ESTADOS, DIAGRAMA DE ATIVIDADES, DIAGRAMA

DE SEQUÊNCIA e DIAGRAMA DE COLABORAÇÃO, além de representarem funcionalidades relevantes do sistema, foram selecionadas pelo fato de disponibilizarem diagramas semelhantes em termos de utilização e finalidade, de acordo com a especificação do UML(GROUP, 2010). Este fato ficou comprovado também em nível de implementação, o que pôde ser demonstrado por meio da utilização de métricas de entrelaçamento, que são exploradas na Seção4.1.2. Com a utilização dessas métricas ficou evidenciado que essas *features* possuem código compartilhado e, e com isso forneceram interessantes cenários para estudo na refatoração efetuada no ArgoUML.

O modelo de *features* da linha de produtos ArgoUML-SPL é mostrado na Figura 13. Nesse modelo, define-se que o diagrama de classes é obrigatório. Isso é demonstrado no modelo através da representação de um círculo preenchido. Essa situação decorre do fato desse diagrama estar situado no centro do processo de modelagem de objetos, sendo considerado o diagrama principal para captura das regras que regem a definição e uso de objetos de um sistema. Por ser um repositório das regras de um sistema, ele é também a principal fonte para a transformação do modelo em código e vice-versa (PENDER, 2003). Os demais diagramas (incluindo diagrama de estados, diagrama de atividades, diagrama de sequência, diagrama de colaboração, diagrama de casos de uso e diagrama de implantação), além do suporte cognitivo e *logging*, são consideradas *features* opcionais.

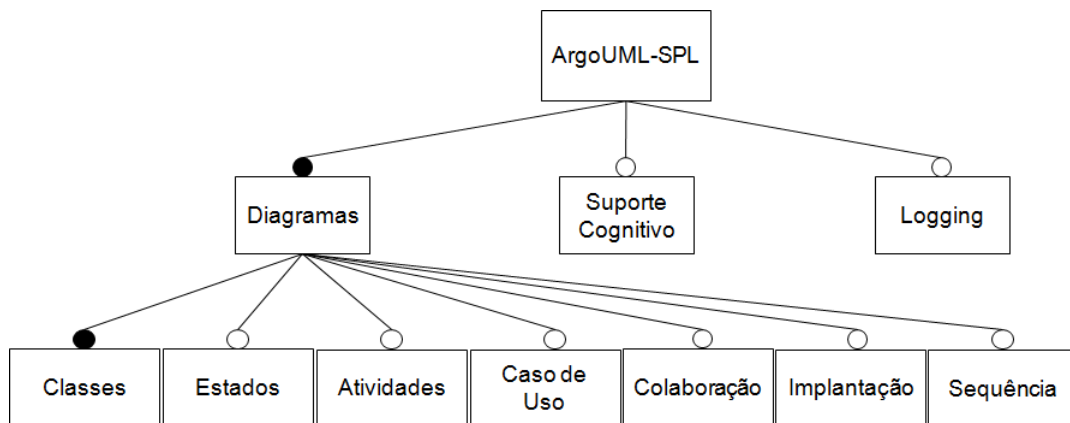


Figura 13: Modelo de *features* do ArgoUML-SPL

Dois critérios principais guiaram a seleção das *features* mencionadas: relevância e complexidade de implementação. Em respeito a relevância, decidiu-se extrair apenas *features* que representam requisitos funcionais típicos do domínio de ferramentas de modelagem (tais como os seis diagramas UML), uma *feature* que representa um requisito não-funcional (LOGGING) e uma *feature* que apresenta um inquestionável comportamento opcional (SUPORTE COGNITIVO).

Em relação ao critério complexidade, a implementação de cada *feature* requer considerável quantidade de código, geralmente não confinado em uma única classe ou pacote. Por exemplo, a implementação de LOGGING e SUPORTE COGNITIVO possui comportamento transversal, impactando várias classes do sistema. Finalmente, duas situações são particularmente comuns na implementação das *features* que representam os diagramas UML: entrelaçamento (quando um bloco de código, geralmente um comando ou expressão, inclui código associado a mais de uma *feature*) e aninhamento (quando um bloco de código associado a uma dada *feature* está lexicalmente aninhado em um bloco de código externo, associado a outra *feature*).

Deve-se mencionar que o ArgoUML-SPL inclui apenas *features* opcionais e mandatórias (isto é, ele não provê suporte para *features* alternativas, por exemplo). Além disto, não há dependências semânticas entre as *features* extraídas. Em outras palavras, qualquer configuração – com quaisquer *features* habilitadas ou desabilitadas – é válida. No entanto, essas limitações não representam uma ameaça para a validade do uso do ArgoUML-SPL na avaliação de técnicas para implementação de linhas de produtos de software. A razão para isso é que eventuais dependências entre duas *features* A e B – incluindo dependências dos tipos `or`, `xor`, ou `requires` – não impactam na anotação do código de A e B (isto é, o código de ambas as *features* deve ser anotado de qualquer maneira). Na realidade, as abordagens baseadas em compilação condicional consideram dependências entre A e B apenas em tempo de derivação de produtos, para prevenir a geração de produtos inválidos.

3.4 Processo de Extração

A tecnologia empregada na extração da linha de produto de software que originou o projeto ArgoUML-SPL foi a de pré-processadores. Conforme discutido na Seção 2.2.1.1, essa técnica apresenta prós e contras que foram levados em consideração para sua escolha. Assim, ao utilizar pré-processadores, pretendeu-se disponibilizar uma linha de produtos que servisse como *benchmark* para outras tecnologias de modularização, como orientação a aspectos. Em outras palavras, pretende-se com o ArgoUML-SPL disponibilizar um sistema-desafio, que contribua para avaliar o real poder de expressão de técnicas modernas de modularização e separação de interesses.

A linguagem Java não possui suporte nativo para diretivas de pré-processamento. Devido a isso, nesse trabalho, foi utilizada a ferramenta `javapp`, apresentada na Seção 2.2.1.1. Para automatizar a geração dos produtos da LPS criada, foi implementado um conjunto de scripts `ant`. Basicamente, esses scripts permitem que os desenvolvedores

informem quais *features* devem ser incluídas em um determinado produto. Feito isso, os scripts se encarregam da execução do pré-processador `javapp` e da compilação do código gerado, a fim de gerar o produto desejado.

3.4.1 Anotação do Código

A identificação dos elementos de código do ArgoUML pertencentes a uma determinada *feature* foi feita manualmente. O primeiro passo para essa extração consistiu em um estudo detalhado da arquitetura do sistema, principalmente com o auxílio de seu *cookbook* (TOLKE; KLINK; WULP, 2010). Após entendimento básico da arquitetura do ArgoUML, o seu código fonte foi estudado a fim de identificar os pontos em que cada *feature* era demandada. O ponto de partida para a extração de cada *feature* foi a identificação dos principais pacotes responsáveis pela sua implementação. Após essa identificação, as classes desses pacotes eram avaliadas, buscando-se as referências para elas em outras partes do sistema. Essas referências e suas dependências eram anotadas como pertencentes à *feature* que estava sendo extraída. A Figura 14 ilustra esse processo.

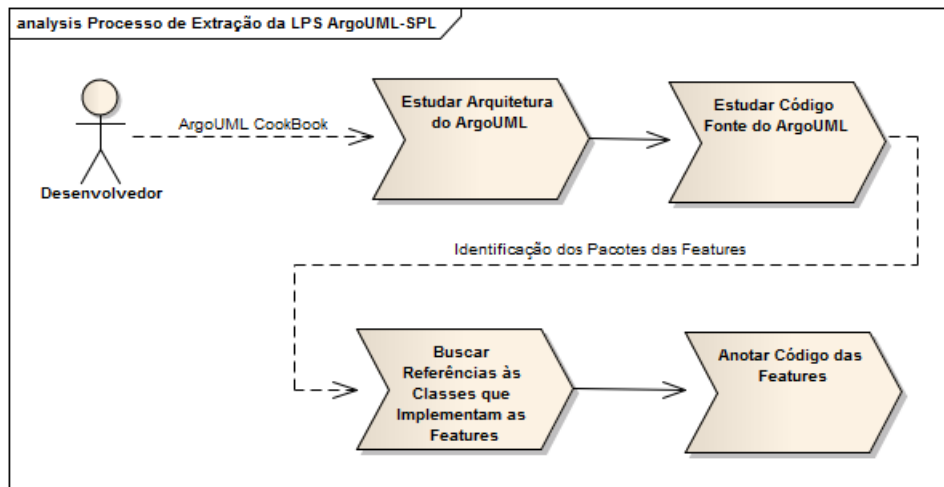


Figura 14: Processo de Extração do ArgoUML-SPL

Nesta tarefa de exploração do código fonte foi utilizado o ambiente de desenvolvimento Eclipse. Os recursos de busca textual e busca por referências dessa IDE foram extensivamente usados para auxiliar na localização dos trechos de código relativos a cada uma das *features*. Nesse processo, uma vez identificado que um determinado trecho de código *X* apenas deveria ser compilado caso uma determinada *feature* *F* fosse selecionada, procedia-se à sua delimitação por meio de `#ifdef` e `#endif`. As constantes definidas para anotação das *features* da linha de produtos ArgoUML-SPL são apresentadas na Tabela 2.

Tabela 2: Constantes utilizadas para anotação de *features* no ArgoUML-SPL

<i>Feature</i>	Constante
SUPORTE COGNITIVO	COGNITIVE
DIAGRAMA DE ATIVIDADES	ACTIVITYDIAGRAM
DIAGRAMA DE ESTADOS	STATEDIAGRAM
DIAGRAMA DE COLABORAÇÃO	COLLABORATIONDIAGRAM
DIAGRAMA DE SEQUÊNCIA	SEQUENCEDIAGRAM
DIAGRAMA DE CASOS DE USO	USECASEDIAGRAM
DIAGRAMA DE IMPLANTAÇÃO	DEPLOYMENTDIAGRAM
LOGGING	LOGGING

O código responsável pela implementação dos diagramas UML encontra-se encapsulado em pacotes específicos, como por exemplo a *feature* DIAGRAMA DE ESTADOS, cuja implementação concentra-se no pacote `org.argouml.uml.diagram.state`. No entanto, classes desse pacote são usadas ao longo de diversas partes do sistema. Por exemplo, a classe `ExplorerPopup`, que faz parte do pacote `org.argouml.ui.explorer` e é responsável pela criação de *menus pop-up*, referencia a classe `UMLStateDiagram`, pertencente ao pacote `org.argouml.uml.diagram.state.ui`.

Como um segundo exemplo, a implementação da *feature* SUPORTE COGNITIVO pode ser encontrada nos seguintes pacotes: `org.argouml.cognitive.critics`, `org.argouml.uml.cognitive.critics` e `org.argouml.pattern.cognitive.critics`. As classes desses pacotes são referenciadas por diversas classes de outros pacotes, como por exemplo pela classe `Profile`, responsável por representar de forma abstrata um Profile UML e pertence ao pacote `org.argouml.profile`. Portanto, a extração das *features* mencionadas é uma tarefa complexa porque não se limita a anotações de pacotes específicos. Para correta extração é necessária a identificação dos pontos do sistema que fazem referência às classes que implementam as *features*.

O ArgoUML possui diversas classes responsáveis por testes unitários. Essas classes, criadas com apoio do *framework* JUnit, não foram incluídas no ArgoUML-SPL.

Validação dos Produtos da LPS: Inicialmente, a validação de diversos produtos gerados pela LPS extraída foi feita por meio de testes funcionais, do tipo caixa preta. Esses testes visaram principalmente a validação do produto gerado de acordo com a seleção de *features* efetuada. Em tais testes, foi avaliada a estabilidade do sistema e comparado o funcionamento do ArgoUML original com o produto gerado pela LPS. Por exemplo, suponha um produto com todas as *features* habilitadas, exceto DIAGRAMA DE ESTADOS.

Durante os testes, esse produto foi gerado e então executado para avaliar o seu correto funcionamento (isto é, se no produto gerado a opção de criação de diagrama de estado tinha sido de fato removida e todos os outros diagramas estavam funcionando corretamente). Esse procedimento foi repetido para todos as *features* opcionais da LPS extraída.

A segunda parte da validação consistiu em uma inspeção manual do código fonte do sistema, para verificar se os elementos sintáticos responsáveis por cada *feature* haviam sido de fato anotados. Também foi validado o código fonte dos produtos gerados. Ou seja, após efetuar o pré-processamento do código fonte do ArgoUML-SPL para geração de cada produto, o código resultante foi analisado a fim de certificar-se de que não havia erros de compilação.

Ao final do processo de extração, a linha de produtos extraída foi enviada para os projetistas do sistema ArgoUML, que após avaliação aprovaram a sua hospedagem no *website* oficial da comunidade. O principal critério para aprovação do ArgoUML-SPL como um dos sub-projetos do ArgoUML foi o seu potencial para contribuir para evolução e modernização da ferramenta. Espera-se que essa hospedagem pública no repositório de versões do ArgoUML possa atrair desenvolvedores interessados em ampliar o número de *features* atualmente disponibilizadas pela LPS. O ArgoUML-SPL encontra-se publicamente disponível no sítio <http://argouml-spl.tigris.org/>.

3.4.2 Coleta de Métricas

Os trechos de código anotados para criação do ArgoUML-SPL foram identificados segundo a sua localização e o tipo da granularidade da anotação. Isso possibilitou a coleta de métricas, o que permitiu estudar a natureza do código anotado, possibilitando caracterizar as anotações efetuadas e analisar tecnologias alternativas para realização dessa tarefa. Uma descrição mais detalhada dessas métricas é realizada no Capítulo 4.

A Figura 15 apresenta um exemplo de como essa identificação foi feita. Nessa figura, é exibido um trecho de código pertencente à *feature* LOGGING. O método mostrado tem por finalidade retornar uma lista de arestas de entrada de um componente do diagrama. A linha 5 informa que o trecho anotado somente será incluído na compilação se a *feature* LOGGING tiver sido selecionada. A linha 6 informa que essa anotação possui o tipo de granularidade **Statement**. Por fim, a linha 7 informa que a localização dessa anotação é do tipo **BeforeReturn**. As métricas de granularidade e localização serão descritas em detalhes, respectivamente, nas Seções 4.1.3 e 4.1.4.

```

1 public List getInEdges(Object port) {
2     if (Model.getFacade().isAStateVertex(port)) {
3         return new ArrayList(Model.getFacade().getIncomings(port));
4     }
5     //#if defined(LOGGING)
6     //@#$LPS-LOGGING: GranularityType: Statement
7     //@#$LPS-LOGGING: Localization: BeforeReturn
8     LOG.debug("TODO: getInEdges of MState");
9     //#endif
10    return Collections.EMPTY_LIST;
11 }

```

Figura 15: Identificação do tipo das anotações

Para coleta dessas métricas, paralelamente à extração da linha de produtos ArgoUML-SPL, foi criado um aplicativo que tem por finalidade ler os arquivos fontes do sistema, buscando e contabilizando cada identificação de métrica encontrada. Esse aplicativo, denominado **SPLMetricsGather**, foi desenvolvido neste estudo. Ele utiliza a linguagem Java e encontra-se publicamente disponível no mesmo sítio em que o ArgoUML-SPL está disponibilizado. Além das métricas de localização e granularidade, a ferramenta **SPLMetricsGather** também é responsável por coletar métricas de tamanho e de transversalidade, descritas, respectivamente, nas Seções 4.1.1 e 4.1.2.

3.5 Comentários Finais

Este capítulo apresentou o sistema base – ArgoUML – utilizado na criação da linha de produtos ArgoUML-SPL, descrevendo a sua arquitetura interna e as principais interações entre seus componentes. Também foram apresentadas as *features* consideradas na linha de produtos, e o processo seguido durante a fase de extração dessas *features*. Descreveu-se também a metodologia utilizada na anotação dos trechos de código de cada *feature*. Basicamente, a anotação desses trechos teve dois propósitos: (1) delimitar os trechos de código pertencentes a cada *feature* para permitir a geração de diferentes produtos a partir da linha de produtos; (2) permitir a coleta de métricas para análise e entendimento do comportamento das *features* consideradas na linha de produtos.

A definição dessas métricas e o estudo e análise dos dados coletados são abordados no próximo capítulo.

4 AVALIAÇÃO

4.1 Caracterização da LPS Extraída

A fim de avaliar e caracterizar a linha de produtos extraída, foi utilizado – e adaptado – um conjunto de métricas recentemente proposto para avaliação de linhas de produtos de software baseadas em pré-processadores (LIEBIG et al., 2010). Basicamente, quatro conjunto de métricas foram coletados: métricas de tamanho, métricas de transversalidade, métricas de granularidade e métricas de localização. Estas métricas – e seus respectivos valores – são detalhados nas próximas subseções.

4.1.1 Métricas de Tamanho

Essas métricas se destinam a avaliar o tamanho dos produtos e das *features* da linha de produtos implementada com o uso de compilação condicional. Mais especificamente, foram coletadas as seguintes métricas de tamanho:

- Linhas de Código (LOC - *Lines Of Code*): conta o total de linhas de código – sem comentários e linhas em branco – para um dado produto P gerado a partir da LPS;
- Número de Pacotes (NOP - *Number Of Packages*): conta o número de pacotes presentes em um dado produto P ;
- Número de Classes (NOC - *Number Of Classes*): conta o número de classes presentes em um dado produto P ;
- Linhas de Features (LOF - *Lines Of Feature*): proposta por Liebig et al. (LIEBIG et al., 2010), essa métrica conta o total de linhas de código – sem comentários e linhas em branco – responsáveis pela implementação de uma dada *feature* F . Basicamente, $LOF(F) = LOC(All) - LOC(All - F)$, no qual All denota o produto com todas as *features* da LPS habilitadas; $All - F$ denota um produto com todas as *features* habilitadas, exceto F .

As Tabelas 3 e 4 apresentam os valores coletados para as métricas de tamanho de produtos e *features*, respectivamente. A Tabela 4 apresenta também o percentual de código dedicado à implementação de cada *feature* (em relação à versão original – não baseada em LPS – do ArgoUML).

Tabela 3: Métricas de Tamanho para Produtos

Produto	LOC	NOP	NOC
Original, não baseada em LPS	120.348	81	1.666
Apenas SUPORTE COGNITIVO desabilitada	104.029	73	1.451
Apenas DIAGRAMA DE ATIVIDADES desabilitada	118.066	79	1.648
Apenas DIAGRAMA DE ESTADOS desabilitada	116.431	81	1.631
Apenas DIAGRAMA DE COLABORAÇÃO desabilitada	118.769	79	1.647
Apenas DIAGRAMA DE SEQUÊNCIA desabilitada	114.969	77	1.608
Apenas DIAGRAMA DE CASOS DE USO desabilitada	117.636	78	1.625
Apenas DIAGRAMA DE IMPLANTAÇÃO desabilitada	117.201	79	1.633
Apenas LOGGING desabilitada	118.189	81	1.666
Todas as <i>features</i> desabilitadas	82.924	55	1.243

Tabela 4: Métricas de Tamanho para *Features*

<i>Feature</i>	LOF	
SUPORTE COGNITIVO	16.319	13,56%
DIAGRAMA DE ATIVIDADES	2.282	1,90%
DIAGRAMA DE ESTADOS	3.917	3,25%
DIAGRAMA DE COLABORAÇÃO	1.579	1,31%
DIAGRAMA DE SEQUÊNCIA	5.379	4,47%
DIAGRAMA DE CASOS DE USO	2.712	2,25%
DIAGRAMA DE IMPLANTAÇÃO	3.147	2,61%
LOGGING	2.159	1,79%
Todas as <i>features</i>	37.424	31,10%

Análise de Tamanho: Conforme mostrado na Tabela 3, a versão original do sistema – com todas as *features* habilitadas – possui 120.348 LOC. Por outro lado, o menor produto que pode ser derivado a partir da linha de produtos extraída – com todas as *features* desabilitadas – possui 82.924 LOC (portanto, 31,1% menor). Essa redução mostra os benefícios em termos de customização que podem ser alcançados quando se muda para uma abordagem de desenvolvimento baseada em linhas de produtos de software. Isto é, ao invés de uma abordagem monolítica (do tipo *one-size-fits-all*), a linha de produtos extraída permite aos seus usuários criar um sistema do tamanho de suas reais necessidades. A Tabela 3 mostra também que LOGGING é a única *feature* cuja remoção não afeta o número

de classes e pacotes do sistema. Essa constatação se deve ao fato de que o ArgoUML conta com uma biblioteca externa, chamada Log4J ¹, para geração de mensagens de *logging*.

De acordo com a Tabela 4, SUPORTE COGNITIVO é a *feature* com o maior LOF. De fato, 13,56% das linhas de código do sistema são dedicados à implementação dessa *feature*. O diagrama com menor LOF é o DIAGRAMA DE COLABORAÇÃO (1.579 LOC). Por outro lado, DIAGRAMA DE SEQUÊNCIA é o diagrama com o maior LOF (5.379 LOC). A Tabela 4 mostra também que o número de linhas de código dedicadas à *feature* LOGGING (2.159 LOC) é próximo ao número de linhas dedicadas às *features* que implementam alguns diagramas UML, tais como DIAGRAMA DE ATIVIDADES (2.282 LOC) e DIAGRAMA DE CASOS DE USO (2.712 LOC). Tais números sugerem a importância de considerar LOGGING como uma *feature* no ArgoUML-SPL.

As *features* extraídas neste experimento representam um total de 37.424 LOC (ou seja, 31,1% do total da versão original do ArgoUML). Em outras palavras, o núcleo da LPS extraída corresponde a aproximadamente 69% do tamanho do sistema, em termos de linhas de código. Basicamente, as classes do núcleo são responsáveis pela interface com o usuário (aproximadamente 38% do tamanho do sistema) e por várias outras *features* não extraídas neste experimento, tais como persistência, internacionalização, geração de código, engenharia reversa etc.

4.1.2 Métricas de Transversalidade

Essas métricas se destinam a medir o comportamento transversal das *features* extraídas em linhas de produtos baseadas em pré-processadores. Mais especificamente, foram coletadas as seguintes métricas:

- Grau de Espalhamento (SD - *Scattering Degree*): proposta por Liebig et al., essa métrica conta o número de ocorrências de constantes `#ifdef` que definem uma dada *feature* F (LIEBIG et al., 2010). Ou seja, dada uma constante que define uma *feature* particular, SD conta o número de ocorrências dessa constante em expressões `#ifdef`. Para ilustrar essa situação, suponha o fragmento de código apresentado na Figura 16. Nesse exemplo, as constantes `STATEDIAGRAM`, `ACTIVITYDIAGRAM`, `SEQUENCEDIAGRAM` e `COLLABORATIONDIAGRAM` indicam código associados, respectivamente, às *features* DIAGRAMA DE ESTADOS, DIAGRAMA DE ATIVIDADES, DIAGRAMA DE SEQUÊNCIA e DIAGRAMA DE COLABORAÇÃO. Desse modo, SD(DIAGRAMA DE ESTADO)

¹Disponível em <http://logging.apache.org/log4j/>

= 2 e $SD(\text{DIAGRAMA DE ATIVIDADES}) = 2$, uma vez que as constantes mencionadas aparecem em duas expressões `#ifdef` (linhas 4 e 18). De forma análoga, $SD(\text{DIAGRAMA DE SEQUÊNCIA}) = 1$ e $SD(\text{DIAGRAMA DE COLABORAÇÃO}) = 1$, uma vez que as respectivas constantes aparecem em uma única expressão `#ifdef` (linha 13);

- Grau de Entrelaçamento (TD - *Tangling Degree*): para cada par de *features* F_1 e F_2 , essa métrica conta o número de expressões `#ifdef` nas quais F_1 e F_2 são combinadas por operadores AND ou OR. No exemplo da Figura 16, $TD(\text{DIAGRAMA DE ESTADO}, \text{DIAGRAMA DE ATIVIDADES}) = 2$, porque há dois `#ifdef` no qual as duas *features* são combinadas por um operador OR (linhas 4 e 18). De forma análoga, $TD(\text{DIAGRAMA DE SEQUÊNCIA}, \text{DIAGRAMA DE COLABORAÇÃO}) = 1$ porque há um `#ifdef` no qual as duas *features* são combinadas por um operador OR (linha 13). O objetivo dessa métrica é mensurar interações e dependências entre as *features* extraídas em uma LPS baseada em pré-processadores.

```

1 public void actionPerformed(ActionEvent e) {
2     super.actionPerformed(e);
3     Object action = createAction();
4     // #if defined(STATEDIAGRAM) or defined(ACTIVITYDIAGRAM)
5     if (getValue(ROLE).equals(Roles.EXIT)) {
6         Model.getStateMachinesHelper().setExit(getTarget(), action);
7     } else if (getValue(ROLE).equals(Roles.ENTRY)) {
8         Model.getStateMachinesHelper().setEntry(getTarget(), action);
9     } else if (getValue(ROLE).equals(Roles.DO)) {
10        Model.getStateMachinesHelper().setDoActivity(getTarget(), action);
11    } else
12        // #endif
13    // #if defined(COLLABORATIONDIAGRAM) or defined(SEQUENCEDIAGRAM)
14    if (getValue(ROLE).equals(Roles.ACTION)) {
15        Model.getCollaborationsHelper().setAction(getTarget(), action);
16    }
17    // #endif
18    // #if defined(STATEDIAGRAM) or defined(ACTIVITYDIAGRAM)
19    else if (getValue(ROLE).equals(Roles.EFFECT)) {
20        Model.getStateMachinesHelper().setEffect(getTarget(), action);
21    } else
22        // #endif
23        if (getValue(ROLE).equals(Roles.MEMBER)) {
24            Model.getCommonBehaviorHelper().addAction(getTarget(), action);
25        }
26    TargetManager.getInstance().setTarget(action);
27 }

```

Figura 16: Exemplo de contagem das métricas SD e TD

As Tabelas 5 e 6 apresentam os valores coletados para as métricas SD e TD, respectivamente. A Tabela 5 mostra ainda o valor médio de linhas de código (ou seja, a razão LOF/SD) em cada trecho de código anotado como pertencente a uma determinada *feature*.

Tabela 5: *Scattering Degree* (SD)

<i>Feature</i>	SD	LOF/SD
SUORTE COGNITIVO	319	51,16
DIAGRAMA DE ATIVIDADES	136	16,78
DIAGRAMA DE ESTADO	167	23,46
DIAGRAMA DE COLABORAÇÃO	89	17,74
DIAGRAMA DE SEQUÊNCIA	109	49,35
DIAGRAMA DE CASOS DE USO	74	36,65
DIAGRAMA DE IMPLANTAÇÃO	64	49,17
LOGGING	1287	1,68

Tabela 6: *Tangling Degree* (TD)

Pares de <i>Features</i>	TD
(DIAGRAMA DE ESTADO, DIAGRAMA DE ATIVIDADES)	66
(DIAGRAMA DE SEQUÊNCIA, DIAGRAMA DE COLABORAÇÃO)	25
(SUORTE COGNITIVO, DIAGRAMA DE SEQUÊNCIA)	1
(SUORTE COGNITIVO, DIAGRAMA DE IMPLANTAÇÃO)	13

Análise do Comportamento Transversal: A Tabela 5 mostra que LOGGING apresenta o maior valor para a métrica SD. De acordo com os dados apresentados nessa tabela, há 1.287 localizações estáticas no sistema nas quais a *feature* LOGGING é requerida. Com isso, esse valor confirma alegações recorrentes na literatura sobre o comportamento transversal de LOGGING. Todavia, conforme apresentado na Tabela 4, LOGGING possui o segundo menor valor para a métrica LOF entre as *features* consideradas nesse estudo. Isto é, LOGGING é usado em 1.287 pontos do sistema, mas requer poucas linhas de código em cada um desses pontos (em média 1,68 linhas de código em cada localização na qual ele é demandado). Por outro lado, existem *features* como DIAGRAMA DE IMPLANTAÇÃO que são requeridas em apenas 64 pontos do sistema, mas que demandam muito mais código em cada localização (em média 49,17 linhas de código em cada localização). Com isso, pode-se afirmar que não é possível analisar uma *feature* apenas pelo número absoluto de localidades em que ela é demandada. Sem dados relativos, tal como a métrica LOF, a

métrica SD pode levar a conclusões incorretas, tal como afirmar sobre a complexidade de uma *feature* com análises baseadas apenas nessa informação.

Com respeito à métrica *Tangling Degree* (TD), a Tabela 6 nos mostra que há relações entre quatro pares de *features* (para as combinações de *features* não apresentadas nessa tabela, $TD = 0$). Relações de entrelaçamento foram detectadas entre diagramas UML similares, particularmente entre as *features* DIAGRAMA DE ESTADO e DIAGRAMA DE ATIVIDADES e entre as *features* DIAGRAMA DE SEQUÊNCIA e DIAGRAMA DE COLABORAÇÃO. É sabido que tais diagramas são similares. Assim, o entrelaçamento encontrado entre eles denota fragmentos de código que devem ser incluídos quando ambos os diagramas estão habilitados (operador AND) ou quando pelo menos um dos diagramas está habilitado (operador OR). Há também relações de entrelaçamento entre SUPORTE COGNITIVO e dois diagramas: DIAGRAMA DE SEQUÊNCIA e DIAGRAMA DE IMPLANTAÇÃO. Neste caso, o entrelaçamento é usado para denotar classes que proveem suporte cognitivo de forma específica para os diagramas mencionados.

Entre as *features* anotadas nesse estudo, LOGGING é a que mostra-se mais frequentemente lexicalmente aninhada em relação a blocos de códigos associados a outras *features*. De fato, verificou-se que LOGGING aparece aninhada a todas as outras *features* consideradas. Essa propriedade de LOGGING pode ser explicada devido ao seu comportamento transversal (tal como mensurado pela métrica SD). Em outras palavras, LOGGING é um requisito não-funcional cuja implementação é transversal à maioria dos requisitos funcionais do sistema. Além de LOGGING, foram encontrados seis pontos nos quais DIAGRAMA DE ATIVIDADES encontra-se lexicalmente aninhado em relação a DIAGRAMA DE ESTADO e vinte e dois pontos nos quais DIAGRAMA DE ESTADO está lexicalmente aninhado em relação a DIAGRAMA DE ATIVIDADES. Por meio de inspeção do código, foi observado que esse aninhamento ocorre devido ao fato que diagramas de atividades são uma especialização de diagramas de estado, tal como definido pela especificação da UML (FOWLER; SCOTT, 2000; GROUP, 2010). Situação semelhante ocorre entre as *features* DIAGRAMA DE SEQUÊNCIA e DIAGRAMA DE COLABORAÇÃO. Esse fato une-se aos dados apresentados pela métrica TD a fim de confirmar a dependência entre o código de diagramas UML similares.

4.1.3 Métricas de Granularidade

Métricas de granularidade quantificam o nível hierárquico dos elementos de código anotados para uma *feature* específica. Essas métricas são capazes de identificar níveis de

granularidade que abrangem *features* de granularidades grossa e fina. De acordo com a definição apresentada, uma *feature* possui granularidade grossa quando sua implementação ocorre principalmente em unidades sintáticas com um nível hierárquico alto, de acordo com a gramática de Java, tal como pacotes, classes e interfaces. Por outro lado, uma *feature* é de granularidade fina quando seu código é composto por unidades sintáticas de baixo nível, tais como comandos e expressões (KÄSTNER; APEL; KUHLEMANN, 2008). As métricas utilizadas para mensuração de granularidade neste estudo são:

- **Package:** quantifica pacotes totalmente anotados (isto é, todas as classes e interfaces) utilizados para implementar uma *feature*;
- **Class:** quantifica classes inteiramente anotadas para implementar uma *feature*;
- **ClassSignature:** conta pedaços de código anotados associados a assinatura de classes, isto é, cláusulas **extends** ou **implements** de uma classe Java. A Figura 17 apresenta um exemplo de ocorrência desse tipo de granularidade. Nessa figura, a classe **FigEdgeModelElement** (linha 1) somente deve implementar **Highlightable** (linha 12) se a *feature* SUPORTE COGNITIVO (linha 10) estiver selecionada para um determinado produto;

```

1 public abstract class FigEdgeModelElement extends FigEdgePoly
2     implements
3         VetoableChangeListener ,
4         DelayedVChangeListener ,
5         MouseListener ,
6         KeyListener ,
7         PropertyChangeListener ,
8         ArgoNotationEventListener ,
9         ArgoDiagramAppearanceEventListener ,
10        // #if defined(COGNITIVE)
11        // @#$LPS-COGNITIVE: GranularityType: ClassSignature
12        Highlightable ,
13        // #endif
14        IItemUID ,
15        ArgoFig ,
16        Clarifiable {
17        ...

```

Figura 17: Exemplo de anotação do tipo **ClassSignature**

- **InterfaceMethod:** conta assinaturas de métodos pertencentes a uma interface Java anotados para implementar uma *feature*. Na Figura 18, é apresentado um exemplo de ocorrência desse tipo de granularidade. Nessa figura, o método **setToDoItem**

(linha 4) somente existirá na interface `Clarifier` (linha 1) se a *feature* `SUPORTE COGNITIVO` (linha 2) estiver selecionada;

```

1 public interface Clarifier extends Icon {
2     //@#if defined(COGNITIVE)
3     //@#$LPS-COGNITIVE:GranularityType:InterfaceMethod
4     public void setToDoItem(ToDoItem i);
5     //@#endif

```

Figura 18: Exemplo de anotação do tipo `InterfaceMethod`

- **Method:** conta métodos inteiramente anotados para implementar uma *feature* (isto é, assinatura e corpo);
- **MethodBody:** conta métodos cujo corpo (mas não a assinatura) foram anotados para implementar uma *feature*. A Figura 19 apresenta um exemplo de ocorrência desse tipo de granularidade. Nessa figura, o código interno ao método `handleProfileEnd` (linhas 4 a 6) somente será incluído na compilação se a *feature* `LOGGING` (linha 2) estiver selecionada. Em produtos gerados sem essa *feature*, esse método ainda existirá, porém como um corpo vazio;

```

1 protected void handleProfileEnd(XMLElement e) {
2     //@#if defined(LOGGING)
3     //@#$LPS-LOGGING:GranularityType:MethodBody
4     if (profiles.isEmpty()) {
5         LOG.warn("No profiles defined");
6     }
7     //@#endif
8 }

```

Figura 19: Exemplo de anotação do tipo `MethodBody`

- **Attribute:** conta o número de atributos de classe e de instância anotados para implementar uma determinada *feature*;
- **Statement:** conta o número de comandos anotados para implementar uma *feature*, incluindo chamadas de método, atribuições, comandos condicionais, comandos de repetição etc;
- **Expression:** conta o número anotações em operandos e/ou operadores de expressões. A Figura 20 apresenta um exemplo de ocorrência desse tipo de granularidade. Nessa figura, o código anotado incide sobre um operando (linha 4) e um operador

(linha 5) de um comando `if`. Dessa maneira, produtos que não possuam a *feature* `DIAGRAMA DE ESTADO` terão apenas um único operando (linha 7) nesse comando `if`.

```

1 ...
2 if (//#if defined(STATEDIAGRAM)
3     //@#$LPS-STATEDIAGRAM: GranularityType: Expression
4     enclosed instanceof FigStateVertex
5     ||
6     //#endif
7     enclosed instanceof FigObjectFlowState) {
8     changePartition(enclosed);
9 }
10 ...

```

Figura 20: Exemplo de anotação do tipo **Expression**

As métricas de granularidade não consideram elementos que foram contados em uma outra métrica. Por exemplo, a métrica **Class** não considera uma classe ou interface pertencente a um pacote totalmente anotado (isto é, um pacote contabilizado pela métrica **Package**). Essa decisão foi tomada para evitar a contagem dupla de um elemento específico.

Nesta dissertação de mestrado, foram consideradas as seguintes categorias para elementos sintáticos de granularidade grossa e fina:

- Granularidade grossa: **Package**, **Class**, **InterfaceMethod**, **MethodBody**, **Method**;
- Granularidade fina: **ClassSignature**, **Statement**, **Attribute** e **Expression**.

Análise de Granularidade: As Tabelas 7 e 8 mostram, respectivamente, os valores coletados para as métricas de granularidade grossa e fina propostas. Duas principais constatações surgem a partir da análise dessas tabelas:

- A implementação das *features* **SUPORTE COGNITIVO**, **DIAGRAMA DE ATIVIDADES** e **DIAGRAMA DE ESTADO** possui mais elementos de granularidade grossa do que as outras *features*. Isso significa que parte do código de tais *features* está propriamente modularizado por elementos tais como pacotes, classes e interfaces. Entretanto, tais componentes de granularidade grossa são referenciados por outros elementos de menor granularidade, tais como chamadas de métodos. Em resumo, a implementação

das *features* anteriormente mencionadas é bem modularizada, mas seu uso é disperso pelo resto do código.

- LOGGING possui sozinha mais elementos de granularidade fina do que o somatório das outras *features*. Ela demanda 789 comandos (**Statement**) e 241 atributos (**Attribute**), enquanto as outras *features* juntas possuem 236 comandos (**Statement**) e 12 atributos (**Attribute**). Isso a caracteriza como uma *feature* de granularidade fina.

Tabela 7: Métricas de Granularidade Grossa

<i>Feature</i>	Package	Class	InterfaceMethod	Method	MethodBody	Total
SUORTE COGNITIVO	11	8	1	10	5	35
DIAG. DE ATIVIDADES	2	31	0	6	6	45
DIAG. DE ESTADO	0	48	0	15	2	65
DIAG. DE COLABORAÇÃO	2	8	0	5	3	18
DIAG. DE SEQUÊNCIA	4	5	0	1	3	13
DIAG. DE CASOS DE USO	3	1	0	1	0	5
DIAG. DE IMPLANTAÇÃO	2	14	0	0	0	16
LOGGING	0	0	0	3	15	18

Tabela 8: Métricas de Granularidade Fina

<i>Feature</i>	ClassSignature	Statement	Attribute	Expression	Total
SUORTE COGNITIVO	2	49	3	2	56
DIAG. DE ATIVIDADES	0	59	2	6	67
DIAG. DE ESTADO	0	22	2	5	29
DIAG. DE COLABORAÇÃO	0	40	1	1	42
DIAG. DE SEQUÊNCIA	0	31	2	3	36
DIAG. DE CASOS DE USO	0	22	1	0	23
DIAG. DE IMPLANTAÇÃO	0	13	1	3	17
LOGGING	0	789	241	1	1031

4.1.4 Métricas de Localização

As métricas de localização fornecem informações sobre a posição estática dos elementos sintáticos anotados para implementação das *features* do ArgoUML-SPL. Essas métricas foram calculadas apenas para elementos de granularidade **Statement** (segundo a classificação da subseção anterior). O objetivo é mostrar a localização dos comandos que foram associados a cada *feature*, isto é, se eles ocorrem logo no início ou no final do

corpo de um método, antes de um comando **return** ou aninhados a outros comandos, por exemplo. Essas métricas foram inspiradas no modelo de pontos de junção implementados por linguagens de programação orientadas a aspectos, tal como AspectJ (KICZALES et al., 2001).

As métricas de localização são definidas conforme se segue:

- **StartMethod**: conta os comandos anotados para uma determinada *feature* que aparecem no início de um método;
- **EndMethod**: conta os comandos anotados para uma determinada *feature* que aparecem no final de um método;
- **BeforeReturn**: conta os comandos anotados para uma determinada *feature* que aparecem imediatamente antes de um comando **return** não anotado;
- **NestedStatement**: conta os comandos anotados para uma determinada *feature* que aparecem aninhados no escopo de um comando mais externo não anotado.

A Figura 21 apresenta um trecho de código anotado cuja localização enquadra-se em dois grupos: **BeforeReturn** e **NestedStatement**. Claramente, esse trecho de código está localizado imediatamente antes de um comando **return** não anotado (linha 11). Além disso, o comando anotado (linha 9) encontra-se aninhado em relação a comandos externos não anotados (linhas 3 a 5).

```

1 private static String getMostRecentProject() {
2     String s = Configuration.getString(Argo.KEY_MOST_RECENT_PROJECT_FILE,
3         "");
4     if (!"".equals(s)) {
5         File file = new File(s);
6         if (file.exists()) {
7             //if defined(LOGGING)
8             //@$LPS-LOGGING:Localization:BeforeReturn
9             //@$LPS-LOGGING:Localization:NestedStatement
10            LOG.info("Re-opening project " + s);
11            //endif
12            return s;
13        }
14    }
15    ...

```

Figura 21: Exemplo de anotação do tipo **BeforeReturn** e **NestedStatement**

A Tabela 9 apresenta os valores coletados para as métricas de localização.

Tabela 9: Métricas de Localização

<i>Feature</i>	StartMethod	EndMethod	BeforeReturn	NestedStatement
SUORTE COGNITIVO	3	5	0	10
DIAG. DE ATIVIDADES	2	20	2	19
DIAG. DE ESTADO	2	19	3	12
DIAG. DE COLABORAÇÃO	1	10	3	3
DIAG. DE SEQUÊNCIA	0	9	3	7
DIAG. DE CASOS DE USO	0	2	0	1
DIAG. DE IMPLANTAÇÃO	0	0	0	3
LOGGING	127	21	89	336

Análise de Localização: Dentre as quatro localizações analisadas, aquelas que podem ser diretamente implementadas pelo modelo de pontos de junção de AspectJ são: **StartMethod**, **EndMethod** e **BeforeReturn**. Entretanto, para **LOGGING** – a *feature* tipicamente transversal analisada nesse estudo – esses três tipos de localização ocorreram menos frequentemente do que, por exemplo, **NestedStatement**. De fato, **NestedStatement** em **LOGGING** é pelo menos duas vezes mais frequente do que outras localizações. Esse resultado é interessante porque *features* transversais podem naturalmente surgir como candidatas para aspectização. Entretanto, a modularização física de código aninhado em relação em comandos externos é mais desafiadora, requerendo, por exemplo, transformações prévias no código orientado por objetos (NASSAU; VALENTE, 2009; NASSAU; OLIVEIRA; VALENTE, 2009).

4.2 Discussão

Esta seção apresenta e discute alguns resultados que podem ser destacados em neste estudo e compara tais resultados com outros trabalhos publicados na literatura. Em particular, as subseções a seguir discutem algumas formas transversais encontradas no ArgoUML-SPL e quão prejudiciais são essas formas transversais (Seção 4.2.1) e a viabilidade do uso de técnicas de desenvolvimento orientadas a aspectos para separação das oito *features* opcionais do ArgoUML-SPL (Seção 4.2.2).

4.2.1 Formas Recorrentes de Features

Trabalhos recentes têm mostrado que nem todas as formas de manifestações transversais de uma *feature* são prejudiciais aos atributos de qualidade de um sistema, tais

como modularidade e estabilidade (DUCASSE; GÎRBA; KUHN, 2006; EADDY et al., 2008; FIGUEIREDO et al., 2009). Assim, é importante estudar e identificar as *features* que apresentam as formas transversais mais prejudiciais. Os resultados apresentados na Seção 4.1 auxiliam na identificação de três formas recorrentes de manifestações transversais documentadas na literatura: *God Concern*, *Black Sheep* e *Octopus* (DUCASSE; GÎRBA; KUHN, 2006; FIGUEIREDO et al., 2009).

O padrão *God Concern* ocorre quando uma *feature* requer uma fração expressiva do código do sistema em sua implementação (FIGUEIREDO et al., 2009). Esse tipo de *feature* foi inspirado na definição de *God Class* feita por Fowler (FOWLER et al., 1999). Ou seja, assim como em *God Class*, elementos que implementam uma *feature God Concern* agregam excessiva responsabilidade no sistema de software. De acordo com os dados apresentados nas Tabelas 4, 7 e 8 pode-se observar que a *feature* SUPORTE COGNITIVO possui a forma definida por *God Concern*. Isto é, a implementação dessa *feature* envolve uma grande quantidade de código fonte do sistema, como por exemplo, 11 pacotes e 9 classes (Tabela 7). Além disso, 16.319 linhas de código do sistema (13,56%) são usados para implementar SUPORTE COGNITIVO (Tabela 4). No entanto, estudos recentes mostraram pouca correlação entre *features God Concern* com problemas de modularidade e estabilidade de um sistema (FIGUEIREDO et al., 2009).

A manifestação transversal oposta a *God Concern* é caracterizada como *Black Sheep*. Esse padrão define uma *feature* transversal implementada por alguns trechos de código espalhados em diferentes classes do sistema (DUCASSE; GÎRBA; KUHN, 2006). Uma *feature Black Sheep* não agrega muita responsabilidade e sua remoção não impacta drasticamente o funcionamento do sistema. Por exemplo, *Black Sheep* ocorre quando nenhuma classe implementa exclusivamente uma determinada *feature*. Nesse caso, essa *feature* é implementada por pequenos trechos de código, tais como comandos ou expressões, espalhados pelo sistema. A *feature* LOGGING segue o padrão *Black Sheep* no ArgoUML-SPL, pois nenhuma classe é completamente dedicada à implementação dessa *feature* (Tabela 7). Entretanto, grande parte do código da *feature* é implementado por elementos de granularidade fina, tais como comandos (**Statement**) e atributos (**Attribute**) (Tabela 8). Engenheiros de software devem estar atentos a uma *feature* que se manifesta como *Black Sheep*, pois estudos revelam correlação moderada entre este tipo de *feature* e a estabilidade de um sistema (FIGUEIREDO et al., 2009).

Diferentemente de *God Concern* e *Black Sheep*, o padrão de transversalidade *Octopus* define uma *feature* transversal que se encontra parcialmente modularizada em uma ou

mais classes (DUCASSE; GÎRBA; KUHN, 2006). Apesar de modularizada em algumas classes, uma *feature Octopus* também se espalha por várias outras classes do sistema. Portanto, pode-se identificar dois tipos de classes nessa manifestação transversal: (a) classes que são completamente dedicadas à implementação da *feature* (isto é, representam o corpo do polvo); (b) classes que usam apenas pequenas partes de seu código para implementar a *feature* (isto é, representam os tentáculos do polvo).

Com base nessa definição e nos dados apresentados na Seção 4.1, pode-se observar que a implementação das *features* que representam os seis diagramas UML – Estados, Atividades, Colaboração, Sequência, Casos de Uso e Implantação – seguem o padrão *Octopus*. Isto é, para todas essas *features*, pode-se identificar classes completamente dedicadas à sua implementação (corpo) e outras classes que utilizam apenas alguns serviços das classes desse primeiro tipo (tentáculos). Assim como *Black Sheep*, *features* que se enquadram no padrão *Octopus* apresentam de moderada a elevada correlação com estabilidade do sistema (FIGUEIREDO et al., 2009). Portanto, essas *features* devem ser cuidadosamente avaliadas pelos engenheiros de software, sendo que possivelmente pode-se aplicar algumas das técnicas de refatoração para sua modularização.

4.2.2 Modularização Por Meio de Aspectos: É Viável?

O conjunto de métricas apresentado na Seção 4.1 fornece importantes dados para que se possa fazer uma avaliação sobre a possibilidade de refatoração das *features* selecionadas por meio de uma tecnologia de modularização tal como aspectos. Isto é, uma análise detalhada das informações mostradas nas tabelas dessa seção permite avaliar quais *features* são boas candidatas a esse tipo de refatoração.

Por exemplo, de acordo com a Tabela 7, SUPORTE COGNITIVO é a *feature* mais bem modularizada no ArgoUML-SPL. Isto é, sua implementação está concentrada em alguns pacotes e classes específicos do sistema. Entretanto, de acordo com os dados apresentados na Tabela 8, esta *feature* também usa diversos elementos de granularidade fina em sua implementação, ficando atrás apenas das *features* LOGGING e DIAGRAMA DE ATIVIDADES. Então, embora SUPORTE COGNITIVO esteja apenas parcialmente modularizada, ela também oferece oportunidade para o uso de aspectos.

Em comparação a SUPORTE COGNITIVO, as *features* DIAGRAMA DE ESTADOS, DIAGRAMA DE ATIVIDADES, DIAGRAMA DE SEQUÊNCIA e DIAGRAMA DE COLABORAÇÃO possuem complexidade extra para serem refatoradas em aspectos, devido à natureza entrelaçada de sua implementação, como pode ser visto na Tabela 6 (métrica TD). As

features que representam os demais diagramas UML também possuem certa complexidade para uma possível aspectização devido a natureza transversal de sua implementação.

LOGGING é frequentemente citada como sendo uma *feature* que apresenta características que a tornam uma boa candidata a refatoração por meio de aspectos (KICZALES et al., 2001; MEZINI; OSTERMANN, 2003). Nesta dissertação, como pode ser visto na Tabela 9, essa *feature* foi a que apresentou maior número de trechos de código em condições favoráveis para interceptação por meio de pontos de junção definidos pelas linguagens que implementam aspectos. Entretanto, em termos relativos, esse número ainda é pequeno frente ao total de comandos que implementam essa *feature* (30%). Além disso, após uma investigação manual no código fonte, verificou-se que as mensagens registradas pelas chamadas de *logging* são diferentes umas das outras em 84,5% dos casos. Esse cenário implica então em uma extensa implementação de diferentes aspectos para cada mensagem. Em outras palavras, caso fosse implementada através de aspectos, essa *feature* certamente iria requerer um baixo grau de quantificação (VALENTE et al., 2010).

Diante desse cenário, pode-se afirmar preliminarmente que não seria produtivo refatorar qualquer uma das *features* extraídas neste trabalho por meio de uma tecnologia para modularização física de interesses transversais, tal como aspectos.

4.3 Comentários Finais

Este capítulo apresentou e discutiu as métricas criadas e adaptadas a partir de trabalhos de outros autores, com a finalidade de avaliar e caracterizar a linha de produtos extraída. Por meio da análise dos números coletados, foram efetuadas avaliações sobre as *features* extraídas, tendo em vista caracterizá-las segundo padrões discutidos na literatura e avaliá-las sobre uma ótica voltada a tecnologia de orientação a aspectos. A maior contribuição desse capítulo consiste, então, na apresentação de um arcabouço para avaliação e caracterização de linhas de produtos baseadas em pré-processadores. Acredita-se que tal arcabouço possa servir de apoio e inspiração para pesquisadores e profissionais interessados em assuntos relacionados a linhas de produtos de software.

O próximo capítulo apresenta as principais contribuições desta dissertação de mestrado e relaciona possíveis linhas de trabalhos futuros.

5 CONCLUSÕES

5.1 Contribuições

Descreveu-se nesta dissertação uma experiência que compreendeu a extração de oito *features* complexas de um sistema real (ArgoUML), tendo como finalidade a geração de uma linha de produtos de software (ArgoUML-SPL). As principais contribuições deste estudo são:

- Extração de uma linha de produtos a partir de um sistema real e complexo. A versão do ArgoUML que foi usada nesse trabalho contém cerca de 120 KLOC e foram anotadas cerca de 37 KLOC por meio da utilização de diretivas de compilação condicional;
- O ArgoUML-SPL é por ele mesmo a principal contribuição do trabalho, uma vez que seu código fonte está publicamente disponível no sítio: <http://argouml-spl.tigris.org>. Esta é uma característica chave para promover o uso do ArgoUML-SPL entre pesquisadores e profissionais interessados em assuntos relacionados a linhas de produtos de software;
- As técnicas utilizadas para anotação do código descritas na Seção 3.4 tendo em vista a posterior coleta de dados para análise, mostraram-se bastante produtivas, uma vez que possibilitaram a coleta automatizada dos dados e, portanto, podem ser aplicadas na avaliação de outros projetos de extração de linhas de produtos;
- Projeto e implementação da ferramenta SPLMetricsGather, a qual provê mecanismos para coleta automatizada de métricas sobre linhas de produtos de software implementadas por meio de compilação condicional. Acredita-se que essa ferramenta pode ser de grande utilidade para pesquisadores e desenvolvedores interessados na avaliação de linhas de produtos;
- Foi proposto um arcabouço para avaliação e caracterização de linhas de produtos baseadas em pré-processadores. Esse arcabouço foi inspirado em um conjunto de

métricas originalmente proposto por Liebig et al. (LIEBIG et al., 2010). Entretanto, esse arcabouço foi estendido com novas métricas, tais como aquelas relacionadas a espalhamento e entrelaçamento. O arcabouço estendido suporta a caracterização das *features* de acordo com diferentes perspectivas, incluindo tamanho, comportamento transversal, granularidade e localização estática no código. A análise combinada dessas diferentes perspectivas permitiu entender e caracterizar as *features* extraídas no ArgoUML-SPL. Este conhecimento foi documentado com detalhes no Capítulo 4 para permitir futuras replicações deste trabalho por diferentes grupos de pesquisa.

Publicações: Os principais resultados desta dissertação de mestrado estão reportados no seguinte artigo:

- Marcus Vinícius Couto, Marco Túlio Valente and Eduardo Figueiredo. Extracting Software Product Lines: A Case Study Using Conditional Compilation. In *15th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 1-10, 2011. (Classificação Qualis: A2; Taxa de aceitação: 29/102).

5.2 Trabalhos Futuros

Trabalhos futuros podem incluir a extração de outras *features*, tais como geração de código, engenharia reversa, internacionalização etc. Um dos objetivos da disponibilização pública da versão atual da linha de produtos é motivar pesquisadores a contribuir com essa extensão. Pode-se também investigar a extração das *features* do ArgoUML-SPL por meio de outros paradigmas de programação, tal como programação orientada a aspectos (AOP), programação orientada por *features* (FOP) e anotações disciplinadas.

Outra linha de trabalho futuro inclui a criação de um aplicativo web destinado aos usuários finais do ArgoUML-SPL. Esse aplicativo deverá prover uma interface com recursos para a seleção das diferentes *features* disponíveis, e a partir dessa seleção, deverá ser capaz de gerar um produto de acordo com as necessidades do usuário.

O ArgoUML está em constante desenvolvimento pela sua comunidade. A versão utilizada como a base para a geração da linha de produtos extraída neste trabalho (0.28.1) foi publicamente disponibilizada em janeiro de 2010 pela comunidade do ArgoUML. No decorrer do ano vários outros *releases* foram lançados, e atualmente o ArgoUML encontra-se na versão 0.30.2. Tendo em vista manter a linha de produtos ArgoUML-SPL alinhada

com as inovações introduzidas no ArgoUML, sugere-se avaliar as mudanças existentes entre a versão utilizada na extração e a versão atual do sistema, replicando-se então todo o código evoluído para a linha de produtos.

Conforme afirmado no Capítulo 3, diretivas de compilação condicional foram selecionadas como a tecnologia primária para a extração de *features* na linha de produtos ArgoUML-SPL. O objetivo foi prover um ponto de partida para comparação com outras técnicas de modularização. Finalmente, a anotação manual de *features* por meio de diretivas `#ifdef` e `#endif` demonstrou ser uma tarefa tediosa e repetitiva. Diante disso, uma linha de trabalho futuro consiste na investigação de técnicas automáticas ou semi-automáticas para marcação de *features*, tais como aquelas descritas em (KÄSTNER; APEL; KUHLEMANN, 2008; BORGES; VALENTE, 2009).

REFERÊNCIAS

- ADAMS, B. et al. Can we refactor conditional compilation into aspects? In: *8th ACM International Conference on Aspect-Oriented Software Development (AOSD)*. Charlottesville, Virginia, USA: ACM, 2009. p. 243–254.
- APEL, S.; KÄSTNER, C. Virtual separation of concerns - a second chance for preprocessors. *Journal of Object Technology*, v. 8, n. 6, p. 59–78, 2009.
- BATORY, D. Feature-oriented programming and the ahead tool suite. In: *26th International Conference on Software Engineering (ICSE)*. Edinburgh, Scotland, UK: [s.n.], 2004. p. 702–703.
- BEZERRA, E. *Princípios de Análise e Projetos de Sistemas com UML*. Rio de Janeiro: Editora Campus, 2001. ISBN 85-352-1032-6.
- BORGES, V.; VALENTE, M. T. Coloração automática de variabilidades em linhas de produtos de software. In: *III Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software (SBCARS)*. Natal, RN, Brasil: [s.n.], 2009. p. 67–80.
- BROOKS, F. P. *The mythical man-month (anniversary ed.)*. [S.l.]: Addison-Wesley, 1995. ISBN 0-201-00650-2.
- CLEMENTS, P.; KREUGER, C. Point/counterpoint. *IEEE Software*, v. 19, p. 28–31, 2002.
- CLEMENTS, P.; NORTHROP, L. M. *Software Product Lines: Practices and Patterns*. [S.l.]: Addison-Wesley, 2002. ISBN 978-0201703320.
- DEREMER, F.; KRON, H. Programming-in-the large versus programming-in-the-small. In: *International Conference on Reliable Software*. Los Angeles, California: ACM, 1975. p. 114–121.
- DUCASSE, S.; GÎRBA, T.; KUHN, A. Distribution map. In: *International Conference on Software Maintenance (ICSM)*. Philadelphia, Pennsylvania, USA: [s.n.], 2006. p. 203–212.
- EADDY, M. et al. Do crosscutting concerns cause defects? *IEEE Transactions on Software Engineering*, v. 34, n. 4, p. 497–515, 2008.
- FIGUEIREDO, E. et al. Evolving software product lines with aspects: an empirical study on design stability. In: *30th International Conference on Software Engineering (ICSE)*. Leipzig, Germany: [s.n.], 2008. p. 261–270.
- FIGUEIREDO, E. et al. Crosscutting patterns and design stability: An exploratory analysis. In: *International Conference on Program Comprehension (ICPC)*. Vancouver, BC, Canada: [s.n.], 2009. p. 138–147.

FOWLER, M. et al. *Refactoring: Improving the Design of Existing Code*. [S.l.]: Addison-Wesley, 1999. ISBN 978-0201485677.

FOWLER, M.; SCOTT, K. *UML distilled*. [S.l.]: Addison-Wesley, 2000. ISBN 978-0201657838.

FURTADO, A.; SANTOS, A.; RAMALHO, G. Streamlining domain analysis for digital games product lines. In: *14th International Software Product Line Conference (SPLC)*. Jeju, South Korea: [s.n.], 2010.

GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. [S.l.]: Addison-Wesley, 1994. ISBN 978-0201633610.

GODIL, I.; JACOBSEN, H.-A. Horizontal decomposition of prevayler. In: *Centre for Advanced Studies on Collaborative Research (CASCON)*. Toronto, Ontario, Canada: [s.n.], 2005. p. 83–100.

GROUP, O. M. *Unified Modeling Language (UML)*. [S.l.], 2010. Disponível em: <http://www.omg.org/spec/UML/>.

HILSDALE, E.; KERSTEN, M. Aspect-oriented programming with aspectj. In: *Mini-course at OOPSLA*. [S.l.: s.n.], 2001.

JOHNSON, R. E. Components, frameworks, patterns. In: *Symposium on Software Reusability (SSR)*. Boston, MA: [s.n.], 1997. p. 10–17.

KÄSTNER, C.; APEL, S.; BATORY, D. A case study implementing features using aspectj. In: *11th International Software Product Line Conference (SPLC)*. Kyoto, Japan: [s.n.], 2007. p. 223–232.

KÄSTNER, C.; APEL, S.; KUHLEMAN, M. Granularity in software product lines. In: *30th International Conference on Software Engineering (ICSE)*. Leipzig, Germany: [s.n.], 2008. p. 311–320.

KERNIGHAN, B.; RITCHIE, D. *The C Programming Language*. [S.l.]: Addison-Wesley, 1988. ISBN 978-0131103627.

KICZALES, G. et al. An overview of AspectJ. In: *15th European Conference on Object-Oriented Programming (ECOOP)*. Budapest, Hungary: Springer-Verlag, 2001. (LNCS, v. 2072), p. 327–355.

KICZALES, G. et al. Aspect-oriented programming. In: *11th European Conference on Object-Oriented Programming (ECOOP)*. Jyväskylä, Finland: Springer-Verlag, 1997. (LNCS, v. 1241), p. 220–242.

KRASNER, G. E.; POPE, S. T. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, SIGS Publications, Denville, NJ, USA, v. 1, p. 26–49, August 1988. ISSN 0896-8438.

KRUEGER, C. W. Easing the transition to software mass customization. In: *4th International Workshop on Software Product-Family Engineering*. Bilbao, Spain: Springer-Verlag, 2001. (LNCS, v. 2290), p. 282–293.

- LIEBIG, J. et al. An analysis of the variability in forty preprocessor-based software product lines. In: *32nd International Conference on Software Engineering (ICSE)*. Cape Town, South Africa: [s.n.], 2010.
- LIU, J.; BATORY, D.; LENGAUER, C. Feature oriented refactoring of legacy applications. In: *28th International Conference on Software Engineering (ICSE)*. Shanghai, China: [s.n.], 2006. p. 112–121.
- LOPEZ-HERREJON, R.; BATORY, D.; COOK, W. R. Evaluating support for features in advanced modularization technologies. In: *19th European Conference on Object-Oriented Programming (ECOOP)*. Glasgow, UK: Springer-Verlag, 2005. (LNCS, v. 3586), p. 169–194.
- LOPEZ-HERREJON, R. E.; BATORY, D. S. A standard problem for evaluating product-line methodologies. In: *3th International Conference on Generative and Component-Based Software Engineering (GPCE)*. [S.l.]: Springer-Verlag, 2001. (LNCS, v. 2186), p. 10–24.
- MARINHO, F. G. et al. A software product line for the mobile and context-aware applications domain. In: *14th International Software Product Line Conference (SPLC)*. Jeju, South Korea: [s.n.], 2010.
- MEZINI, M.; OSTERMANN, K. Conquering aspects with caesar. In: *2nd International Conference on Aspect-Oriented Software Development (AOSD)*. Boston, Massachusetts: [s.n.], 2003. p. 90–99.
- NASSAU, M.; OLIVEIRA, S.; VALENTE, M. T. Guidelines for enabling the extraction of aspects from existing object-oriented code. *Journal of Object Technology*, v. 8, n. 3, p. 1–19, 2009.
- NASSAU, M.; VALENTE, M. T. Object-oriented transformations for extracting aspects. *Information and Software Technology*, Elsevier, v. 51, n. 1, p. 138–149, 2009.
- PARNAS, D. L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, v. 15, n. 12, p. 1053–1058, 1972.
- PARNAS, D. L. On the design and development of program families. *IEEE Transactions on Software Engineering*, v. 2, n. 1, p. 1–9, 1976.
- PENDER, T. *UML Bible*. [S.l.]: Wiley Publishing, Inc., 2003. ISBN 0-7645-2604-9.
- PRESSMAN, R. S. *Software Engineering: A Practitioner's Approach*. 6. ed. [S.l.]: McGraw-Hill, 2005. 77–126 p.
- ROBBINS, J.; REDMILES, D. Cognitive support, UML adherence, and XMI interchange in Argo/UML. In: *Information and Software Technology*. [S.l.: s.n.], 1999. p. 79–89.
- SPENCER, H. #ifdef considered harmful, or portability experience with C News. In: *USENIX Conference*. San Diego, CA: [s.n.], 1992. p. 185–197.
- SUGUMARAN, V.; PARK, S.; KANG, K. C. Introduction to the special issue on software product line engineering. *Communications ACM*, v. 49, n. 12, p. 28–32, 2006.

TOLKE, L.; KLINK, M.; WULP, M. van der. *ArgoUML Cookbook*. [S.l.], 2010. Disponível em: <http://argouml.tigris.org/wiki/Cookbook>.

VALENTE, M. T. et al. On the benefits of quantification in aspectj systems. *Journal of the Brazilian Computer Society*, v. 16, n. 2, p. 133–146, 2010.