Pontifícia Universidade Católica
do Rio de Janeiro

Uirá Kulesza

Uma Abordagem Orientada a Aspectos para o Desenvolvimento de Frameworks

Tese de Doutorado

Tese apresentada ao Programa de Pós-Graduação em Informática da PUC-Rio como requisito parcial para obtenção do grau de Doutor em Informática.

Orientador: Carlos José Pereira de Lucena

Rio de Janeiro
Abril de 2007

Uirá Kulesza

Uma Abordagem Orientada a Aspectos para o Desenvolvimento de Frameworks

Tese apresentada ao Programa de Pós-graduação em Informática da PUC-Rio como requisito parcial para obtenção do grau de Doutor em Informática. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Carlos José Pereira de Lucena
Orientador
Departamento de Informática – PUC-Rio

Prof. Arndt von Staa
Departamento de Informática – PUC-Rio

Prof. Renato Cerqueira
Departamento de Informática – PUC-Rio

Prof. Paulo Henrique Monteiro Borba
Centro de Informática – UFPE

Prof. Paulo Cesar Masiero
Instituto de Ciências Matemática e de Computação – USP

Prof. José Eugenio Leal
Coordenador Setorial do Centro
Técnico Científico – PUC-Rio

Rio de Janeiro
25 de Abril de 2007

Uirá Kulesza

Graduou-se no Curso de Bacharelado em Ciência da Computação da Universidade Federal de Campina Grande (UFCG) em 1998. Obteve o título de Mestre em Ciência da Computação na Universidade de São Paulo (USP) em 2000. Atuou como coordenador de projetos, engenheiro de processo e engenheiro de software no Centro de Estudos e Sistemas Avançados do Recife (CESAR) e na Qualiti Software Processes de 2000 a 2002. É pesquisador na área de Desenvolvimento de Software Orientado a Aspectos no Laboratório de Engenharia de Software (LES) da PUC-Rio, desde 2003.

Ficha Catalográfica

Kulesza, Uirá

Uma abordagem orientada a aspectos para o desenvolvimento de frameworks / Uirá Kulesza ; orientador: Carlos José Pereira de Lucena. – Rio de Janeiro : PUC, Departamento de Informática, 2007.

205 f. ; 30 cm

Tese (Doutorado em Informática) – Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2007.

Inclui referências bibliográficas.

1. Informática – Teses. 2. Frameworks orientados a objetos. 3. Desenvolvimento de software orientado a aspectos. 4. Arquiteturas de família de sistemas. 5. Desenvolvimento generativo. 6. Projeto de software. I. Lucena, Carlos José Pereira de. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Ao nosso bondoso Pai que nunca
deixa de nos guiar nas nossas caminhadas.

Agradecimentos

Durante os anos de doutorado na PUC-Rio, tive a oportunidade de não apenas aprender a encarar problemas desafiadores e complexos vivenciados pela comunidade de engenharia de software, mas também desfrutar da convivência, conhecimento e amizade de vários parceiros de pesquisa do Laboratório de Engenharia de Software (LES) e de outras universidades.

Professor Lucena me acolheu no LES, me manteve sempre motivado e focado na minha pesquisa. Ele também transmitiu tremendo conhecimento na arte de coordenar e sincronizar atividades de vários grupos de pesquisa atuando em um mesmo laboratório. Passado todos esses anos no Rio, posso dizer que tive não apenas um excelente orientador para enveredar na carreira científica, mas ganhei sobretudo a amizade de uma grande pessoa.

Alessandro Garcia (o "Véio") foi um parceiro constante e ativo de vários trabalhos de pesquisa durante os últimos anos. Meu doutorado se iniciou como *spin-off* da sua tese, e as direções seguintes que trilhei receberam sempre inestimável contribuição de sua parte. A parceira continou mesmo com sua partida para Lancaster, e eu posso dizer com toda certeza que ele tem uma presença "*crosscutting*" nesse trabalho. Agradeço também pela oportunidade de realização conjunta dos estudos empíricos quantitativos com outros membros do grupo de Aspectos da PUC-Rio.

Vander Alves também contribuiu significativamente para os resultados alcançados nesse trabalho. Tivemos excelentes e estimulantes discussões durante o desenvolvimento de nossas respectivas teses. Juntamente com Alberto Costa Neto e professor Paulo Borba, todos do *Software Productivity Group* (SPG) da UFPE, eles contribuíram para validar e evoluir a abordagem proposta na tese, aplicando-a em um estudo de caso.

Roberta Coelho e Elder Cirilo do grupo de aspectos da PUC-Rio também tiveram participação substancial e ativa na realização desse trabalho. Roberta na

definição e validação da abordagem com colaboração nos estudos de caso, e Elder na implementação da ferramenta que contempla o modelo generativo proposto na tese e que é o tema central da sua dissertação de mestrado. Raoni Kulesza e Klessis Dias estão usando muito dos conceitos propostos na tese no desenvolvimento de frameworks orientado a aspectos para diferentes domínios.

Cláudio Sant´Anna (o "Baiano") me acolheu desde o meu primeiro dia de trabalho no grupo de Aspectos do LES, foi um amigo extremamente presente e trouxe sempre sugestões diárias construtivas para evolução do trabalho. Ele foi também parceiro constante em várias outras pesquisas realizadas durante o doutorado.

Diversos outros membros/amigos do grupo de aspectos do LES contribuíram indiretamente na realização do trabalho e também para o meu crescimento profissional: (i) professor Arndt von Staa ofereceu várias sugestões e novas perpectivas do trabalho, sobretudo em relação ao uso da tecnologia de geradores de aplicação; (ii) Christina Chavez da UFBA, doutora e ex-pesquisadora do grupo de aspectos da PUC-Rio, trouxe sempre sugestões/questionamentos e ofereceu novas direções para a realização do trabalho; e finalmente, (iii) Cidiane Lobato e Eduardo Magno compartilharam comigo os resultados de suas pesquisas e estimularam discussões "cruzadas" de nossos trabalhos.

Outros pesquisadores que agradeço pelos trabalhos conjuntos ou sugestões oferecidas em algum momento da realização desse trabalho são: André van der Hoek, Awais Rashid, Birgit Geppert, David Weiss, Don Cowan, Jean-Pierre Briot, Itana Gimenes, Krzysztof Czarnecki, Kyo Kang, Lyrene Fernandes, Marcílio Mendonça, Michal Antkiewicz, Mira Mezini, Paulo Alencar, Peter Kim, Thais Batista, Toacy Cavalcante.

Foi um prazer aprender e colaborar com todos vocês, amigos e parceiros, e espero que a colaboração se perpetue ainda por vários anos. E a amizade dure para sempre!

Agradeço a todos os membros da banca pelas sugestões oferecidas para

melhoria do texto apresentado nesse documento e novas perspectivas de reflexão do trabalho.

Agradeço as agências de fomento FAPERJ e CNPq que forneceram o apoio financeiro necessário para realização desse trabalho.

Não poderia também deixar de agradecer a imensa "família" de amigos que me apoiaram imensamente durante esses anos de doutorado.

Roberta Coelho, minha esposa querida, sempre me motivou e me manteve animado nos momentos difíceis de realização da tese. Ao seu lado encarar o doutorado foi muito mais fácil e tranquilo.

Vera Menezes me ofereceu seu sorriso diário e o apoio necessário a qualquer problema adicional encontrado no doutorado. Obrigado Verita, por tudo! Sentirei imensa saudade sua no ambiente de trabalho.

Sílvia Passos, minha madrinha e amiga, me ofereceu sempre seu lar e sua divertida e agradável companhia durante vários finais de semana. Ricardo Rocha e Renata foram também amigos importantes durante todos esses anos de caminhada no Rio. Juba e Dilene foram companhias constantes em programas de lazer, shows e choppinhos de relaxamento nos finais de semana.

Vários outros amigos do LES e da PUC tornaram a minha vida de doutorando em diferentes fases mais divertida e fácil, entre eles (juntamente com os já citados acima): Akeo, Anarosa, Andrew, Bruno, Caculé, Carol, Chicão, Daflon, Dani "Gaúcha", Dani Kussel, Daniel, Felipe, Firmo, Guilherme, Guga, Hana, Ivan, Karla, Léo Cunha, LF, Maíra, Mariela, Michel, Miriam, Pádua, Renato Maia, Ricardo Choren, Ricardo Gralhoz, Rodrigo "Alagoano", Vagner, Viviane.

Agradeço também a toda equipe e amigos do Centro de Pastoral Anchieta da PUC-Rio por todos os ensinamentos durante esses anos que contribuíram imensamente para a minha formação humana/social. As sementes plantadas por vocês, estão certamente germinando.

Outros amigos fundamentais nessa caminhada foram a família Coelho/Cordeiro, família Mitsunaga/Kulesza e todos do DreamTeam (turma de graduação da UFCG).

Finalmente agradeço meu pai, minha mãe, minha esposa e cada um dos meus irmãos (incluindo Nandinho), pelo apoio incondicional, sorrisos, pensamento positivo, aprendizado mútuo e atenção dedicada durante todos esses anos. Sem vocês ao lado tudo isso ficaria sem sentido! Obrigado por me fazerem enxergar e sentir os verdadeiros valores da vida!

Resumo

Kulesza, Uirá. **Uma Abordagem Orientada a Aspectos para o Desenvolvimento de Frameworks.** Rio de Janeiro, 2007. 205p. Tese de Doutorado - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Esse trabalho propõe uma abordagem sistemática para o desenvolvimento de frameworks usando técnicas orientadas a aspectos (OA). O objetivo central da abordagem é melhorar a capacidade de extensão e configuração de frameworks orientados a objetos (OO) para diferentes cenários de reutilização, através de uma melhor gerência de suas características. A abordagem é composta por: (i) um conjunto de diretrizes para o projeto e implementação de frameworks usando programação orientada a aspectos; e (ii) um modelo generativo usado para a instanciação automática do framework e suas variabilidades OO e OA. As diretrizes propõem a definição de um conjunto de pontos de junção de extensão (EJPs – *extension join points*) no código do framework, os quais podem ser usados para estender a funcionalidade básica do framework através da implementação de aspectos de extensão. Tais aspectos são responsáveis pela implementação de características transversais opcionais, alternativas ou de integração demandadas por usuários do framework. A abordagem é demonstrada com a implementação / refatoração de 3 frameworks OO pertencentes a diferentes domínios de aplicação. Uma avaliação da abordagem por meio de um estudo qualitativo e um estudo quantitativo é também apresentada. Finalmente, diversas lições aprendidas e discussões resultantes da experiência de uso da abordagem são descritas.

Palavras-chave

Frameworks Orientados a Objetos; Desenvolvimento de Software Orientado a Aspectos; Arquiteturas de Família de Sistemas; Desenvolvimento Generativo; Projeto de Software.

Abstract

Kulesza, Uirá. **An Aspect-Oriented Approach to Framework Development.** Rio de Janeiro, 2007. 205p. PhD Thesis - Computer Science Department, Pontifical Catholical University of Rio de Janeiro.

This work proposes a systematic approach to framework development which relies on the use of aspect-oriented (AO) techniques. The main goal of the approach is to improve the extensibility and configurability of object-oriented (OO) frameworks. It is composed of: (i) a set of guidelines to design and implement frameworks using aspect-oriented programming; and (ii) a generative model which allows the automatic instantiation of the framework and its respective OO and AO variabilities. Our guidelines propose the definition of extension join points (EJPs) in the framework code, which can be used to extend the framework basic functionality by means of extension aspects. The extension aspects are responsible for implementing optional, alternative and integration crosscutting features required by the framework users. Since such aspects can be automatically unplugged from the framework code, our approach makes it easier to customize the framework to specific needs. Three cases studies are presented to illustrate the applicability of our approach to the development of frameworks from different domains. The approach is also evaluated through both a qualitative and a quantitative study. Finally, several lessons learned and discussions resulting from the use of the approach are described.

Keywords

Object-Oriented Frameworks; Aspect-Oriented Software Development; System Family Architectures; Generative Development; Software Design.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Siglas e Abreviaturas

ADL	*Architecture Description Language*
AWT	*Abstract Window Toolkit*
DAO	*Data Access Object*
DG	Desenvolvimento Generativo
DSL	*Domain Specific Language*
DSOA	Desenvolvimento de Software Orientado a Aspectos
EJP	*Extension Join Point* ou Ponto de Junção de Extensão
EMF	*Eclipse Modeling Framework*
FOP	*Feature Oriented Programming*
GUI	*Graphical User Interface*
JET	*Java Emitter Template*
J2ME	*Java 2 Micro Edition*
LPS	Linha de Produto de Software
POA	Programação Orientada a Aspecto
OA	Orientado a Aspecto
OO	Orientado a Objeto
SoI	Separação de Interesses
UML	*Unified Modeling Language*
XPI	*Crosscutting Interface* ou Interface Transversal

1
Introdução

Métodos e técnicas para o desenvolvimento de famílias de sistemas e linhas de produto [27, 33, 100, 117] têm demonstrado ao longo dos últimos anos ser uma abordagem promissora para tratar de problemas centrais da engenharia de software, tais como, aumento da produtividade e melhoria na qualidade do software produzido. Uma família de sistemas pode ser caracterizada como um conjunto de programas que possuem propriedades comuns e propriedades específicas que variam de acordo com o membro da família sendo considerado [98]. Uma linha de produto é definido como um conjunto de sistemas de software que compartilham um conjunto de funcionalidades comuns e que satisfazem necessidades específicas de um determinado segmento de mercado [27].

Abordagens de desenvolvimento de famílias de sistemas ou linhas de produto [27, 33, 100, 117] definem como aspecto central o projeto e implementação de uma arquitetura comum para todas as aplicações dentro de um escopo específico. Tal arquitetura contempla as funcionalidades comuns e varíaveis compartilhadas por aplicações pertencentes ao escopo/domínio definido pela família de sistemas. Ela define um conjunto de componentes que são necessários para construir cada um dos sistemas (ou produtos) presentes em uma família de sistemas (ou linha de produto).

Frameworks Orientados a Objetos (OO) [37, 38, 64] representam atualmente uma tecnologia comum e relevante para a implementação de arquiteturas de família de sistemas ou linhas de produto. Eles permitem o reuso em larga escala e modular através: (i) do encapsulamento de interesses recorrentes de um dado domínio; e (ii) da disponibilização de diversas variabilidades e opções de configuração para as aplicações alvos. No desenvolvimento baseado em frameworks, aplicações são implementadas através do reuso da implementação da arquitetura definida pelo framework e da extensão de seus respectivos pontos de extensão/variação (*hot-spots*) [37]. Conseqüentemente a adoção da tecnologia de

framework traz, em geral, produtividade e qualidade para o desenvolvimento de novas aplicações.

1.1. Problema

Apesar dos benefícios trazidos pela tecnologia de frameworks OO, diversos pesquisadores e desenvolvedores têm identificado e relatado recentemente [5, 7, 14, 73, 80, 88, 89, 94] a inadequação de mecanismos OO para lidar com a modularização e composição de muitas características[1] (*features*) transversais encontradas em frameworks e arquiteturas de famílias de sistemas, trazendo dificuldades para a definição de variabilidades[2] no framework e integração com outros artefatos existentes. Exemplos de tais funcionalidades são: características opcionais, características alternativas transversais e características de integração transversal com outros artefatos (frameworks, componentes ou bibliotecas). Tal limitação na modularização de determinadas funcionalidades e propriedades de um framework OO traz como conseqüência: (i) o aumento da complexidade do framework com a inclusão de várias funcionalidades que não são úteis a muitas de suas instâncias; e (ii) dificuldade para configuração e customização do framework para atender diferentes cenários de reuso.

Trabalhos recentes [5, 7, 8, 73, 89, 94, 118] têm explorado o uso de técnicas de desenvolvimento de software orientadas a aspectos para facilitar a implementação de arquiteturas de famílias de sistemas flexíveis e customizáveis. Desenvolvimento de Software Orientado a Aspectos (DSOA) [42, 68] foi proposta como uma abordagem de engenharia de software que oferece mecanismos avançados para modularizar os chamados interesses tranversais. Interesses tranversais são funcionalidades ou propriedades do sistema que usualmente entrecortam diversos módulos em um sistema de software. Acredita-se que o isolamento dos interesses transversais possa trazer melhorias para o reuso e evolução de sistemas. DSOA pode ser vista como uma técnica complementar à

[1] Nesta tese de doutorado, uma característica é definida como sendo uma funcionalidade ou propriedade de uma família de sistemas ou linha de produto.

[2] Variabilidade é uma característica de uma família de sistemas ou linha de produto que varia de acordo com o membro da família ou produto sendo instanciado. Esse trabalho se concentra, principalmente, em variabilidades internas que são aquelas diretamente relacionadas com o projeto e implementação da arquitetura da família de sistemas ou linha de produto.

aquelas propostas por OO, a qual permite a modularização dos interesses tranversais encontrados em sistemas OO.

Nesse contexto, este trabalho explora problemas existentes na modularização de determinadas características em frameworks OO e realiza uma análise sistemática de como técnicas orientadas a aspectos (OA) podem ajudar na sua resolução.

1.2. Limitações das Abordagens Atuais

Diversas abordagens baseadas em técnicas de modularização avançadas têm sido propostas ao longo dos últimos anos. Muitas delas se fundamentam em técnicas orientadas a aspectos. Zhang & Jacobsen [118] propõem um conjunto de princípios e refatorações para a modularização de arquiteturas OA flexíveis e configuráveis. Apel & Batory [7] demonstram como programação orientada a características [15, 111] pode ser usada em conjunção com orientação a aspectos para oferecer um melhor suporte a modularização de características em linhas de produto. Embora várias das abordagens propostas apresentem diretrizes gerais para o uso de aspectos na modularização de arquiteturas mais flexíveis e configuráveis, nenhuma delas investiga sistematicamente como a programação orientada a aspectos pode ser usada de forma complementar a técnica de frameworks OO auxiliando na modularização de suas características.

Também poucas das abordagens propostas discutem ou investigam mecanismos automáticos para a instanciação de arquiteturas de famílias de sistemas ou linha de produto considerando o uso da abstração de aspectos. A investigação de como novas abordagens de desenvolvimento de famílias de sistemas (exemplos: Desenvolvimento Generativo [32, 33] e Fábricas de Software [53, 54]) podem ser combinadas com técnicas OA para habilitar a instanciação de frameworks e suas variabilidades OO e OA também não tem sido tratada. Uma das poucas exceções é o trabalho de Lougran & Rashid [89]. Eles propõem a abordagem *Framed Aspects* que combina *frames* e aspectos para a implementação de arquiteturas flexíveis e usa os mecanismos de parametrização e geração de código oferecido pela técnica de *frames* para habilitar a instanciação automática das variabilidades presentes na arquitetura. *Frame* [11] é uma tecnologia que

permite particionar o código-fonte de um sistema ou componente em uma hierarquia de unidades menores, os chamados *frames*. Cada *frame* especifica a funcionalidade comum e variável de um dado elemento de implementação.

1.3. Solução Proposta

Esse trabalho propõe uma abordagem sistemática para o desenvolvimento de frameworks OO usando técnicas de DSOA [42, 68, 77]. O objetivo central da abordagem é melhorar a capacidade de extensão e configuração de frameworks OO para diferentes cenários, através de uma melhor gerência de suas características. A abordagem é composta por: (i) um conjunto de diretrizes para o projeto e implementação de frameworks usando programação orientada a aspectos; e (ii) um modelo generativo usado para a instanciação automática do framework e suas variabilidades, incluindo aquelas implementadas usando aspectos.

As diretrizes para o projeto e implementação de frameworks usando aspectos, são centradas no conceito de Pontos de Junção de Extensão (EJPs, do inglês, *Extension Join Points*) [73, 75, 82, 84] como uma forma unificada de projetar e documentar pontos de extensão transversais de frameworks. Os EJPs oferecem novas formas de extensão do núcleo do framework, através da exposição de pontos específicos da sua execução (eventos gerados ou estados atingidos), os quais podem ter sua funcionalidade estendida por meio da codificação de aspectos. A adição de novas funcionalidades no framework é feita através do uso de aspectos de variabilidade e de integração. Considerando que os aspectos podem ser facilmente adicionados e removidos do núcleo do framework, nossa abordagem facilita dessa forma a customização do framework para necessidades específicas. Os EJPs podem também ser vistos como a especialização do conceito de interfaces transversais (XPIs) para auxiliar a modularização e composição de frameworks OO. As XPIs são usadas para especificar interfaces entre o código base[3] e um conjunto de aspectos.

[3] Em DSOA, o termo código base é usado para se referir às classes do sistema que são potencialmente afetadas pelos aspectos.

Nosso trabalho também se fundamenta em técnicas de Desenvolvimento Generativo (DG) [32, 33]. DG envolve o estudo de métodos e ferramentas que habilitam a produção automática de membros de uma família de software a partir de especificações de alto nível. Nossa abordagem contempla a definição de um modelo generativo [32, 33] o qual permite instanciar automaticamente arquiteturas orientadas a aspectos, permitindo a geração e customização de variabilidades OO e OA. O modelo generativo pode ser adotado na derivação de instâncias de frameworks OA projetados e implementados usando nossas diretrizes. Ele é centrado na definição de: (i) um modelo de característica – usado para especificar e coletar informações necessárias para a customização de variabilidades OO e OA, existentes em classes e aspectos; (ii) um modelo de arquitetura – que agrega os elementos de implementação de uma arquitetura de família de sistemas, na forma de um conjunto de componentes; e (iii) um modelo de configuração – que permite definir o mapeamento entre características e elementos de implementação, presentes nos modelos de características e arquitetura, respectivamente. Nosso modelo generativo trata explicitamente da instanciação de variabilidades OA por meio da customização de pontos de corte de aspectos usando relações tranversais entre características e mapeamento de características `<<joinpoint>>` em pontos de corte definidos nos EJPs.

1.4. Organização do Texto

Essa tese de doutorado está organizada em mais 8 capítulos, além deste introdutório.

O Capítulo 2 apresenta as três principais abordagens de desenvolvimento de famílias de sistemas investigadas nesse trabalho: (i) frameworks orientados a objetos, (ii) desenvolvimento de software orientado a aspectos; e (iii) desenvolvimento generativo. O capítulo apresenta as deficiências trazidas por mecanismos OO para implementação de características transversais existentes em frameworks, e mostra o potencial que DSOA e DG trazem para a resolução de muitos desses problemas.

O Capítulo 3 traz uma visão geral da abordagem orientada a aspectos de desenvolvimento de frameworks proposta neste trabalho. Ela é composta de: (i)

um conjunto de diretrizes para implementação de frameworks usando aspectos; e (ii) um modelo generativo que permite a instanciação automática de arquiteturas OA.

O Capítulo 4 detalha as diretrizes para implementação de características transversais existentes em frameworks usando técnicas orientadas a aspectos. É apresentado como os principais elementos da abordagem, tais como pontos de junção de extensão (EJPs) e aspectos de variabilidade e integração podem ser implementados em AspectJ.

O Capítulo 5 apresenta um modelo generativo a aspectos, cujo objetivo central é habilitar a instanciação automática de arquiteturas de linha de produto que contenham variabilidades OO e OA, a partir de informações coletadas por um modelo de característica.

O Capítulo 6 descreve três estudos de caso de frameworks e arquiteturas de linha de produto que foram implementados em AspectJ usando a abordagem proposta. O modelo generativo OA de cada um dos estudos de caso é também apresentado.

O Capítulo 7 apresenta uma análise geral da abordagem proposta a partir dos estudos de caso realizados. São apresentados os benefícios gerais trazidos pela abordagem, assim como um estudo qualitativo e quantitativo da aplicação da abordagem em um dos estudos de caso. Diversas discussões e lições aprendidas a partir da experiência de uso da abordagem são também apresentadas.

O Capítulo 8 discorre sobre trabalhos relacionados com a proposta apresentada nesta tese. Finalmente, o Capítulo 9 conclui esta tese sumarizando a sua lista de contribuições e sugerindo direções para a realização de trabalhos futuros.

2
Abordagens de Desenvolvimento de Família de Sistemas

Este capítulo apresenta uma visão geral das três diferentes abordagens de desenvolvimento de família de sistemas que são exploradas nesse trabalho. A tecnologia de Frameworks Orientados a Objetos (OO), uma das mais populares para a implementação de arquiteturas de famílias de sistemas, é inicialmente apresentada (Seção 2.1). Problemas de modularização de determinados tipos de características encontrados durante o desenvolvimento de frameworks OO são também descritos. Em seguida, a abordagem de Desenvolvimento de Software Orientado a Aspectos (DSOA), cujo objetivo é a modularização de interesses e características transversais, é apresentada (Seção 2.2). Finalmente, a abordagem de Desenvolvimento Generativo (DG) é descrita (Seção 2.3). DG endereça o estudo de métodos e ferramentas que habilitam a produção automática de membros de uma família de software a partir de especificações de alto nível. O capítulo é concluído com uma análise do potencial de integração entre as abordagens (Seção 2.4).

2.1.
Frameworks Orientados a Objetos

Frameworks Orientados a Objetos (OO) [37, 64] representam uma técnica comum e pragmática para a implementação de arquiteturas de família de sistemas e linhas de produto. Cada framework OO define uma arquitetura flexível na forma de código-fonte em uma linguagem de programação, que pode ser customizada para a implementação de diferentes aplicações para um mesmo domínio. Um framework OO é composto por um conjunto de classes que juntas colaboram para implementar um projeto abstrato para um domínio específico. Uma parte do comportamento definido nas classes do framework é fixo, enquanto uma outra parte do comportamento de suas classes é extensível, sendo passível dessa forma de ser customizado de forma distinta para cada aplicação. Frameworks OO são, em geral, implementados usando linguagens e mecanismos de programação OO.

Os pontos de extensão (*hot-spots*) do framework são especificados através da definição de classes abstratas ou interfaces. Diversos padrões de projeto [45] podem ser usados na implementação de tais pontos de extensão. O desenvolvimento de aplicações baseada em frameworks requer a criação de classes que estendem seus pontos de extensão, em um processo que é denominado instanciação.

Um framework OO mantém as seguintes propriedades: (i) define um conjunto de classes (núcleo do framework) que juntas colaboram para implementar uma arquitetura de família de sistemas; (ii) oferece um conjunto de pontos de extensão na forma de classes abstratas ou interfaces; e (iii) especifica o fluxo de controle principal da aplicação através do comportamento das classes que representam o seu núcleo e que são responsáveis pela invocação das classes que estendem seus pontos de extensão.

A tecnologia de framework OO traz em geral um impacto positivo na produtividade e qualidade de desenvolvimento de aplicações. Frameworks OO possibilitam a reutilização não apenas de código de implementação, mas também reúso de projeto de soluções arquiteturais para um dado domínio. Frameworks OO maduros contribuem para a melhoria da qualidade das aplicações finais gerada, por oferecerem uma implementação estável e bem testada, além de possibilitar a codificação de menos linhas de código.

2.1.1. Problemas de Modularização

Apesar dos benefícios da tecnologia de frameworks na implementação de famílias de sistemas, diversos pesquisadores têm demonstrado a incapacidade dos mecanismos OO em modularizar determinados tipos de características encontrados em frameworks, tais como, características opcionais [14, 73, 118] e de composição tranversal [78, 94]. Basicamente, tem se concluído que o uso de técnicas OO na modularização de várias dessas características pode contribuir para (i) o aumento da complexidade do framework com a inclusão de várias funcionalidades que não são usadas por todas as instâncias do mesmo (esse fenômeno é denominado na literatura de "*overfeatured*" [28]); assim como traz (ii) dificuldades ou até imposibilita o reúso do framework em diferentes contextos

e cenários. As seções seguintes apresentam os principais problemas de modularização identificados pela comunidade e descrevem sintomas de tais problemas no contexto do framework JUnit.

2.1.1.1. Framework JUnit: Um Exemplo

O propósito principal do framework JUnit é possibilitar a implementação e execução de casos e suítes de teste para aplicações Java. Ele é usado principalmente para implementar testes de unidade, mas pode também ser usado para implementar testes de integração entre módulos. A Figura 1 apresenta as classes principais do framework JUnit[4]. Os seguintes componentes principais definem a estrutura do JUnit:

(i) **Framework**: esse componente agrega as classes responsáveis por especificar o comportamento básico de execução de casos e suítes de teste. As principais classes desse módulo são `TestCase` e `TestSuite`. Elas representam os pontos de extensão principais do framework. Usuários do framework estendem essas classes para criar e especificar casos e suítes específicos para o teste de suas aplicações;

(ii) **Runner**: as classes desse componente se responsabilizam pela disponibilização de uma interface gráfica que permita a inicialização e monitoramento da execução de casos e suítes de teste. São oferecidas três implementações alternativas para essa interface: (1) `textui` – baseada em linha de comando; (2) `awtui` – baseado na biblioteca AWT; e (3) `swingui` – baseado na biblioteca Java Swing;

(iii) **Extensions**: define classes que estendem o comportamento básico das classes de teste do JUnit. Exemplos de extensões disponíveis são: execução concorrente de testes em threads distintas; execução repetida de determinados casos e/ou suítes de teste; e configurações *default* durante a inicialização ou finalização de casos e/ou suítes de teste específicos.

As seções seguintes apresentam exemplos no contexto do JUnit dos principais problemas de modularização existentes em frameworks. Embora o framework JUnit tenha uma arquitetura bem organizada e estruturada, a

[4] JUnit Framework. URL: http://www.junit.org. 2007.

identificação desses problemas de modularização refletem dificuldades que poderiam ser encontradas durante a extensão ou evolução do framework. No contexto de frameworks maiores e mais complexos, esses problemas poderiam causar um desgaste gradual de sua arquitetura.

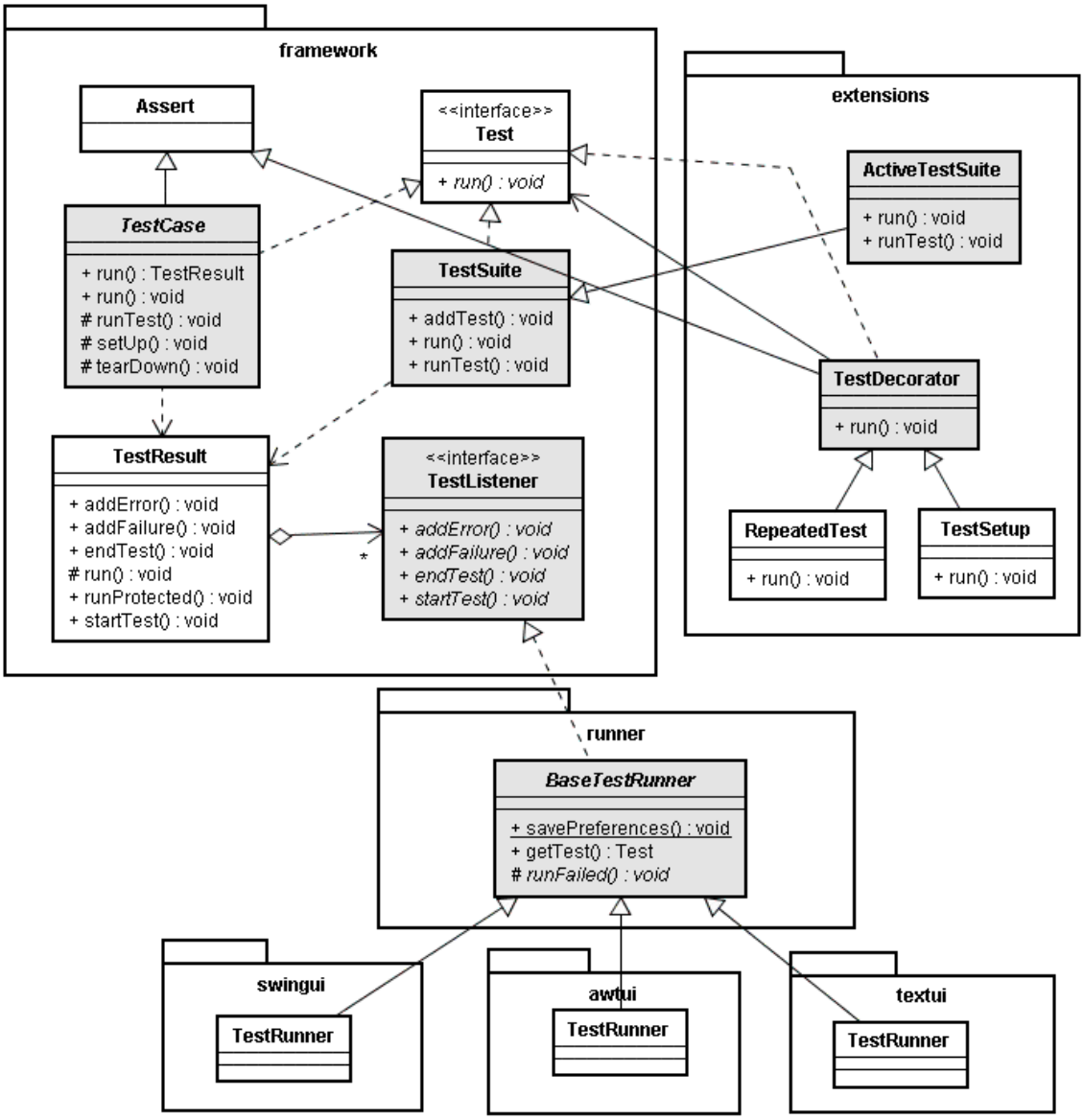

Figura 1. Diagrama de Classes do framework JUnit

2.1.1.2. Complexidade de Colaboração entre Objetos

Um framework OO define um conjunto de classes reusáveis abstratas e concretas que implementam uma arquitetura para uma família de sistemas. Colaborações complexas entre essas classes devem ser implementadas. Essas

colaborações representam funcionalidades comuns compartilhadas por diversas aplicações no domínio do framework. Cada classe do framework, em geral, tem de desempenhar diversos papéis [104], o que significa que elas precisam colaborar com diferentes classes para implementar suas respectivas responsabilidades. Conseqüentemente, o entendimento e manutenção das classes do framework pode se tornar uma tarefa difícil. Riehle et al [104] analisam os problemas de colaborações complexas entre objetos, e seu impacto no projeto e implementação de frameworks. Eles mostram como a complexidade de um framework OO aumenta quando suas classes desempenham diversos papéis.

No framework JUnit, por exemplo, o propósito principal da classe `TestResult` é coletar os resultados da execução de casos de teste. Para endereçar os requisitos do framework, entretanto, essa classe deve assumir um diferente papel adicional. A característica de monitoramento da execução dos testes está sobreposta no código das classes de execução dos testes para possibilitar a notificação para as classes da interface gráfica sobre o estado de execução (execução iniciada, correta, com falha, finalizada) dos testes. Tal característica é implementada através da declaração de um conjunto de objetos do tipo `TestListener` dentro da classe `TestResult`. A interface `TestListener` define os métodos necessários para implementar tal protocolo de notificação. Ela é implementada por classes do componente Runner, para apresentar os resultados do monitoramento da execução dos testes. A implementação dessa característica pode também ser vista como uma instanciação do padrão de projeto *Observer* [45]. A Figura 2 ilustra o espalhamento e entrelaçamento da implementação dessa característica por entre diferentes classes do framework JUnit.

De forma geral, a medida que novas características vão sendo incluídas nas classes pertecentes ao framework, o entendimento e evolução de sua estrutura interna pode se tornar uma atividade bastante complexa. A sobreposição de determinados padrões de projeto na estrutura de classes do framework pode ocasionar o surgimento de código entrelaçado e espalhado, tal como no exemplo ilustrado para o JUnit. Também a inclusão de novos papéis de classes para endereçar características opcionais ou de integração do framework com outros componentes, a serem exploradas nas seções seguintes, também contribui ainda mais para o aumento da complexidade do núcleo do framework.

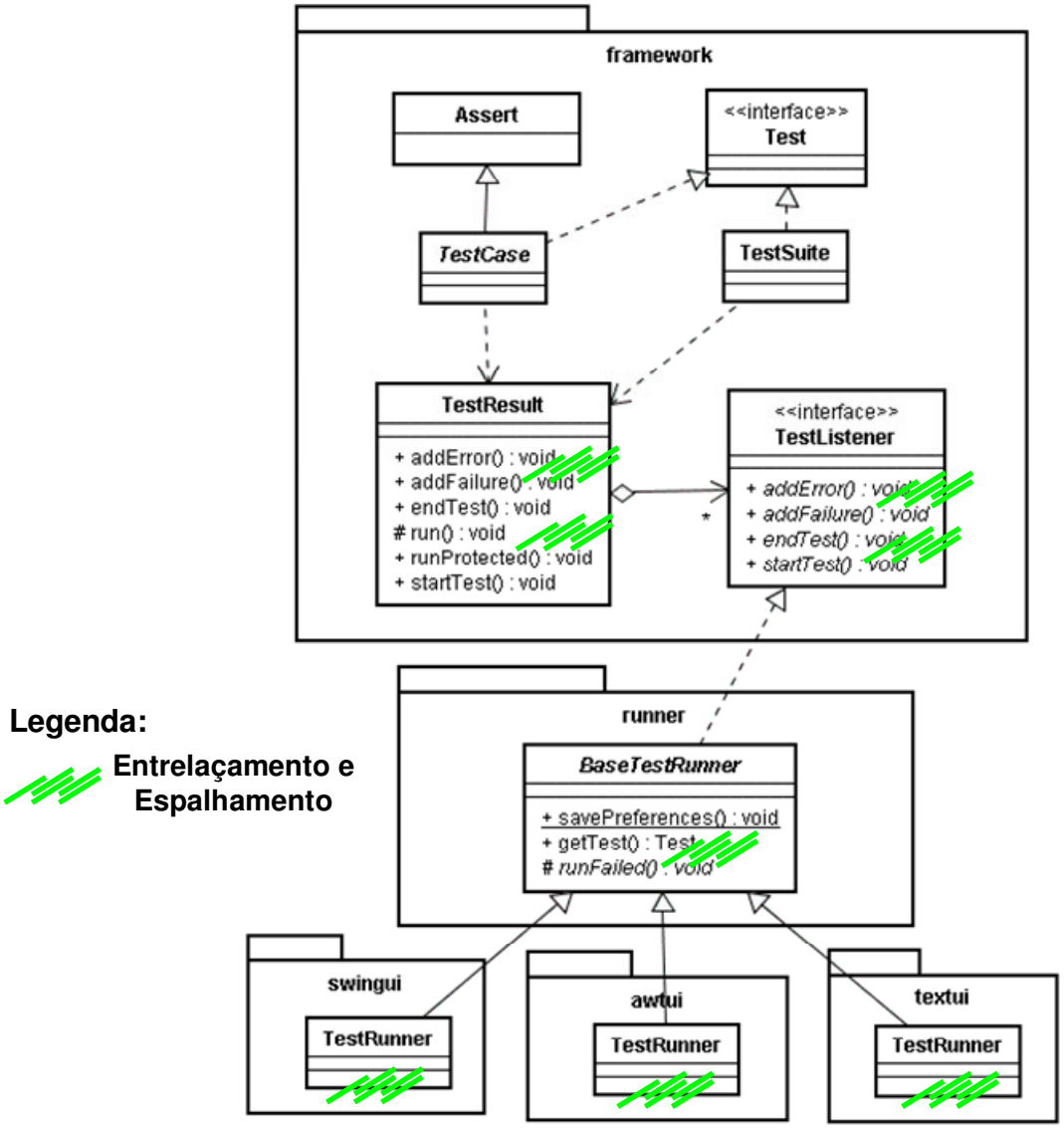

Figura 2. Implementação do monitoramento de testes no JUnit

2.1.1.3. Dificuldade de Modularização de Características Opcionais

Batory et al [14] discutem a dificuldade da técnica de framework em modularizar características opcionais. Uma característica opcional pode ser vista como uma funcionalidade do framework que não é usada em todas as suas instâncias. Esses pesquisadores ilustram [14] alternativas que desenvolvedores usualmente adotam para lidar com tal problema, tais como: (i) implementar a característica opcional no código de classes que implementam os pontos de extensão do framework durante a sua instanciação; e (ii) criar dois diferentes

frameworks, um contendo a característica opcional e outro sem ele, o que conseqüentemente traz problemas para lidar com a evolução dos dois artefatos.

Uma outra prática bastante adotada na implementação de características opcionais em frameworks é a especificação de relações de herança para definir comportamento adicional nas classes já existentes do framework. Essas classes acabam por representar uma característica do framework que foi estendida. No framework JUnit, por exemplo, a implementação da propriedade de execução concorrente de suítes de teste é realizada através do uso de relação de herança. A Figura 3 mostra a classe `ActiveTestSuite` herdando da classe `TestSuite` de forma a possibilitar a execução de suítes de teste em threads distintas.

Vários padrões de projeto podem também ser usados para prover implementações de características opcionais. Eles ilustram o uso sistemático de mecanismos de composição OO (herança e agregação), oferecendo flexibilidade para inclusão de novas implementações para um dado interesse do sistema. Exemplos de tais padrões são [45]: *Decorator*, *Observer* e *Adapter*. No framework JUnit, o padrão de projeto *Decorator* é usado para possibilitar a inclusão de novas propriedades de extensão de casos ou suítes de teste. A Figura 3 ilustra o uso do *Decorator* para implementação de tais propriedades. A classe `TestDecorator` representa o participante *Decorator* do padrão. As propriedades de extensão são definidas como especializações de tal classe, através da especificação de decoradores concretos (`ConcreteDecorator`). Exemplos de tais propriedades no JUnit são: (i) configurações de comandos de inicialização ou finalização de casos ou suítes de teste, implementado pela classe `TestSetup`; e (ii) repetições de testes implementado pela classe `RepeatedTest`. A associação de tais propriedades a casos ou suítes de teste específicos requer a instanciação de tais classes, passando como parâmetro em seus respectivos construtores, instâncias dos casos ou suítes de teste a serem estendidos.

O uso de padrões de projeto e relações de herança para implementar características opcionais pode trazer dificuldades para o entendimento e posterior manutenção do projeto de frameworks. Em particular, as seguintes dificuldades podem ser enfrentadas: (i) a inclusão de características opcionais através de relações de herança, implica no entendimento de novas classes que representam novos pontos de extensão do framework (tais como, as classes `TestDecorator`, `TestSetup`, `RepeatedTest` e `ActiveTestSuite` no caso do JUnit,); (ii) a

modularização de muitos padrões de projeto usando os mecanismos de OO incorre em problemas [51, 58] de entrelaçamento com outras funcionalidades e/ou espalhamento por entre várias classes, e isso como conseqüência também traz dificuldades para gerenciar as variabilidades implementadas no framework; e (iii) as formas de combinação e composição de várias dessas características opcionais precisa ser explicitamente codificada no código de instâncias do framework (classes `TestScenarioComposition1` e `TestScenarioComposition2` da Figura 3) e pode sofrer limitações em função dos mecanismos e soluções de projeto OO usados.

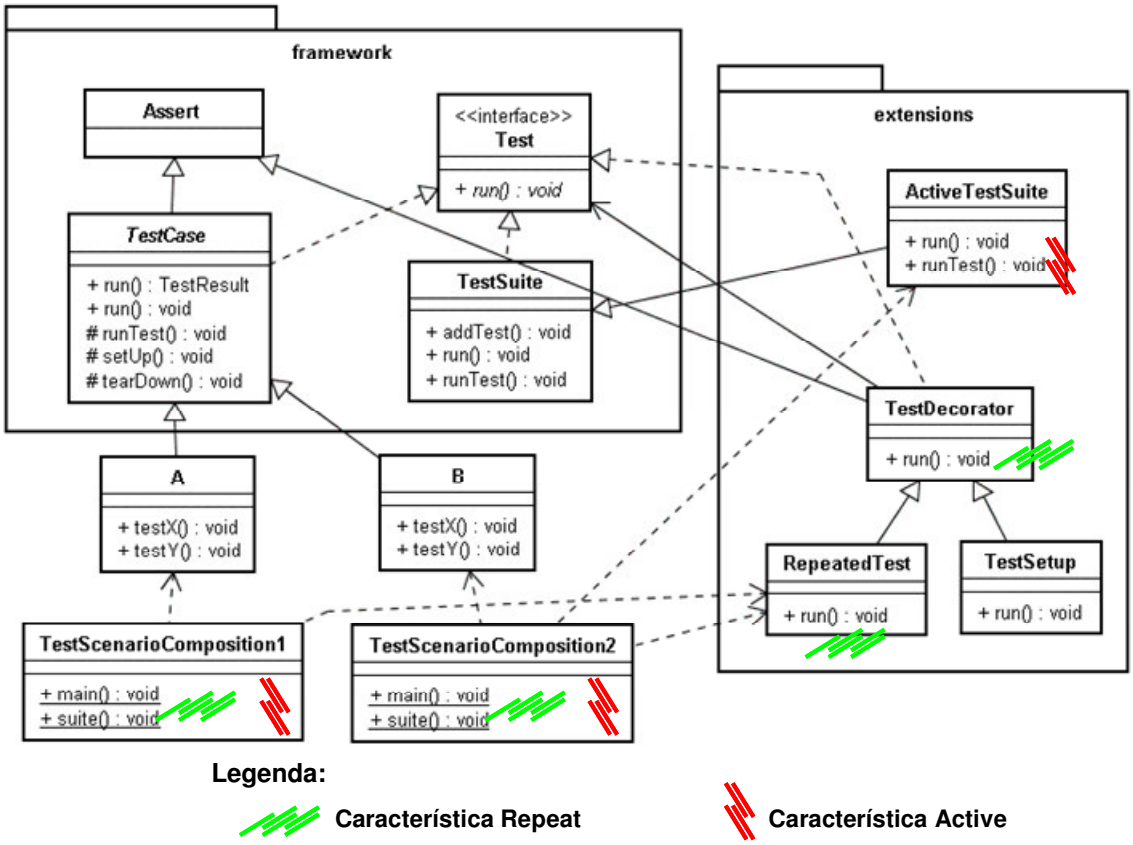

Figura 3. Implementação de propriedades de extensão no JUnit

A combinação e composição de características de extensão de casos e suítes de testes no caso do framework JUnit é explicitada diretamente no código de classes das suas instâncias. A Figura 4 mostra o código de 2 cenários possíveis de combinação das características de execução concorrente (`ActiveTestSuite`) e

execução repetitiva (`RepeatedTest`), aplicados a duas subclasses[5] (`A` e `B`) da classe `TestCase` do JUnit. Na classe `TestScenarioComposition`, os dois casos de testes são adicionados a um `ActiveTestSuite`, e em seguida, essa suíte de teste é executada 5 vezes. Na classe `TestScenarioComposition2`, os dois casos de teste são inicialmente caracterizados como repetitivos, e cada um dos casos de teste (total=2) é executado numa diferente *thread*. A diferença central entre esses os dois cenários implementados por essas classes é que: (i) o primeiro demanda a criação de 10 *threads*, pois o `ActiveTestSuite` é repetido 5 vezes, e cria uma thread para cada um dos 2 métodos de teste das classes A e B; e (ii) o segundo cenário ocasiona a criação de apenas 2 *threads*, um para cada `RepeatedTest`. Diferentes cenários de composição das características existentes do JUnit demandam a criação de diferentes classes na instância sendo criada pelo framework.


```
01 public class TestScenarioComposition {
02     public static void main(String[] args) {
03       junit.textui.TestRunner.run(AllTestsScenario2.suite());
04     }
05     public static Test suite() {
06        ActiveTestSuite suite= new ActiveTestSuite();
07
08        suite.addTestSuite(A.class);
09        suite.addTestSuite(B.class);
10
11        // Execute each suite 5 times
12        Test test= new RepeatedTest(suite, 5);
13        return test;
14     }
15 }
```

```
01 public class TestScenarioComposition2 {
02   public static void main(String[] args) {
03     junit.textui.TestRunner.run(AllTestsScenario4.suite());
04   }
05   public static Test suite() {
06     // Execute each suite 5 times
07     RepeatedTest test= new RepeatedTest(new TestSuite(A.class), 5);
08     RepeatedTest test2=new RepeatedTest(new TestSuite(B.class), 5);
09      ActiveTestSuite suite= new ActiveTestSuite();
10
11     suite.addTest(test);
12     suite.addTest(test2);
13     return suite;
14   }
15}
```

Figura 4. Composição de propriedades de extensão no JUnit

[5] Cada uma dessas subclasses definem 2 métodos de teste.

O problema de combinação e composição de várias características tem sido também abordado por vários outros trabalhos de pesquisa [20, 59, 118]. Zhang & Jacobsen [118] mostram, por exemplo, que no contexto do desenvolvimento de middlewares para sistemas distribuídos, várias características precisam ser compostas e coordenadas de diferentes formas e que o uso de mecanismos OO para a sua modularização remete ao problema de convolução de implementação [118]. Tal problema diz respeito à dificuldade de modularização de determinadas características e ao seu natural entrelaçamento com outras características existentes no middleware. Conseqüências diretas da perda de modularidade de tais características são: (i) perda de configurabilidade devido à impossibilidade de plugar/desplugar a característica do núcleo do middleware; e (ii) penalidades no desempenho do sistema devido à execução de código desnecessário quando a característica não deve estar presente.

2.1.1.4. Composições Transversais na Integração de Frameworks

Mattsson et al [90, 91] analisam os problemas e causas relacionados com a composição de frameworks. Para cada problema apresentado, eles propõem um conjunto de soluções OO. Uma composição de dois frameworks, como descrita pelos autores, pode ser vista como uma composição de um novo conjunto de características (representado por um framework) na estrutura de um outro. A seguir é apresentado um exemplo de composição entre frameworks.

Considere, por exemplo, um framework de medições e análise de qualidade de produtos para sistemas de manufatura. O framework *Measurement* apresentado em [17] endereça tal propósito. A Figura 5 apresenta as classes principais desse framework. Itens de um dado produto são analisados pelo sistema de medição (uma instância do framework *Measurement*), através de um ciclo composto pelos seguintes passos: (i) a classe `Trigger` indica que um item de produto será analisado pelo sistema de medição; (ii) sensores coletam informações (classes `PhysicalSensor`, `Sensor` e `UpdateStrategy`) sobre o produto. Um exemplo seria tirar fotografias do mesmo; (iii) em seguida, as informações coletadas do produto são comparadas com valores esperados (classe `MeasurementValue`) de forma a classificar o produto de acordo com critérios de qualidade; e (iv)

finalmente, ações são tomadas por acionadores baseadas na classificação do produto (classes `Actuator`, `PhysicalActuator` e `ActuationStrategy`). Um exemplo de possível ação seria emitir um rótulo no produto de acordo com sua categoria de qualidade.

Suponha agora que se deseja integrar o framework *Measurement* com um framework de interface gráfica (GUI) disponível, para endereçar a funcionalidade de visualização de informações do processo de medição, tais como, itens já processados por sensores e acionadores, tempo de processamento e rótulo atribuído a cada item de produto. A Figura 5 também apresenta um framework GUI baseado na biblioteca Java Swing, que permite apresentar dados em tabelas visuais. As classes abstratas `DataTableModel` e `ObjectTableModel` são usadas como fontes de dados das tabelas visuais. Três subclasses (`SensorTableModel`, `ActuatorTableModel` e `ProcessedItemTableModel`) da classe `ObjectTableModel` foram definidas para permitir a apresentação das informações do processo de medição. A composição entre os frameworks também requer, entretanto, que sejam feitas chamadas internas de determinadas classes (`Trigger`, `PhysicalSensor`, `PhysicalActuator` e `ActuationStrategy`) do framework *Measurement* para repassar informações para as subclasses de `ObjectTableModel`. Dessa forma, a composição entre os frameworks pode ser vista como transversal, uma vez que ela requer mudanças invasivas em várias classes e métodos do framework *Measurement* de forma a notificar classes do framework GUI sobre o andamento do processo de medição. A Figura 5 ilustra as modificações invasivas e transversais que devem ser feitas no framework Measurement para endereçar a composição.

Um estudo de análise [78] das soluções de composição de frameworks propostas por Mattsson et al [90, 91] foi conduzido por nosso grupo de pesquisa. O estudo concluiu que várias das soluções OO propostas pelos autores demandam modificações invasivas e trazem dificuldades para a implementação, entendimento e manutenção do código de composição do framework. Nossa análise envolveu a realização de um estudo de caso com a composição de características de quatro diferentes frameworks endereçando interesses de domínios verticais e horizontais [33]. A análise também mostrou que de nove soluções OO descritas por tais autores, seis delas possuem problemas de modularização e uma natureza transversal, e demandaram mudanças internas

invasivas no código do framework. Detalhes adicionais desse estudo serão apresentados no capítulo 7.

De forma geral, o estudo concluiu que o uso de mecanismos OO para implementar código de composição transversal entre frameworks traz dificuldades: (i) para entender e evoluir o código de composição dos frameworks que pode se entrelaçar e se espalhar por entre várias classes; e (ii) para desacoplar os dois frameworks e tratá-los como entidades individuais. O capítulo 7 apresenta o estudo de caso de composição do framework Measurement com outros diferentes frameworks de GUI, Persistência e Estatística, o qual foi usado para avaliar o uso de aspectos no projeto de diferentes composições transversais existentes entre frameworks.

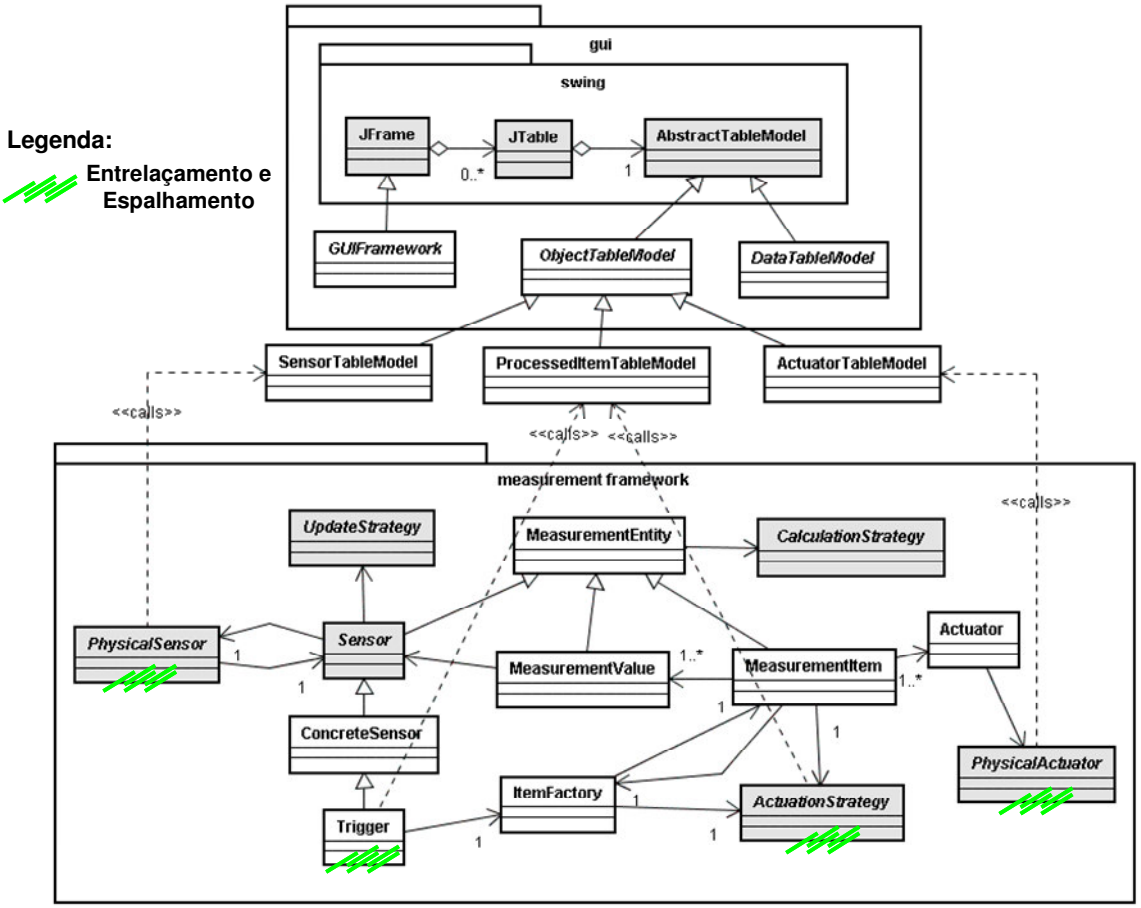

Figura 5. Exemplo de Composição entre Frameworks

2.2. Desenvolvimento de Software Orientado a Aspectos

Desenvolvimento de Software Orientado a Aspectos (DSOA) [42, 68] é uma abordagem de engenharia de software que objetiva a modularização dos chamados interesses transversais. Interesses traversais são aqueles que entrecortam diversos módulos dentro de um sistema de software. DSOA encoraja a descrição modular de sistemas de software complexo oferecendo mecanismos para separar claramente a funcionalidade básica do sistema, a qual pode ser endereçada por abordagens já propostas (tais como OO), dos interesses traversais. DSOA permite a modularização dos interesses tranversais propondo uma nova abstração, denominada aspecto, e novos mecanismos que permitem compor aspectos e abstracões dos paradigmas vigentes (exs: classes, interfaces, métodos, construtores). A composição entre aspectos e abstrações OO são realizadas dentro de pontos específicos, denominados pontos de junção. DSOA como a maioria dos paradigmas de desenvolvimento de software surgiu inicialmente no nível de implementação, sendo caracterizada pelo termo Programação Orientada a Aspectos (POA) [68].

Diversos trabalhos de pesquisa apresentados pela comunidade demonstram os benefícios que programação orientada a aspectos pode trazer para a modularização de propriedades sistêmicas ou requisitos não funcionais, tais como, rastreamento [31], auditoria [31], delimitação de transações [113], persistência [102, 113], segurança [85], tratamento de exceções [40], monitoramento e distribuição [113] . Trabalhos recentes [5, 7, 8, 73, 89, 94, 118] têm explorado o uso de técnicas orientadas a aspectos (OA) no projeto e implementação de arquiteturas de famílias de softwares, tais como, frameworks e linhas de produto de software. Nesses trabalhos, a abstração de aspectos é usada para melhorar a modularização de características transversais encontradas no domínio endereçado.

Esta tese propõe uma abordagem para o desenvolvimento de frameworks baseado em técnicas OA. São apresentadas diretrizes para a modularização de características transversais opcionais, alternativas e de integração existentes em frameworks usando a abstração de aspectos. As subseções seguintes apresentam: (i) uma visão geral da linguagem orientada a aspectos AspectJ utilizada nos

estudos de caso da tese; e (ii) o conceito de interface transversal, o qual está diretamente relacionado com a proposta de nossa abordagem.

2.2.1. AspectJ

AspectJ [69, 70] é uma extensão orientada a aspectos para a linguagem de programação Java. Ela é atualmente a abordagem mais amadurecida e popular de orientação a aspectos. AspectJ propõe uma nova abstração para modularização do software, denominada aspecto. Um aspecto em AspectJ é composto por pontos de corte (*pointcuts*), adendos (*advices*) e declarações inter-tipos (*inter-type declarations*). Os pontos de corte especificam os pontos de execução, denominados pontos de junção (*join points*) em AspectJ, de um programa Java que o aspecto deseja atuar/interceptar. Em AspectJ, diversos pontos de execução de classes Java são expostos para serem interceptados por aspectos, entre eles: execução de métodos e construtores, chamada de métodos ou construtores, execução de manipuladores de exceções, consulta ou modificação de valores de atributos, etc. Adendos definem o comportamento que será invocado durante a ocorrência daqueles pontos de junção. Três tipos de adendos podem ser definidos em AspectJ: (i) adendos `before` – são invocados sempre que um dado ponto de junção ocorre e antes do prosseguimento normal da sua execução; (ii) adendos `after` – são invocados depois da computação que define o ponto de junção; e (iii) adendos `around` – são executados sempre que um dado ponto de junção é alcançado, e têm o controle para decidir se a computação que caracteriza o ponto de junção deve ou não ser executada.

Um aspecto em AspectJ pode também definir declarações inter-tipos. Elas permitem modificar a estrutura de classes definidas em Java. Os seguintes tipos de declarações inter-tipos podem ser definidas: (i) introdução de atributos ou métodos em classes; (ii) introdução de relações de herança entre classes; e (iii) introdução de relações de implementação entre classes e interfaces.

A Figura 6 mostra o código de um aspecto exemplo, denominado `FaultHandler`. Esse aspecto é apresentado no guia de programação de AspectJ. O propósito desse aspecto é manipular eventuais faltas que ocorram em um servidor. O aspecto `FaultHandler` define: (i) a introdução do atributo `disabled` na classe

`Server` (linha 3); (ii) dois métodos internos do aspecto, o método privado `reportFault()` usado para indicação de faltas ocorridas (linhas 5-7) e o método público estático `fixServer()` que determina que o servidor voltou a operar normalmente (linhas 9-11); (iii) o ponto de corte `services()`, que intercepta todas as chamadas para métodos de escopo público da classe `Server` (linha 13); e (iv) dois adendos, associados ao ponto de corte `services()`, um do tipo `before` - que determina o lançamento de uma exceção quando o servidor não está operando normalmente (linhas 15-17), e outro do tipo `after throwing` - que modifica o estado do servidor para desabilitado sempre que uma exceção do tipo `FaultException` é lançada nos métodos públicos da classe `Server` (linhas 19-22).

```
01 public aspect FaultHandler {
02
03   private boolean Server.disabled = false;
04
05   private void reportFault() {
06     System.out.println("Failure! Please fix it.");
07   }
08
09   public static void fixServer(Server s) {
10     s.disabled = false;
11   }
12
13   pointcut services(Server s): target(s) && call(public * *(..));
14
15   before(Server s): services(s) {
16     if (s.disabled) throw new DisabledException();
17   }
18
19   after(Server s) throwing (FaultException e): services(s) {
20     s.disabled = true;
21     reportFault();
22   }
23 }
```

Figura 6. Aspecto FaultHandler

2.2.2. Interfaces Transversais

Duas propriedades, denominadas Inconsciência e Quantificação (*Obliviousness* e *Quantification*), têm sido identificadas como fundamentais para Programação Orientada a Aspectos por alguns pesquisadores [43]. A propriedade Quantificação diz respeito à capacidade dos programadores em escrever trechos de código com a seguinte forma: "*No programa P, quando quer que a condição C*

ocorra, execute a ação A". A linguagem AspectJ, por exemplo, oferece suporte para tal propriedade por meio das construções de ponto de corte, pontos de junção e adendos, citadas anteriormente. A propriedade de Inconsciência estabelece que desenvolvedores do código base - as classes do sistema que são potencialmente afetadas pelos aspectos - não precisam estar conscientes dos aspectos que as afetarão. Isso significa que os programadores não precisam preparar o código para ser afetado por aspectos. A seguinte sentença dos autores sintetiza ambas as propriedades [43]: "*POA pode ser entendida como o desejo de especificar comandos com o uso da quantificação sobre o comportamento de programas, e garantir que tais comandos são aplicados sobre programas escritos por programadores inconscientes da sua existência*".

Em um estudo recente, Sullivan et al [115] compara a metodologia baseada na propriedade de *Inconsciência* com uma nova abordagem para desenvolvimento OA baseado em *design rules*[6] [9]. Nessa abordagem, os autores propõem a especificação de interfaces entre o código base e os aspectos. Dessa forma, é necessária a especificação antecipada de pontos de junção do código base antes da sua própria codificação. Tais pontos de junção são posteriormente usados na codificação dos aspectos. Tal abordagem baseada em regras de projeto endereça o desacoplamento do código base e dos aspectos através de uma especificação explícita das interações e contratos entre tais elementos permitindo dessa forma o seu desenvolvimento paralelo. No estudo conduzido pelos autores [115] foram observados os benefícios que a abordagem traz para reduzir ou eliminar diversas desvantagens da abordagem *Inconsciente*, tais como: (i) a codificação de pontos de corte frágeis e complexos; e (ii) o forte acoplamento dos aspectos para detalhes de implementação do código base.

Griswold et al [56] mostram como as interfaces entre o código base e os aspectos, chamadas de interfaces transversais (XPIs - *crosscutting interfaces*) e propostas anteriomente pela abordagem baseada em *design rules*, podem ser implementadas em AspectJ. As XPIs são usadas para abstrair um comportamento existente no código base. A implementação das XPIs em AspectJ é composta de: (i) uma parte sintática – que permite expor pontos de junção específicos através da especificação de pontos de corte em AspectJ; e (ii) uma parte semântica – a qual

detalha o significado dos pontos de junção especificados e pode definir restrições (tais como, pré e pós-condições) que devem ser satisfeitas quando estendendo tais pontos de junção. A parte semântica pode ser parcialmente implementada com aspectos de *enforcement* [31] (implementados por meio das construções `declare error` e `declare warning` de AspectJ) ou definindo aspectos com contratos que garantem restrições específicas a serem satisfeitas antes e depois da execução de adendos.

Essa tese define o conceito de pontos de junção de extensão (EJPs), os quais são usados para estender o comportamento básico de um framework orientado a objeto. Os EJPs são inspirados na proposta de XPIs e podem ser vistos como o uso especializado das mesmas para auxiliar a modularização e composição de frameworks OO.

2.3. Desenvolvimento Generativo

Desenvolvimento Generativo (DG) [32, 33] endereça o estudo de métodos e ferramentas que habilitam a produção automática de membros de uma família de software a partir de especificações de alto nível. Ele promove a separação dos espaços de problema e solução dando flexibilidade de evoluir ambos de forma independente. Para prover tal separação, Czarnecki & Eisenecker [33] propõem o conceito de modelo de domínio generativo. Este modelo é composto de: (i) espaço de problema – que representa os conceitos e características existentes em um domínio específico; (ii) espaço de solução – que consiste na arquitetura de software e componentes usados para construir membros de uma família de sistemas; e (iii) conhecimento de configuração – que define como combinações específicas de características no espaço de problema são mapeadas para um conjunto de componentes de software no espaço de solução. Geradores de código representam o conhecimento de configuração dentro de um modelo generativo. A Figura 7 apresenta os elementos principais do modelo generativo.

DG foi proposto como uma abordagem baseada em engenharia de domínio [101]. Dessa forma, métodos de engenharia de domínio [33] podem ser usados

[6] *Design Rule* é definido como um parâmetro de projeto que pode ser usado como uma interface entre módulos com o objetivo de reduzir o acoplamento entre os mesmos.

para apoiar a definição de um modelo de domínio generativo. Atividades comuns encontradas em tais métodos são: (i) análise de domínio – a qual está interessada na definição de um domínio para uma familia de software específica e na identificação de características comuns e variáveis dentro desse domínio; (ii) projeto de domínio – a qual se concentra na definição de uma arquitetura e componentes comuns para este domínio; e (iii) implementação de domínio – que envolve a implementação de uma arquitetura e componentes anteriormente especificados no projeto de domínio. Duas novas atividades [33] precisam ser introduzidas em tais métodos de engenharia de domínio de forma a endereçar os objetivos de DG, são elas: (i) desenvolvimento de mecanismos apropriados para especificar membros da família de sistemas sendo desenvolvida. Linguagens específicas de domínio (DSLs - *Domain Specific Languages*) devem ser desenvolvidas para lidar com tal requisito; e (ii) modelagem detalhada do conhecimento de configuração de forma a automatizá-lo através do uso de geradores de código.

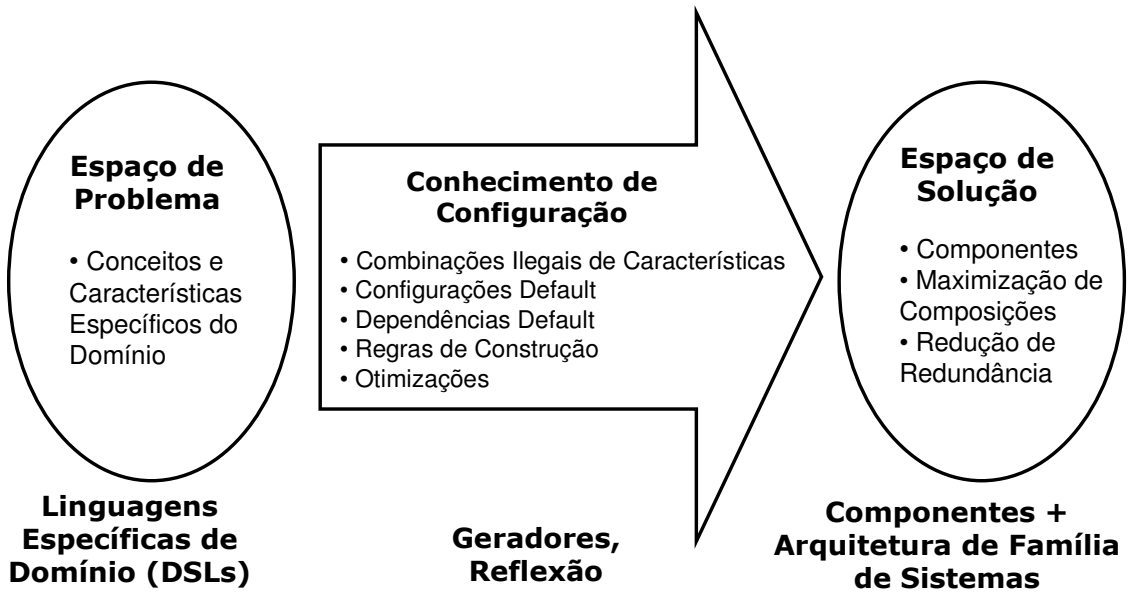

Figura 7. Modelo Generativo

De acordo com Czarnecki [32], o mapeamento entre o espaço de problema e o espaço de solução pode ser visto sobre duas perspectivas: (i) *visão de configuração* – nessa perspectiva o espaço de problema é visto como um conjunto de características a serem selecionadas e o espaço de solução consiste em um conjunto de componentes que podem ser combinados e customizados para derivar

uma instância de uma família de sistemas ou linha de produto. O mapeamento entre tais espaços é definido por regras de construção que indicam como certa combinação de características deve ser traduzida para determinadas configurações de componentes; e (ii) *visão transformacional* – na qual o espaço de problema é representado por uma linguagem específica de domínio (DSL) e o espaço de solução por uma linguagem de programação. O mapeamento entre tais espaço é definido através de transformações que geram código-fonte em uma linguagem de programação a partir de um programa especificado na DSL.

Essa tese propõe uma abordagem para desenvolvimento de frameworks usando programação orientada a aspectos. A abordagem define um modelo generativo orientado a aspectos cujo objetivo é facilitar a instanciação automática de variabilidades existentes em frameworks implementados com o uso da tecnologia de aspectos. O modelo generativo proposto contempla a visão de configuração. Também um conjunto de diretrizes de projeto e implementação de características transversais encontradas em frameworks compõe nossa abordagem. Essas diretrizes podem ser usadas para endereçar a modelagem e codificação de uma arquitetura de família de aplicações que determina o espaço de solução de nosso modelo generativo.

2.4. Potencial de Integração entre as Abordagens

Desenvolvimento Generativo e Desenvolvimento Orientado a Aspectos são abordagens com um grande potencial para serem integradas e serem usadas de forma complementar. Técnicas orientadas a aspectos podem ser usadas na definição de abordagens generativas, tanto: (i) no espaço de problema para, por exemplo, permitir a representação de relações tranversais entre características existentes no domínio endereçado; assim como (ii) no espaço de solução, para permitir a codificação de características transversais que não podem ser bem modularizadas por mecanismos OO. Esse uso integrado pode trazer benefícios diretos [74, 79, 80], tais como: (i) separação clara entre características não transversais e transversais nos espaços de problema e solução, e ao longo de todo o desenvolvimento de software; (ii) mapeamento direto de características transversais em aspectos; (iii) diminuição da complexidade de desenvolvimento

de geradores de código, porque a composição de características transversais pode ser realizada por compiladores de aspectos (*aspect weavers*); e (iv) melhoria no reuso de artefatos relacionados a características transversais.

Baseado na nossa experiência de desenvolvimento de uma abordagem generativa orientada a aspectos [79, 80], foram identificados requisitos úteis para a integração das abordagens generativas e de desenvolvimento orientado a aspectos. Tais requisitos guiaram o desenvolvimento dessa tese. A seguir é apresentada uma síntese dos mesmos:

(i) **suporte para representação de características transversais na análise de domínio**: a modelagem de características transversais em fases preliminares do desenvolvimento (tais como, modelagem de requisitos e da arquitetura) tem sido reconhecida recentemente como um tema de pesquisa importante em DSOA, denominado *Early Aspects* [10, 103]. Dessa forma, a extensão de modelos usados na análise e projeto de domínio para representar características transversais é fundamental para a sua modelagem e de suas respectivas interações com outras características de uma família de sistemas;

(ii) **especificação de arquiteturas orientadas a aspectos**: uma atividade fundamental quando construindo famílias de sistemas e linhas de produto é a modelagem de uma arquitetura de software que endereça características comuns e variáveis. O uso da abstração de aspectos traz novas possibilidades para a modularização de características transversais existentes em tais arquiteturas. Dessa forma, a definição de diretrizes, notações, princípios e padrões para a especificação de arquiteturas OA, com a adequada modularização e composição de suas características não tranversais e transversais, é um tópico importante que deve também ser tratado;

(iii) **especificação do conhecimento de configuração**: o conhecimento de configuração em um modelo generativo define como combinações de características no espaço de problema podem ser mapeadas para combinações específicas de componentes no espaço de solução. Diferentes tecnologias podem ser usadas para automatizar o conhecimento de configuração [32, 33], tais como, configuradores de produto [26] e tecnologias de geração de código baseada em templates [33, 34], em sistemas de transformação [33, 34], etc. A definição explícita de regras de mapeamento entre (I) características transversais presentes em DSLs usadas para modelar ou especificar produtos no espaço de problema, e

(II) artefatos de implementação (aspectos, frameworks, componentes, classes, templates) existentes no espaço de solução, é um requisito fundamental para habilitar a geração automática de membros de uma arquitetura de família de sistemas ou linha de produto;

(iv) **implementação de linguagens específicas de domínio**: DSLs permitem especificar membros de uma família da aplicação a serem instanciados ou gerados a partir de um conjunto de artefatos definidos para uma abordagem generativa. A modelagem de características variáveis transversais em arquiteturas de família de sistemas pode também requerer a sua explícita representação em DSLs, de forma a habilitar a sua customização. DSLs aspectuais [33, 110] têm sido propostas para endereçar tal requisito. O uso combinado de DSLs aspectuais e não aspectuais para expressar a composição de características no espaço de problema é outro tópico importante para ser endereçado na integração entre DG e DSOA;

(v) **implementação de frameworks com aspectos**: frameworks OO são uma tecnologia comum e bastante útil para a implementação de famílias de sistemas. Entretanto, como mencionado anteriormente (Seção 2.1.1), a modularização de determinadas características transversais em frameworks com mecanismos OO pode trazer dificuldades para o entendimento, gerência e manutenção de tais características. A tecnologia de aspectos habilita desenvolvedores a oferecer implementações modulares para características transversais. Dessa forma, a definição de diretrizes que auxiliem desenvolvedores no uso integrado das técnicas de framework e aspectos é também um elemento fundamental para a integração das abordagens generativa e de desenvolvimento orientado a aspectos.

A abordagem proposta nessa tese endereça vários dos requisitos discutidos para a integração das abordagens. Nossa abordagem é composta por: (1) um conjunto de diretrizes para modularização de características transversais encontradas em frameworks OO usando programação orientada a aspectos (Capítulo 4) que endereça o requisito (v); e (2) um modelo generativo orientado a aspectos (Capítulo 5) que propõe a extensão do modelo de características para explicitar a existência de características transversais endereçando assim o requisito (i) e um conjunto de regras de mapeamento entre características transversais e elementos de implementação baseados em aspectos que endereçam

o requisito (iii) permitindo a especificação de modelos de configuração. O requisito (ii) de especificação de arquiteturas orientadas a aspectos foi também explorado, embora não no contexto direto desta tese de doutorado, através da investigação do uso da abstração de aspectos em linguagens de descrição de arquiteturais (ADLs) [12, 13] e notações baseadas em UML [23, 24]. Finalmente, o requisito (iv) foi endereçado parcialmente já que nossa extensão do modelo de características para representar relações tranversais pode ser usada como uma DSL para a derivação de frameworks e linhas de produtos baseadas em aspectos.

2.5. Sumário

Esse capítulo apresentou três abordagens que vêm sendo exploradas no desenvolvimento de famílias de sistemas e linhas de produto, são elas: (i) frameworks orientados a objetos; (ii) desenvolvimento de software orientado a aspectos (DSOA); e (iii) desenvolvimento generativo (DG). Foram descritos diversos problemas de modularização de cararacterísticas transversais que são encontrados durante o desenvolvimento de frameworks, quando usando mecanismos OO para implementá-las. Dada a natureza transversal de tais características, elas são candidatas naturais a serem implementadas com técnicas orientadas a aspectos. O capítulo foi concluído apresentando uma discussão sobre o potencial de sinergia existente entre as três abordagens. O capítulo seguinte apresenta uma abordagem para o desenvolvimento de frameworks, centrada em técnicas oferecidas pelas abordagens de DSOA e DG. Tal abordagem lida com os problemas de modularização de frameworks descritos nesse capítulo.

3
Uma Abordagem Orientada a Aspectos para o Desenvolvimento de Frameworks

Este capítulo apresenta uma visão geral da contribuição principal deste trabalho: uma abordagem orientada a aspectos para o desenvolvimento de frameworks. A abordagem tem como objetivo uma melhor modularização de características transversais encontradas em frameworks de forma a facilitar a sua customização para diferentes cenários. Ela é composta por: (i) um conjunto de diretrizes que auxiliam o desenvolvedor a definir quais funcionalidades do framework devem ser projetadas e implementadas usando aspectos; e (ii) um modelo generativo que permite a instanciação automática das variabilidades do framework, incluindo aquelas implementadas usando programação orientada a aspectos, a partir de um modelo de características.

3.1.
Diretrizes para Implementação de Frameworks com Aspectos

Essa seção apresenta os elementos que compõem as diretrizes para modularização de frameworks usando aspectos, sendo eles: (i) pontos de junção de extensão; (ii) núcleo do framework; e (iii) aspectos de extensão.

3.1.1.
Pontos de Junção de Extensão (EJPs)

Os pontos de extensão (*hot-spots*) de um framework OO são tipicamente implementados através de classes abstratas ou interfaces. Eles possibilitam a extensão do comportamento comum de colaboração fornecido pelo framework oferecendo implementações concretas para seus respectivos métodos abstratos.

Na abordagem proposta, um framework OO especifica e implementa não apenas suas funcionalidades comuns e variáveis usando classes, mas também expõe um conjunto de pontos de junção de extensão [73, 75, 84] (EJPs, do inglês *Extension Join Points*). Os EJPs expõem pontos específicos da execução do

framework, os quais podem ter sua funcionalidade estendida por meio da codificação de aspectos. Dessa forma, eles são adotados como uma forma de facilitar a implementação de variabilidades transversais e de integração. Os EJPs também estabelecem contratos entre as classes do framework e o conjunto de aspectos que estendem sua funcionalidade básica. Eles podem ser usados para dois diferentes propósitos:

(i) expor um conjunto de eventos do framework que podem ser usados para notificar ou facilitar uma integração transversal com outros módulos de software, tais como, frameworks, componentes ou bibliotecas;

(ii) oferecer pontos de execução pré-definidos que estão espalhados e/ou entrelaçados no framework e nos quais pode ser incluída a implementação de características opcionais ou alternativas transversais.

Dessa forma, os EJPs documentam pontos de extensão transversais para desenvolvedores de software que desejam instanciar e estender o framework. Eles podem também ser vistos como um conjunto de restrições no espaço total de pontos de junção existentes no framework. O Capítulo 4 mostra como os EJPs podem ser implementados em AspectJ.

3.1.2. Núcleo do Framework e Aspectos de Extensão

Nossa abordagem promove o desenvolvimento de frameworks como uma composição de uma estrutura núcleo e um conjunto de aspectos de extensão. Um aspecto de extensão pode endereçar: (i) a implementação de características opcionais ou alternativas transversais do framework; ou (ii) a integração transversal com um componente, biblioteca ou framework adicional. A composição entre o núcleo do framework e suas extensões é realizada por diferentes tipos de aspectos. Cada aspecto define uma composição transversal com o framework através dos EJPs expostos. A seguir, os principais elementos da abordagem são descritos:

(i) **núcleo do framework** – implementa a funcionalidade obrigatória de uma família de software. De forma similar a um framework OO tradicional, a estrutura do núcleo contém a implementação de classes que representam seus pontos fixos (*frozen-spots*) com a funcionalidade comum da família de software, e de classes

que representam pontos flexíveis (*hot-spots*) não transversais do domínio endereçado;

(ii) **aspectos do núcleo** – implementam e modularizam interesses ou papéis tranversais existentes nas classes do núcleo do framework. Eles representam o uso tradicional de POA para simplificar o entendimento e evolução do núcleo;

(iii) **aspectos de variabilidade** – implementam características opcionais e alternativas transversais existentes no núcleo do framework. Tais elementos estendem os EJPs do framework com algum comportamento transversal adicional;

(iv) **aspectos de integração** – definem composições transversais entre o núcleo do framework e outras extensões existentes, tais como, uma biblioteca, um componente ou um framework. Esses elementos também dependem dos EJPs para definir sua implementação.

A Figura 8 mostra o projeto do framework OO com aspectos seguindo as diretrizes de nossa abordagem. Como podemos ver na figura, os aspectos de integração e variabilidade podem atuar apenas nos EJPs oferecidos pelo framework e devem obedecer todas as restrições definidas pelos mesmos.

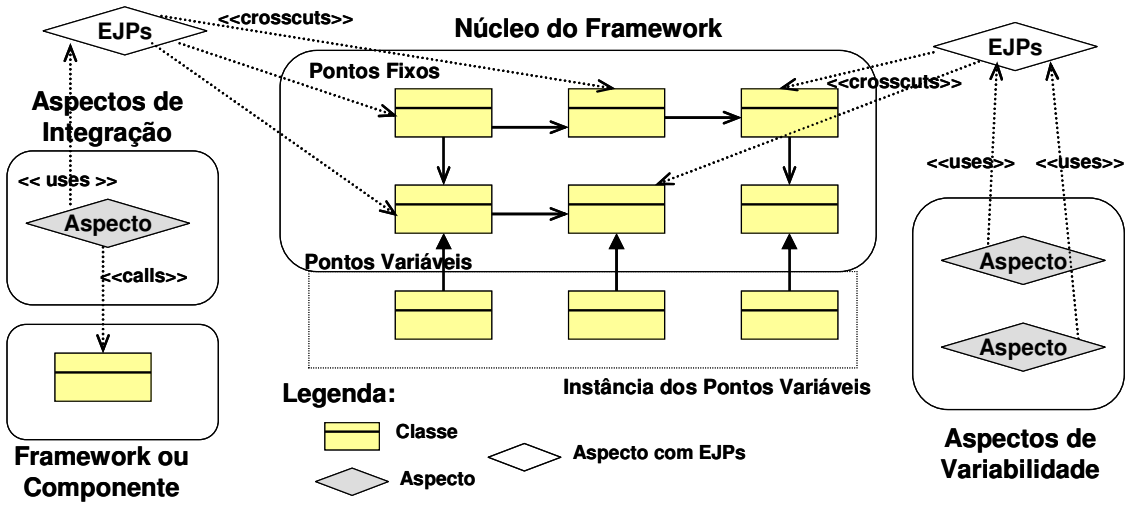

Figura 8. Elementos de Implementação da Abordagem Proposta

3.2. Um Modelo Generativo Orientado a Aspectos

O resultado do projeto e implementação de uma arquitetura de família de sistemas ou linha de produto usando as diretrizes para modularização de características de frameworks é um conjunto de artefatos (classes, aspectos, arquivos de configuração, etc) os quais endereçam as características comuns e variáveis de um dado domínio. Muito das classes e aspectos definidos para a arquitetura serão instanciados apenas se eles forem necessários para implementar uma aplicação ou produto específico. Dessa forma, a seleção manual e customização de tais elementos pode se tornar uma atividade complexa a qual torna extremamente custoso ou até impraticável o uso da abordagem. Para facilitar a instanciação do framework e seus respetivos aspectos de extensão, um modelo generativo orientado a aspectos é proposto. O objetivo principal de tal modelo generativo é permitir a configuração de uma instância do framework a partir de um modelo de características.

Nosso modelo generativo OA [74, 76, 81] segue a estrutura geral apresentada por Czarnecki e Eisenecker [33] (Seção 2.3). Entretanto, é proposta a extensão de tal modelo para suportar a instanciação de arquiteturas OA, permitindo a geração e customização de variabilidades OO e OA. Nosso modelo é composto por:

(i) **um modelo de característica** – este modelo funciona como uma linguagem específica de domínio de configuração [33]. Ele é responsável pela especificação e coleta de informação necessária para a customização das variabilidades OO e OA, existentes em classes e aspectos. Um conjunto de relacionamentos tranversais entre características é usado em tal modelo para auxiliar na customização de pontos de corte de aspectos;

(ii) **um modelo de arquitetura OA** – este modelo define os principais componentes de uma arquitetura de família de sistemas ou linha de produto. A arquitetura contém um conjunto de variabilidades que precisam ser customizadas para definir uma aplicação completa. Variabilidades transversais são implementadas como aspectos nessa arquitetura. Cada componente é definido como um conjunto de classes, aspectos, templates e arquivos extras. Os templates representam elementos (classes, aspectos, arquivos de configuração) que serão

customizados durante o processo de instanciação automático. As diretrizes para modularização de características em frameworks de nossa abordagem (Seção 3.1.1) são usadas como a base principal para implementação da arquitetura OA;

(iii) **um modelo de configuração** – esse modelo especifica o mapeamento entre os elementos definidos no modelo de característica e os componentes (e respectivos sub-elementos, tais como, classes, aspectos, arquivos extras ou templates) do modelo de arquitetura OA. Tal modelo também permite definir explicitamente o mapeamento entre features transversais e aspectos possibilitando a geração de pontos de corte específicos em aspectos. Todas as informações providas pelo modelo de configuração são usadas para auxiliar no processo de decidir quais componentes devem ser instanciados e quais customizações devem ser realizadas em tais componentes considerando uma aplicação específica.

Os elementos existentes (modelos de característica, de arquitetura e de configuração) em nosso modelo generativo são definidos para habilitar seu uso por uma ferramenta de customização e geração de código. O Capítulo 5 detalha cada um desses modelos e apresenta diretrizes de tecnologias que podem ser utilizadas para a sua definição. Um algoritmo de processamento de tais modelos que gera como resultado uma aplicação específica é também apresentado.

3.3. Fluxo de Atividades da Abordagem

Existem diversas atividades envolvidas no processo de desenvolvimento dos elementos de nossa abordagem. As Figuras 9 e 10 apresentam esse conjunto de atividades, organizadas sob as perspectivas da engenharia de domínio [33] e da engenharia de aplicação [33], respectivamente. Na engenharia de domínio, nossa abordagem endereça atualmente apenas as fases de projeto e implementação de domínio, oferecendo um conjunto de atividades para o desenvolvimento de frameworks usando mecanismos OO e OA (atividades I, II, III, IV e V). Também outras atividades endereçam a especificação dos diversos elementos que compõem o nosso modelo generativo (atividade V, VI, VII, VIII e IX), de forma a apoiar a instanciação automática do framework e suas variabilidades durante a engenharia de aplicação. A seguir são apresentadas descrições de cada uma das atividades da engenharia de domínio e de aplicação. Embora tais atividades

estejam organizadas de forma seqüencial, elas não necessariamente precisam ocorrer nessa ordem. A especificação do modelo de característica, por exemplo, pode ocorrer em paralelo com as demais atividades.

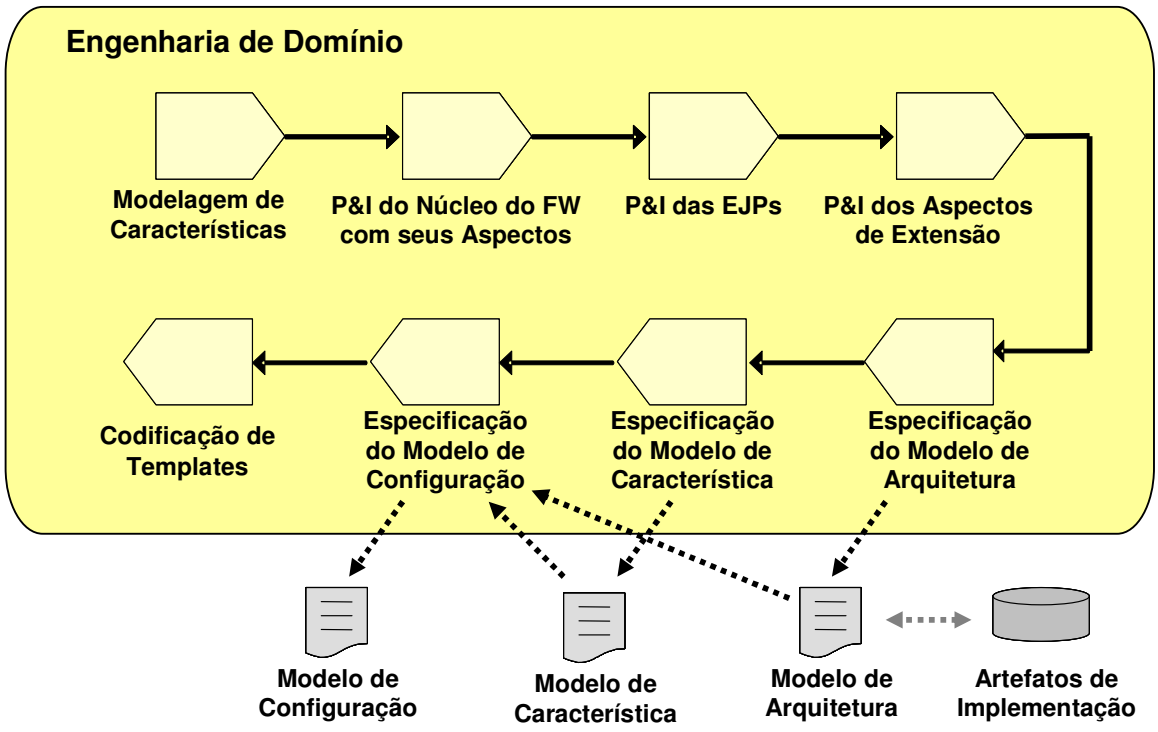

Figura 9. Atividades de Engenharia de Domínio da Abordagem

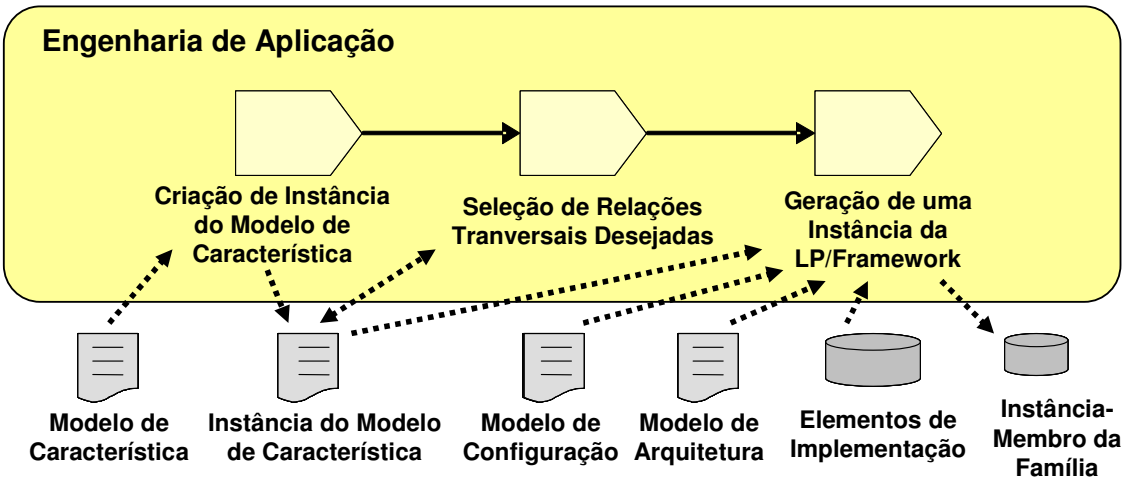

Figura 10. Atividades de Engenharia da Aplicação da Abordagem

FASE: ENGENHARIA DE DOMÍNIO

Atividades de Desenvolvimento do Framework

(I) **modelagem de características comuns e variáveis do framework** – essa atividade se concentra na representação de características comuns e variáveis a serem implementadas pelo framework, em um modelo de característica. Ela é, em geral, precedida por atividades de análise de domínio que buscam a identificação e elicitação das características comuns e variáveis existentes no domínio do framework;

(II) **projeto e implementação do núcleo do framework** – essa atividade objetiva a implementação da funcionalidade comum a qualquer uma das instâncias do framework. Ela envolve o desenvolvimento das características obrigatórias (*frozen-spots*) do framework e também das características variáveis não transversais (*hot-spots*) que devem sempre ser instanciadas pelos usuários do framework, usando mecanismos OO tradicionais, tais como, herança, agregação e composição. Padrões de projeto (tais como, *Strategy* e *Template Method*) são bastante úteis para o projeto OO de características variáveis não transversais;

(III) **projeto e implementação de aspectos do núcleo** – essa atividade busca oferecer uma melhor modularização dos interesses transversais existentes no núcleo, aspectos podem ser codificados para facilitar o entendimento e/ou manutenção do núcleo do framework;

(IV) **projeto e implementação dos EJPs** – cada framework pode, além de oferecer pontos de extensão OO (por meio de interfaces ou classes abstratas), expor um conjunto de pontos de junção para a inclusão de extensões transversais. Essa atividade objetiva a identificação de eventos ou estados relevantes da execução do framework que possam ser relevantes para a realização de alguma composição transversal com o framework;

(V) **projeto e implementação dos aspectos de extensão** – nessa atividade, aspectos de extensão são implementados para endereçar as características transversais opcionais, alternativas e de integração do framework. Cada um dos aspectos de extensão deve atuar exclusivamente nos EJPs expostos pelo framework. Aspectos de extensão podem também por sua vez oferecer alguma variabilidade na sua implementação.

Atividades de Especificação do Modelo Generativo

(VI) **especificação do modelo de arquitetura** – essa atividade envolve a representação de todos os elementos implementados para a arquitetura da família de sistemas ou framework, como um conjunto de classes, aspectos, templates e arquivos extras. Uma ferramenta pode auxiliar na geração semi-automática de tal modelo, a partir do processamento do diretório contendo os artefatos de implementação da arquitetura;

(VII) **especificação do modelo de características** – após a implementação do framework, e seus EJPs e aspectos de extensão, é necessário revisitar e atualizar o modelo de característica produzido como resultado da atividade (I), com eventuais modificações realizadas nas variabilidades do framework;

(VIII) **especificação do modelo de configuração** – essa atividade envolve, principalmente, a definição de um conjunto de relações de dependência entre características e elementos de implementação do framework (classes, aspectos, templates, arquivos de configuração, etc);

(IX) **codificação de templates** – envolve o uso de atributos/propriedades capturadas pelo modelo de característica para codificação de um dado elemento (classe, aspecto ou arquivo de configuração), cuja customização depende de informações oferecidas por tais atributos/propriedades;

FASE: ENGENHARIA DA APLICAÇÃO

A engenharia de aplicação é simplificada devido à preparação do framework para ser automaticamente instanciado durante a engenharia de domínio. Essa fase é composta das seguintes atividades:

(I) **criação de uma instância do modelo de características** – criação de uma instância do modelo de características da família de sistemas, e seleção das variabilidades (características varíaveis) que se deseja incluir no framework, com a respectiva configuração de suas propriedades;

(II) **seleção de relações transversais** – envolve a escolha ou seleção de relações tranversais válidas no modelo de características. Ela permite a customização de pontos de junção específicos onde aspectos de extensão irão atuar;

(III) **criação de uma instância da arquitetura de linha de produto / framework** – essa atividade envolve a solicitação de criação de uma instância da arquitetura da família de sistemas, a partir: da instância do modelo de característica definido (atividade I) com respectivas relações traversais entre características (atividade II), e dos modelos de arquitetura e configuração da família ou linha de produto sendo considerada. Uma ferramenta de instanciação é responsável por processar tais modelos e gerar uma aplicação que atenda a solicitação do engenheiro de aplicação.

3.4. Sumário

Este capítulo apresentou uma visão geral da abordagem OA para desenvolvimento de framework proposta nesse trabalho. Ela é centrada na definição de um conjunto de pontos de junção de extensão (EJPs) no núcleo de framework de forma a facilitar a implementação de extensões transversais. Os capítulos seguintes detalham as diretrizes de modularização de frameworks usando aspectos e o modelo generativo proposto para instanciação de suas variabilidades.

4
Diretrizes para o Projeto e Implementação de Frameworks usando Orientação a Aspectos

Nossa abordagem para desenvolvimento de frameworks usando aspectos oferece um conjunto de diretrizes (seção 3.3.1) para a modularização de determinados tipos de características encontradas no projeto e implementação de frameworks. Seguindo nossas diretrizes, tais características são implementadas usando aspectos e estendem o framework em pontos de junção de extensão bem definidos, denominados EJPs. Esse capítulo detalha tais diretrizes, mostrando como EJPs e aspectos de extensão podem ser implementados usando os mecanismos de AspectJ. O framework JUnit é usado para ilustrar as diretrizes de implementação.

4.1.
Implementando EJPs em AspectJ

Um EJP na nossa abordagem declara: (i) um conjunto de pontos de junção do framework que são candidatos a serem estendidos por aspectos; e (ii) um conjunto de contratos que são usados para regular as relações entre o framework, EJPs e aspectos de extensão. Nas subseções seguintes, os mecanismos específicos de AspectJ usados para codificação de EJPs e seus respectivos contratos são detalhados.

4.1.1.
Documentação Visual de EJPs e Aspectos de Extensão

Para facilitar a documentação e visualização dos elementos que compõem a abordagem de desenvolvimento de frameworks, foram definidos alguns estereótipos em diagramas de classes UML [16] que buscam facilitar a identificação e visualização do núcleo do framework, e seus respectivos EJPs e aspectos de extensão e do núcleo. Essa documentação é usada ao longo desse capítulo e na apresentação dos estudos de caso descritos no capítulo 6.

A Figura 11 apresenta a documentação do framework JUnit [73, 75, 84] com seus respectivos aspectos do núcleo, EJP e aspectos de extensão. Diferentes pacotes podem ser definidos para agregar cada um desses elementos. Na Figura 11 foram definidos os seguintes pacotes: *framework core*, *extension join points*, *variability aspects* e *integration aspects*. Aspectos EJPs são representados usando o esteréotipo `<<EJP>>`, e o compartimento de métodos é usado para definir seus diferentes pontos de corte. Aspectos do núcleo e de extensão são representados usando o esteréotipo `<<aspect>>`, a distinção entre eles pode ser feita pelo pacote do qual fazem parte. Dois novos tipos de estereótipos foram também definidos para especificar relações de dependência: (i) `<<crosscuts>>` – indica que determinados aspectos interceptam pontos de junção de classes do núcleo do framework; e (ii) `<<uses>>` – útil para especificar que um dado aspecto de extensão usa a definição de pontos de corte expostos por aspectos ou EJPs ou que utiliza a implementação de uma dada classe.

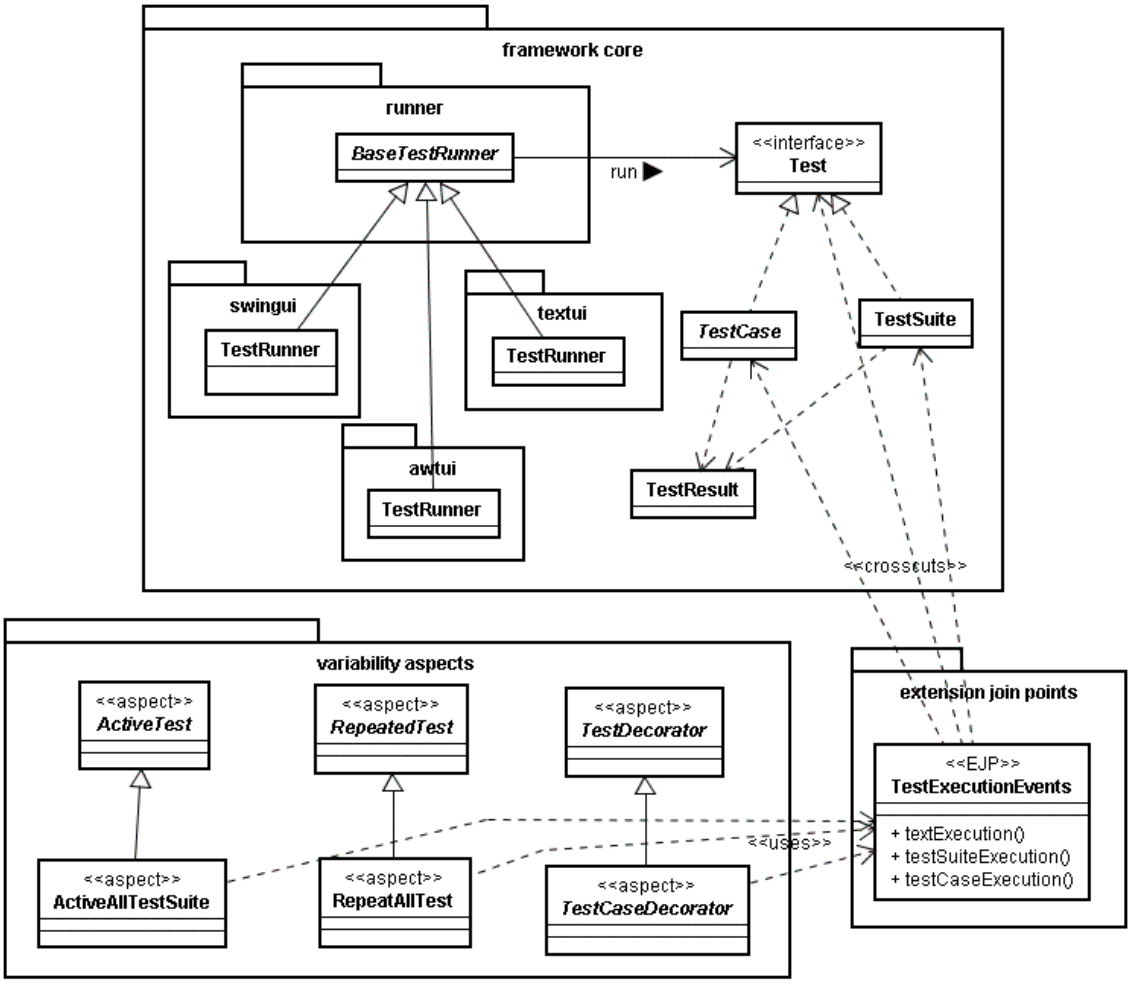

Figura 11. Diagrama de Classes e Aspectos do JUnit

4.1.2. Estrutura de EJPs

EJPs são implementados na nossa abordagem usando a abstração de aspectos de AspectJ. Tais aspectos declaram pontos de junção públicos os quais representam os EJPs do framework. Dependendo da quantidade de pontos de junção a serem expostos pelo framework, vários aspectos podem ser criados, cada um englobando um subconjunto de pontos de junção fortemente relacionados. A Tabela 1 apresenta os elementos que compõem a especificação de cada EJP em AspectJ.

Elemento	Propósito
Nome	• Especifica o nome do aspecto EJP; • É representado pelo nome do aspecto em AspectJ.
Pontos de Extensão Transversais	• Quantificam o conjunto de pontos de junção do framework; • Representam eventos ou estados de transição relevantes que ocorrem durante a execução das funcionalidades do núcleo do framework.
Escopo	• Define o escopo do núcleo do framework, contemplando todos os elementos de implementação (classes, aspectos); • É representado pela especificação de pontos de corte AspectJ, que utilizam a construção `within`, para incluir todos os pacotes dos componentes do framework.

Tabela 1. Estrutura Geral de EJPs

A Figura 12 apresenta o aspecto EJP `TestExecutionEvents`, o qual expõe um conjunto de pontos de junção no framework JUnit, através de vários pontos de corte, entre eles: (i) ponto de corte `testExecution()` – o qual expõe o evento de execução de casos ou suítes de teste (instância do tipo `Test`) (linhas 3-4); (ii) ponto de corte `testSuiteExecution()` – expõe o evento de execução de todos os suítes de teste (linhas 11-14); (iii) ponto de corte `testCaseExecution()` – expõe o evento de execução de todos os casos de teste (linhas 17-20). Esses dois últimos pontos de corte são definidos em função do ponto de corte `testExecutionGeneral()`, o qual também é usado para auxiliar na especialização de EJPs (Seção 4.1.3).

Cada aspecto EJP pode também definir o escopo das classes que fazem parte do núcleo do framework, através da especificação de pontos de corte com o uso da construção `within` de AspectJ. Tais pontos de corte são usados na especificação dos contratos dos EJPs. Na Figura 12 foi definido o ponto de corte `FWScope()` com tal propósito, ele define que o núcleo do framework JUnit compreeende todas as classes dentro do pacote `junit` e respectivos subpacotes.

```
01 public aspect TestExecutionEvents {
02   // Expose the test execution event
03   public pointcut testExecution(Test test):
04      target(test) && call (void Test.run(TestResult));
05
06   // Expose the method of Test Execution
07   public pointcut testExecutionGeneral(TestResult result):
08           call (void *.run(TestResult)) && args(result);
09
10   // Expose the test suite execution event
11   public pointcut testSuiteExecution (TestSuite testSuite,
12                                       TestResult result):
13        testExecutionGeneral (result) &&
14        target(testSuite) && target(TestSuite);
15
16   // Expose the test case execution event
17   public pointcut testCaseExecution (TestCase testCase,
18                                      TestResult result):
19        testExecutionGeneral (result) &&
20        target(testCase) && target(TestCase);
21   ...
22   // Framework Scope
23   protected pointcut FWScope(): within(junit..*);
24   ...
25 }
```

Figura 12. Aspecto EJP `TestExecutionEvents`

Uma vez exposto o conjunto de EJPs de um framework, aspectos de extensão podem ser codificados para estender a funcionalidade do seu núcleo. A Figura 13 ilustra a definição de um aspecto de variabilidade para o JUnit, que estende o framework para possibilitar a repetição da execução de casos de teste. O aspecto `RepeatedAllTests` define um adendo do tipo `around` (linhas 6-11), o qual intercepta o evento de execução de casos de teste para introduzir a funcionalidade de execução repetida. O ponto de corte `testCaseExecution()` do aspecto EJP `TestExecutionEvents` é usado para introduzir a funcionalidade do adendo dentro do framework (linha 7).

```
01 public aspect RepeatedAllTests {
02    public int getTimesRepeat(){
03      return 10;
04    }
05
06    void around(TestCase testCase, TestResult testResult):
07         TestExecutionEvents.testCaseExecution(testCase, result) {
08       for (int i=0; i < getTimesRepeat(); ++i){
09          proceed(test, testResult);
10       }
11    }
12 }
```

Figura 13. Aspecto de Extensão RepeatAllTests

4.1.3. Especialização de EJPs

Cada EJP expõe um conjunto de pontos de junção do framework nos quais podem ser inseridos novos comportamentos tranversais codificados em aspectos de extensão. Dentre os pontos de junção expostos, alguns podem ser métodos abstratos ou métodos gancho (*hooks*) de classes que representam pontos flexíveis OO (*hot-spots*) do framework. Durante a instanciação do framework, as classes que representam pontos flexíveis são especializadas com a definição de métodos concretos para criar comportamento específico para uma dada aplicação. Assim, uma vez que alguns dos métodos gancho ou abstratos podem ser especializados, também os pontos de junção definidos nos EJPs podem ser especializados para afetar apenas instâncias específicas de pontos flexíveis.

No framework JUnit, por exemplo, alguns dos seus pontos de junção podem ser customizados para afetar apenas instâncias específicas de casos e suítes de teste. Dessa forma, o desenvolvedor que está implementando os aspectos de extensão precisa especializar os EJPs para indicar que ele deseja estender apenas instâncias específicas de casos e suítes de teste.

Em AspectJ, a especialização de EJPs pode ser implementada através do uso de composições de pontos de corte e do designador `target()`. A Figura 14 apresenta o aspecto de extensão `RepeatSpecificTestCases`, o qual afeta apenas subclasses de casos de teste específicas. O novo ponto de corte `testCaseExecutionSpecialization()` é definido como uma especialização do ponto de corte `testCaseExecutionGeneral()`, presente no aspecto EJP `TestExecutionEvents`. Para afetar apenas instâncias específicas, o designador de

ponto de corte `target()` é usado. Ele permite especificar os tipos de subclasses que se deseja afetar (linha 5). No caso específico desse aspecto, ele irá afetar apenas as subclasses `TestCaseA` e `TestCaseB`.

```
01 public aspect RepeatSpecificTestCases {
02    public pointcut testCaseExecutionSpecialization(
03            Test testCase, TestResult result):
04       TestExecutionEvents.testCaseExecutionGeneral(result)
05       && (target(TestCaseA) || target(TestCaseB))
06       && target(testCase)
07       && !adviceexecution();
08
09    public int getTimesRepeat(){
10       return 10;
11    }
12    void around(TestCase testCase, TestResult testResult):
13         TestExecutionEvents.testCaseExecutionSpecialization
14                                        (testCase, result) {
15        for (int i=0; i < getTimesRepeat(); ++i){
16          proceed(testCase, testResult);
17        }
18     }
19 }
```

Figura 14. Exemplo de Especialização de EJP

4.1.4. Contratos de EJPs

Na nossa abordagem, foi também definido um conjunto de contratos [92] que podem existir entre framework, EJPs e aspectos de extensão. Tais contratos são organizados nas seguintes categorias: (i) *contratos internos do framework* – que definem restrições a serem obedecidas pelo framework; e (ii) *contratos de extensão do framework* – definem restrições a serem seguidas pelos aspectos de extensão.

Os contratos internos do framework definem restrições (*constraints*) cujo propósito é garantir que refatorações e evoluções do framework não irão afetar a funcionalidade de seus aspectos de extensão. Eles são classificados em: (i) *estruturais* – cujo objetivo é garantir que o framework implementa interfaces específicas definidas pelos EJPs; e (ii) *comportamentais* – os quais buscam garantir que os EJPs expõem todos (e apenas) os eventos ou estados desejados.

Os contratos de extensão do framework são usados para garantir que cada aspecto de extensão respeita restrições e invariantes do framework. As seguintes

categorias foram definidas: (i) estruturais – estes contratos objetivam garantir que aspectos apenas estendem os pontos de junção do framework que foram expostos pelos EJPs; (ii) comportamentais – definem os métodos de classes do framework que podem ser invocados pelos aspectos de extensão; e (iii) invariantes – definem um conjunto de pré e pós-condições que devem ser preservadas antes e depois da execução dos adendos dos aspectos de extensão.

As Tabelas 2 e 3 apresentam a categorização de contratos, assim como os respectivos mecanismos de AspectJ que podem ser usados para implementá-los. AspectJ oferece diversos mecanismos úteis para implementar cada tipo de contrato. Durante a escolha de qual mecanismo usar para cada tipo de contrato, foram priorizados mecanismos estáticos ao invés de dinâmicos, uma vez que os primeiros podem ser verificados em tempo de compilação. Esse é o caso, por exemplo, dos mecanismos de AspectJ: `declare parents`, `declare error` e `declare warning`. Alguns tipos de contrato, entretanto, dependem de informações dinâmicas para serem implementados. Para esses casos específicos, tal como a verificação de invariantes do framework, a construção `adviceexecution` de AspectJ que permite interceptar a execução de adendos de aspectos pode ser utilizada. Alguns tipos de contratos não podem ser codificados usando os mecanismos disponíveis em AspectJ, tais como, os contratos de extensão estrutural (Tabela 3).

Tipo do Contrato	Descrição e Mecanismos de Implementação em AspectJ
Estrutural	• Especificação de interfaces que devem ser implementadas pelas classes do framework; • A obrigação de implementar essas interfaces é atribuída pelos contratos estruturais dos EJPs usando a construção `declare parents` de AspectJ. As interfaces são também declaradas dentro dos aspectos que representam os contratos dos EJPs.
Comportamental	• Implementação de políticas de reforço que garantem que os EJPs são chamados em todos e apenas lugares apropriados dentro do código das classes do framework. • Esses contratos podem ser especificados usando os comandos `declare warning` e `declare error` de AspectJ.

Tabela 2. Contratos Internos do Framework.

A Figura 15 apresenta um aspecto contendo a implementação de dois contratos para o framework JUnit. O primeiro é um contrato interno

comportamental definido usando a construção `declare error` (linhas 7-11). Ele restringe que o método `run(TestResult)` da interface `Test` deve ser invocado apenas por determinados métodos das classes `TestCase` e `TestSuite`. Isso garante que apenas tais métodos podem efetivamente demandar a execução dos testes dentro do framework JUnit. O método `run()` da interface `Test` é o responsável direto pela execução dos testes no JUnit. Durante a refatoração ou evolução do framework JUnit, caso novas classes passem a invocar tal método, o contrato acusará um erro que deverá ser analisado pelo desenvolvedor para garantir que aquele contrato interno continua sendo preservado, se necessário o código do contrato pode evoluir para incluir novos métodos que fazem chamadas ao método `run()` da interface `Test`.

Tipo do Contrato	Descrição e Mecanismos de Implementação em AspectJ
Estrutural	• Esse contrato garante que aspectos de extensão apenas estendem os pontos de junção do framework expostos pelos EJPs; • Esse contrato não pode ser implementado em AspectJ, devido a limitação corrente da linguagem a qual não permite restringir pontos de junção específicos para serem estendidos; • Uma forma de garantir esses contratos é estabelecer que desenvolvedores sigam a prática de programação de estender o framework apenas nos pontos de junção especificados nas EJPs.
Comportamental	• Determina quais métodos de classes do framework podem ser invocados pelos aspectos de extensão; • Tais contratos podem ser especificados em AspectJ, usando os comandos `declare warning` e `declare error` de AspectJ, o qual permite a verificação estática das políticas de acesso.
Invariantes	• Definem pré e pós-condições que devem ser garantidas antes e depois da execução dos adendos; • O comando `adviceexecution()` o qual permite interceptar os adendos de aspectos pode ser usado com a finalidade de implementar tais pré e pós condições nos aspectos de extensão.

Tabela 3. Contratos de Extensão do Framework.

A Figura 15 também apresenta um contrato de extensão comportamental que visa garantir que apenas os aspectos de extensão do JUnit podem invocar métodos internos das classes do framework JUnit (linhas 14-22). Esse contrato foi

definido usando dois mecanismos sofisticados de AspectJ, são eles: (i) `adviceexecution` - que permite interceptar a invocação de adendos de aspectos; e (ii) `cflow` - que permite interceptar todos pontos de junção que ocorrem no contexto do fluxo de controle de um dado ponto de junção. Assim, o adendo `before` de tal contrato lança uma `RuntimeException`, sempre que houver chamadas para classes do framework, ocorrendo dentro de adendos de aspectos (`adviceexecution`) ou originadas a partir de tais adendos (`cflow`) e que não estejam dentro do pacote de aspectos de extensão (`!extensionAspectsScope()`).

```
01 public aspect JUnitEJPContracts {
02   //Behavioral Internal Contract
03   public pointcut EJPMethodsCallScope():
04     withincode (public TestResult TestCase.run())  ||
05     withincode (public void TestSuite.runTest(
06                                   Test, TestResult));
07   declare error:
08      (!EJPMethodsCallScope()
09        && call(void Test.run(TestResult))):
10      "Contract violation: Test execution should occur" +
11      "through one of the methods specified above";
12
13   //Behavioral Extension Contract
14   public pointcut extensionAspectsScope():
15             within(junit.aop.extensions..*);
16   before() :
17     cflow ( adviceexecution() && !extensionAspectsScope())
18     && ( call(* *(..)) && TestExecutionEvents.FWScope() ){
19      throw new RuntimeException("Contract Violation: no " +
20        "aspects, except variability aspects, can access " +
21        "the elements of JUnit framework.");
22    }
23 }
```

Figura 15. Exemplo de Contratos de EJPs no Contexto do JUnit

Esta tese apenas propõe uma categorização de contratos de EJPs. Tais contratos foram utilizados na implementação de dois de nossos estudos de caso [29, 75]. Uma abordagem para verificação e teste de características transversais

[30], a qual detalha a utilização de tais contratos em conjunção com a definição de testes automatizados de unidade e integração vem sendo desenvolvida em um outro trabalho de pesquisa.

4.2. Implementando Aspectos de Extensão em AspectJ

Na nossa abordagem, todas as extensões tranversais do framework são introduzidas no seu núcleo usando aspectos de variabilidade e integração. Cada aspecto introduz comportamentos transversais no conjunto de pontos de junção expostos pelos EJPs. Os aspectos desempenham um papel específico que está diretamente relacionado com a funcionalidade transversal que eles introduzem no framework. Eles podem, por exemplo, desempenhar o papel de observadores de eventos internos do framework para notificar módulos externos interessados em tais eventos. Eles podem também mediar a comunicação entre classes específicas do framework e outras extensões OO dentro de particulares pontos de junção expostos pelos EJPs. Ou eles podem ainda decorar EJPs com implementações de características transversais opcionais ao núcleo do framework. Assim, muitos aspectos desempenham o papel de padrões de projeto tradicionais (tais como, *Observer*, *Mediator* ou *Decorator*) [45] com o objetivo de estender o núcleo do framework.

No estudo de caso do framework JUnit, por exemplo, foram definidos vários aspectos de variabilidade, os quais "decoram" o comportamento oferecido por classes do núcleo do framework, são eles: (i) `RepeatedTests` – aspecto abstrato que é concretizado para permitir a execução de casos de teste repetidamente; (ii) `ActiveTest` – aspecto abstrato que é especializado para permitir a execução de suítes de teste em *threads* distintas; e (iii) `TestCaseDecorator` – aspecto abstrato que é usado para introduzir comportamentos adicionais antes ou depois de casos e suítes de teste. Esses aspectos atuam como decoradores de suítes e casos de teste.

As Figuras 16 e 17 apresentam, respectivamente, o código dos aspectos de extensão `ActiveTest` e `ActiveAllTestSuite`. O aspecto abstrato `ActiveTest` define o comportamento comum de execução de casos ou suítes de teste em threads distintas, através de um conjunto de adendos e pontos de corte abstratos. O aspecto `ActiveAllTestSuite` especializa o aspecto `ActiveTest`, concretizando

os pontos de junção do sistema onde se deseja aplicar tal funcionalidade. Observe que na definição dos pontos de corte do aspecto `ActiveAllTestSuite`, pontos de corte expostos pelo aspecto EJP `TestExecutionEvents` são reusados.

```
01 public abstract aspect ActiveTest {
02   private volatile int fActiveTestDeathCount;
03   private int testCaseQuantity;
05   public abstract pointcut activeTestsPoints(
06              TestSuite testSuite, TestResult testResult);
08   before (TestSuite testSuite, TestResult result):
09                activeTestsPoints(testSuite, result){
10      fActiveTestDeathCount= 0;
11   }
12   after (TestSuite testSuite, TestResult result):
13                activeTestsPoints(testSuite, result){
14      waitUntilFinished();
15   }
16   public abstract pointcut activeInternalTestsPoints(
17     TestSuite testSuite, Test test, TestResult testResult);
19    void around(TestSuite testSuite,
20              Test test, TestResult result):
21      activeInternalTestsPoints(testSuite, test, result){
22        testCaseQuantity = testSuite.countTestCases();
23        runTest(test, result);
24    }
25    public void runTest(final Test test,
26                        final TestResult result){
27       Thread t= new Thread() {
28          public void run() {
29            try {  test.run(result);
30            } finally {
31              ActiveTest.this.runFinished(test);
32            }
33          }
34        };
35        t.start();
36    }
37    synchronized void waitUntilFinished() {  ... }
38 }
```

Figura 16. Aspecto de Variabilidade `ActiveTest`

```
01 public aspect ActiveAllTestSuite extends ActiveTest {
02    public pointcut activeTestsPoints(TestSuite testSuite,
03                                      TestResult result):
04      TestExecutionEvents.testSuiteExecution(testSuite,
05                                             result);
06
07    public pointcut activeInternalTestsPoints(
08        TestSuite testSuite, Test test, TestResult result):
09      TestExecutionEvents.testMethodExecutionInsideSuite(
10                            testSuite, test, result);
11
12 }
```

Figura 17. Aspecto de Variabilidade `ActiveAllTestSuite`

4.2.1. Variabilidades em Aspectos de Extensão

A implementação dos aspectos de extensão pode também requerer a separação entre código comum e variável presente nos mesmos. Isso significa que aspectos de extensão podem também conter variabilidades. Os aspectos `ActiveTest` e `ActiveAllTestSuite`, apresentados na seção anterior, representam a separação entre código comum e variável para a característica de execução de suíte de teste em *threads* distintas.

O aspecto `RepeatedAllTests` (apresentado na Seção 4.1.2) também pode ser modularizado de tal forma a postergar para tempo de instanciação da aplicação, a definição da quantidade de vezes de repetição de suítes ou casos de teste e a quais casos de teste específicos se deseja aplicar tal funcionalidade de extensão. Diferentes tipos de variabilidades podem existir nos aspectos de extensão, tais como:

(i) **comportamento variável do aspecto** – nesse caso parte da funcionalidade do aspecto depende de informação disponível na aplicação que será gerada a partir da instanciação do framework. Os mecanismos de definição de aspectos e métodos abstratos oferecidos por AspectJ permitem a implementação desse comportamento variável;

(ii) **ponto de junção varíavel do aspecto** – esse tipo de variabilidade ocorre quando o comportamento geral do aspecto já está definido, mas os pontos de junção onde ele irá atuar só são conhecidos em tempo de instanciação do framework. Em AspectJ podem ser usados os mecanismos de aspectos e pontos de corte abstratos para implementar tal tipo de variabilidade;

(iii) **introdução de um conjunto de atributos e métodos em interfaces e classes** – um conjunto de atributos e métodos pode ser introduzido numa interface (usando os mecanismos de declarações inter-tipos de AspectJ), e posteriormente, serem introduzidos em uma dada classe do sistema (através da construção `declare parents` de AspectJ).

Esses mecanismos podem ser combinados quando um dado aspecto de extensão precisa implementar mais do que um tipo de variabilidade. Hannenberg et al [57] apresentam um conjunto de idiomas AspectJ que pode ser utilizado para

implementar e compor esses diferentes mecanismos. Idiomas [19] são padrões de implementação específicos de uma dada linguagem de programação. Os idiomas AspectJ propostos endereçam o projeto e implementação de aspectos que podem ser reutilizados em diferentes aplicações.

A Figura 18 mostra a refatoração do aspecto `RepeatedAllTests` (Seção 4.1.2) em um novo aspecto abstrato, denominado `RepeatedTests`. Ele define dois elementos abstratos: (i) o ponto de corte `repeatedTestsPoints()` – que deve ser especializado para indicar os casos de teste que terão sua execução repetida (linhas 2-3); e (ii) o método `getTimesRepeat()` – que retorna a quantidade de vezes que determinados casos de teste serão repetidos (linha 4). A Figura 19 apresenta dois subaspectos de `RepeatedTests`. O subaspecto `RepeatAllTests` concretiza o ponto de corte `repeatedTestsPoints()` determinando a execução repetida de todos os testes, através do reuso do ponto de corte específico exposto pelo aspecto EJP `TestExecutionEvents`. O subaspecto `RepeatSpecificTestCases` especializa o ponto de corte `testCaseExecutionGeneral()` do aspecto EJP `TestExecutionEvents` para introduzir a funcionalidade de repetição apenas nos casos de teste `TestCaseA` e `TestCaseB`.

Durante a descrição dos estudos de caso (Capítulo 6), serão apresentados diferentes exemplos de ocorrência de variabilidades em aspectos de extensão.


```
01 public abstract aspect RepeatedTests {
02    public abstract pointcut repeatedTestsPoints(
03               Test test, TestResult testResult);
04    public abstract int getTimesRepeat();
05
06    void around(Test test, TestResult testResult):
07            repeatedTestsPoints(test, testResult) {
08       for (int i=0; i < getTimesRepeat(); ++i){
09         proceed(test, testResult);
10       }
11    }
12 }
```

Figura 18. Aspecto de Variabilidade `RepeatedTests`

```
01 public aspect RepeatAllTests extends RepeatedTests {
02   public pointcut repeatedTestsPoints(
03        Test testCase, TestResult result):
04    TestExecutionEvents.testCaseExecution(testCase, result);
05
06    public int getTimesRepeat(){  return 10; }
07 }

01 public aspect RepeatSpecificTestCases extends RepeatedTests {
03    public pointcut repeatedTestsPoints(
04       Test testCase, TestResult result):
05            testCaseExecutionGeneral(result)
06            (target(TestCaseA) || target(TestCaseB));
07            && target(testCase)
08            && !adviceexecution();
09    public int getTimesRepeat(){  return 10; }
10 }
```

Figura 19. Exemplos de Especialização de Aspectos de Extensão

4.3. Sumário

A codificação dos elementos (EJPs e aspectos de extensão) de nossa abordagem em AspectJ traz sistematização para a sua adoção, oferecendo um conjunto concreto de diretrizes de implementação. Os seguintes benefícios podem ser observados como fruto dessas diretrizes: (i) habilita o desenvolvedor a expor um conjunto de pontos de junção que estão espalhados no framework em um conjunto restrito de aspectos, que por sua vez podem ser usados para estender a funcionalidade do framework, através da codificação de aspectos de variabilidade e integração; (ii) permite a especificação de vários contratos a serem satisfeitos quando estendendo pontos de junção do framework. A especificação de tais contratos permite que os mesmos sejam não apenas declarados, mas também garantidos em tempo de compilação e execução; (iii) permite a especificação de variabilidades existentes nos EJPs, através do uso do mecanismo de composição e especialização de pontos de junção; e (iv) permite implementar variabilidades encontradas nos aspectos de extensão, através do uso de herança entre aspectos, métodos e pontos de corte abstratos, e declarações intertipo.

5
Um Modelo Generativo Orientado a Aspectos

Nesse capítulo é apresentado um modelo generativo orientado a aspectos que é usado para instanciação de variabilidades OO e OA encontradas em arquiteturas de famílias de sistemas. Esse modelo generativo compõe nossa abordagem OA para desenvolvimento de frameworks. Seu objetivo central é habilitar a instanciação automática de arquiteturas orientadas a aspectos baseado em informações coletadas por um modelo de características.

5.1.
Visão Geral

Nosso modelo generativo orientado a aspectos contempla o uso de técnicas OA para habilitar a implementação e instanciação de arquiteturas de famílias de sistemas. Ele pode ser visto como uma instanciação do modelo genérico proposto por Czarnecki e Eisenecker [33]. Três modelos são definidos para habilitar a instanciação automática de uma arquitetura de família de sistemas, são eles: (i) modelo de características com relações tranversais – responsável pela especificação e coleta de características a serem instanciadas na arquitetura de família de sistemas; (ii) modelo de arquitetura – define os principais componentes e elementos de implementação que compõem a arquitetura; e (iii) modelo de configuração – especifica o mapeamento entre características e componentes (e respectivos sub-elementos) provenientes dos modelos de característica e arquitetura, respectivamente.

A Figura 20 apresenta as relações existentes entre os diferentes modelos existentes na nossa abordagem. Ela apresenta os modelos sob as perspectivas da engenharia de domínio e de aplicação. Na engenharia de domínio, cada um dos modelos (característica, arquitetura e configuração) é especificado considerando uma arquitetura OA de família de sistemas particular. Durante a instanciação de um membro da família (engenharia da aplicação), uma instância do modelo de característica é definida pelo engenheiro de aplicação. Uma ferramenta usa tal

modelo para customizar e instanciar a arquitetura OA, usando os modelos de arquitetura e configuração. As seções seguintes detalham as atividades de especificação e uso do modelo generativo.

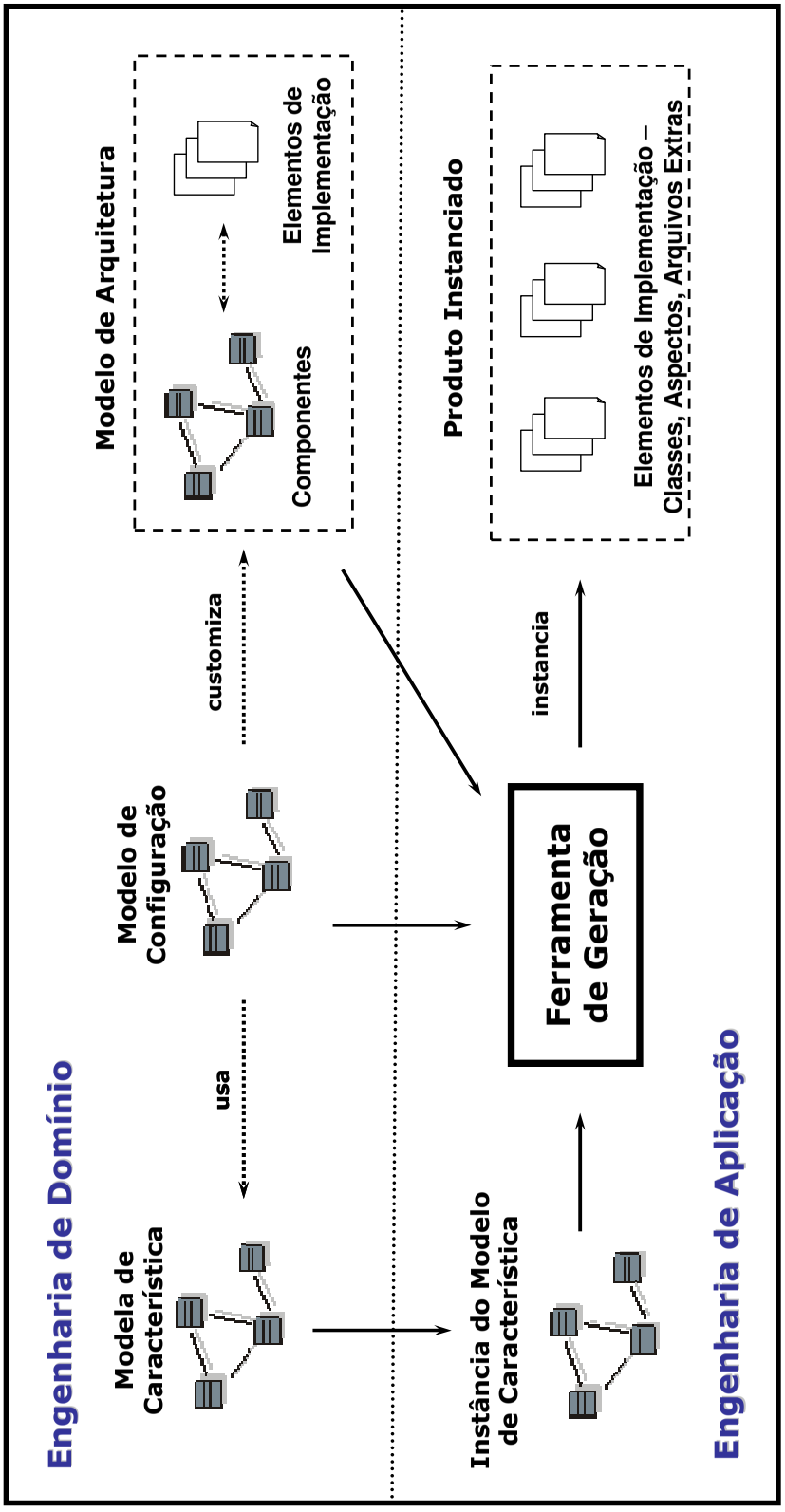

Figura 20. Visão Geral dos Elementos do Modelo Generativo

5.2. Engenharia de Domínio: Definição do Modelo Generativo OA

Nossa abordagem endereça na engenharia de domínio, a produção dos modelos de características, de arquitetura e de configuração que compõem o modelo generativo orientado a aspectos. Esses modelos devem ser produzidos (ou revisitados caso já tenham sido desenvolvidos anteriormente), após a implementação do framework (ou arquitetura de família de sistemas) usando as diretrizes apresentadas no Capítulo 4.

A arquitetura OA do JUnit (apresentada no Capítulo 4) será usada para ilustrar as atividades de definição do modelo generativo de forma a permitir a instanciação automática de suas variabilidades OO e OA. Nas seções seguintes são apresentadas as atividades de concepção dos elementos do modelo generativo, mostrando como as variabilidades do JUnit podem ser preparadas para serem instanciadas automaticamente.

5.2.1. Especificação do Modelo de Arquitetura

A implementação de uma arquitetura de família de sistemas resulta tipicamente em um conjunto de artefatos/elementos de implementação, tais como, classes, interfaces, templates, aspectos e arquivos extra. Uma agregação de vários desses elementos concretiza os componentes definidos anteriormente no projeto de uma arquitetura OA. Nossa abordagem OA para desenvolvimento de frameworks, por exemplo, produz um conjunto de elementos de implementação bem definidos, tais como, núcleo do framework, conjunto de classes abstratas e interfaces que representam os pontos flexíveis do framework, aspectos do núcleo, aspectos EJPs, e aspectos de variabilidade e integração.

A primeira atividade de definição do modelo generativo, é a especificação do seu modelo de arquitetura. Ele permite relacionar os elementos de implementação de uma arquitetura de família de sistemas com a especificação de seus componentes arquiteturais. O propósito principal do modelo de arquitetura é criar uma representação visual dos elementos de implementação de forma a relacioná-los com um modelo de característica que expressa as variabilidades existentes nos componentes. Ele é desenvolvido para ser usado e processado por

uma ferramenta de derivação de membros da família de sistemas. Nosso modelo de arquitetura é formado por um conjunto de componentes. Cada componente é responsável por implementar um conjunto de funcionalidades relacionadas. Eles agregam elementos de implementação, tais como, classes, interfaces, aspectos, arquivos extras e templates resultantes das atividades de projeto e implementação da arquitetura de família de sistemas. Durante a especificação do modelo de arquitetura, esses elementos devem ser agregados em componentes para facilitar o processo de instanciação da arquitetura. Os arquivos extras são aqueles necessários para a implementação do componente, tais como, arquivos de configuração, de imagens, etc. Já os templates são usados para implementar elementos da arquitetura (classes, interfaces, aspectos ou arquivos de configuração) que necessitam ser customizados durante o processo de instanciação da arquitetura. Para auxiliar na modularização de um componente, cada componente pode também ele próprio ser formado por um conjunto de sub-componentes.

Os templates são os únicos elementos que precisam ser implementados para complementar a especificação do modelo de arquitetura. Eles podem ser usados para representar: (i) especializações de classes que representam pontos flexíveis de um framework; (ii) subaspectos que definem implementações concretas para um dado interesse transversal; e (iii) alguma classe, aspecto ou arquivo de configuração que necessita sofrer customizações baseado em informação coletada pelo modelo de característica. Existem diversas tecnologias que possibilitam a implementação de templates, tais como, JET/EMF [18], Velocity[7], XSLT[8], etc.

A Figura 21(c) apresenta uma representação hierárquica do modelo de arquitetura da implementação OA do JUnit. Essa representação foi gerada usando o plugin EMF (*Eclipse Modeling Framework*) [18], a partir da interpretação de um meta-modelo simplificado para representação de modelos arquiteturais.

Como pode ser visto na Figura 21(c), dois componentes principais foram definidos para agregar os elementos de implementação presentes na arquitetura OA do JUnit: core e extensions. O componente core agrega os elementos de implementação que determinam o núcleo do framework, sendo composto por dois sub-componentes: (i) testing - que agrega as classes responsáveis pela definição

[7] The Apache Velocity Project. URL: http://velocity.apache.org/. 2007.

dos testes propriamente ditas; e (ii) runner - que agrega três diferentes sub-componentes cada um responsável pela implementação de uma diferente alternativa de interface gráfica para o JUnit. O componente testing é também responsável por agregar o aspecto EJP `TestExecutionEvents`, assim como os dois templates `TestSuiteTemplate` e `TestCaseTemplate` que são usados para criar, respectivamente, casos e suites de teste para uma dada aplicação.

O componente extensions agrega três sub-componentes (active, repeat, setup) que definem as extensões a serem aplicadas a suítes e casos de teste. Esses componentes definem aspectos abstratos de variabilidade, responsáveis por definir a funcionalidade de uma dada extensão, assim como templates que possibilitam a geração de subaspectos de cada um desses aspectos de variabilidade, sendo eles: (i) `RepeatTestTemplate` – define um subaspecto que será customizado para introduzir código de repetição em casos de teste específicos. Esse template pode ser usado para gerar automaticamente o aspecto concreto `RepeatAllTests` apresentado na Seção 4.2.1 (Figura 19); (ii) `ActiveTestTemplate` – define um subaspecto que será customizado para introduzir a funcionalidade de execução concorrente em suítes de teste específicos; e (iii) `TestDecoratorTemplate` – é usado para permitir a adição de código de configuração no início ou final de casos ou suítes de teste.

5.2.2. Especificação do Modelo de Características

A segunda atividade de definição do modelo generativo é a especificação do modelo de característica da família de aplicações sendo desenvolvida. Um modelo de característica permite representar as características comuns e varíaveis existentes em um dado domínio. Como o propósito do nosso modelo generativo é habilitar a instanciação automática de variabilidades existentes numa dada arquitetura, a especificação de um modelo de característica na nossa abordagem focaliza, principalmente, a representação das características varíaveis existentes em tal arquitetura.

Existem várias ferramentas disponíveis atualmente que oferecem suporte para a criação de modelos de características. Nesse trabalho foi utilizado o modelo

[8] XSL Transformations (XSLT) Specification. URL: http://www.w3.org/tr/xslt. 2007.

de característica proposto em [35], o qual permite modelar características obrigatórias, opcionais e alternativas. Além disso, esse modelo de característica também permite a representação de cardinalidades para especificação da quantidade de ocorrência de características, assim como a criação de propriedades e atributos em cada uma das características definidas. O plugin FMP (*Feature Modelling Plugin*) [6] oferece suporte para a modelagem dessa proposta de modelo de característica.

A Figura 21(a) apresenta o modelo de características com as variabilidades do framework JUnit modeladas usando o plugin FMP. Ele é composto de três características principais: Testing, Runner e Extensions. A característica Testing é modelada como obrigatória, porque ela deve existir em qualquer instanciação do JUnit. Ela é usada para especificar as suítes e casos de teste que serão criados numa dada instanciação. A característica Runner modela as três alternativas (TXT, AWT e Swing) de interface gráfica que são oferecidas pelo framework, apenas uma delas deve ser escolhida. Finalmente, a característica opcional Extensions permite definir diferentes extensões a serem aplicadas a suítes e casos de teste.

Nossa abordagem define uma extensão simples para o modelo de característica com o objetivo de permitir a customização de aspectos presentes no modelo de arquitetura. Nossa extensão define duas propriedades, denominadas `<<crosscutting>>` e `<<joinpoint>>`, as quais podem ser atribuídas a determinadas características. Uma característica `<<crosscutting>>` é usada para representar aspectos do modelo de arquitetura que podem estender o comportamento de outras características do sistema. Uma característica `<<joinpoint>>` é usada para representar pontos de junção específicos de classes (ou aspectos) do sistema, os quais são candidatos a serem estendidos por aspectos no espaço de solução. Relações transversais podem ser definidas entre esses dois tipos de características, durante o processo de instanciação automática, para permitir a customização de pontos de corte de aspectos para afetar classes/métodos específicos do sistema. Na nossa abordagem, características `<<crosscutting>>` e `<<joinpoint>>` são mapeadas, respectivamente, para os seguintes elementos de implementação: aspectos de extensão e pontos de corte existentes em EJPs.

A Figura 21(a) mostra exemplos de tais propriedades atribuídas ao modelo de característica do JUnit. As características RepeatedTest, ConcurrentTest e Configuration Test são modeladas como sendo <<crosscutting>>, porque podem ser aplicadas a outras características difusas no modelo de características. Já as características Test Suite e Test Case são modeladas como sendo <<joinpoint>>, porque são candidatas a serem estendidas por características <<crosscutting>>. Durante o processo de engenharia de aplicação, são definidas relacões tranversais, que indicam como as características Extensions são aplicadas às características Test Suite e Test Case.

5.2.3. Especificação do Modelo de Configuração

A atividade de especificação do modelo de configuração permite relacionar elementos dos modelos de característica e arquitetura definidos anteriormente. O modelo de configuração é usado para definir como uma configuração específica de características deve ser mapeada para uma configuração de componentes da arquitetura. Dessa forma, ele representa a especificação do conhecimento de configuração existente no desenvolvimento generativo [33]. A especificação de modelos de configuração possibilita entender, modificar e evoluir o conhecimento de configuração de forma independente do espaço de problema (modelo de característica) e do espaço de solução (modelo de arquitetura).

Nosso modelo de configuração é composto por três diferentes elementos: (i) relações de dependência entre componentes (e elementos de implementação) provenientes do modelo de arquitetura e características presentes no modelo de característica; (ii) relações tranversais válidas entre características <<crosscutting>> e <<joinpoint>>; e (iii) especificação do mapeamento entre características <<joinpoint>> e pontos de junção concretos existentes em classes do modelo de arquitetura. A Tabela 4 apresenta os elementos de nosso modelo de configuração e seu respectivo propósito para o processo de instanciação. Os parágrafos seguintes detalham cada um desses elementos.

As relações de dependência entre os componentes e sub-elementos (tais como classes, aspectos, templates e arquivos extras) e características são usadas para especificar quais componentes e sub-elementos devem ser instanciados

quando um conjunto de características é selecionado na etapa de engenharia de aplicação. Elas representam a concretização de modelos de decisão [117]. A Tabela 5 descreve regras de mapeamento entre: (i) diferente tipos de características que podem ser definidas; e (ii) elementos de implementação de nossa abordagem OA para desenvolvimento de frameworks. Essas regras de mapeamento podem ser usadas como base para derivar as relações de dependência entre elementos de implementação e características.

Elemento do Mod. de Configuração	Propósito Principal
Relações de Dependência entre Elementos de Implementação e Características	• Escolha de Variabilidades
Relações Transversais Válidas entre Características `<<crosscutting>>` e `<<joinpoint>>`	• Restringir Relações Transversais no Espaço de Problema
Mapeamento entre Características `<<join point>>` e pontos de junção concretos de EJPs	• Customização de Pontos de Corte de Aspectos

Tabela 4. Elementos do Modelo de Configuração

Além de usar as regras de mapeamento, as seguintes diretrizes podem ser usadas para especificar tais relações de dependência: (i) se um componente (ou sub-elemento) deve ser instanciado para todo membro da família de sistemas, nenhuma relação de dependência partindo do mesmo deve ser especificada; (ii) se um componente (ou sub-elemento) depende da ocorrência de uma característica específica, uma relação de dependência deve ser criada entre eles. As relações de dependência são usadas por uma ferramenta de geração de código para decidir quais módulos serão incluídos na aplicação sendo instanciada, baseado em uma instância do modelo de característica definido por engenheiros de aplicação.

As seguintes diretrizes são usadas para especificar tais relações de dependência: (i) se um componente (ou sub-elemento) deve ser instanciado para

todo membro da família de sistemas, nenhuma relação de dependência partindo do mesmo deve ser especificada; (ii) se um componente (ou sub-elemento) depende da ocorrência de uma característica específica, uma relação de dependência deve ser criada entre eles. As relações de dependência são usadas por uma ferramenta de instanciação para decidir quais módulos serão incluídos na aplicação sendo instanciada, baseado em uma instância do modelo de característica especificado por engenheiros de aplicação. No caso específico de elementos de implementação do tipo template, as relações de dependência definem se eles serão processados e incluídos no produto final gerado. Todo template depende necessariamente de uma característica, a qual oferece informação necessária para a sua instanciação.

Tipo de Característica	**Elemento de Implementação**
Característica Obrigatória	• Núcleo do Framework • Aspectos do Núcleo
Característica Alternativa Existente no Núcleo[9]	• Classes de Pontos de Extensão (*Hot-Spots*) do Núcleo do Framework
Característica `<<joinpoint>>`	• EJPs (aspectos)
Característica Opcional	• Aspectos de Integração e Variabilidade
Característica Alternativa `<<crosscutting>>`	• Aspectos de Integração e Variabilidade

Tabela 5. Regras de Mapeamento entre Características e Elementos de Implementação

A Figura 21(b) ilustra as diferentes relações de dependência existentes no modelo de configuração do JUnit. O modelo mostra que cada um dos sub-componentes do componente runner (textui, awtui, swingui) e seus respectivos elementos de implementação (classes, interfaces, etc) serão instanciados baseado na alternativa selecionada para a característica Runner. Como podemos ver na Figura 21, todo template definido para o JUnit (seção 5.2.1) possui uma relação de dependência com elementos do modelo de característica. Os templates `TestCaseTemplate` e `TestSuiteTemplate`, por exemplo, dependem das características Test Case e Test Suite, respectivamente. Isso significa que para cada característica desse tipo que for solicitada pelo engenheiro de aplicação, uma nova classe de caso ou suíte de teste será criada, como fruto do processamento dos respectivos templates. Informação coletada pelo modelo de característica é usada durante esse processamento. O nome de cada caso ou suíte de teste definido no modelo de característica, por exemplo, será usado para nomear cada classe de teste (suíte ou caso de teste) a ser criada.

Embora as relações de dependência sejam usualmente especificadas para relacionar um determinado componente ou elemento de implementação a apenas uma característica, relações de dependência mais complexas podem também ser definidas, caso necessário, para permitir relacionar componentes ou elementos de implementação a um conjunto de características organizadas em expressões booleanas. Tais expressões booleanas podem ser usadas para indicar que um dado componente ou elemento de implementação depende: (i) da ocorrência de um conjunto de duas ou mais características; ou (ii) da ocorrência de determinadas características e não ocorrência de outras, simultaneamente.

Nosso modelo de configuração também define um conjunto de relações válidas que podem existir entre características `<<crosscutting>>` e `<<joinpoint>>`. Na prática essa informação é usada para especificar quais aspectos (representados por características `<<crosscutting>>`) podem afetar quais pontos de junção de classes/aspectos do sistema (representados por características `<<joinpoint>>`). Assim, uma ferramenta de instanciação pode usar tal informação para checar se engenheiros de aplicação estão especificando

[9] Tais características exigem a instanciação de no mínimo uma das alternativas existentes.

relacionamentos válidos entre características transversais e pontos de junção. A Tabela 6 mostra as relações transversais válidas para o estudo de caso do JUnit. Ela especifica que: (i) a característica <<crosscutting>> Repeat pode apenas estender o comportamento da característica <<joinpoint>> Test Case; (ii) a característica Concurrent Execution pode modificar apenas o comportamento da característica Test Suite; e (iii) a característica Test Setup pode estender ambas as características Test Case e Test Suite.

O terceiro e último elemento que deve ser definido no modelo de configuração é o mapeamento entre características <<joinpoint>> e pontos de junção concretos presentes em classes/aspectos do modelo de arquitetura. Essa informação é usada por uma ferramenta de instanciação para customizar pontos de corte durante a geração do código de aspectos. O mapeamento envolve a identificação de quais trechos de código de classes (ex: execução ou chamada de métodos e construtores) correspondem a uma dada característica <<joinpoint>>. Se todos os aspectos de extensão têm pontos de corte fixos, não há necessidade de especificar esse mapeamento. Na nossa abordagem, os pontos de junção concretos podem ser diretamente encontrados e extraídos dos aspectos EJP definidos para nossa arquitetura de família de sistemas. A Tabela 6 mostra o mapeamento das características <<joinpoint>> para pontos de junção concretos expostos pelos EJPs do framework JUnit.

Modelo de Configuração	Framework JUnit
Relações Transversais Válidas	• Repeat feature <<crosscuts>> Test Case feature • Concurrent Execution <<crosscuts>> Test Suite feature • Test Setup <<crosscuts>> Test Suite feature • Test Setup <<crosscuts>> Test Case feature
Mapeamento entre Características <<joinpoint>> e pontos de corte de EJPs	• Test Case feature <<maps>> TestExecutionEventsEJP. testCaseExecution(…); • Test Suite feature <<maps>> TestExecutionEventsEJP.testSuiteExecution(…);

Tabela 6. Elementos do Modelo de Configuração do JUnit.

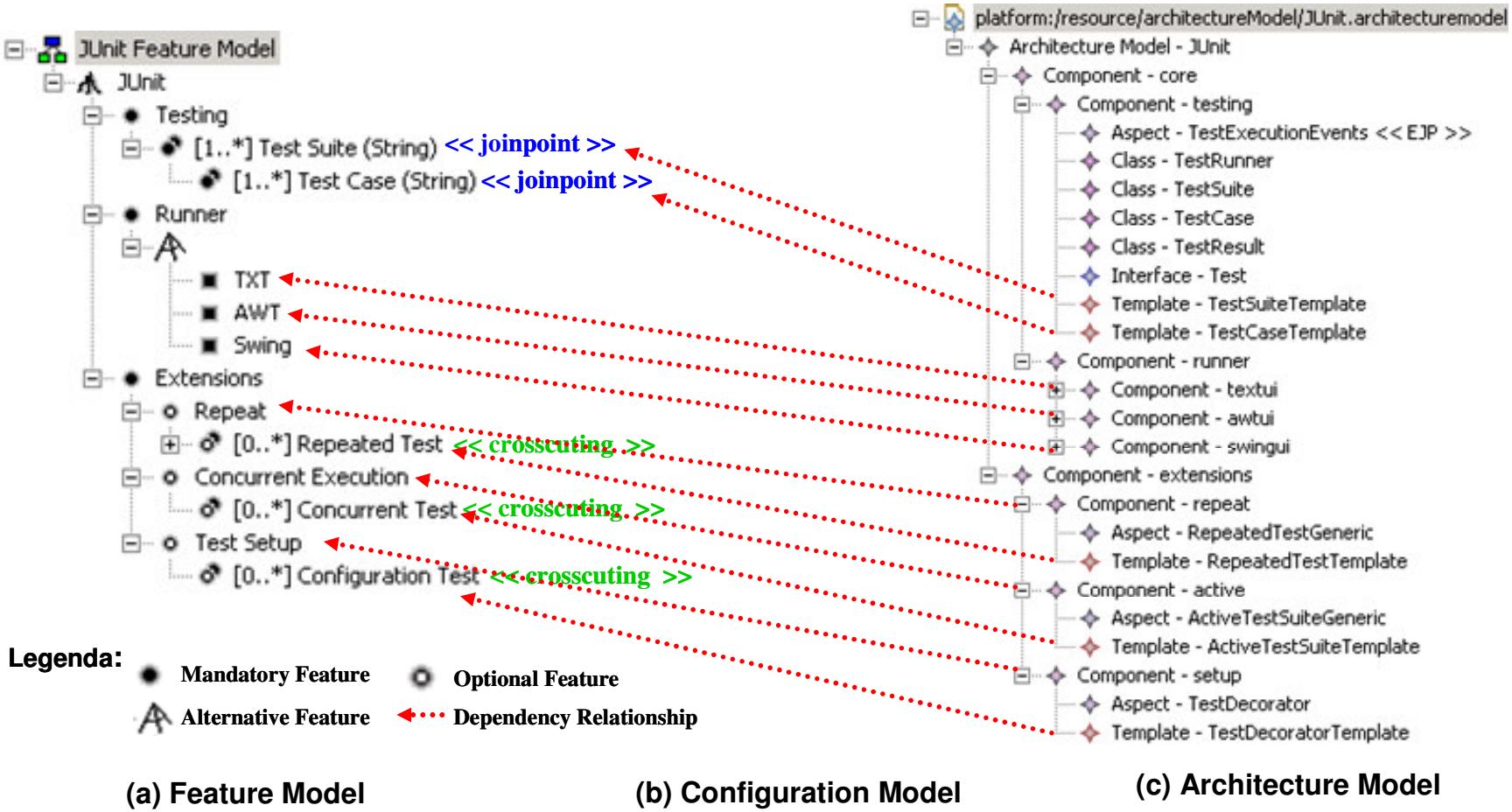

Figura 21. Modelo Generativo do Framework JUnit

5.2.4. Codificação de Templates

A última atividade de definição do modelo generativo OA é a codificação de templates. Conforme mencionado anteriormente, os templates são usados para codificar elementos de implementação que precisam ser customizados durante a instanciação de um membro da família de sistemas. Embora a estrutura geral dos templates possa ser definida durante a atividade de especificação do modelo de arquitetura, sua codificação só pode ser completamente realizada, após a especificação dos modelos de característica e de configuração. Isso ocorre porque estes últimos modelos oferecem informações relevantes para a especificação dos templates. O modelo de configuração indica as características que o template depende para ser instanciado. O modelo de característica por sua vez é usado para coletar os dados que auxiliam na customização das variabilidades presentes no template.

Existem muitas ferramentas disponíveis que implementam a tecnologia de templates [34]. Em uma implementação protótipo [79] de uma ferramenta de instanciação, a tecnologia *Java Emitter Templates* (JET) foi utilizada na codificação de nossos templates. JET é o motor de templates do plugin *Eclipse Modelling Framework* (EMF). Ele pode ser usado para implementar templates que representem qualquer tipo de elemento, tais como, código de classes e aspectos ou arquivos de configuração. A Figura 22 mostra um exemplo de implementação do template `TestSuiteTemplate` usando o JET. O template contém inicialmente seu código de configuração (linhas 1-5). A varíavel `testSuite` do tipo `Feature`[10] é usada para armazenar uma referência para uma sub-árvore de objetos que representam as informações coletadas pelo modelo de característica. No caso específico do template `TestSuiteTemplate`, como no modelo de configuração foi definido que ele depende de características Test Suite, o gerador de código irá processar esse template para cada característica desse tipo que for encontrada e passará como informação toda a sub-árvore definida por tais características. A implementação restante do template define como será o código gerado para a

[10] O tipo `Feature` é proveniente do meta-modelo do modelo de característica, sendo capturado pela ferramenta de instanciação e repassado para o template durante seu processamento.

classe de suíte de teste, oferecendo possíveis trechos de customização, tais como: (i) o nome da classe de suíte de teste (linha 9); e (ii) quais classes de caso de teste serão incluídas em tal suíte (linhas 13-18). Como pode ser observado no exemplo apresentado na Figura 22, diretivas específicas do JET são usados para processar as informações provenientes do modelo de características, tais como: (i) `<%= %>` – para acessar o valor de uma dada variável (linhas 9, 12 e 17); e (ii) `<% %>` – para definir um conjunto de comandos Java a serem executados (linhas 5 e 13-18). A Seção 5.3.3 apresenta em detalhes um algoritmo de customização do modelo de arquitetura baseado nos modelos de característica e configuração. Templates de aspectos devem também usar a informação sobre o mapeamento de ponto de junção do modelo de configuração para customizar pontos de corte[11].

```
01 <%@ jet package="translated"
02    imports="org.eclipse.emf.common.util.Elist ..."
03    class="TestSuiteTemplate" %>
04
05 <% FeatureElement testSuite = (FeatureElement) argument;%>
06 import junit.framework.Test;
07 import junit.framework.TestSuite;
08
09 public class <%=testSuite.getName()%>TestSuite {
10
11   public static Test suite(){
12     TestSuite suite = new TestSuite("<%=testSuite.getName()%>");
13     <% EList features = testSuite.getChildren();
14       for (Iterator iter=features.iterator(); iter.hasNext();){
15         FeatureElement testCase= (FeatureElement) iter.next(); %>
16         suite.addTest(
17               new TestSuite(<%=testCase.getName()%>Test.class));
18     <% } %>
19     return suite;
20   }
21 }
```

Figura 22. Template `TestSuiteTemplate`

[11] Essa informação de mapeamento deve também ser repassada pela ferramenta de instanciação ao template. Uma opção no exemplo acima, seria incluir um método chamado

5.3. Engenharia de Aplicação: Instanciação da Arquitetura OA

Durante a etapa de engenharia de aplicação, desenvolvedores solicitam a geração de uma instância da arquitetura OA. Esse processo é feito a partir da escolha das diferentes variabilidades presentes na arquitetura e que estão expressas no modelo de característica. Duas atividades principais estão presentes nessa etapa: (i) escolha das variabilidades em uma instância do modelo de característica; e (ii) escolha de relações tranversais válidas entre características. Uma ferramenta de instanciação usa a informação coletada nessas atividades e o modelo de configuração para gerar uma instância da arquitetura de família de sistemas, a partir da customização do modelo de arquitetura.

5.3.1. Escolha de Variabilidades no Modelo de Característica

A primeira atividade da engenharia de aplicação é a escolha de variabilidades e definição de seus respectivos parâmetros dentro do modelo de característica. Uma instância do modelo de característica é criada para possibilitar tal escolha. Essa instância representa um membro da família de sistemas que será solicitado à instanciação. Diversas ferramentas disponíveis na indústria, tais como, pure::variants[12], Gears[13] e FMP [6], suportam a criação de instâncias de modelos de características.

A Figura 23 mostra uma instância do modelo de características do JUnit. Nessa instância, o engenheiro de aplicação solicita duas suítes de teste (`TestSuiteModule1` e `TestSuiteModule2`) cada um com 2 casos de teste (`TestCaseA`, `TestCaseB`, `TestCaseC` e `TestCaseD`). A interface Swing é escolhida para a característica Runner. Finalmente, o engenheiro também define duas extensões a serem aplicadas a casos e/ou suítes de teste, são elas: (i) duas características Repeated Test com atributos de repetição (Quantity) 10 e 5; e (ii) uma característica Concurrent Test.

`getJoinPointFeatures()` na classe `Feature` de forma a poder acessar as características `<<join point>>` que uma dada característica `<<crosscutting>>` afeta.

[12] Pure::Variants. URL: http://www.pure-systems.com/, 2007.

[13] Big Lever. Gears. URL: http://www.biglever.com. 2007.

5.3.2.
Escolha de Relações Transversais no Modelo de Característica

Uma vez escolhida as variabilidades, através da especificação de uma instância do modelo de característica, a próxima atividade envolve a escolha de relações tranversais entre características `<<crosscutting>>` e `<<join point>>`. Essas relações transversais são usadas para habilitar a customização de pontos de corte de aspectos. Elas definem, portanto, como será a composição dos aspectos com elementos do núcleo do framework. Nossa proposta é permitir a especificação de tais relações tranversais, de forma separada da escolha das variabilidades, dando mais organização, simplicidade e sistematização para cada uma das atividades. A especificação visual de tais relações tranversais pode ser feita através do uso de diagramas ou mesmo *wizards* que permitam conectar características `<<crosscutting>>` às características `<<join point>>`.

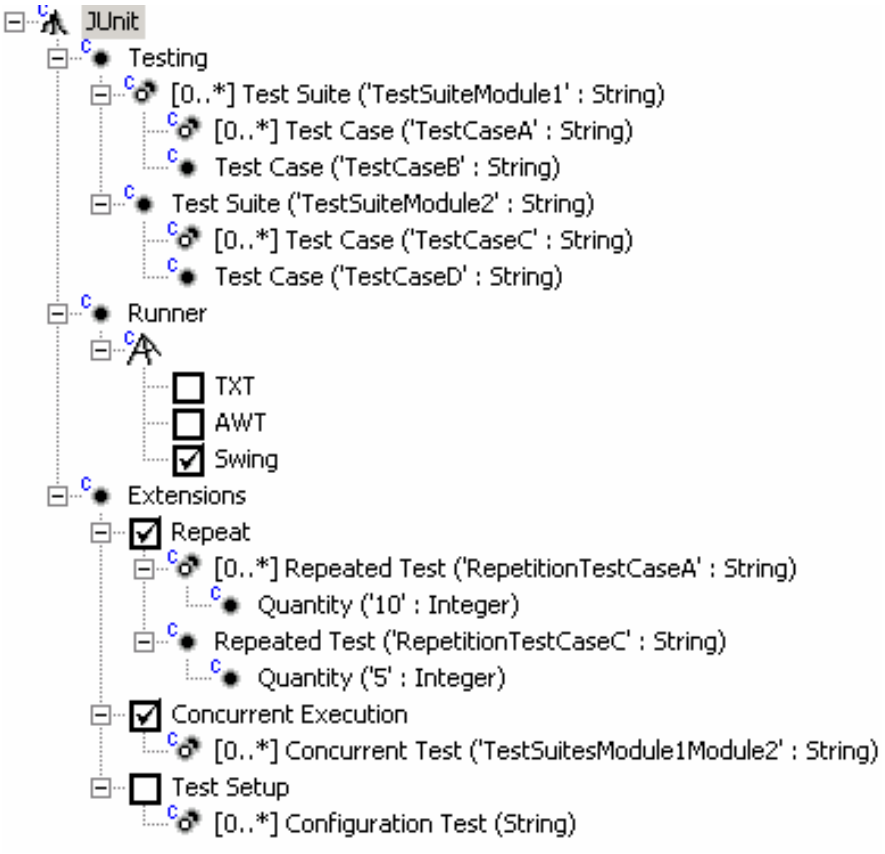

Figura 23. Instância do Modelo de Característica

A Figura 24 mostra a definição de relações transversais considerando a instância do modelo de característica do JUnit apresentada na seção anterior. Duas características <<crosscutting>> Repeated Test são aplicadas individualmente a duas características <<join point>> Test Case definidas. Já a característica <<crosscutting>> Concurrent Test é aplicada às duas suítes de teste solicitadas pelo engenheiro de aplicação.

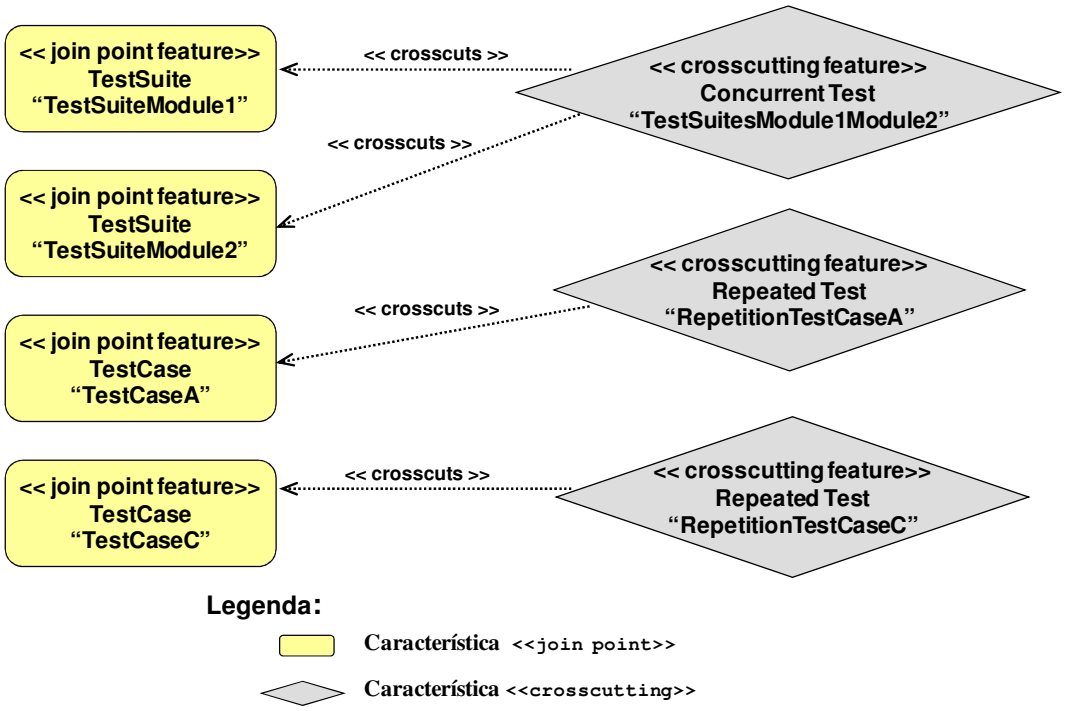

Figura 24. Definição de Relações Transversais

5.3.3. Algoritmo de Instanciação de Arquiteturas OA

A atividade final da engenharia de domínio é a solicitação de geração do código da arquitetura de um membro da família de sistemas, a partir de uma instância do modelo de característica e dos modelos de arquitetura e de configuração especificados para a família. Dessa forma, uma ferramenta de instanciação deve ser codificada para processar cada um desses modelos[14].

[14] Vale ressaltar que o processamento dos modelos de característica, arquitetura e configuração pela ferramenta de instanciação envolve o entendimento de seus respectivos meta-modelos como fonte de informação para navegar na estrutura de cada modelo criado.

Durante o processo de instanciação da arquitetura, a ferramenta deve realizar uma seqüência de 3 passos principais:

(i) *detecção de relações tranversais inválidas* – inicialmente, deve ser verificado se alguma relação transversal entre características que foi criada é inválida[15]. A detecção de relações transversais inválidas deve interromper o processo de geração, de forma a permitir que o engenheiro de aplicação resolva tal inconsistência;

(ii) *processamento do modelo de arquitetura* – no passo seguinte, o modelo de arquitetura é processado pela ferramenta da seguinte forma: para cada componente encontrado a ferramenta de instanciação verifica no modelo de configuração se ele depende de uma característica específica. Caso isso ocorra, a ferramenta apenas instancia tal componente (e processa seus respectivos sub-elementos) se existir uma ocorrência daquela característica na instância do modelo repassada para o gerador. Os componentes são instanciados através da criação de um pacote Java correspondente. Quando processando os elementos de implementação de um componente (classes, interfaces, aspectos, templates e arquivos extra) o mesmo processo é usado. Isso significa que a instanciação de cada elemento de implementação depende da ocorrência de características que eles eventualmente dependem. A instanciação de cada elemento de implementação implica em: (I) recuperá-lo de algum repositório ou do sistema de arquivos contendo todos os artefatos da família de sistemas; e (ii) carregá-lo em um projeto dentro de um ambiente integrado de desenvolvimento (IDE). Componentes e elementos de implementação que não possuem dependências de características são sempre instanciados. Elementos do tipo template devem sempre depender de alguma característica. Conseqüentemente, eles são processados para cada ocorrência daquela característica. Durante o processamento do template, a informação sobre a característica (e respectivas sub-características) da qual ele depende é usada para contemplar a customização do mesmo.

(iii) *processamento de relações transversais* – a ferramenta de instanciação usa as relações transversais entre características para determinar quais aspectos devem afetar quais classes do sistema. Templates de aspectos são definidos para representar aspectos com pontos de corte a serem customizados. Todo template

que representa um aspecto deve necessariamente depender de uma característica <<crosscutting>>. Dessa forma, quando ele é processado, o template pode obter a informação de quais características <<join point>> estão sendo afetadas pela respectiva característica <<crosscutting>> que o representa. De posse das características <<join point>> que se deve afetar, o template de aspecto usa o mapeamento entre características <<join point>> e pontos de corte concretos definidos no modelo de configuração, para realizar a geração dos pontos de corte dos aspectos.

Vamos considerar, por exemplo, a instanciação do framework JUnit, como resultado do processamento: dos modelos de característica, arquitetura e configuração da Figura 21; e da instância de modelo de característica da Figura 23. A Figura 25 apresenta as classes e aspectos gerados durante o processo de instanciação. Dessa forma, quando esses modelos são processados por uma ferramenta, o framework JUnit é instanciado da seguinte forma:

(i) criação de dois suítes de teste (TestSuiteModule1 e TestSuiteModule2) cada um referenciando duas classes de teste (TestCaseA, TestCaseB, TestCaseC e TestCaseD), através do processamento, respectivamente, dos templates TestSuiteTemplate e TestCaseTemplate;

(ii) inclusão das classes do componente swingui e não inclusão das classes dos componentes awtui e txtui;

(iii) criação de dois aspectos que implementam a característica de repetição para dois diferentes casos de teste. Esses aspectos são criados a partir do processamento do template RepeatTestTemplate. Os pontos de corte de tais aspectos são customizados baseados em informação de mapeamento de características em pontos de corte definidos no modelo de configuração;

(iv) inclusão dos aspectos ActiveTest e RepeatedTest e não inclusão do aspecto TestDecorator;

(v) criação de um aspecto que entrecorta as classes de suítes de teste (TestSuiteModule1 e TestSuiteModule2) para executá-las em *threads* separadas. O template ActiveTestTemplate é usado nesse processo..

[15] Dependendo da forma de modelagem visual das relações transversais isso pode ser garantido em tempo de especificação durante o processo de escolha das relações transversais (Seção 5.3.2)

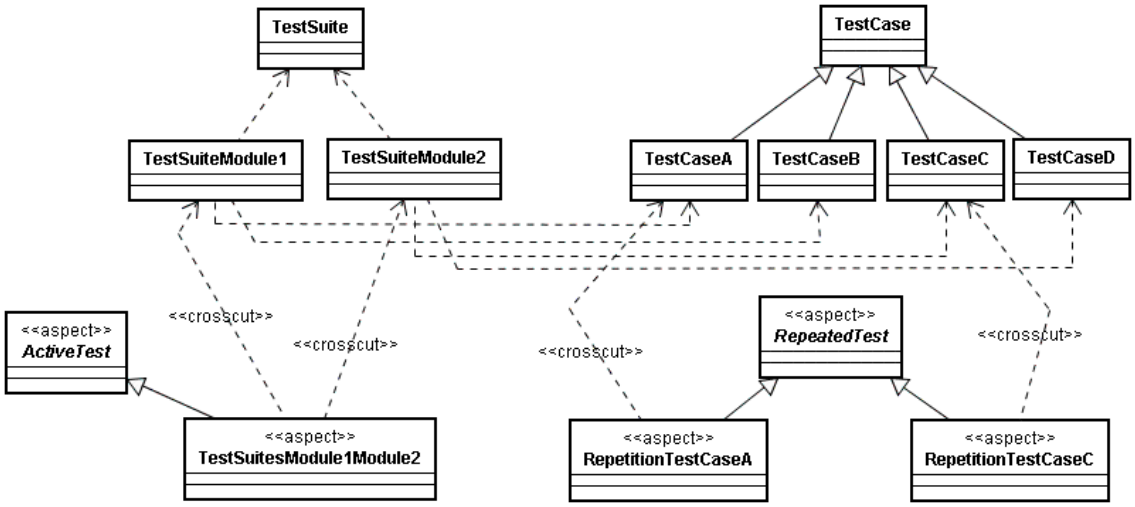

Figura 25. Classes e Aspectos Gerados para uma Configuração do JUnit

5.4. Sumário

Técnicas generativas vêm sendo cada vez mais adotadas no desenvolvimento de aplicações e famílias de sistemas. Neste capítulo foi apresentado um modelo generativo que incorpora abstrações de DSOA. O objetivo central do modelo é habilitar a instanciação de variabilidades OO e OA presentes em arquiteturas de famílias de sistemas e frameworks, usando como base modelos de característica. Uma implementação protótipo do modelo generativo foi implementada na criação de uma abordagem generativa OA para sistemas multi-agentes [77, 79]. A ferramenta foi implementada como um *plugin* da plataforma Eclipse usando como base: (i) o EMF (*Eclipse Modeling Framework*) – para criação e manipulação dos modelos de arquitetura e característica; e (ii) o JET (*Java Emitter Template*) – para criação e processamento de templates. Tal implementação não contemplava ainda a criação explícita do modelo de configuração. Uma nova ferramenta está sendo desenvolvida como parte das pesquisas de continuidade dessa tese de doutorado [26]. Novas tecnologias para desenvolvimento dirigido por modelos [114] disponíveis na plataforma Eclipse [109], tais como o OpenArchitectureWare[16], estão sendo adotadas. O capítulo seguinte apresenta três estudos de caso, os quais exemplificam o uso do modelo generativo.

[16] OpenArchitectureWare. URL: http://www.eclipse.org/gmt/oaw/. 2007.

6
Estudos de Casos

Este capítulo apresenta diferentes estudos de caso de frameworks que foram projetados e implementados usando a abordagem orientada a aspectos proposta na tese. Para cada um dos frameworks são apresentados: (i) seu núcleo; (ii) seus pontos de junção de extensão (EJPs); (iii) seus aspectos de variabilidade e integração; e (iv) o modelo generativo OA de cada um deles.

6.1.
Framework Measurement

Measurement [17] é um framework orientado a objetos que endereça a automação de processos de medições e controle de qualidade de produtos em sistemas de manufatura. Um dos objetivos do framework é categorizar um conjunto de produtos que estão sob análise, de forma a classificá-los em aceitáveis de acordo com critérios de qualidade bem definidos. O framework *Measurement* foi implementado como parte de um estudo sistemático [78] de avaliação do uso da tecnologia de aspectos na composição entre frameworks OO. Seu projeto e implementação foi baseado em documentação publicada anteriormente [17].

6.1.1.
Núcleo do Framework

O processo de controle de qualidade do framework *Measurement* define um ciclo de análise de um item de produto. Esse ciclo de análise representa a funcionalidade principal do núcleo do framework. A Figura 26 apresenta as principais classes do núcleo do framework *Measurement*. As classes que representam pontos de extensão (*hot-spots*) são destacadas. A seguir as classes do framework são relacionadas com a funcionalidade definida para os ciclos de análise.

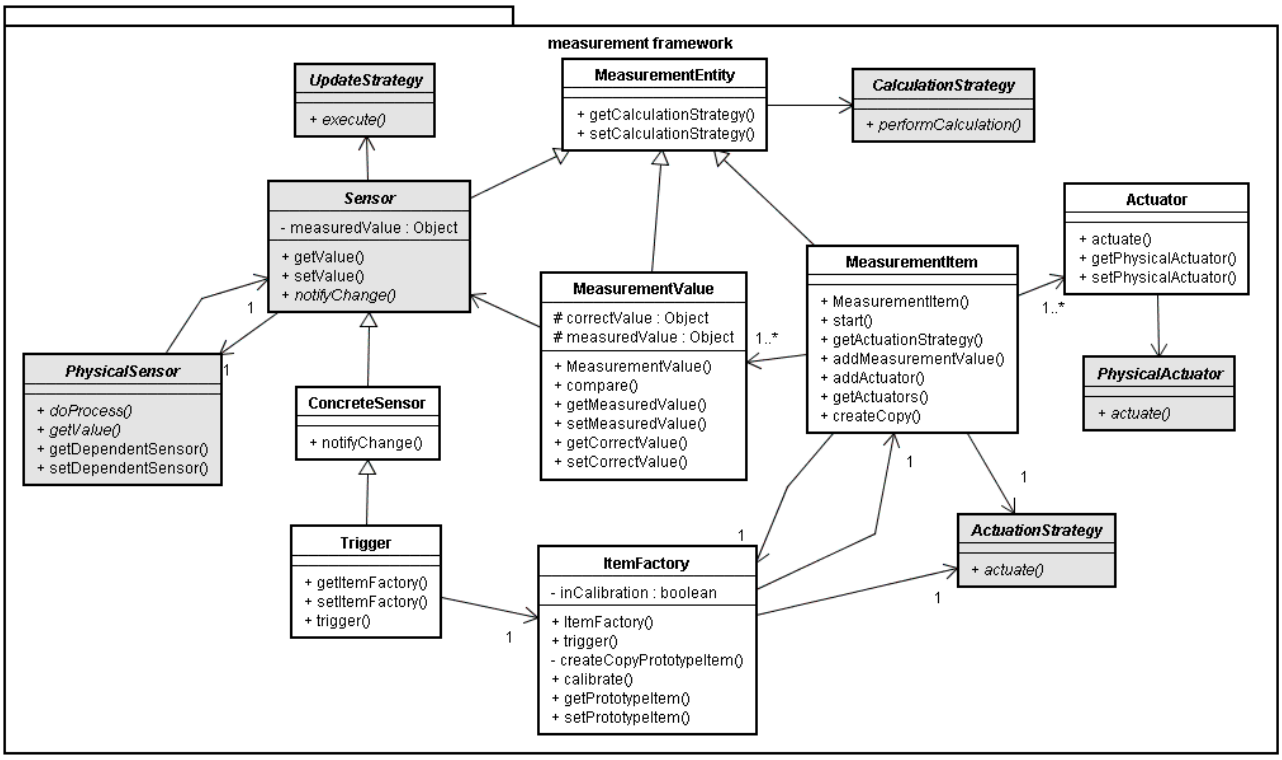

Figura 26. Diagrama de Classes do Framework Measurement

Cada ciclo de análise de um item de produto é composto pelos seguintes passos: (i) etapa de coleção de dados – um *Trigger* (classe `Trigger`) determina que um dado item de produto está entrando no sistema de medição, e a partir daí, sensores (classes `PhysicalSensor`, `Sensor` e `UpdateStrategy`) coletam propriedades relevantes do item de produto sendo avaliado; (ii) etapa de análise – os dados coletados (classe `MeasurementValue`) pelos sensores são então convertidos para uma representação comum e, em seguida, são comparados com valores ideais esperados (através das classes `MeasurementItem` e `CalculationStrategy`). Baseado nessa comparação, os itens medidos são então classificados; (iii) etapa de ação – nessa etapa ações são tomadas de acordo com a classificação dos itens (classes `Actuator`, `PhysicalActuator` e `ActuationStrategy`). Diversos tipos de ações podem ser executadas, tais como, tirar o item da linha de produção ou emitir um rótulo sobre sua qualidade. A Figura 27 mostra um diagrama de seqüência UML, ilustrando as colaborações entre as classes do framework para endereçar a implementação de um ciclo de medição.

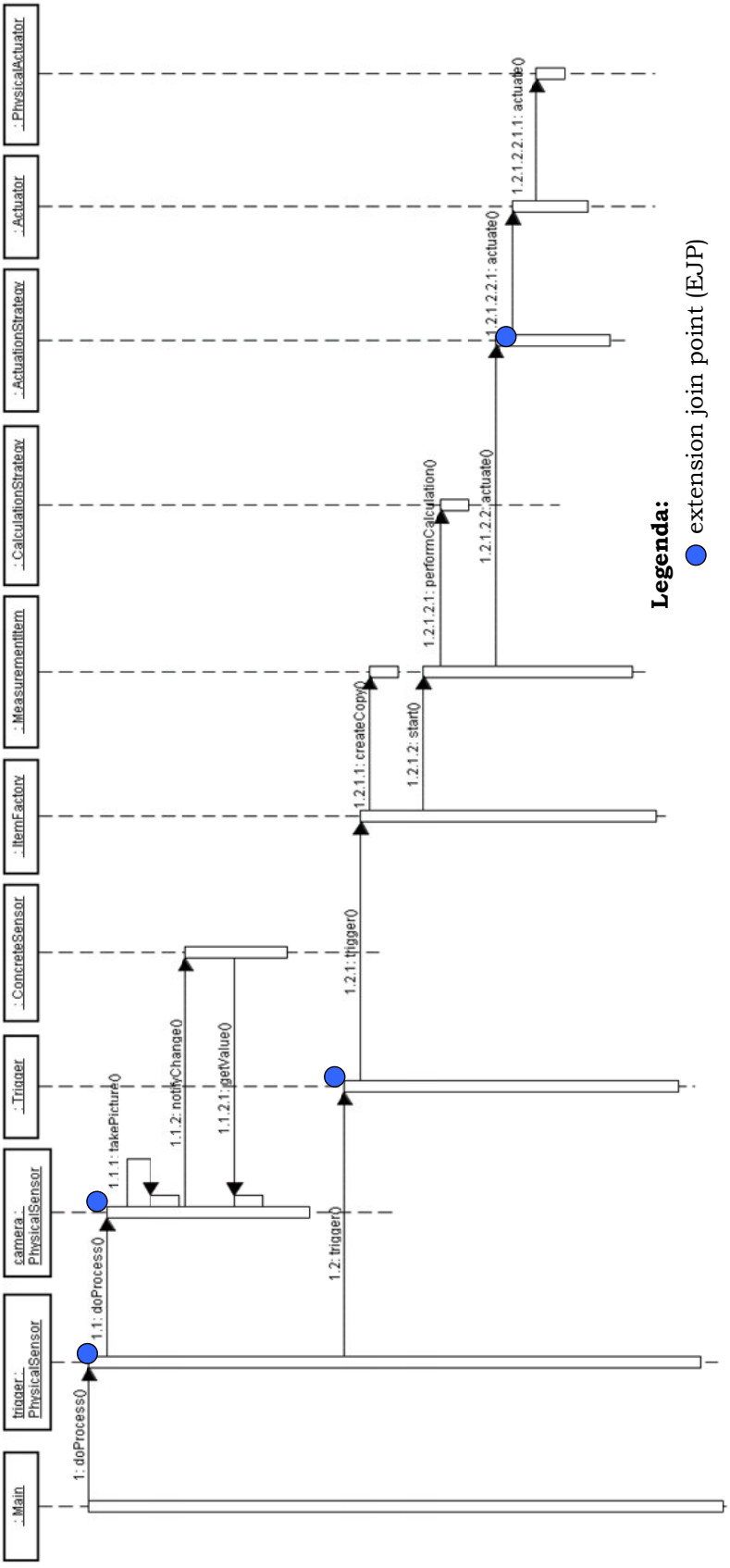

Figura 27. Diagrama de Seqüência do Framework Measurement

A Figura 28 mostra um exemplo de instância do framework para um sistema de análise de garrafas de cerveja (AGC), anteriormente apresentado em [17]. As seguintes classes foram criadas com tal propósito: (i) `Main` - classe principal da aplicação responsável por configurar e inicializar o framework; (ii) `BeerCanCamera` e `BeerCanTrigger` - subclasses de `PhysicalSensor` que representam, respectivamente, uma câmera e um trigger físico que atuam no processo de análise de cervejas; (iii) `BeerCanActuationStrategy` - define uma estratégia de atuação para o processo de análise que determina a ativação de atuadores em situações específicas; (iv) `BeerCanActuator` - representa um atuador físico concreto do sistema de AGC; (v) `Picture` - representa uma foto tirada pelo sensor; e (vi) `BeerCanCalculationStrategy` - determina o algoritmo para processamento de cada item medido pelos sensores.

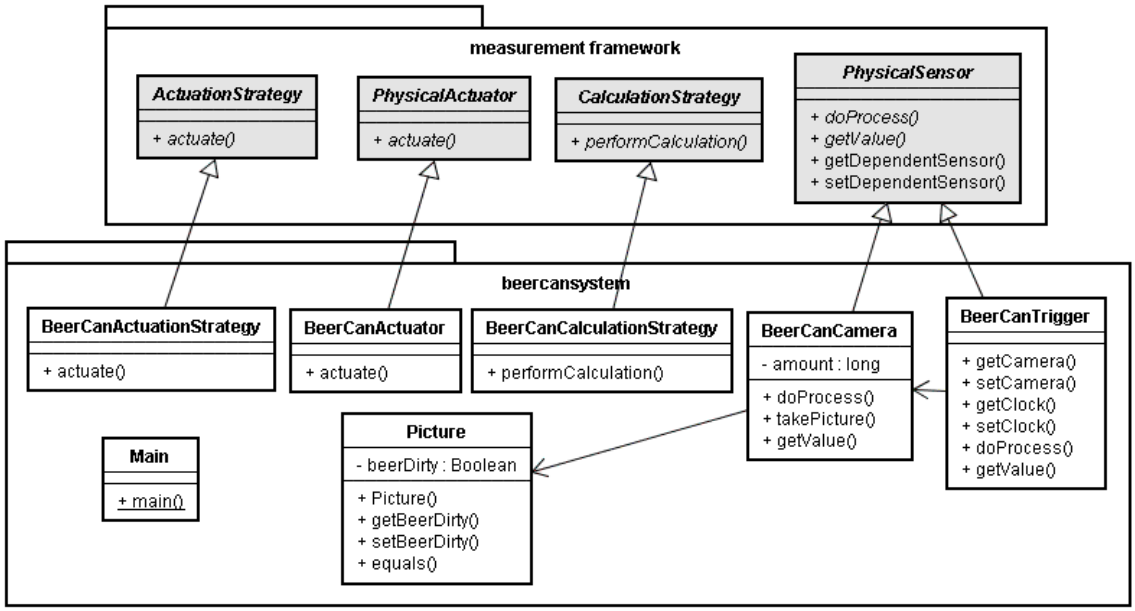

Figura 28. Exemplo de Instância do Framework *Measurement*

6.1.2. Pontos de Junção de Extensão

Durante a realização do estudo de composição do framework *Measurement* com outros frameworks [ref], foram identificadas que diversas composições tinham uma natureza transversal, requerendo dessa forma a interceptação de eventos internos específicos acontecendo na execução do framework

Measurement. Assim, os seguintes pontos de junção de extensão (EJPs) foram definidos para o framework *Measurement*;: (i) ativação de *triggers* – que representa o evento de inicialização do processo de análise da qualidade do produto; (ii) ativação de sensores – determina o evento de coleta de dados do produto; e (iii) ativação de acionadores (*actuators*) – esse evento representa o passo final de análise da qualidade do produto. Para expor esses pontos de junção de extensão, o aspecto EJP `MeasurementEvents` foi implementado em AspectJ, contendo três diferentes pontos de corte, que interceptam a execução de classes do framework. A Figura 29 mostra o código do aspecto EJP `MeasurementEvents`. A Figura 27 destaca no diagrama de seqüência do framework *Measurement*, os pontos de junção de extensão (EJPs) expostos.

```
01 public aspect MeasurementEvents {
02    // Event of data gathering by sensors
03    public pointcut physicalSensor(
04                       PhysicalSensor physicalSensor):
05       execution(public void PhysicalSensor+.doProcess())
06       && target(physicalSensor);
07
08    // Init the processing of an item
09    public pointcut triggerSW(Trigger triggerSW):
10          execution(public void Trigger.trigger()) &&
11          target(triggerSW);
12
13    // Finalize the processing of an item
14    public pointcut actuation(ActuationStrategy actuator):
15      execution(public void ActuationStrategy+.actuate(..))
16      && target (actuator);
17 }
```

Figura 29. Aspecto EJP `MeasurementEvents`

6.1.3. Aspectos de Integração

Nosso estudo de composição envolveu a composição do framework *Measurement* com outros três diferentes frameworks, sendo eles: (i) framework de interface gráfica (GUI) baseado na tecnologia Java Swing [44]; (ii) framework de análise estatística; e (iii) framework de persistência baseado no Hibernate[17]. Nosso objetivo foi avaliar diferentes tipos de composições entre frameworks de acordo

[17] Hibernate - Object/Relational Persistence and Query Service. URL: http://www.hibernate.org/. 2007..

com uma categorização de problemas de composição proposta por Mattsson et al [90, 91]. Cada integração entre o *Measurement* e os demais frameworks endereça um dos problemas de composição apresentados por aqueles autores. Nas subseções seguintes são apresentados os frameworks e os respectivos aspectos de integração que foram usados para prover uma composição com o framework *Measurement*.

6.1.3.1. Composição com o Framework GUI

O framework de interface gráfica (GUI), baseado em componentes Java Swing, permite a apresentação de dados da aplicação em diferentes tabelas visuais. A Figura 30 apresenta o diagrama de classe do framework GUI. Duas classes abstratas (`DataTableModel` e `ObjectTableModel`) definem abstrações de dados a serem apresentados em uma tabela Swing baseado, respectivamente, em um array ou uma lista de objetos. Essas classes herdam da classe `AbstractTableModel` da biblioteca Swing. Os dados da aplicação são apresentados visualmente usando a classe Swing `JTable`. Durante a instanciação do framework, o usuário deve também estender a classe `GUIFramework` e definir uma implementação para o seu método abstrato `createTables()`. A implementação desse método deve criar e retornar uma lista de objetos `JTable` configurado com seus respectivos objetos `AbstractTableModel`. Toda informação escrita em objetos `AbstractTableModel` são automaticamente apresentadas nos respectivos objetos `JTable` baseado em um protocolo de notificação implementado na biblioteca Swing [44].

Aspectos de Integração foram codificados para mostrar visualmente no framework GUI, detalhes dos itens que estão sendo processados pelo framework *Measurement*. O objetivo desses aspectos é interceptar a execução de métodos ocorrendo no framework *Measurement* e capturar informação para ser apresentada no framework GUI. Os EJPs do framework *Measurement* podem prover suporte para a implementação de tais aspectos de integração. A informação capturada pelos aspectos de integração é repassada para objetos `AbstractTableModel` do framework GUI. O protocolo de notificação entre objetos `AbstractTableModel` e `JTable` já implementado no framework GUI garante a visualização dos dados.

Para obter a referência para os objetos `AbstractTableModel` disponíveis, nossos aspectos de integração requereram a exposição de tais objetos. Dessa forma, a codificação de tais aspectos demandou a criação de um aspecto EJP do framework GUI, denominado `GUIEvents`, cujo objetivo é expor o método de criação `createTables()` do framework GUI. Esse método retorna um objeto do tipo `Map` que armazena os objetos `AbstractTableModel` criados. O aspecto EJP `GUIEvents` também expõe eventos relacionados a atualização de `ObjectTableModel` e `DataTableModel` no framework GUI. O código do aspecto EJP `GUIEvents` é apresentado na Figura 31.

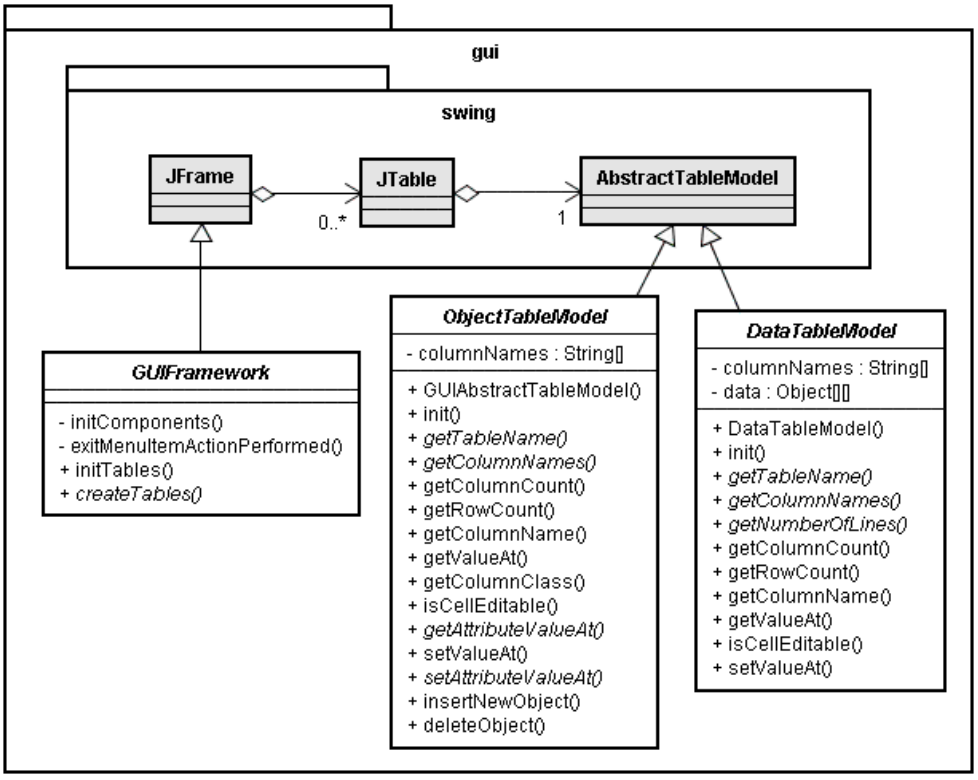

Figura 30. Framework GUI

A Figura 32 mostra os aspectos de integração. O aspecto `MeasurementGUIAspect` define o código comum que é sempre reusado quando é necessária uma composição entre os frameworks. Ele é responsável por interceptar os pontos de corte `triggerSW()` e `actuation()` do aspecto EJP `MeasurementEvents` de forma a capturar a informação dos itens de produtos que estão sendo processados por uma instância do framework *Measurement*. O subaspecto `BeerCanMeasurementGUIAspect` define o código variável que depende das instâncias criadas dos frameworks, tal como: (i) a inicialização da

instância do framework GUI durante a inicialização do framework *Measurement*; e (ii) a atualização de objetos `AbstractTableModel` específicos que irão apresentar os dados de uma instância do framework *Measurement*. A classe `BeerCanTableModel` é um exemplo de um objeto `TableModel` da aplicação de análise de qualidade de cervejas. Essa classe apresenta atributos de qualidade de cervejas que estão sendo processadas pelo framework *Measurement*.

```
01 public aspect GUIEvents {
02    public pointcut initializeTables():
03      execution(public Map GUIFramework+.createTables());
04
05    public pointcut updateObjectTable(Object object):
06      execution(public void
07          GUIAbstractTableModel.insertNewObject(Object))
08          && args(object);
09
10    public pointcut updateDataTable(Object object,
11                                    int row, int col):
12      execution(public void
13          DataTableModel.setValueAt(Object, int, int))
14          && args(object, row, col);
15 }
```

Figura 31. Aspecto EJP `GUIEvents`

Figuras 33 e 34 mostram o código parcial AspectJ dos aspectos de integração. O aspecto `MeasurementGUIAspect` define as seguintes funcionalidades: (i) um conjunto de métodos e pontos de corte abstratos que garantem a inicialização do framework GUI quando o framework *Measurement* é criado (linhas 5 e 9); (ii) ele salva uma referência para todos os objetos `AbstractTableModel` criados no framework GUI de forma a notificá-los quando ocorre atualização nos dados de itens de produto(linhas 2 e 13); e finalmente, (iii) ele intercepta a execução de métodos no framework *Measurement* para capturar informação a ser escrita nos objetos `AbstractTableModel`, tal como, o inicio do processamento de um item (linhas 17-29), a ativação de sensores e a finalização do processamento de um item nos objetos `ActuationStrategy` (linhas 35-44). O aspecto `MeasurementGUIAspect` também mantém internamente dados sobre os itens com o tempo inicial e final de processamento (linha 03). Esses dados são repassados para uma instância da classe `ProcessedItemTableModel` do framework GUI.

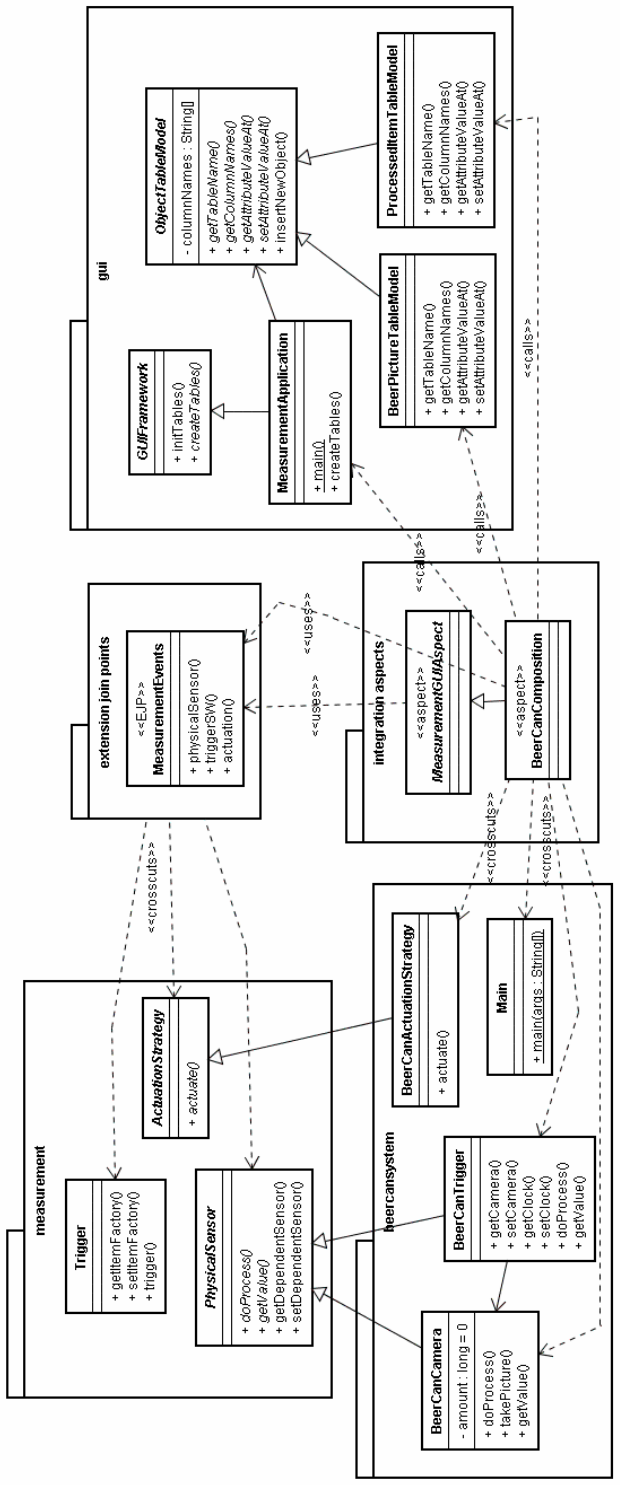

Figura 32. Integração entre os frameworks Measurement e GUI

```
01 public abstract aspect MeasurementGUIAspect {
02   private Map tables = null;
03   private Map itemsProcessingTime = new HashMap();
04
05   public abstract pointcut initIntegration();
06   before(): initIntegration() {
07       initIntegration();
08   }
09   public abstract void initIntegration();
10   ...
11   after() returning(Map tables):
12                GUIEvents.initializeTables(){
13      this.tables = tables;
14      this.initStandardTableModels();
15      this.initSpecificTableModels();
16   }
17   before(Trigger triggerSW):
18                MeasurementEvents.triggerSW(triggerSW){
19     this.createProcessedItem();
20     long threadId = Thread.currentThread().hashCode();
21     ProcessedItem currentItem =
22            (ProcessedItem) this.itemsProcessingTime.get(
23                                          new Long(threadId));
24     AbstractTableModel tableModel =
25       (AbstractTableModel) this.getTableModel(triggerSW);
26     if (currentItem != null && tableModelGeneral != null){
27           tableModelGeneral.insertNewObject(currentItem);
28     }
29   }
30   private void storeItemProcessingTime(){ ... }
31
32   public Map getItemsProcessingTime(){
33     return this.itemsProcessingTime;
34   }
35   after(ActuationStrategy actuator):
36            MeasurementEvents.actuation(actuator){
37     ...
38     ProcessedItem currentItem =
39            (ProcessedItem) this.itemsProcessingTime.get(
40                                           new Long(threadId));
41          currentItem.setEndTime(new Date());
42          // Updates the respective table model
43      ...
44   }
45 }
```

Figura 33. Aspecto de Integração `MeasurementGUIAspect`

O subaspecto `BeerCanMeasurementGUIAspect` especifica o ponto de corte que representa a inicialização do framework *Measurement* e implementa o método `initIntegration()` chamando o método `main()` que inicializa o framework GUI (linhas 3-8). Ele também implementa o método `getSpecificTableModel()` o qual retorna o objeto `AbstractTableModel` associado com um tipo específico de objeto (linhas 9-12). Esse método é chamado pelo método `getTableModel()` no aspecto `MeasurementGUIComposition`. O subaspecto `BeerCanMeasurementGUIAspect` pode também definir adendos e pontos de corte, que se responsabilizam pela atualização de objetos `AbstractTableModel` de uma instância do framework. A Figura 34 mostra, por exemplo, a definição de um adendo associado com o ponto de corte `physicalSensor()` o qual é exposto pelo aspecto EJP `MeasurementEvents`. Esse adendo atualiza o objeto `BeerCanTableModel` baseado em informação capturada pela classe `BeerCanCamera`, uma subclasse de `PhysicalSensor` (linhas 13-21).


```
01 public aspect BeerCanMeasurementGUIAspect extends
02                                  MeasurementGUIAspect {
03   public pointcut initIntegration():
04       execution(public static void BeerCanMain.main(..));
05
06   public void initIntegatrion(){
07      GUIApplication.main(null);
08   }
09   public AbstractTableModel getSpecificTableModel(
10                                Object object, Map tables){
11     ...
12   }
13   after (PhysicalSensor physicalSensor):
14         MeasurementEvents.physicalSensor(physicalSensor){
15     AbstractTableModel tableModel = (AbstractTableModel)
16                 this.getTableModel(physicalSensor);
17     Object object = physicalSensor.getValue();
18     if (object != null && tableModel != null) {
19        tableModel.insertNewObject(object);
20     }
21   }
22 }
```

Figura 34. Aspecto `BeerCanMeasurementGUIAspect`

6.1.3.2. Composição com o Framework de Estatística

O estudo de composição também contemplou a implementação de um framework de análise estatística simplificado. Esse framework oferece: (i) a classe facade `StatisticalFrameworkFacade` através da qual os serviços estatísticos são oferecidos; e (ii) duas classes abstratas, `StatisticalAlgorithm` e `Statistic`, que representam, respectivamente, pontos de extensão para implementar algoritmos estatísticos e a estatística calculada a partir do processamento do algoritmo.

O framework de Estatística foi usado para complementar a análise dos itens de produto processados pelo framework *Measurement*, de forma a calcular informações estatísticas relacionada a tal processo. As seguintes funcionalidades foram implementadas: (i) calcular o melhor e pior tempo de processamento de itens de produto; (ii) calcular o tempo médio de processamento de todos os itens de produto; e (iii) calcular o índice percentual de itens de produto de qualidade aceitável e não aceitável. A Figura 35 mostra as classes do framework de Estatística, assim como as classes necessárias para a sua instanciação no contexto do sistema de avaliação da qualidade de cervejas. Foram criadas classes para implementar as seguintes funcionalidades: (i) dois algoritmos de análise estatística (classes `ProcessingTimeAlgorithm` e `BeerCanQualityAlgorithm`) e (ii) os dados estatísticos concretos resultantes do processamento de tais algoritmos (classes `ProcessingTimeStatistic` e `BeerCanQualityStatistic`).

Para integrar a funcionalidade oferecida pelo framework de Estatística juntamente com os frameworks *Measurement* e GUI, foi criado um novo aspecto de integração, denominado `MeasurementGUIStatisticAspect`. A Figura 36 apresenta a relação de tal aspecto com os aspectos EJPs dos frameworks *Measurement* e GUI, e com o aspecto de integração `BeerCanMeasurementGUIAspect`. O aspecto `MeasurementGUIStatisticAspect` é responsável por configurar e usufruir dos serviços oferecidos pelo framework de Estatística para calcular informações de interesse da aplicação e, em seguida, apresentá-las visualmente no framework GUI, através da classe `ProcessingTimeTableModel`.

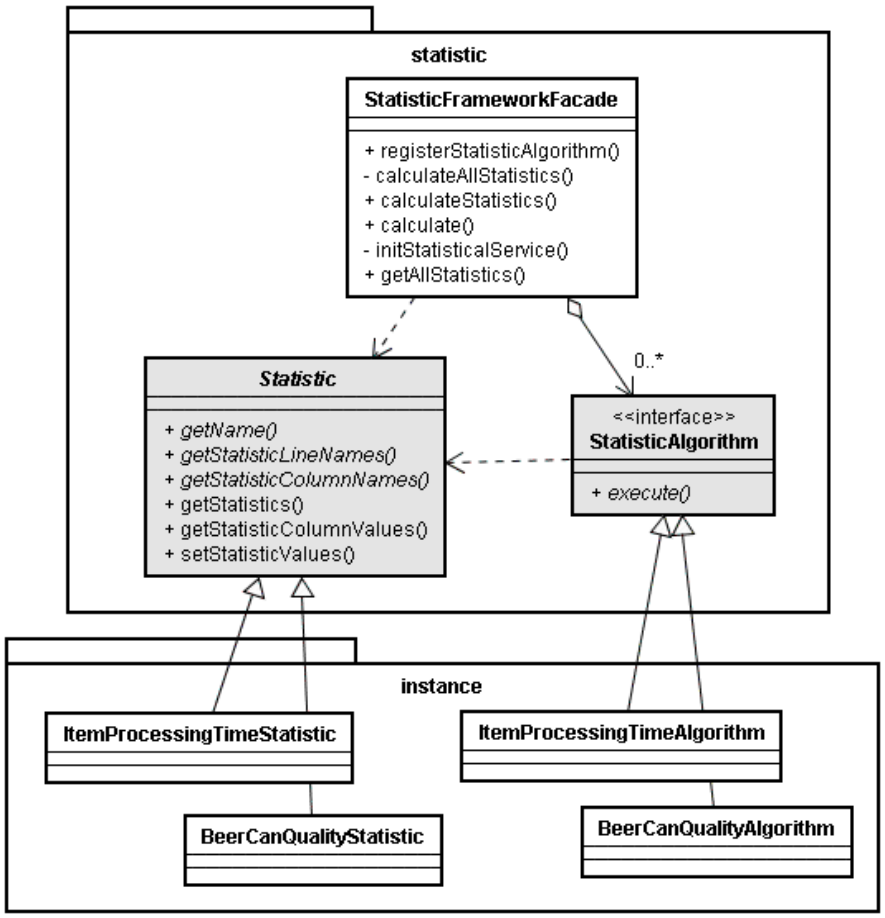

Figura 35. Framework de Estatística

A Figura 37 mostra o código do aspecto `MeasurementGUIStatisticAspect` responsável pela implementação das funcionalidade estatísticas. Ele possui as seguintes responsabilidades: (i) configurar o framework de Estatística para registrar algoritmos específicos a serem executados (linhas 13-22); (ii) criar os objetos `JTable` e respectivos objetos `AbstractTableModel` que representarão visualmente os dados estatísticos (linhas 23-28); e (iii) inicializar uma *thread* que calcula periodicamente os novos dados estatísticos através da invocação dos serviços do framework de Estatística, e atualiza os objetos `AbstractTableModel` com essas novas informações (linhas 29-52). Vale ressaltar que as informações dos itens sendo processados com seus respectivos instantes de processamento, são obtidas através de consulta ao aspecto de integração `BeerCanMeasurementGUIAspect` (linhas 50-51), definindo assim uma dependência entre os aspectos de integração. Essas informações são usadas para alimentar os algoritmos estatísticos.

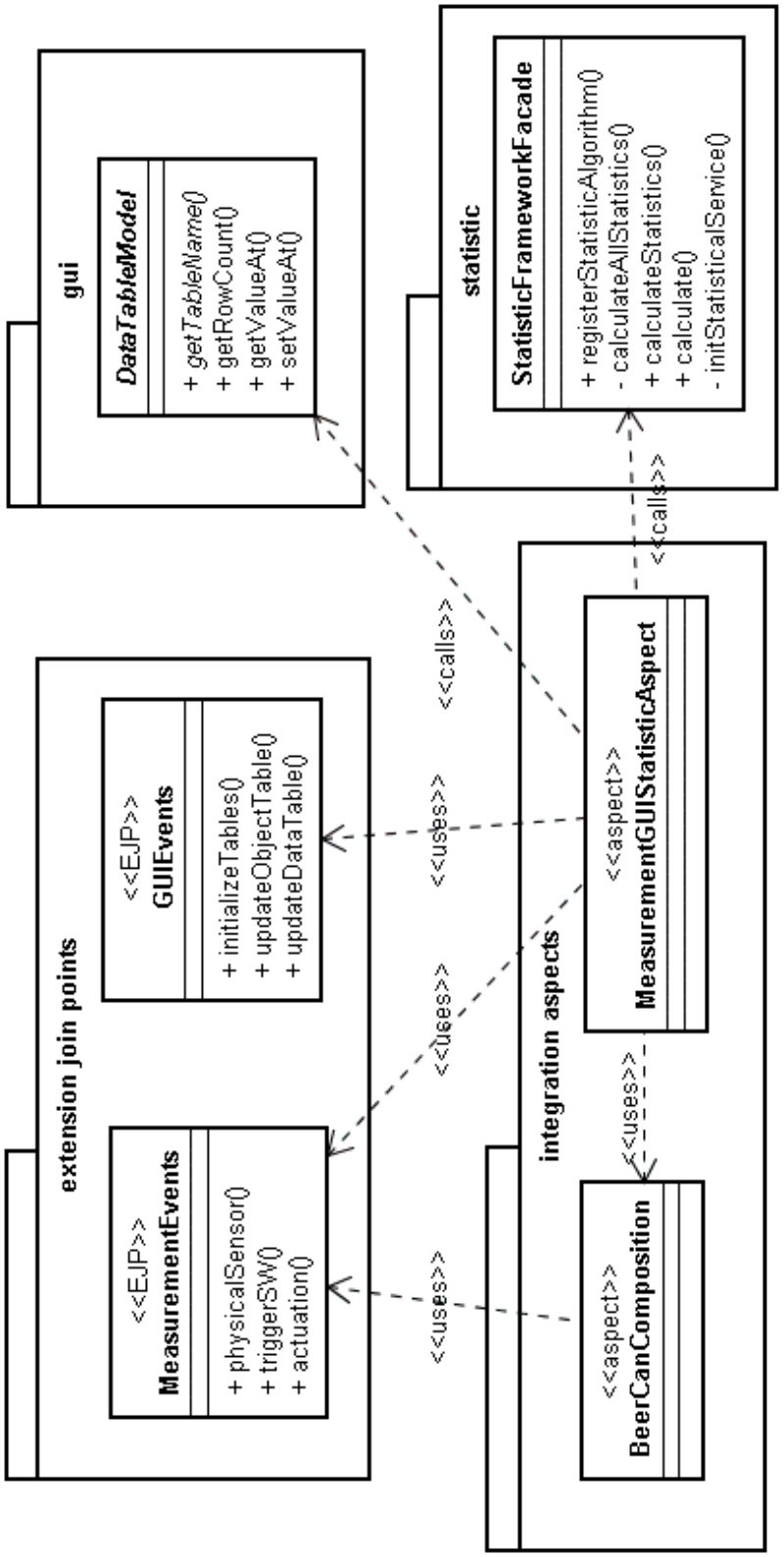

Figura 36. Integração entre os FWs Measurement, GUI e Estatística

```
01 public aspect MeasurementGUIStatisticAspect {
02   private Thread statisticalService = null;
03   private DataTableModel statisticTableModel = null;
04   public pointcut initIntegration():
05            MeasurementEvents.initApplication();
06   before(): initIntegration(){
07      this.configureStatisticFramework();
08   }
09   after() returning(Map tables):
10                GUIEvents.initializeTables(){
11     this.initStatisticTableModel(tables);
12   }
13   public void configureStatisticFramework(){
14      StatisticFrameworkFacade statisticFramework =
15                StatisticFrameworkFacade.getInstance();
16      StatisticAlgorithm algorithm =
17                new BeerProcessingTimeAlgorithm();
18      statisticFramework.registerStatisticAlgorithm(
19                  "Measurement", algorithm,
20                  this.getItemsProcessingTime());
21      this.initStatisticalService(10000);
22   }
23   public void initStatisticTableModel(Map tables){
24      this.statisticTableModel = new
25                          ProcessingTimeTableModel();
26      JTable table = new JTable(this.statisticTableModel);
27      tables.put("Processing Time Statistics", table);
28   }
29   private void initStatisticalService(final long delay){
30     if (statisticalService == null){
31        statisticalService = new Thread() { ...
32        };
33        statisticalService.start();
34      }
35   }
36   private void processStatistics(){
37      StatisticFrameworkFacade statisticFramework =
38                  StatisticFrameworkFacade.getInstance();
39      statisticFramework.updatePopulation("Measurement",
40                    this.getItemsProcessingTime());
41      Map statistics =
42      statisticFramework.getStatistiscs("Measurement");
43      this.updateStatistics(statistics);
44   }
45   private void updateStatistics(Map statistics){
46      // Update statistics in their respective table models
47      ...
48   }
49   private Collection getItemsProcessingTime(){
50      return BeerCanMeasurementGUIAspect.aspectOf().
51                    getItemsProcessingTime().values();
52   }
53   ...
54 }
```

Figura 37. Aspecto `MeasurementGUIStatisticAspect`

6.1.3.3. Composição com o Framework de Persistência

No nosso estudo de composição, o último interesse endereçado foi o de persistência. As seguintes informações foram persistidas: (i) dados dos itens de produto analisados pelo framework *Measurement*; e (ii) dados estatísticos oriundos desse processo de análise. Para endereçar a persistência de tais informações, foi utilizado o framework Hibernate. O Hibernate permite a definição de mapeamentos entre classes de negócio e tabelas de banco de dados, assim como oferece vários serviços para acesso e atualização do banco de dados.

Para possibilitar a persistência dos dados resultantes do processo de análise da qualidade de itens de produto, através do Hibernate, foi definida a classe `PersistenceFrameworkFacade`, responsável por prover os serviços de persistência e por agregar um conjunto de objetos de acesso a dados. Tais objetos de acessos a dados representam implementações concretas do padrão de projeto DAO (*Data Acess Object*) [1]. Cada DAO é responsável pela persistência de objetos de uma dada classe do domínio do sistema. No framework de persistência, a classe abstrata `EntityDAO` especifica um conjunto de métodos de persistência a serem implementados por DAOs concretos. Essa classe herda da classe `HibernateDAOSupport` do framework Spring [65], de forma a reusar serviços de banco de dados oferecidos pelo framework Hibernate, tais como, consultas simples de acesso ao BD, gerenciamento de conexões e demarcação de transações.

A Figura 38 mostra os aspectos de integração e as classes DAOs responsáveis pela implementação do interesse de persistência. Para capturar as informações a serem persistidas, o aspecto de persistência `PersistenceAspect` intercepta: (i) o ponto de execução do framework GUI, exposto pelo aspecto EJP `GUIEvents`, que representa a atualização de dados em objetos `ObjectTableModel`; e (ii) o ponto de execução do framework de Estatística, exposto pelo aspecto EJP `StatisticEvents`, o qual expõe a execução de algoritmos estatísticos. Essas informações coletadas pelo aspecto `PersistenceAspect` são repassadas para a classe `PersistenceFrameworkFacade`, que se encarrega de chamar a classe DAO apropriada para persistir no banco de dados tais informações. A configuração dos DAOs a serem usados pelo framework de persistência é feita também pelos

aspectos de integração, o aspecto `PersistenceAspect` configura DAOs que serão usados em diferentes configurações de composição, enquanto seu subaspecto `BeerCanPersistenceAspect` se responsabiliza pela configuração de DAOs específicos.

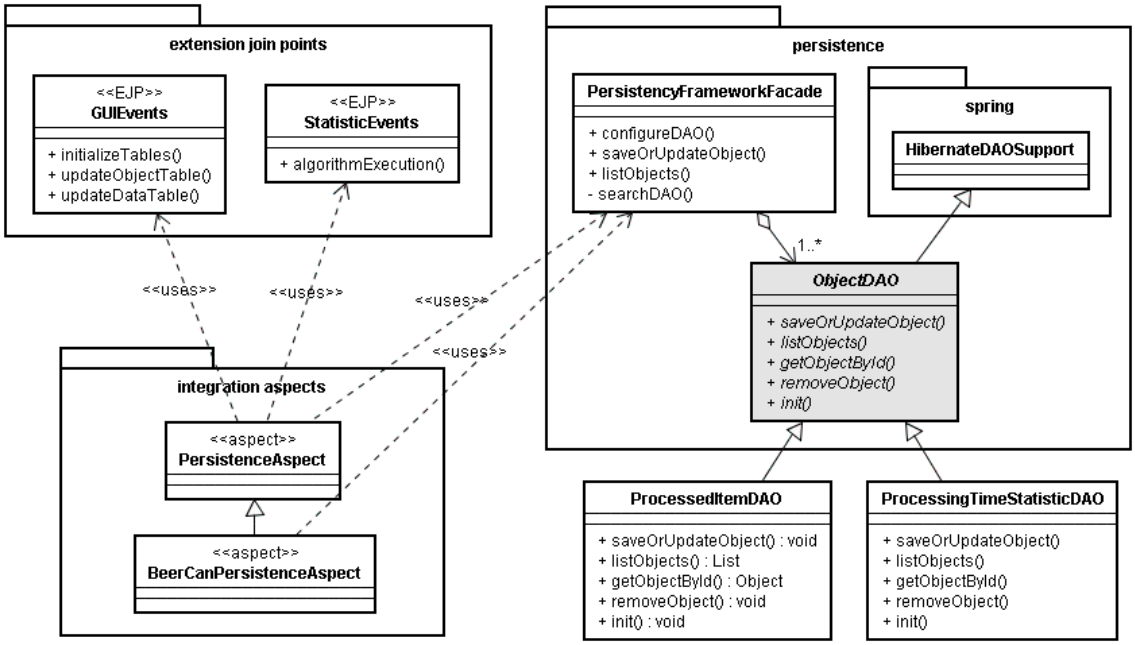

Figura 38. Integração entre os FWs GUI, de Estatística e de Persistência

As Figuras 39 e 40 mostram o código dos aspectos de integração de persistência. O aspecto `PersistenceAspect` define os seguintes pontos de corte: (i) `initializePersistenceService()` – o qual é usado para disparar a inicialização do framework de persistência com a configuração de seus respectivos DAOs (linha 4). Esse ponto de corte é concretizado por subaspectos; (ii) o ponto de corte `processedItemsTableModel()` – o qual intercepta a atualização de objetos `ObjectTableModel` no framework GUI, de forma a persistir tais informações (linhas 19-24); e (iii) `statisticalDataPersistence()` – responsável por interceptar a execução de algoritmos de estatística no framework respectivo, de forma a coletar os dados estatísticos resultantes de tal processamento e invocar o método `saveOrUpdateObject()` do framework de persistência (linhas 26-31). O subaspecto `BeerCanPersistenceAspect`, por sua vez, concretiza o ponto de corte de inicialização do framework GUI (linhas 3-5) e inicializa, caso existam, novos DAOs concretos (linhas 7-11).

```
01 public abstract aspect PersistenceAspect {
02   PersistencyFrameworkFacade persistenceFramework = null;
03
04   public abstract pointcut initializePersistenceService();
05
06   after(): initializePersistenceService(){
07     persistenceFramework =
08               PersistencyFrameworkFacade.getInstance();
09     this.initCommonDAOs();
10     this.initSpecificDAOs();
11   }
12   public void initCommonDAOs(){
13     this.persistenceFramework.configureDAO(
14               ItemProcessingTime.class.getName(),
15               new ProcessedItemDAO());
16   }
17   public abstract void initSpecificDAOs();
18
19   public pointcut processedItemsTableModel(Object object):
20                GUIEJPs.updateObjectTable(object);
21
22   after(Object object): processedItemsTableModel(object){
23     this.persistenceFramework.saveOrUpdateObject(object);
24   }
25
26   public pointcut statisticalDataPersistence():
27      StatisticEvents.algorithmExecution();
28
29   after() returning(Statistic statistic):
30               statisticalDataPersistence(){
31    this.persistenceFramework.saveOrUpdateObject(statistic);
32
33   }
34 }
```

Figura 39. Aspecto de Integração `PersistenceAspect`

```
01 public aspect BeerCanPersistenceAspect
02                          extends PersistenceAspect {
03   public pointcut initializePersistenceService():
04      execution(public static void
05                       MeasurementApplication.main(..));
06
07   public void initSpecificDAOs(){
08      this.persistenceFramework.configureDAO(
09             BeerProcessingTimeStatistic.class.getName(),
10             new ProcessingTimeStatisticDAO());
11   }
12 }
```

Figura 40. Aspecto de Integração `BeerCanPersistenceAspect`

6.1.4. Modelo Generativo OA

O modelo generativo do framework *Measurement* integrado com os frameworks de GUI, Estatística e Persistência é apresentado na Figura 41. O modelo de arquitetura agrega os quatro componentes principais que representam cada um dos frameworks da composição, são eles: measurement, gui, statistic, e persistence. Cada um deles agrega: (i) as classes que implementam o núcleo do framework; (ii) templates úteis para a instanciação do framework que são especializações de classes abstratas ou interfaces que representam os pontos flexíveis do framework; e (iii) seus respectivos EJPs, os quais expõem eventos/estados específicos oriundos de colaborações de classes do framework, e que podem ser usados para prover alguma integração transversal com um outro framework/componente. O componente measurement, por exemplo, define os templates `ActuatorTemplate`, `SensorTemplate`, `ActuationStrategyTemplate` e `CalculationStrategyTemplate` para a criação de efetuadores, sensores e estratégias relacionadas ao processo de análise dos produtos no framework *Measurement*. O aspecto EJP `MeasurementEvents` também é definido dentro de tal componente. Por questões de espaço apenas parte dos elementos de implementação de cada componente são apresentados.

Para os demais componentes (gui, statistic, persistence), foram também definidos no modelo de arquitetura: (i) um sub-componente integration - responsável por agregar aspectos e classes relacionados com a integração entre os frameworks; e (ii) classes diretamente relacionadas com a instanciação do framework para o contexto da composição com o framework *Measurement*. Os aspectos e classes de integração entre os frameworks *Measurement* e GUI, por exemplo, são definidos dentro do sub-componente integration do componente gui. As classes `SensorTableModel` e `ActuatorTableModel` representam uma instanciação do framework GUI para representar informações sobre o processo de análise de qualidade dos produtos.

O modelo de característica da composição dos frameworks expõe as variabilidades de cada um deles para serem escolhidas pelos engenheiros de aplicação. No que se refere ao processo de medição, o usuário pode definir sensores (Sensor) e acionadores (Actuator) concretos a serem usado em tal

processo. A característica GUI permite ao usuário definir que informações ele deseja visualizar do processo de medição, tais como, informações de itens já processados por sensores e efetuadores (Sensor e Actuator Visualization) e/ou informações detalhadas do processo de análise de qualidade do produto (Product Information Visualization). A característica Statistic permite ao usuário decidir pela visualização ou não de informações estatísticas sobre os itens sendo processados pelo framework *Measurement*. Finalmente, a característica Persistence permite escolher que informação manipulada pelos frameworks se deseja persistir em banco de dados.

No modelo de configuração da composição dos frameworks de *Measurement*, GUI, Estatística e Persistência, diferentes relações de dependência são criadas entre os elementos de implementação dos frameworks e características. Templates de classes representando sensores, efetuadores e estratégias relacionadas ao processo de medição, dependem das respectivas variabilidades existentes no modelo de característica. O template `ActuatorTemplate`, por exemplo, depende explicitamente da criação de uma característica Actuator. Cada um dos componentes gui, statistic e persistence dependem, respectivamente, da seleção das características GUI, Statistic e Persistence. Isso significa que todos os elementos de implementação internos a tais componentes e que não possuem alguma outra relação de dependência explícita, serão incluídos em um produto instanciado sempre que as características dos quais dependem forem selecionadas. Finalmente, os diferentes elementos que implementam as alternativas de interface gráfica (`SensorTableModel`, `ActuatorTableModel`, `ItemTableModel`) e persistência (`ProcessedItemDAO`, `ProcessingTimeStatisticDAO`) também dependem explicitamente das características alternativas de GUI e Persistence.

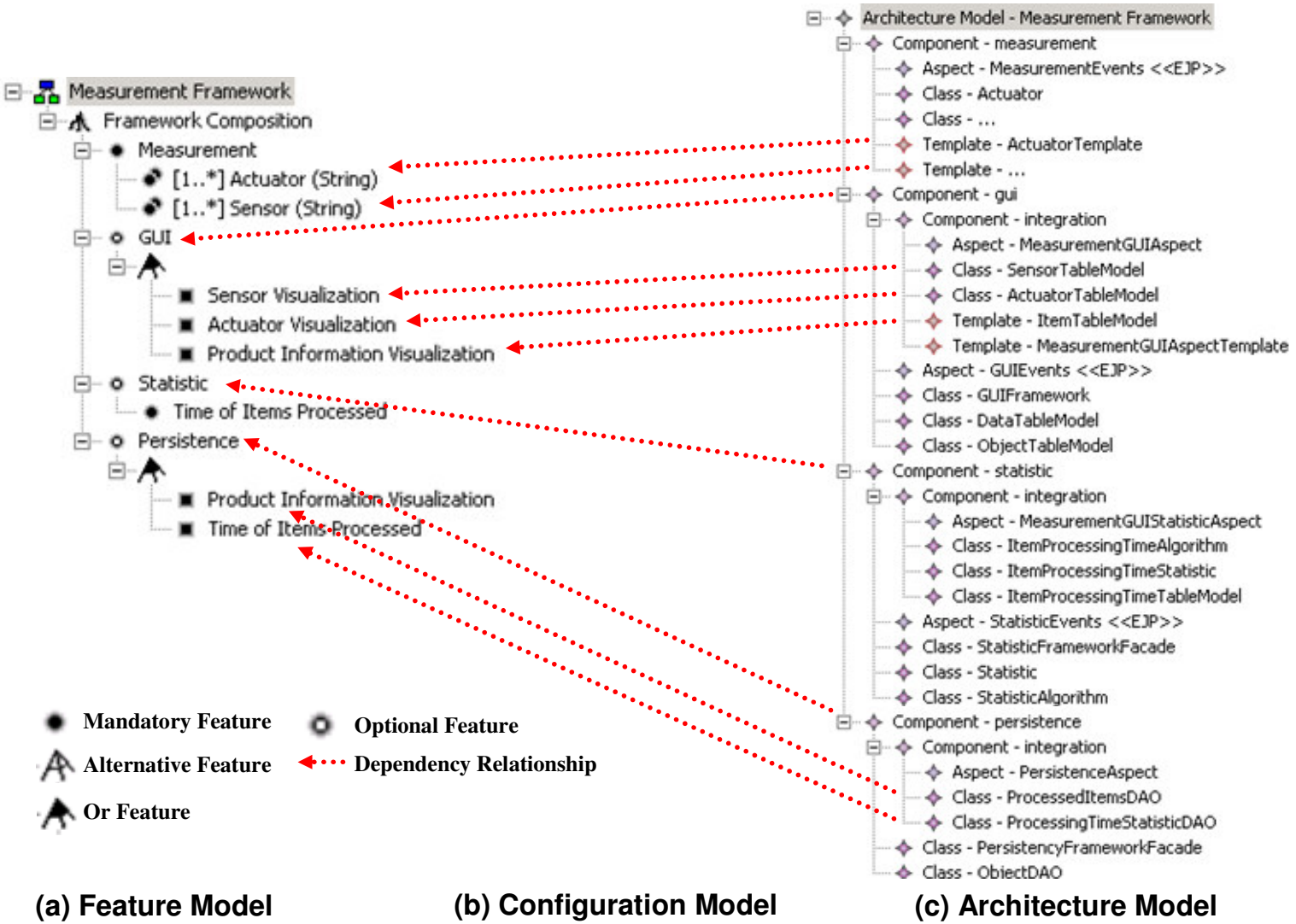

Figura 41. Modelo Generativo da Composição dos Frameworks

6.2. AspectT

Nossa abordagem foi também aplicada na reestruturação do framework orientado a aspectos AspectT [46, 50, 80]. AspectT é um framework usado na implementação de arquiteturas de agentes que endereçam diferentes propriedades. Ele foi desenvolvido para endereçar uma melhor modularização do projeto e implementação de diferentes tipos de agentes que precisam endereçar um conjunto de funcionalidades e propriedades (interação, adaptação, autonomia, aprendizado, mobilidade) que são transversais entre si.

Assim, o framework AspectT busca a modularização de propriedades transversais encontrados na definição da arquitetura de agentes, oferecendo mais flexibilidade para a composição transversal entre tais propriedades. A separação de interesses transversais oferecida pelo AspectT pode também trazer os benefícios: (i) de diminuição de linhas de código do sistema e (ii) de facilitar a manutenibilidade e reusabilidade [46, 50] do sistema.

6.2.1. Núcleo do Framework

O núcleo do AspectT define um conjunto de serviços básicos do agente para manipulação do seu conhecimento e para interação com o mundo externo. Cada agente contém crenças, objetivos e planos. As crenças do agente definem informação sobre o próprio agente e sobre o ambiente no qual ele está inserido. Para alcançar seus objetivos, um agente executa planos específicos. Durante a execução do plano, o agente manipula suas crenças. Cada agente também mantém um conjunto de sensores e efetuadores os quais habilitam e permitem sua interação com o mundo externo.

A Figura 42 apresenta um diagrama de classes parcial do núcleo do framework AspectT. A classe `Agent` define o comportamento básico de um agente e agrega instâncias das classes `Goal` e `Plan`. Essas últimas, por sua vez, especificam a estrutura e comportamento geral que deve existir para a definição de objetivos e planos do agente. A classe `Agent` também mantém atributos e métodos para a troca de mensagens com o ambiente, através da classe `Interaction`. Essa classe é responsável pela definição de: (i) atributos `inbox` e

`outbox` para armazenamento de mensagens recebidas e enviadas, respectivamente; e (ii) um conjunto de sensores (subclasses de `Sensor`) e efetuadores (subclasses de `Effector`) para interação com o ambiente externo. A Figura 43 mostra o diagrama de seqüência de recebimento e envio de mensagens do agente.

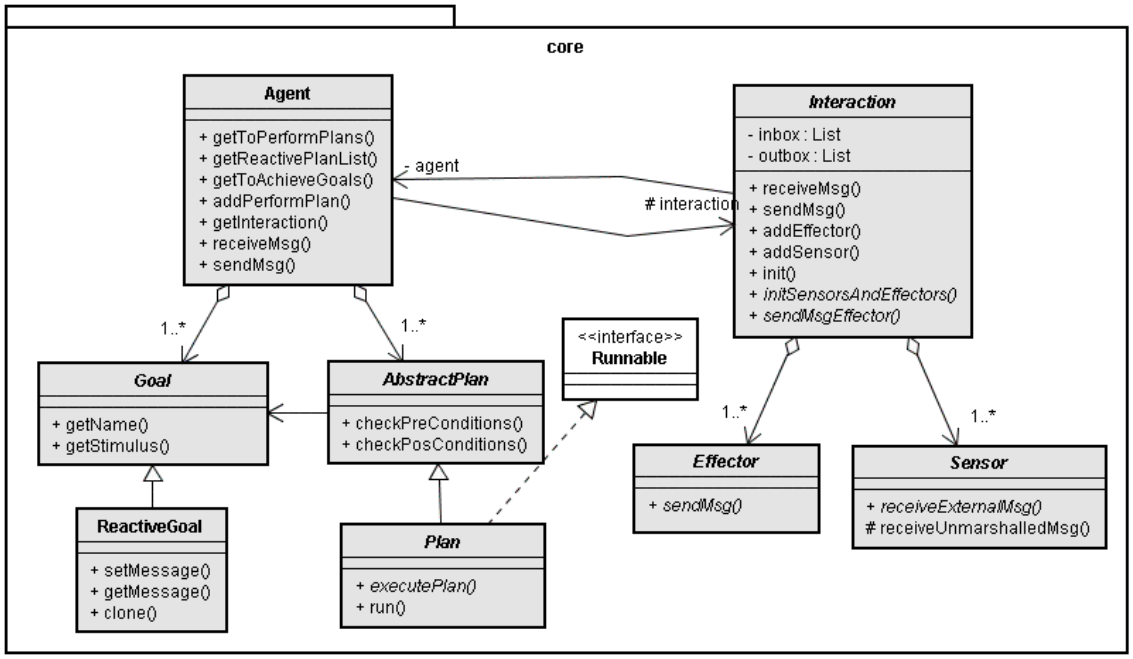

Figura 42. Núcleo do Framework AspectT

Várias classes do núcleo do AspectT representam pontos flexíveis do framework os quais podem ser estendidos para implementar agentes específicos. A classe `Agent` pode ser estendida para definir novos tipos de agentes. Crenças específicas de determinados tipos de agentes podem ser implementadas diretamente como classes de domínio que são agregadas pelas subclasses de `Agent`. As classes `Goal`, `ReactiveGoal`, `Plan` e `ReactivePlan` podem, por sua vez, ser especializadas para definir planos e objetivos específicos de um dado tipo ou papel de agente. Finalmente, as classes `Interaction`, `Sensor` e `Effector` definem uma estrutura geral para a criação dos elementos de comunicação do agente. Subclasses das mesmas devem ser criadas para definir sensores e efetuadores usados por um dado agente.

A Figura 44 mostra um exemplo de instância do framework AspectT para um sistema de gerência do processo de revisão de artigos de conferência,

denominado Expert Committtee (EC) [46]. O sistema EC define agentes do usuário que representam assistentes de software para auxiliar em atividades de gerência e distribuição de artigos. As seguintes classes foram definidas para instanciação do framework AspectT: (i) classe `ResearcherUserAgent` – implementa um agente do usuário que representa pesquisadores, tais agentes podem assumir os papéis *Chair* e *Reviewer* no contexto do EC; (ii) classes `BlackboardSensor` e `BlackboardEffector` – implementam sensores e efetuadores para comunicação via blackboard [19] entre agentes; (iii) subclasses das classes `ReactiveGoal` e `ReactivePlan`, são também criadas para endereçar objetivos e planos a serem alcançados pelos papéis *Chair* e *Reviewer*.

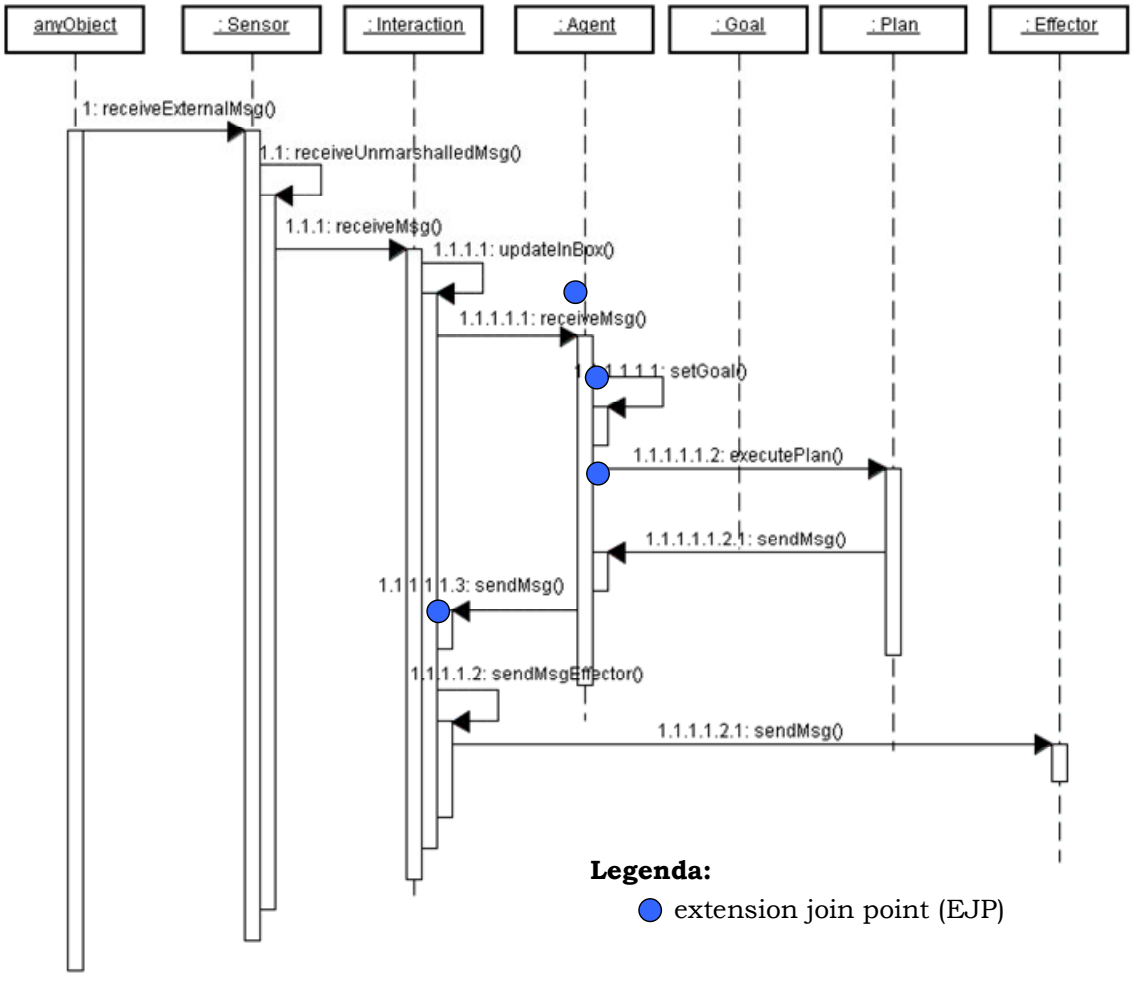

Figura 43. Diagrama de Seqüência do Framework AspectT

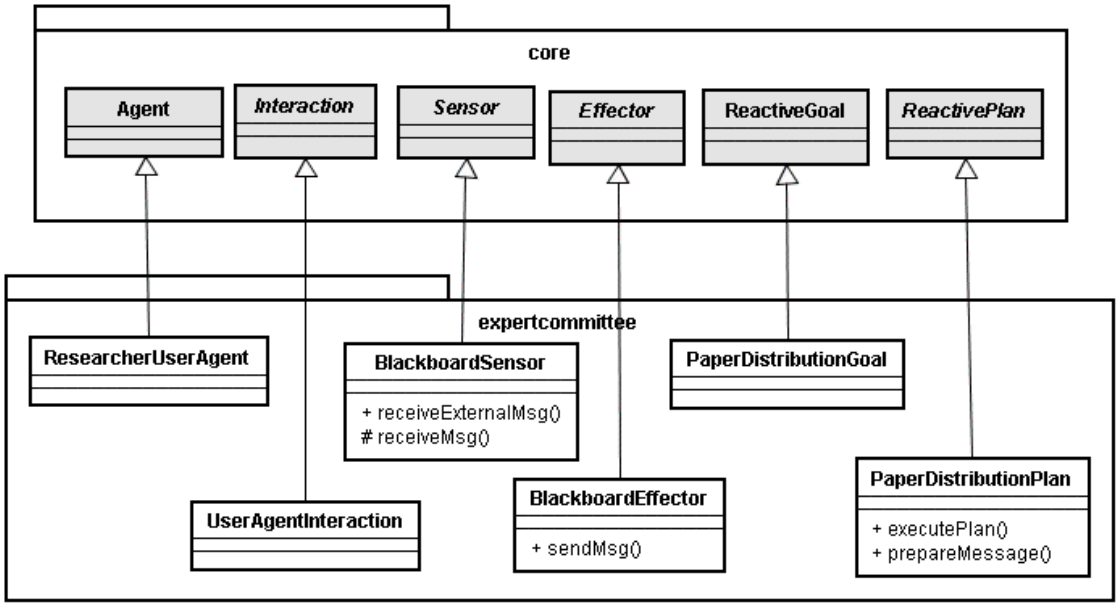

Figura 44. Exemplo de Instância do Framework AspectT

6.2.2. Pontos de Junção de Extensão

O framework AspectT expõe os seguintes EJPs: (i) eventos de recepção e envio de mensagens; (ii) evento de instanciação de objetivos do agente; e (iii) evento de execução de planos. A Figura 45 mostra o código do aspecto EJP `AspectTEvents` do framework AspectT, com os respectivos pontos de corte que expõem eventos de interesse para implementação de aspectos do núcleo e de extensão do framework.

```
01 public aspect AspectTEvents {
02   public pointcut messageReceiving(Agent agent, Message msg):
03     args(msg) && this(agent)
04     && execution(void Agent.receiveMsg(Message));
05
06   public pointcut goalCreation(Agent agent, Goal agentGoal):
07     args(agentGoal) && this(agent)
08     && execution(void setGoal(Goal));
09
10   public pointcut planExecution(Plan plan):
11     call(public void Plan+.executePlan(..))
12      && target(plan);
13
14   public pointcut messageSending(Agent agent, Message msg):
15       execution(void sendMsg(Message msg, Agent agent))
16       && args(msg, agent);
17 }
```

Figura 45. Aspecto EJP `AspectTEvents`

6.2.3. Aspectos do Núcleo

A funcionalidade básica oferecida pelo núcleo do framework AspectT contempla também a definição de dois aspectos do núcleo, os quais implementam duas características obrigatórias do agente, são elas: (i) Adaptação – essa característica define mudanças a serem realizadas nas crenças ou comportamento do agente, a partir da percepção de eventos ocorridos no ambiente ou na própria execução do agente; e (ii) Autonomia – determina a instanciação de objetivos a partir da ocorrência de eventos internos e externos ao agente. As subseções seguintes apresentam os aspectos do núcleo que foram implementados para endereçar as características de Adaptação e Autonomia.

6.2.3.1. Adaptação

A característica de Adaptação é implementada no AspectT pelo aspecto abstrato `Adaptation` e por um conjunto de subaspectos de tal aspecto que especificam comportamento de adaptação específico para um dado tipo ou papel de agente. Dois tipos de adaptação são endereçados por esses aspectos: (i) adaptação de crenças – responsável por interpretar mensagens recebidas do ambiente e manipular/atualizar as crenças do agente baseado em informações contidas em tais mensagens; e (ii) adaptação de planos – determina o plano que o agente deve executar quando determinados objetivos precisam ser alcançados. A Figura 46 apresenta a estrutura geral de solução do AspectT para endereçar a característica de Adaptação, através do aspecto `Adaptation` e seus subaspectos.

A Figura 47 apresenta o código do aspecto abstrato `Adaptation` e os elementos principais que ele interage. A adaptação de crenças desse aspecto é definida através da interceptação do evento de recepção de mensagens – ponto de corte `messageReceiving()` – definido no aspecto EJP `AspectTEvents` (linhas 2-3). Advices e métodos específicos são responsáveis pela atualização de crenças do agente a partir da recepção de mensagens do ambiente (linhas 5-18). A adaptação de planos do aspecto `Adaptation` intercepta o evento de instanciação de objetivos do agente – ponto de corte `goalCreation()` (linhas 20-21), assim

como de falha na execução de planos – ponto de corte `exceptionalPlanExecution`) (linhas 39-42). O objetivo é definir novos planos, através de chamadas aos métodos `findPlan()` e `findSpecificPlan()` (linhas 34-37), para serem executados pelos agentes durante essas situações específicas. O aspecto `Adaptation` pode ser especializado para permitir a definição de adaptação de crenças e planos específicos para um dado tipo ou papel de agente. Os métodos abstratos `adaptSpecificBelief()` e `findSpecificPlan()` podem ser redefinidos por subaspectos para implementar tais funcionalidades. A seção 6.2.4.3 apresenta exemplos de subaspectos da característica de Adaptação.

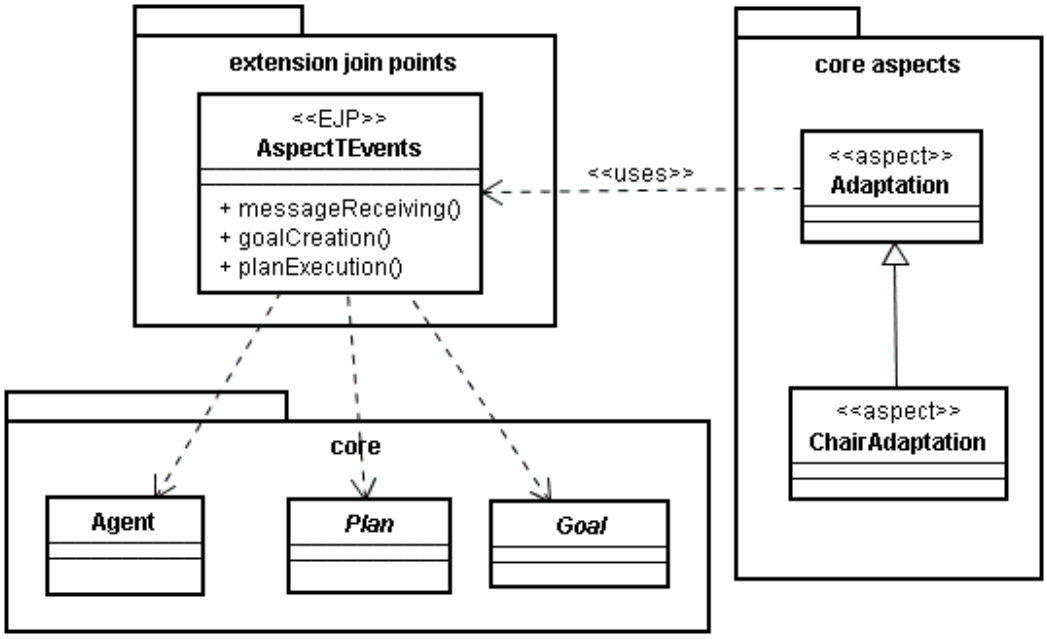

Figura 46. Aspecto do Núcleo Adaptation

```
01 public abstract aspect Adaptation {
02   pointcut beliefAdaptation(Agent agent, Message msg):
03          AspectTEvents.messageReceiving(agent, msg);
04
05   after(Agent agent, Message msg):
06          beliefAdaptation(agent, msg) {
07      adaptBelief(agent, msg);
08   }
09   public void adaptBelief(Agent agent, Message msg){
10     adaptGeneralBelief(agent, msg);
11     adaptSpecificBelief(agent, msg);
12   }
13   public void adaptGeneralBelief(Agent agent, Message msg){
14     // Update the agent belief, based on the received msg
15     ...
16   }
17   public abstract void adaptSpecificBelief
18                            (Agent agent, Message msg);
19
20   pointcut planAdaptation(Agent agent, Goal agentGoal):
21         AspectTEvents.goalCreation(agent, agentGoal);
22   after(Agent agent, Goal agentGoal) returning() :
23          planAdaptation(agent, agentGoal) {
24     Hashtable agentPlans = new Hashtable();
25     Plan plan = findSpecificPlan(agent, agentGoal);
26     if (plan == null) {
27       agentPlans = agent.getReactivePlanList();
28       plan = findPlan(agent, agentGoal, agentPlans);
29     }
30     if (plan != null) {
31       agent.addPerformPlan(plan);
32     }
33   }
34   public abstract Plan findSpecificPlan(Agent agent,
35                                         Goal agentGoal);
36   public Plan findPlan(Agent agent, Goal agentGoal,
37                        Hashtable agentPlans) { ... }
38
39   pointcut exceptionalPlanExecution(Plan plan):
40       AspectTEvents.planExecution(plan);
41   after(Plan plan) throwing(FailedPlanException exception):
42      exceptionalPlanExecution(plan) { ... }
43
44   protected pointcut planFinalization(Plan plan):
45      AspectTEvents.planExecution(plan);
46   after(Plan plan) returning():
47         planFinalization(plan) { ... }
48 }
```

Figura 47. Aspecto do Núcleo `Adaptation`

6.2.3.2. Autonomia

A característica de Autonomia endereça a funcionalidade de instanciação e gerência de objetivos. Ela lida com três tipos de objetivos: (i) reativos – aqueles que são instanciados a partir de requisições externas de agentes ou demanda do próprio ambiente; (ii) proativos – são criados a partir da ocorrência de eventos internos (tais como, finalização de execução de planos ou alcance de determinado estado de execução); e (iii) de decisão – são instanciados devido a eventos internos ou externos e usados para decidir se objetivos reativos ou proativos devem ser criados. A característica Autonomia também se responsabiliza pela definição de uma estratégia de concorrência para execução dos planos do agente.

A Figura 48 mostra os aspectos e classes responsáveis pela implementação da característica Autonomia. O aspecto `Autonomy` é o responsável principal pela extensão do núcleo do framework AspectT para endereçar a propriedade de Autonomia. Ele é usado para definir o comportamento básico de: (i) instanciação e gerência de objetivos; e (ii) execução concorrente de planos. Tipos de agentes mais sofisticados podem estender o aspecto `Autonomy` para definir uma gerência de objetivos específica para um dado tipo ou papel de agente.

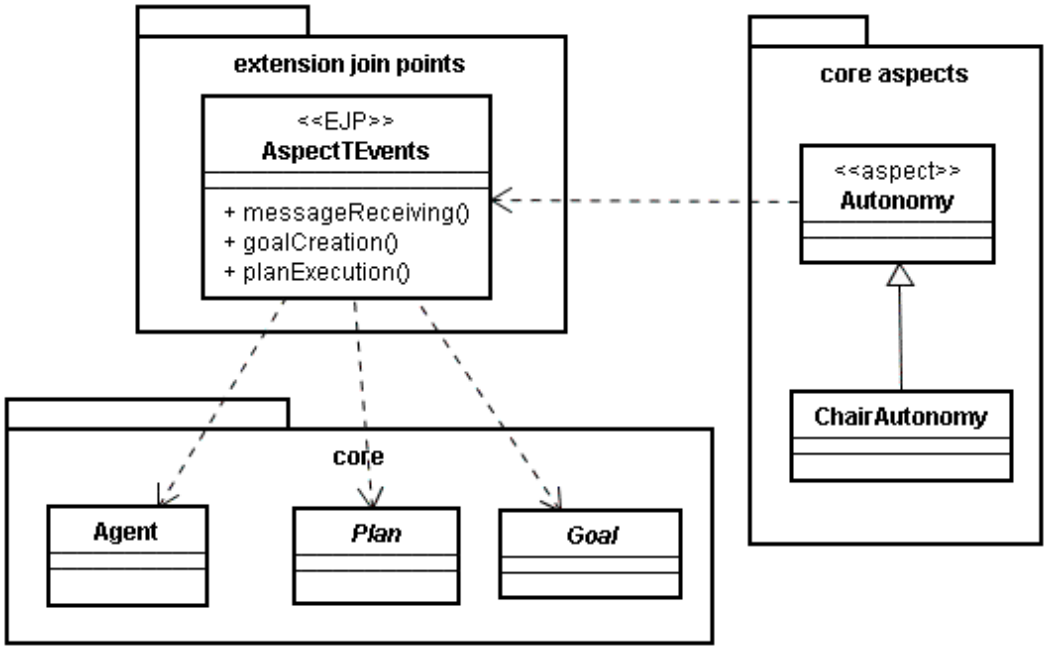

Figura 48. Aspecto do Núcleo Autonomy

O aspecto `Autonomy` demanda a instanciação de objetivos reativos a partir da interceptação do evento de recebimento de mensagens do agente, exposto pelo ponto de corte `messageReceiving()` do aspecto EJP `AspectTEvents` (linhas 33-54). Caso necessário, ele (ou seus sub-aspectos) pode também especificar comportamento para instanciação de objetivos proativos, a partir do monitoramento de eventos internos do agente. Nesse caso diferentes eventos internos expostos pelos aspectos EJPs podem ser utilizados. Adendos e métodos específicos (tais como, `instantiateSpecificReactiveGoal()` e `instantiateSpecificProactiveGoal()`) definem o comportamento propriamente dito de instanciação de objetivos na ocorrência de tais eventos. A autonomia de decisão do agente é definida pelo método `makeDecision()` do aspecto `Autonomy` (linhas 40-44). Esse método verifica se é necessário executar algum plano de decisão na ocorrência de eventos internos ou externos. Subaspectos de `Autonomy` podem também ser codificados para definir comportamento de instanciação de objetivos (reativos, proativos e de decisão) específicos de um dado tipo ou papel de agente, através da concretização de vários métodos abstratos, tais como: `instantiateSpecificReactiveGoal()`, `instantiateSpecificProactiveGoal()` e `makeSpecificDecision()`.

Finalmente, o aspecto `Autonomy` também define um *Active Object* [86] (linhas 20-31). Ele é usado para: (i) monitorar a lista de planos do agente que foram selecionados pela característica de adaptação para serem executados; e (ii) selecionar uma estratégia de concorrência que permite a execução de planos usando diferentes *threads*.

```
01 public abstract aspect Autonomy {
02   private GoalFinder goalFinder;
03   ...
04   /* Goal Creation */
05   protected abstract pointcut agentInstantiation
06                                   (Agent agent);
07   after (Agent agent): agentInstantiation(agent) {
08     goalFinder = new GoalFinder(FILE_AGENT);
09     initGoals(agent);
10     initThread(agent);
11   }
12   public void initGoals(Agent agent) {
13     initGeneralGoals();
14     initSpecificGoals(agent);
15   }
16   public void initGeneralGoals() { ... }
17   public abstract void initSpecificGoals(Agent agent);
18
19   /* Execution Autonomy */
20   private int maxNumberOfThreadsPerAgent = 5;
21   private ActiveObject autonomyActiveObject;
22
23   public void initThread(Agent agent) {
24     // Instantiate the ConcurrencyStrategy
25     ConcurrencyStrategy concurrencyStrategy =
26                       new ThreadPerRequestStrategy();
27     autonomyActiveObject =
28               new ActiveObject(agent.getToPerformPlans(),
29                                concurrencyStrategy);
30     autonomyActiveObject.start();
31   }
32   /* Goal Management */
33   pointcut decisionMaking(Agent agent, Stimulus msg):
34        AspectTEvents.messageReceiving(agent, msg);
35
36   after(Agent agent, Stimulus stimulus):
37             decisionMaking(agent, stimulus) {
38       makeDecision(agent, stimulus);
39   }
40   public void makeDecision(Agent agent, Stimulus stimulus){
41     // Instantiate the agent goals based on the
42     // internal and external stimulus
43     ...
44   }
45   public void instantiateReactiveGoal(Agent agent,
46                                       Message msg){
47     instantiateGeneralReactiveGoal(agent, msg);
48     instantiateSpecificReactiveGoal(agent, msg);
49   }
50   public void instantiateGeneralReactiveGoal(Agent agent,
51                                   Message msg){ ... }
52   public abstract void instantiateSpecificReactiveGoal(
53                            Agent agent, Message msg);
54   ...
55 }
```

Figura 49. Aspecto do Núcleo `Autonomy`

6.2.4.
Aspectos de Extensão

O núcleo do framework AspectT foi estendido em nosso estudo de caso por diversos aspectos de extensão. Cada um deles endereça uma característica opcional relevante de agentes heterogêneos, sendo elas: (i) Aprendizagem – essa propriedade oferece ao agente a capacidade de refinar seu conhecimento, a partir da experiência obtida como resultado de suas ações, interações com o ambiente e com outros agentes; (ii) Mobilidade – essa característica endereça o comportamento necessário do agente para trafegar entre diferentes ambientes remotos; e (iii) Colaboração – define um conjunto de papéis que oferece ao agente a habilidade de colaborar com diferentes tipos de agentes.

A implementação dos aspectos de extensão que endereçam as características de Aprendizagem, Mobilidade e Colaboração de agentes, utiliza os mecanismos de declarações inter-tipos de AspectJ para estender as classes do núcleo do framework AspectT com comportamento necessário para implementar tais características. Tal comportamento adicional (implementada por meio de novos atributos e/ou métodos), é então invocado através de pontos de corte e advices definidos também por cada aspecto de extensão. As seções seguintes apresentam os respectivos aspectos de extensão que endereçam cada uma dessas características no AspectT.

6.2.4.1.
Aprendizagem

A característica de Aprendizagem está relacionada com o comportamento do agente responsável por refinar ou obter novo conhecimento. Agentes cognitivos aprendem baseado na sua experiência como resultado de suas ações, erros, interações com o ambiente e colaborações com outros agentes [95, 105]. Diferentes técnicas de aprendizado podem ser empregadas por agentes, um protocolo de aprendizado é em geral composto pelos seguintes passos [95, 105]: (i) um evento relevante é detectado; (ii) o evento é capturado e informação relevante sobre o mesmo é recolhida para a realização do processo de aprendizado; (iii) um algoritmo de aprendizado processa a informação coletada;

(iv) a informação é armazenada e como resultado pode trazer novas conclusões; e (v) caso uma nova conclusão seja obtida, o conhecimento do agente é atualizado de forma a influenciar diretamente no seu comportamento.

A característica de Aprendizagem é endereçada no framework AspectT por um padrão de projeto o qual oferece diretrizes para o projeto e implementação de tal característica usando mecanismos de orientação a aspectos. O padrão de projeto *Learning Aspect* [49] tem como propósito modularizar o interesse de aprendizado, separando completamente a estrutura básica do agente do seu protocolo de aprendizado. A estrutura geral do padrão é apresentada na Figura 50. O padrão possui 4 participantes, sendo eles: (i) *Learning Aspect* – o qual define o protocolo de aprendizagem; (ii) *Specific Learning* – responsável por implementar a parte específica da característica de aprendizagem de um tipo ou papel de agente; (iii) *Learning Component* – implementa estratégias de aprendizado específicas; e (iv) *Knowledge Element* – oferece eventos e informações contextuais que são relevantes para o processo de aprendizagem.

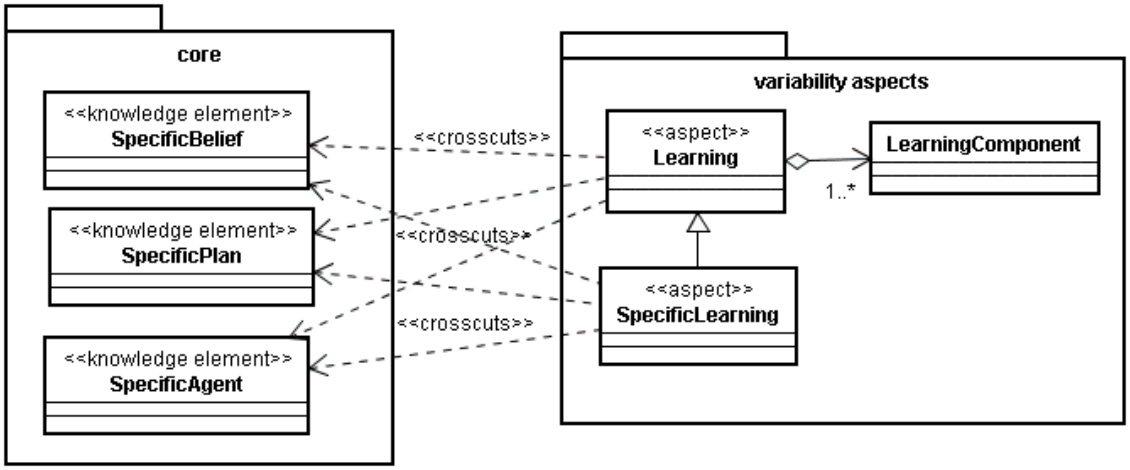

Figura 50. Estrutura do Padrão de Projeto Learning

Para ilustrar o uso do padrão de projeto *Learning Aspect* na construção de aspectos de variabilidade, apresentaremos a solução adotada na instanciação do framework AspectT para o sistema ExpertCommittee (EC). Uma das técnicas de aprendizado usadas em tal sistema para capturar as preferências dos usuários foi a *Temporal Different Learning* (TD-Learning) [95]. Os papéis de agente *Chair* e *Reviewer* do EC usam ambos a técnica de TD-Learning. O papel *Reviewer* utiliza tal técnica para capturar as preferências do usuário nos assuntos de interesse do revisor que o agente representa. Já o papel *Chair* usa tal técnica para aprender as

preferências de cada um dos revisores do sistema. É necessário a coleta de diversas informações do sistema de forma a permitir a execução efetiva de tais técnicas de aprendizado, tais como: escolhas feitas pelos usuários reais do sistema e resultado das interações entre agentes *Chair* e *Reviewer* durante avaliação de propostas de revisões de artigos. A seguir é descrita a implementação dos aspectos de aprendizagem do papel *Chair* no contexto do sistema EC.

A Figura 51 apresenta o código do aspecto abstrato `Learning`. Tal aspecto especifica: (i) um conjunto de atributos e métodos a serem introduzidos na classe plano `RevisionProposal` (linhas 7-24), os quais representam informações e comportamento úteis para implementação da característica de Aprendizagem; (ii) uma referência para uma instância da classe `TDLearning` que representa o algoritmo de aprendizado a ser utilizado pelo sistema (linha 5); e (iii) o ponto de corte abstrato `interestDegreeLearning()` – que representa pontos de execução nos quais os algoritmos de aprendizado serão executados (linhas 25-26) e que deve, portanto, ser concretizado por subaspectos de `Learning`.

```
01 public abstract aspect Learning {
02   public static final int ACCEPTED_PAPER = 200;
03   public static final int REJECTED_PAPER = 0;
04   public static final double LR = 0.1;
05   protected TDLearning learningAlgorithm;
06
07   private Hashtable RevisionProposal.proposalEvaluation =
08                            new Hashtable();
09   private int RevisionProposal.currentPaperInterest = 0;
10
11   public Hashtable RevisionProposal.getEvaluation() {
12     return proposalEvaluation;
13   }
14   public void RevisionProposal.setEvaluation(
15                                 Hashtable evaluation) {
16     this.proposalEvaluation = evaluation;
17   }
18   public int RevisionProposal.getPaperInterest() {
19     return currentPaperInterest;
20   }
21   public void RevisionProposal.setPaperInterest(
22                                           int interest) {
23     this.currentPaperInterest = interest;
24   }
25   protected abstract pointcut interestDegreeLearning(
26                 RevisionProposal proposal, Plan plan);
27 }
```

Figura 51. Aspecto de Extensão `Learning`

A Figura 52 mostra o aspecto `ChairLearning` que implementa as funcionalidades de aprendizado do papel *Chair* do EC. Ele define: (i) o ponto de corte `learningInitialization()` com um adendo associado, os quais são responsáveis pela inicialização das áreas de interesses de pesquisa de cada revisor (linhas 5-13); e (ii) uma concretização para o ponto de corte `interestDegreeLearning()` (linhas 14-18), o qual define a invocação dos algoritmos definidos pela classe `TDLearning` sempre que um agente `Chair` recebe o resultado de uma proposta de revisão (linhas 20-41). A proposta de revisão é analisada para definir se atualizações devem ser feitas nas preferências de áreas de interesse de revisores. A mudança em tais preferências, a partir da execução dos algoritmos de aprendizado, ocasiona a modificação do comportamento do sistema. Detalhes adicionais sobre os aspectos de aprendizado implementados para o EC podem ser encontradas em [46, 49].

```
01 public aspect ChairLearning extends Learning {
02   //(reviewer name, table of research interests)
03   public Hashtable reviewers = new Hashtable();
04
05   before(Agent agent, Reviewer reviewer, List papers):
06        learningInitialization(agent, reviewer, papers) {
07     String reviewerName = reviewer.getName();
08     Hashtable reviewer_interests =
09                 (Hashtable) reviewers.get(reviewerName);
10
11     // Update the reviewer interests
12     ...
13   }
14   protected pointcut interestDegreeLearning(
15               RevisionProposal proposal, Plan plan): (
16     this(plan) && args(proposal) &&
17     execution(void ProposalJudgementReceptionPlan.
18                 verifyReviewerResponse(RevisionProposal)));
19
20   after(RevisionProposal proposal, Plan plan):
21               interestDegreeLearning(proposal, plan) {
22     boolean acceptedProposal = proposal.isAccepted();
23     ResearchArea area =
24                proposal.getPaper().getResearchArea();
25     Vector paperKeywords = area.getResearchKeywords();
26
27    Hashtable reviewerEvaluation = proposal.getEvaluation();
28     int reviewerInterest = proposal.getPaperInterest();
29     Reviewer reviewer = proposal.getReviewer();
30     String reviewerName = reviewer.getName();
31
32     Hashtable reviewerPreferences =
33                (Hashtable) reviewers.get(reviewerName);
34
35     Hashtable newPreferences =
36          learningAlgorithm.learnAgentPreference(
37                   reviewerPreferences, paperKeywords,
38                   acceptedProposal, reviewerInterest);
39     learningAlgorithm.updatePreferences(
40                   reviewerPreferences,newPreferences);
41   }
42 }
```

Figura 52. Aspecto de Extensão `ChairLearning`

6.2.4.2. Mobilidade

A característica de Mobilidade está relacionada com o comportamento do agente necessário a sua movimentação para ambientes remotos. Agentes podem se mover de um ambiente/máquina em uma rede de computadores para outro(a) de forma a alcançar seus objetivos. A característica de Mobilidade envolve o endereçamento de vários interesses [116], tais como: (i) a especificação dos elementos móveis; (ii) a descrição de situações nas quais os agentes devem se mover e quando devem retornar, bem como as respectivas ações a serem tomadas antes de tais deslocamentos; e (iii) o controle dos itinerários dos agentes.

No framework AspectT, a característica de Mobilidade também foi endereçada pela elaboração de um padrão de projeto. O padrão de projeto *Mobility Aspect* [48] propõe a modularização do interesse de mobilidade de forma separada dos interesses básicos do agente e de colaboração. Ele possui 5 participantes: (i) *Mobile Element* – representa elementos do agente móvel, cujo principal propósito é modularizar outros interesses do agente, tais como um tipo ou papel de agente; (ii) *Mobility Aspect* – implementa a parte genérica do interesse de mobilidade; (iii) *Specific Mobility* – endereça a parte específica do interesse de mobilidade de determinados tipos ou papéis de agentes; e (iv) *Mobility Framework* – oferece um conjunto de serviços de mobilidade que podem ser usados na aplicação.

Para ilustrar o uso do padrão de projeto *Mobility Aspect*, vamos apresentar a instanciação do mesmo no contexto do sistema ExpertCommittee (EC). A Figura 53 apresenta a aplicação do padrão na implementação do papel *Chair* no EC. O aspecto `Mobility` afeta as classes `Agent`, `Goal` e `Plan` para introduzir comportamento de mobilidade. O subaspecto `ChairMobility` introduz o comportamento específico de mobilidade relacionado ao papel *Chair* entrecortando as classes `ResearcherUserAgent` e `InformationSearchingPlan`. Finalmente, a classe `MobilityModule` se responsabiliza pela comunicação com a plataforma JADE de forma a poder usufruir dos serviços de mobilidade oferecidos.

A Figura 54 mostra o código do aspecto abstrato de extensão `Mobility`. Ele é responsável por definir o comportamento básico do interesse de mobilidade. Ele introduz diversos atributos (linhas 4-8) e métodos (17-33) na interface

`MobileElement`, através do uso de declarações inter-tipos. Essa interface é usada para representar um elemento móvel. Subaspectos de `Mobility` definem quais tipos de agente devem implementar tal interface. Os seguintes atributos e métodos são introduzidos em tal interface: (i) atributo `home` – o qual armazena o computador original do agente (linha 4); (ii) atributo `location` – armazena a localização atual do agente (linha 5); (iii) atributo `mobilityComponent` – oferece serviços para o agente se deslocar para outros ambientes usando um framework de mobilidade específico (linha 7). No sistema EC, o componente de mobilidade foi implementado usando o framework JADE; (iv) atributo booleano `agentOut` – indica se o agente se moveu para outro ambiente diferente de sua localização original (linha 8); e (v) métodos `prepareMessageDeparture()` e `prepareMessageArrival()` – que são usados para a definição de mensagens de notificações durante a partida ou chegada, respectivamente, de um agente a um dado ambiente (linhas 17-24).

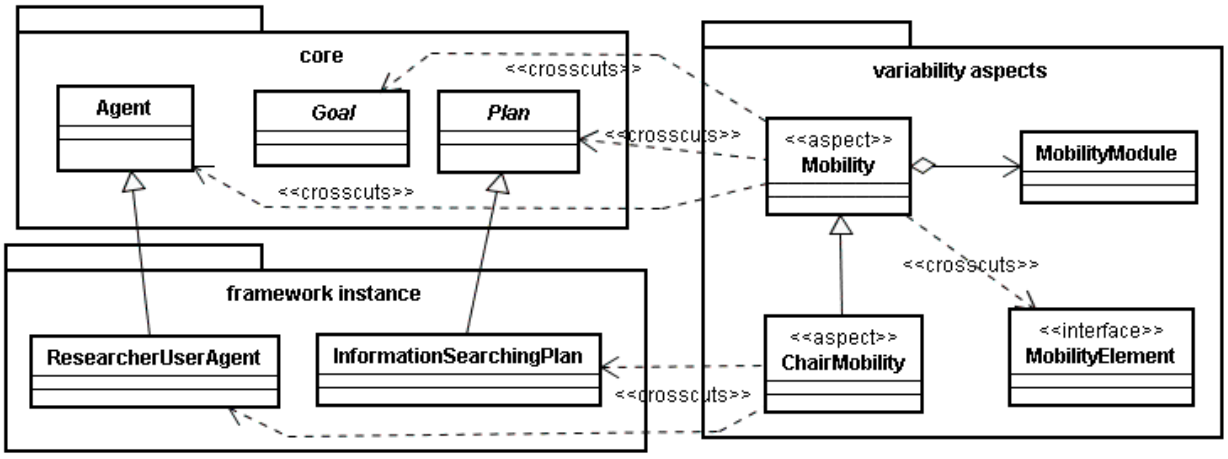

Figura 53. Aspecto de Extensão `Mobility`

Como também pode ser visto na Figura 54, o aspecto `Mobility` também define diversos pontos de corte e métodos abstratos a serem implementados por aspectos concretos de mobilidade, entre eles: (i) ponto de corte `moving()` – define pontos de junção na execução do agente que são candidatos para inicializar a sua mobilidade (linha 35); (ii) método `init()` – inicializa os serviços de mobilidade do agente em uma dada plataforma (linha 15) ; (iii) método `move()` – usado para definir as ações a serem executadas para movimentar um agente para um outro

ambiente (linha 16); (iv) método `initializeAgentInNewEnvironment()` – define a inicialização do agente após a sua movimentação para um outro ambiente (linha 44-45); e (v) método `checkMobilityNeed()` – verifica a necessidade de movimentação ou não do agente (linhas 46-47). Subaspectos de mobilidade devem concretizar tais pontos de corte e métodos de forma a endereçar a implementação do interesse de mobilidade para algum tipo ou papel de agente.

A Figura 55 apresenta o subaspecto de mobilidade `ChairMobility`. Ele é responsável por implementar o interesse de mobilidade para o papel *Chair* do EC. Agentes que assumem tal papel precisam se mover quando há uma falha na obtenção de alguma informação por algum de seus planos. Esse aspecto define, por exemplo, que o ponto de corte `moving()` deve interceptar o método `searchInformation()` da classe `InformationSearchPlan`, para decidir pela necessidade de mobilidade ou não do agente (linhas 21-23). No EC, a plataforma de mobilidade utilizada foi baseada no framework JADE. Esse framework oferece um conjunto de serviços para implementação do interesse de mobilidade do agente. A utilização de tais serviços requer a especialização da classe `Agent` do framework JADE. A classe `JADEModule` assume tal propósito, e é usada pelo aspecto `ChairMobility` para implementar as funcionalidades de mobilidade associadas aquele papel. O método `init()`, por exemplo, é usado para inicializar as funcionalidades de mobilidade do agente na plataforma JADE (linhas 8-13). Ele invoca o método `createJADEReference()` da classe `JADEModule`, para criar um agente JADE que permitirá usufruir dos serviços de mobilidade de tal plataforma (linha 10). Já o método `move()` do aspecto `ChairMobility`, invoca o método `doMove()` da classe `JADEModule` (linhas 14-20).

```
01 public abstract aspect Mobility {
02   declare parents:
03       Agent || AbstractPlan || ... implements Serializable;
04   private String MobileElement.home;
05   private String MobileElement.location;
06
07   protected JADEModule MobileElement.mobilityComponent;
08   private boolean MobileElement.agentOut;
09   protected abstract pointcut agentInstantiation(
10                                          Agent agent);
11   after(Agent agent) : agentInstantiation(agent) {
12     agent.setAgentOutFalse();
13     init(agent);
14   }
15   public abstract void init(Agent agent);
16   public abstract void move(Agent agent);
17   public Message
18       MobileElement.prepareMessageDepartureNotification(){
19     ...
20   }
21   public Message
22      MobileElement.prepareMessageArrivalNotification() {
23      ...
24   }
25   public boolean MobileElement.isAgentOut() {
26      return this.agentOut;
27   }
28   public void MobileElement.setAgentOutTrue() {
29     this.agentOut = true;
30   }
31   public void MobileElement.setAgentOutFalse() {
32     this.agentOut = false;
33   }
34   ...
35   protected abstract pointcut moving(Plan plan);
36   after(Plan plan) returning(Object result): moving(plan) {
37     Agent agent = plan.getAgent();
38     boolean moveAgent = checkMobilityNeed(agent, result);
39     if (moveAgent && (!agent.isAgentOut())) {
40       agent.prepareDepartureNotification();
41       move(agent);
42     }
43   }
44   public abstract void initializeAgentInNewEnvironment(
45                          Object mobileAgent);
46   protected abstract boolean checkMobilityNeed(Agent agent,
47                   Object result);
48   ...
49 }
```

Figura 54. Aspecto de Extensão `Mobility`

```
01 public aspect ChairMobility extends Mobility {
02   declare parents: ResearcherUserAgent implements
03                                              MobileElement;
04   declare parents: Agent || Agenda implements Serializable;
05   protected pointcut agentInstantiation(MobileElement agent):
06              this(agent) &&
07              initialization(ResearcherUserAgent+.new(..));
08   public void init(MobileElement agent) {
09     JADEModule mobilityComponent =
10            JADEModule.createJADEReference((Agent)agent);
11      agent.setMobilityModule(mobilityComponent);
12      ...
13   }
14   public void move(MobileElement agent) {
15     Location destination = new
16         jade.core.ContainerID(agent.getCurrentDestination(),
17                               null);
18     ((JADEModule)agent.getMobilityModule()).
19                                    doMove(destination);
20   }
21   protected pointcut moving(Plan plan):
22     (this(plan) && execution(Hashtable
23     InformationSearchingPlan.searchInformation(..)));
24
25   protected boolean checkMobilityNeed(MobileElement agent,
26                                       Object result) {
27    ...
28   }
29   protected pointcut afterMove(Object JADEagent):
30      this (JADEagent) &&
31      execution(* JADEModule.afterMove());
32   public void initializeAgentInNewEnvironment(Object
33     JADEagent) {
34     ...
35   }
36   pointcut informationNeedChecking(Plan plan):
37     this(plan) &&
38     execution(void PaperDistributionPlan.executePlan(..));
39
40   before(Plan plan): informationNeedChecking(plan) {
41     ...
42   }
43 }
```

Figura 55. Aspecto de Extensão `ChairMobility`

6.2.4.3. Colaboração

A característica de Colaboração possibilita a um dado agente interagir com outros agentes através do desempenho de papéis. Um papel de agente oferece capacidades extras de conhecimento, interação, adaptação ou autonomia. Durante sua execução, é comum um agente assumir diferentes papéis. Na nossa implementação, cada papel é implementado por uma interface e um aspecto. A interface especifica o comportamento do papel, através da definição de métodos. O aspecto define que a classe `Agent` ou algum de seus subtipos deverá implementar tal interface. Atributos que devem existir no papel do agente podem também ser incluídos na interface pelo aspecto, usando declarações inter-tipo. Esses atributos representam crenças específicas determinadas pelos papéis. Planos e objetivos específicos do papel podem também ser definidos. A definição de novas propriedades para um papel de agente, podem ser implementadas, através da especialização dos aspectos abstratos `Autonomy`, `Adaptation`, `Mobility` e `Learning`.

Dois papéis de agentes foram implementados para o sistema EC, são eles: `Chair` e `Reviewer`. A Figura 56 mostra o código do aspecto de extensão `Chair` do EC. Ele é responsável pela introdução de diversos atributos e métodos na classe `ResearcherUserAgent`, que são relacionados a implementação do papel `Chair` no sistema. Os seguintes atributos são definidos, entre eles: (i) um plano para a distribuição de artigos para o comitê de programa (linha 3); (ii) uma lista de artigos que foram submetidos (linha 4); uma lista de revisores de artigos (linha 5); e (iv) a data limite para submissão de artigos (linha 6). Métodos acessores para tais atributos podem também ser definidos, assim como outros que representem serviços específicos daquele papel de agente. A implementação de um papel de agente, envolve também a definição de novas classes representando objetivos, planos e crenças. Além disso, propriedades específicas de um dado papel de agente, podem também ser implementadas na forma de subaspectos dos aspectos do núcleo e extensão apresentados e exemplificados nas seções anteriores.

```
1 public aspect Chair {
2    ...
3    private DistributionPlan UserAgent.distributionPlan;
4    private List UserAgent.papersList;
5    private List UserAgent.reviewerList;
6    private GregorianCalendar UserAgent.paperSubmissionDate;
7    ...
8    public List ResearcherUserAgent.getPapersList() {
9      return this.papersList;
10   }
11   public List ResearcherUserAgent.getReviewerList() {
12     return this.reviewerList;
13   }
14   public void ResearcherUserAgent.setPapersList(
15                                        List papersList) {
16     this.papersList = papersList;
17   }
18   public void ResearcherUserAgent.requestInformation(){
19     ...
20   }
21 }
```

Figura 56. Aspecto de Extensão de Colaboração `Chair`

6.2.5. Modelo Generativo

O modelo generativo do framework AspectT é apresentado na Figura 57. O modelo de arquitetura contempla os seguintes componentes principais: (i) core – o qual agrega todas as classes e aspectos do núcleo do framework na forma de um conjunto de subcomponentes relacionados com cada uma das características do framework, tais como, knowledge, interaction, adaptation, autonomy e ejps. O componente ejps é responsável por agregar o aspecto que expõe os pontos de junção de extensão definidos pelo AspectT; e (ii) extensions - agrega aspectos e classes relacionadas com a implementação de extensões ao framework AspectT, é organizado nos seguintes subcomponentes: learning, mobility e collaboration. Cada um dos componentes do modelo de arquitetura do AspectT também agrega um conjunto de templates que, em geral, representam: (i) especializações de classes ou aspectos abstratos do framework (exemplos: templates `UserAgentTemplate`, `InteractionTemplate` e `AdaptationTemplate`); e (ii) arquivos de configuração do sistema, tais como, um arquivo que especifica a adaptação de planos (template `PlanAdaptationXMLFile`) – determinando quais objetivos demandam a execução de quais planos), e um arquivo que especifica a

automonia reativa (template `ReactiveAutonomyXMLFile`) – o qual determina quais objetivos devem ser instanciados a partir do recebimento de determinadas mensagens.

O modelo de característica do AspectT permite a definição de vários agentes (característica Agent). Cada agente pode definir suas características e propriedades obrigatórias, tais como: (i) Knowledge – onde podem ser definidos crenças, objetivos e planos do agente; (ii) Interaction – essa característica permite especificar os sensores e efetuadores a serem usados pelo agente, assim como mensagens específicas que ele pode processar; (iii) Adaptation – permite a associação de planos a determinados objetivos (Plan Adaptation) e a atualização de crenças a partir do recebimento de determinadas mensagens (Belief Adaptation); e (iv) Autonomy – permite a definição de instanciação de objetivos a partir do recebimento de determinadas mensagens (Reactive Autonomy), assim como a escolha da estratégia de concorrência a ser usada no processamento dos planos (Execution Autonomy). Para cada agente pode também ser definido um conjunto de características opcionais, tais como, Learning, Mobility e Roles. As duas primeiras são especificadas para permitir a criação dos aspectos de extensão responsáveis pela sua implementação. A característica Role representa a especificação dos papéis de agente. Ela permite especificar características de Knowledge, Interaction, Adaptation e Autonomy, Learning e Mobility específicas do papel do agente.

O modelo de configuração do AspectT define uma série de relações de dependência entre os elementos de implementação e características. A implementação de várias das características transversais de agentes usando aspectos, facilita a criação de tais relações de dependência, contribuindo para uma correspondência direta entre características e elementos de implementação. Cada um dos templates que representam especializações de classes ou aspectos abstratos do framework é diretamente relacionado com a característica responsável por coletar informações para a sua instanciação. Exemplos são: `PlanTemplate` relacionado com a característica Plan; `AutonomyTemplate` relacionado com a característica Autonomy, e assim por diante. Em seu estágio atual, o modelo de configuração do AspectT apenas define relações de dependência entre elementos de implementação e características. Relações tranversais válidas e mapeamentos entre características `<<joinpoint>>` e EJPs

não foram necessárias nesse estudo de caso, porque a maioria dos aspectos possui pontos de corte fixos, e, portanto pré-definidos. Em trabalhos futuros pretendemos explorar a extesão do modelo generativo do AspectT para endereçar variabilidades em pontos de corte de aspectos representando as características Learning e Mobility, principalmente.

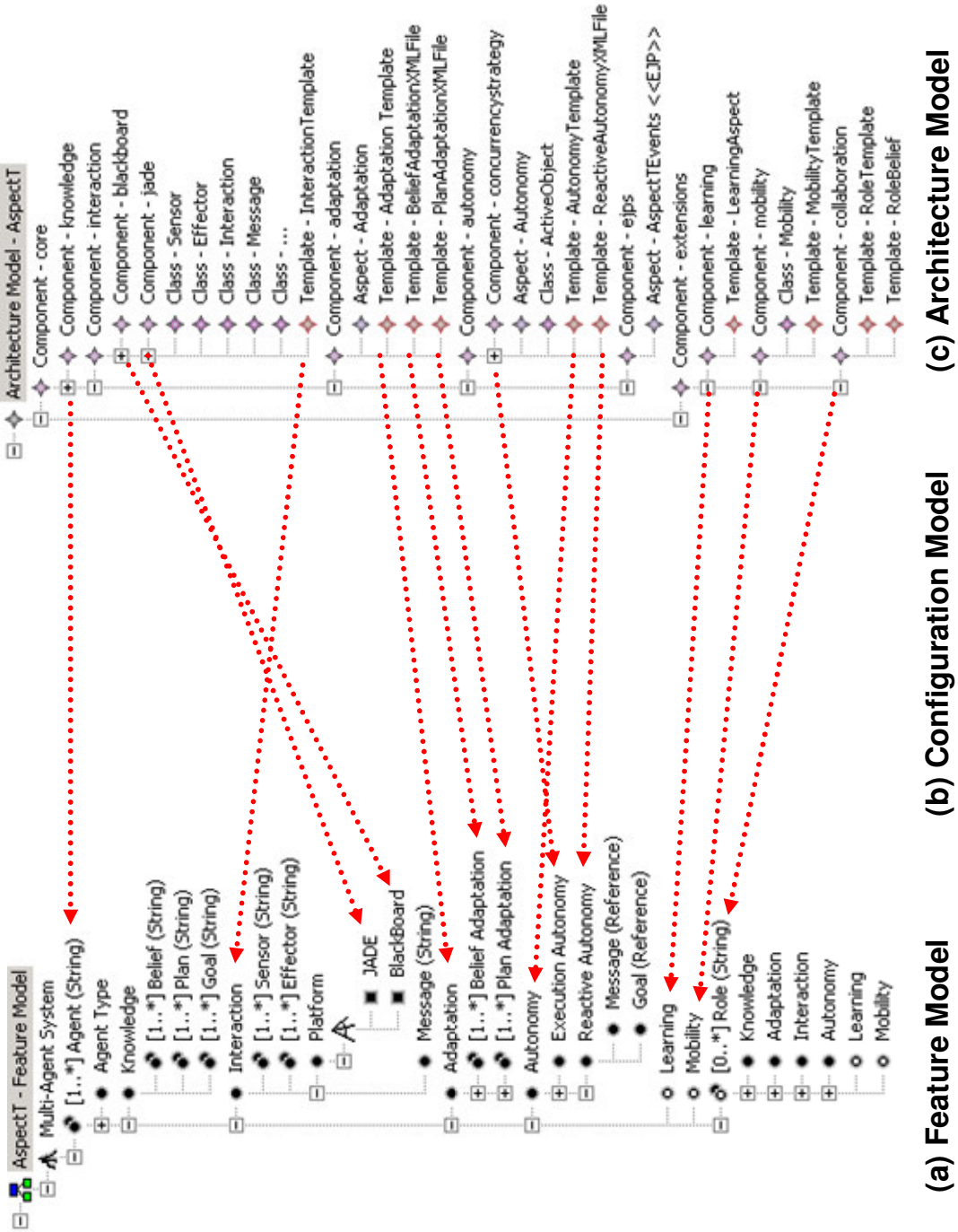

Figura 57. Modelo Generativo do AspectT

6.3. Linha de Produto de Jogos J2ME

Java 2 Micro Edition[18] (J2ME) é uma plataforma de tecnologias para o desenvolvimento de aplicações para dispositivos móveis. Nos últimos anos, J2ME tem sido usada para o desenvolvimento de jogos para celulares com complexidade considerável. Nossa abordagem também foi usada em um estudo de caso de uma linha de produto de jogos J2ME, da empresa *Meantime Mobile Creations*. O estudo de caso envolveu a refatoração da linha de produto do jogo *Rain of Fire*, originalmente desenvolvida em J2ME. Tal estudo foi conduzido por membros do *Software Productivity Group* da Universidade Federal de Pernambuco (UFPE), que participaram diretamente no desenvolvimento e evolução da abordagem baseada em EJPs.

A adaptação de jogos desenvolvidos usando a tecnologia J2ME para diferentes dispositivos portáteis, ocasiona o surgimento de várias variabilidades [3]. Essas variabilidades surgem, principalmente, em função de características do dispositivo no qual o jogo será instalado, tais como, restrições de hardware oferecidas e bibliotecas proprietárias. Exemplos de variabilidades [3, 5] para esse domínio de jogos são: características de interface gráfica (tais como, tamanho de tela, número de cores, som), memória disponível, tamanho da aplicação e bibliotecas proprietárias para manipulação de imagens. Assim, cada jogo construído para a plataforma J2ME pode ser visto como uma linha de produto em função da sua adaptação para diferentes dispositivos.

6.3.1. Núcleo do Framework

A estrutura e comportamento de linhas de produto de jogos é tipicamente definida por um framework conhecido nesse domínio como *Game Engine*. Um *Game Engine* pode ser caracterizado como uma máquina de estados cujas transições são definidas em função do tempo transcorrido do jogo e interações do usuário através do teclado. As mudanças de estado afetam objetos visuais do jogo

[18] Sun Microsystems. Java 2 Platform, Micro Edition (J2me). URL: http://java.sun.com/j2me/. 2004.

(tais como, atores e ambiente) e a forma como eles interagem. Dessa forma, cada mudança de estado demanda modificações visuais nos objetos sendo representados na tela do jogo. Pontos de extensão do framework incluem tipicamente classes abstratas que definem operações básicas para desenho de atores do jogo. A Figura 58 apresenta as classes principais do núcleo do framework do jogo *Rain of Fire*. A Figura 59 mostra como essas classes colaboram para implementar o fluxo definido para o *game engine*.

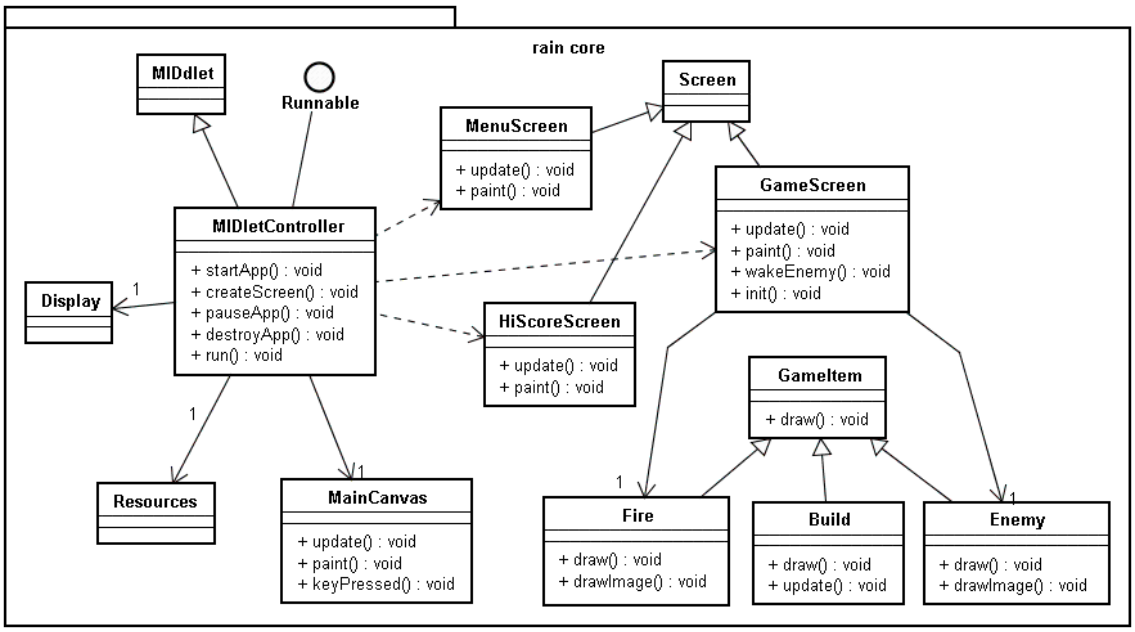

Figura 58. Diagrama de Classes do núcleo do *Rain of Fire*

6.3.2. Pontos de Junção de Extensão

O *Game Engine* que define o núcleo do framework da linha de produto deve definir pontos de junção de extensão (EJPs) para permitir a composição de extensões tranversais na sua funcionalidade básica. Os seguintes EJPs foram definidos: (i) operações de inicialização e uso de imagens do jogo; (ii) operações de desenho de imagens específicas; e (iii) eventos de inicialização e de mudanças de tela do jogo. Esses EJPs foram definidos porque representam eventos relevantes do fluxo de execução do *game engine*. A Figura 59 mostra alguns dos EJPs expostos pelo *game engine* dentro do fluxo principal de colaboração de suas classes. A Figura 60 apresenta a arquitetura geral do jogo *Rain of Fire* com seus

respectivos aspectos EJPs e aspectos de extensão. A seção seguinte descreve como vários dos EJPs são usados pelos aspectos de extensão para estender o comportamento do *game engine*.

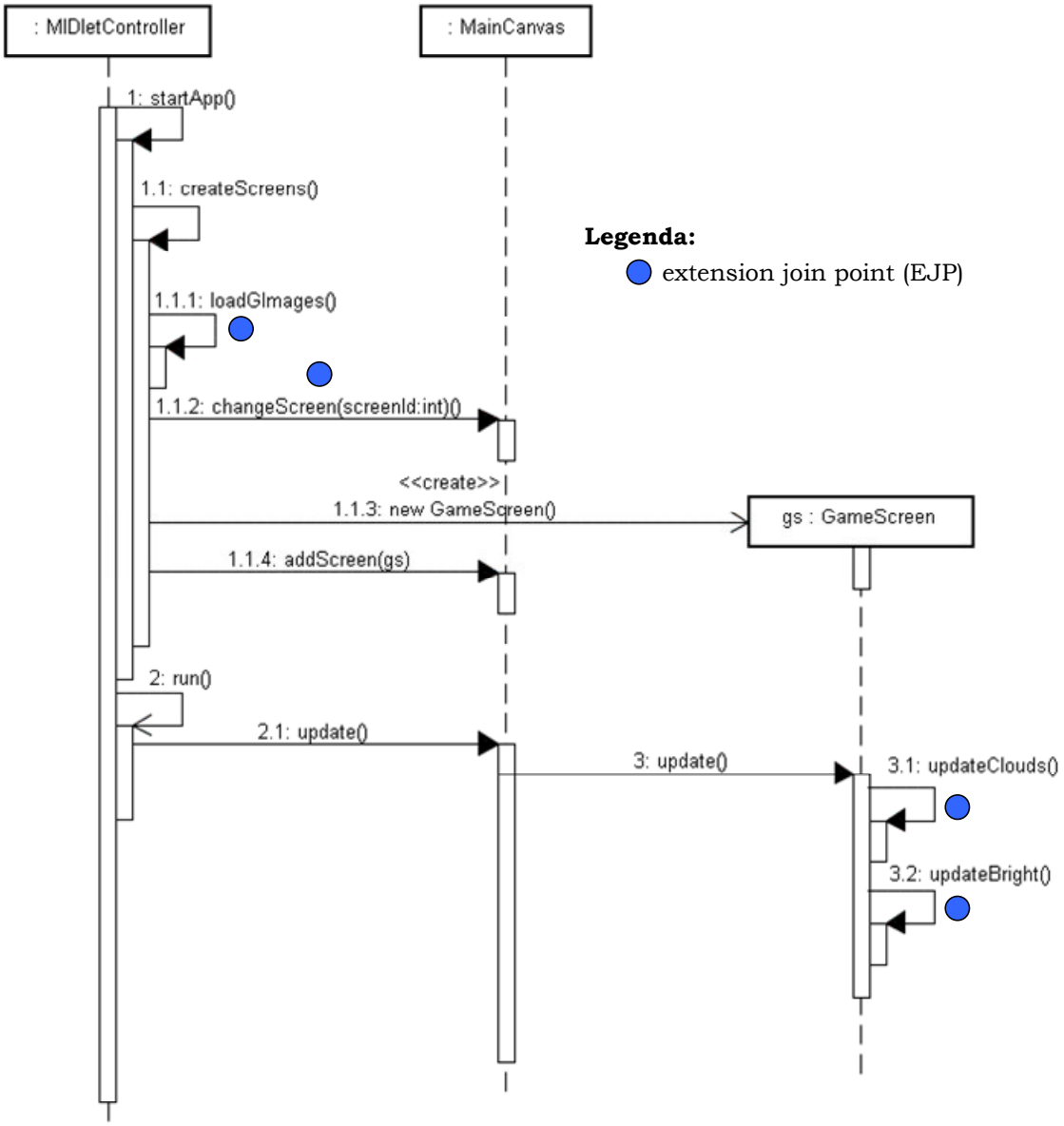

Figura 59. Diagrama de Seqüência do *Rain of Fire*

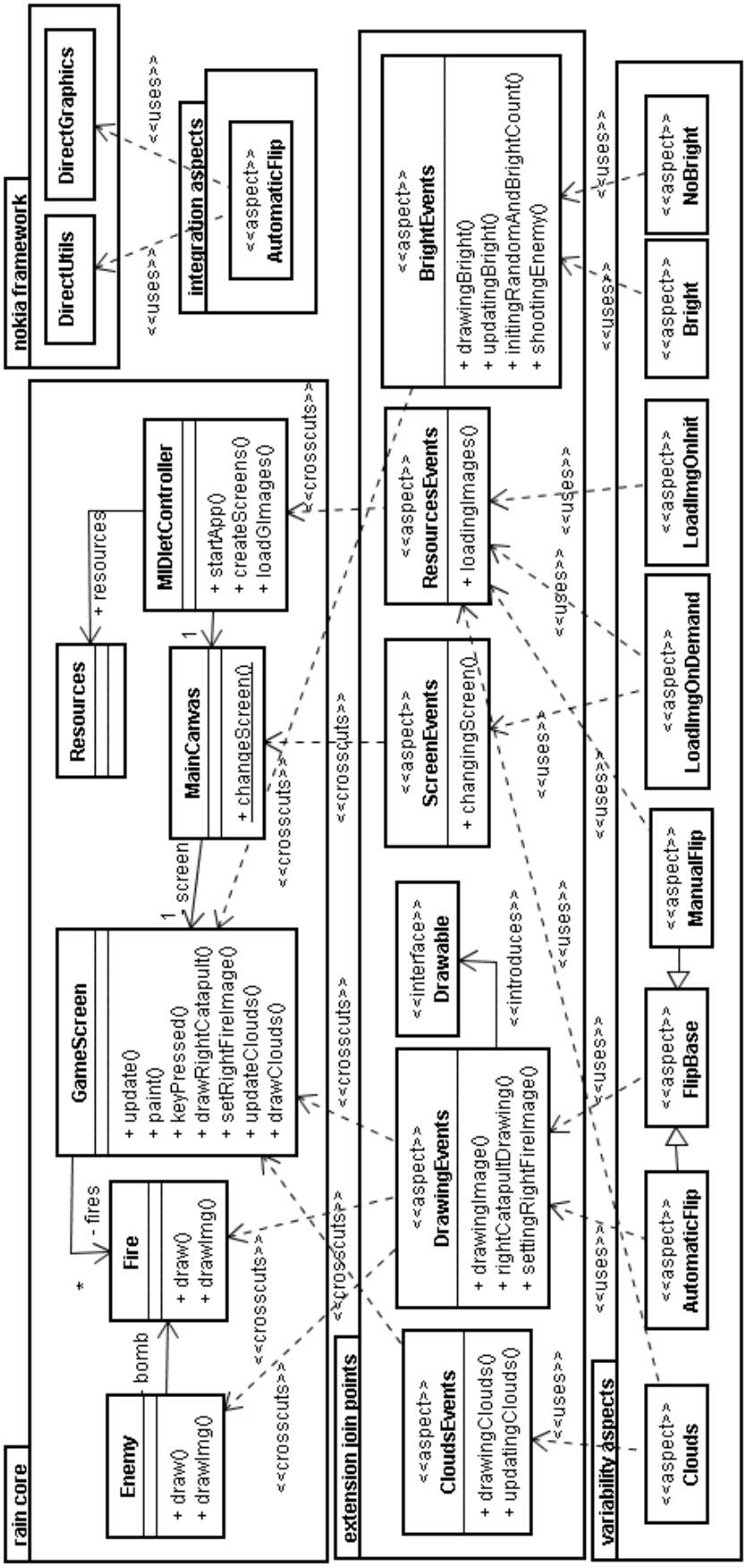

Figura 60. Arquitetura da Linha de Produto *Rain of Fire*

6.3.3. Aspectos de Extensão

Diversos aspectos de extensão foram implementados para endereçar características transversais opcionais e alternativas existentes na linha de produto. Esses aspectos representam a implementação de variabilidades presentes em jogos J2ME. A Figura 60 mostra o núcleo do framework, com seus respectivos EJPs e aspectos de extensão.

Aspectos de variabilidade foram definidos para implementar a característica opcional Croma que representa imagens decorativas presentes no cenário de um jogo. Um exemplo é a existência de imagens representando nuvens que passam no fundo de tela de determinados cenários do jogo. A característica Croma é considerada opcional na linha de produto, porque dispositivos com baixa capacidade de recursos podem não incluir tais características. O aspecto de variabilidade `Clouds` representa a implementação da característica Croma para a linha de produtos do jogo *Rain of Fire*. Observe na Figura 60 que esse aspecto faz uso dos EJPs `CloudsEvents` e `ResourceEvents` para inserir tal funcionalidade no núcleo do framework que define o jogo. A Figura 61 mostra o código parcial do aspecto `Clouds`. A implementação dele envolve: (i) declarar atributos para carregar tais arquivos de imagens (linhas 2-7); (ii) carregar arquivos de imagens (linhas 15-25); e (iii) desenhar e atualizar tais imagens em função de mudanças no estado do jogo (linhas 8-14 e linhas 26-40). O código responsável pelo desenho dessas imagens decorativas é encontrado em diferentes trechos de código de várias classes. Dessa forma, ele pode ser visto como sendo uma característica transversal.

Aspectos de variabilidade e integração foram também codificados para implementar características alternativas de desenho de imagens. Imagens específicas do jogo podem ser desenhadas em vários trechos de código e, em algumas circunstâncias, podem ser transformadas (rotacionadas, flipped), manualmente usando novas imagens (aspecto `ManualFlip`) ou manipulando as imagens originais através do uso de bibliotecas proprietárias (aspecto `AutomaticFlip`). O aspecto `AutomaticFlip` é considerado de integração porque ele envolve interação com bibliotecas proprietárias. A Figura 60 apresenta tais aspectos e os respectivos EJPs `DrawingEvents` e `ResourcesEvents` que garantem

uma implementação modular para a implementação de suas respectivas características.

```
01 public privileged aspect Clouds {
02   public static Image clouds01 = null;
03   public static Image clouds02 = null;
04   ...
05   public int clouds01_x;
06   public int clouds02_x;
07   ...
08   void around(GameScreen gs):
09            CloudsEvents.updatingClouds(gs) {
10     updateClouds(gs);
11   }
12   void around(Graphics g): CloudsEvents.drawingClouds(g) {
13     drawClouds(g);
14   }
15   before(): ResourcesEvents.loadingImages() {
16     loadImages();
17   }
18   public void loadImages() {
19     try {
20       clouds01 = Image.createImage(... + "clouds01.png");
21       ...
22     } catch(IOException ioe) {
23       ...
24     }
25   }
26   protected void updateClouds(GameScreen gs) {
27     if (clouds01_x <= -clouds01.getWidth()&&gs.aux==255){
28       clouds01_x = Resources.CANVAS_WIDTH;
29     }
30     ...
31     // Updates clouds movement
32     if (MainCanvas.frame % 8 == 0) {
33       clouds01_x -= 1;
34     }
35     ...
36   }
37   protected void drawClouds(Graphics g) {
38     g.drawImage(clouds01, clouds01_x, 87, g.TOP | g.LEFT);
39     ...
40   }
41 }
```

Figura 61. Aspecto de Variabilidade `Clouds`

Finalmente, aspectos de variabilidade foram também definidos para implementar a característica opcional de otimização. Essa característica envolve a carga de imagens por demanda quando alternando entre telas do jogo. Essa política de carga é usada para dispositivos que possuem restrições nos recursos disponíveis (ex: memória). Dispositivos sem tais restrições não necessitam da implementação de tal característica, e seu código carrega todas as imagens do jogo

durante a inicialização do mesmo. A Figura 60 mostra os aspectos `LoadImgOnDemand` e `LoadImgOnInit` que realizam a carga de imagens, respectivamente, por demanda ou na inicialização. Os EJPs `ScreenEvents` e `ResorcesEvents` são usados por tais aspectos para permitir a extensão do núcleo do framework. Detalhes adicionais sobre a implementação desse estudo de caso podem ser encontradas em [5, 73].

6.3.4. Modelo Generativo

O modelo generativo da linha de produtos do jogo J2ME *Rain of Fire* é apresentado na Figura 62. Ele apresenta o modelo de arquitetura agregando os elementos de implementação do jogo em 2 componentes principais: core e extensions. O componente core agrega as classes que definem o comportamento básico do jogo, assim como os aspectos EJPs (`ResourceEvents` e `DrawingEvents`) que são compartilhados por vários dos aspectos de variabilidade. O componente extensions agrega vários sub-componentes cada um deles representando as variabilidades da linha de produto, sendo eles: bright, flip, clouds e imageloading. Observe que cada um desses componentes agrega os respectivos aspectos de variabilidade associados. Os componentes bright e clouds agregam ainda os aspectos EJPs (`BrightEvents` e `CloudEvents`, respectivamente) que são associados apenas aquelas variabilidades.

O modelo de características do jogo *Rain of Fire* apresenta apenas as variabilidades da linha de produto, contendo: (i) 2 características opcionais (Bright e Clouds) e (ii) 2 características alternativas (Flip e Image Loading). Todas essas características são `<<crosscutting>>` porque representam a implementação de aspectos de extensão. Nesse estudo de caso não foi necessário representar nenhuma característica `<<join point>>`, porque todos os aspectos do modelo de característica possuem pontos de corte fixos. Como conseqüência, o modelo de configuração desse estudo de caso não necessita definir: (i) relações transversais válidas entre características `<<crosscutting>>` e `<<join point>>`; e (ii) o mapeamento entre características `<<join point>>` e pontos de junção concretos.

O modelo de configuração desse estudo de caso se resume, portanto, as relações de dependência entre elementos de implementação e características. A Figura 62 apresenta tais relações de dependência. O aspecto `Bright` depende da seleção da característica Bright. Os aspectos do componente flip dependem, respectivamente, de cada uma das alternativas (Automatic, Manual) da característica Flip. O mesmo ocorre para os aspectos do componente imageloading que dependem das alternativas oferecidas para as características Image Loading. Dois elementos de implementação dependem da da característica alternativa On Demand: (i) o aspecto `LoadImgOnDemand` e (ii) o aspecto EJP `ScreenEvents` usado pelo primeiro.

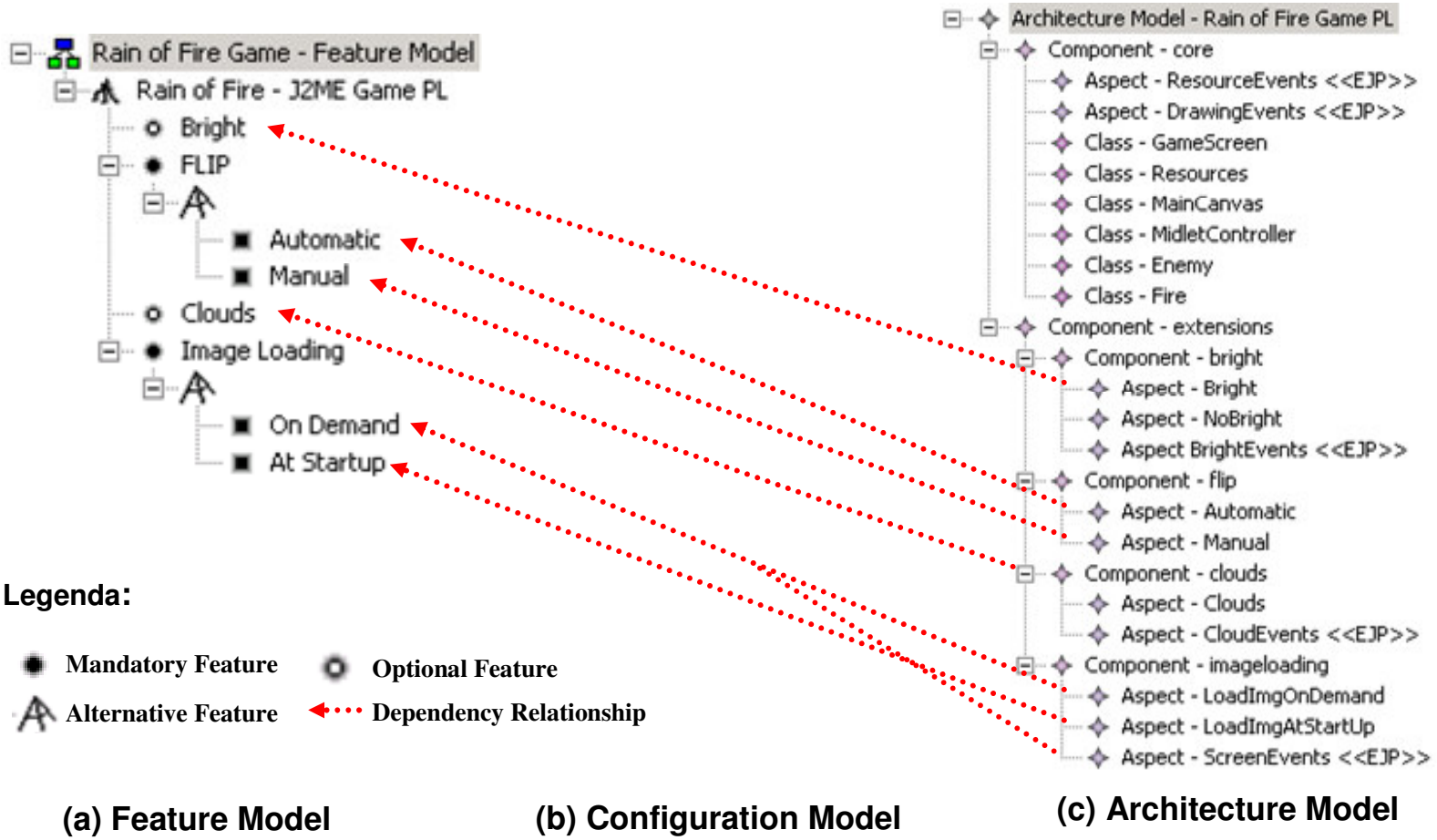

Figura 62. Modelo de Configuração da Linha de Produto do Rain of Fire

6.4. Sumário

Este capítulo apresentou, em detalhe, três diferentes estudos de caso de frameworks desenvolvidos com o uso da nossa abordagem. Os estudos envolveram frameworks de três diferentes domínios (*measurement*, sistemas multi-agentes, jogos para celulares) demonstrando a generalidade da abordagem proposta. Os pontos de junção de extensão (EJPs) encontrados para cada estudo de caso representam eventos ou estados relevantes que ocorrem no domínio do framework. Diferentes tipos de aspectos do núcleo e de extensão foram também definidos para cada um dos frameworks. No capítulo seguinte são discutidos os benefícios observados assim como lições aprendidas a partir da realização desses estudos de caso.

7
Análise dos Estudos de Caso, Discussões e Lições Aprendidas

Este capítulo apresenta uma análise da abordagem OA proposta a partir dos estudos de caso realizados. Inicialmente são apresentados os benefícios gerais trazidos pelas diretrizes de implementação e pelo modelo generativo da abordagem. Em seguida, é apresentado um estudo qualitativo e quantitativo do uso de nossa abordagem no framework *Measurement*. Finalmente, diversas discussões e lições aprendidas a partir da experiência de uso da abordagem são apresentadas.

7.1.
Benefícios da Abordagem

A abordagem orientada a aspectos para o desenvolvimento de frameworks contribui para uma melhor modularização de características transversais encontradas em frameworks OO. Ela mostra como mecanismos OA podem ser usados de forma complementar aos oferecidos pelo paradigma OO. Os seguintes benefícios foram observados a partir da realização dos estudos de caso apresentados (Capítulo 6) e, por conseqüência, estão associados ao uso de nossa abordagem:

(i) **melhor gerenciamento de variabilidades**: a abordagem permite o gerenciamento sistemático de diferentes tipos de variabilidades encontradas no desenvolvimento de um framework OO. Características obrigatórias, opcionais, alternativas e de integração transversais e não transversais podem ser modularizadas separadamente e compostas através dos EJPs. Nossa abordagem oferece um conjunto explícito de regras de mapeamento entre tipos de características e elementos de implementação em um framework (Seção 5.2.3). Os EJPs podem também contribuir para identificar possíveis conflitos existentes entre características que agem sobre um mesmo ponto de extensão do núcleo (Seção 7.3.1). O uso de aspectos de extensão para implementar variabilidades do

framework permite que se decida facilmente pela inclusão ou remoção das respectivas funcionalidades associadas;

(ii) **configurabilidade**: como conseqüência do gerenciamento sistemático de variabilidade e da facilidade oferecida para plugar e desplugar aspectos, a abordagem permite definir diferentes configurações do framework. Isso traz benefícios para customizar o framework para ser reutilizado em diferentes cenários;

(iii) **instanciação automática**: nosso modelo generativo OA endereça a instanciação automática das variabilidades OO e OA encontradas na implementação do framework. Instanciação de frameworks é notadamente reconhecido como sendo um processo complexo e que oferece obstáculos a sua adoção em uma linha de desenvolvimento. O modelo generativo proposto endereça a automatização desse processo, através da especificação de diferentes modelos durante a engenharia de domínio, que trazem facilidades para o engenheiro de aplicação. A instanciação de variabilidades OA é tratada explicitamente no nosso modelo generativo através da customização de pontos de corte de aspectos usando relações tranversais entre características e mapeamento entre características `<<joinpoint>>` e pontos de corte de EJPs.

A Tabela 7 descreve os benefícios trazidos pelos EJPs e por cada tipo de aspecto existente na nossa abordagem. Ela também indica como os aspectos do núcleo, de variabilidade e de integração endereçam cada um dos problemas de modularização apresentados anteriormente (Seção 2.1.1).

Elemento da Abordagem	Benefícios	Problema de Modularização Endereçado
Aspectos do Núcleo	• Simplificam o entendimento e evolução do núcleo do framework; • Modularizam interesses ou papéis tranversais encontrados no núcleo do framework; • Podem auxiliar na descoberta de EJPs.	Interesses ou Papéis Transversais
Pontos de Junção de Extensão	• Sistematizam o processo de extensão e composição do framework facilitando seu processo de reuso; • Habilitam a composição entre o núcleo do framework e suas extensões; • Encapsulam o framework e expõe apenas pontos de junção adequados.	Forte Acoplamento entre o Núcleo do Framework e suas Extensões.
Aspectos de Variabilidade	• Facilitam o reuso e extensão do framework; • Modularizam características opcionais e alternativas transversais do framework; • Tornam possível inserir ou remover facilmente do framework características opcionais ou alternativas.	Características Opcionais ou Alternativas Transversais
Aspectos de Integração	• Facilitam o reuso e extensão do framework; • Modularizam a composição do framework com outras extensões; • Tornam possível inserir ou remover composições transversais com o framework.	Composições Transversais com o Framework

Tabela 7. Análise dos Benefícios das Diretrizes da Abordagem

7.2. Estudo de Composição de Frameworks

Essa seção apresenta os resultados do estudo de composição envolvendo o framework *Measurement*. Conforme descrito no capítulo 6, nesse estudo foram definidos uma série de composições transversais a serem realizadas entre os frameworks *Measurement*, GUI e dois frameworks simples de estatística e persistência. O estudo envolveu uma análise comparativa entre um conjunto de soluções OO relatadas em [90, 91] e as soluções OA baseadas na abordagem de EJPs apresentadas anteriormente (Seção 6.1). As soluções foram comparadas e avaliadas tanto do ponto de vista qualitativo quanto quantitativo. As subseções seguintes relatam os resultados de nosso estudo.

7.2.1. Estudo Qualitativo

Mattson et al [90, 91] discutem problemas, causas e soluções relacionados com a composição de frameworks OO. Os autores apresentam um conjunto de 5 problemas/desafios principais encontrados na composição de frameworks. Para cada um desses problemas, eles definem 3 soluções OO para sua resolução, as quais podem ser usadas de forma complementar ou separadas de acordo com o contexto. Finalmente, eles também discutem um conjunto de causas que podem contribuir para o agravamento do problema, tais como, intenções de projeto, cobertura do domínio e acesso ao código-fonte.

No estudo de composição conduzido por Mattson et al [90, 91], dois tipos de frameworks foram definidos: (i) frameworks *calling* – que são aqueles responsáveis por controlar e invocar todas as outras partes da aplicação. Exemplos de frameworks *calling* são o Measurement e o GUI (Seção 6.1); e (ii) frameworks *called* – representam entidades passivas as quais podem ter seus serviços requisitados por outras partes da aplicação. Os frameworks de estatística e persistência (Seção 6.1) são exemplos de *called*. Ambos os tipos, endereçam propriedades comuns (Seção 2.1) compartilhadas por frameworks OO, que são: (i) definem um conjunto de classes que juntas colaboram para implementar características comuns de um dado domínio; (ii) oferecem pontos de extensão para

ser estendidos; e (iii) são responsáveis pela invocação e controle de suas extensões.

7.2.1.1. Análise Qualitativa das Soluções OO

Nosso estudo de composição endereçou 3 dos 5 problemas apresentados por Mattson et al [90, 91], são eles:

(i) *composição do fluxo de controle de frameworks* – esse problema ocorre quando dois frameworks *calling* estão sendo instanciados e combinados no contexto de uma aplicação. Uma vez que é esperado que cada um dos frameworks detenha o controle da aplicação, a composição entre eles pode requerer a resolução de conflitos;

(ii) *espaço vazio ou lacuna entre os frameworks* (*framework gap*) – esse problema ocorre quando dois frameworks precisam ser compostos para endereçar os requisitos de uma aplicação, mas eles juntos não são suficientes para satisfazer completamente tais requisitos; e

(iii) *composição de funcionalidade de entidade* – esse problema ocorre quando é necessário combinar a representação de uma entidade de domínio oferecido por um dado framework com uma funcionalidade adicional oferecida por um outro framework.

As Tabelas 8, 9 e 10 listam e descrevem cada uma das 3 soluções OO propostas por Mattson et al para os problemas investigados no nosso estudo. Foram preservados os nomes originais das soluções atribuídos por tais autores. Detalhes adicionais das soluções podem ser encontradas em [90, 91].

Nosso estudo avaliou os benefícios e limitações das soluções OO propostas por Mattson et al, através do uso de um conjunto de 5 propriedades de modularidade. Três das propriedades são adaptadas de critérios de comparação propostos por Hanneman & Kiczales [58] para avaliar soluções OO e OA de padrões de projeto clássicos. Outras duas propriedades (entrelaçamento e espalhamento) foram usadas para avaliar o grau de separação de interesse das soluções.

As seguintes propriedades de modularidade foram utilizadas em nosso estudo: (i) *localidade* – indica se o código de composição está completamente

modularizado em módulos; (ii) *transparência de composição* – indica se é possível raciocinar sobre as diferentes composições de frameworks de forma independente; (iii) *facilidade de plugar/desplugar* – determina o quão fácil é adicionar e remover o código de composição entre os frameworks; (iv) *entrelaçamento* – indica se o código de composição está entrelaçado com o código de classes de pelo menos um dos frameworks envolvidos na composição; e (v) *espalhamento* – define se o código de composição da solução proposta está disperso por várias classes do(s) framework(s). A Tabela 11 apresenta a análise das propriedades para cada uma das soluções OO de composição de frameworks. A seguir são discutidos os benefícios e desvantagens das soluções OO.

PROBLEMA: Composição do Fluxo de Controle de Frameworks
Concurrency. Essa solução propõe atribuir uma *thread* de controle separada para cada framework, ao invés de compor seus fluxos de controle. Ela pode ser usada apenas quando não existe notificação de eventos entre os frameworks. Para algumas aplicações, pode ser necessário definir código de sincronização em classes específicas da aplicação.
Wrapping. Essa solução pode ser usada quando na composição entre os frameworks precisam ser definidas notificações de eventos entre os mesmos. Ela consiste no encapsulamento do framework, através de um *wrapper*. Cada *wrapper* é responsável por interceptar eventos de entrada e saída do framework. Todas as classe do framework que se comunicam com classes externas precisam ser estendidas para notificar o *wrapper*. As soluções *Concurrency* e *Wrapping* podem ser usadas de forma complementar.
Removal and Rewriting. Essa solução é adotada quando é necessário prover algum tipo de notificação de eventos internos entre os frameworks. Ela consiste na modificação interna do código do framework de forma a definir um único loop de controle que endereça os requisitos da composição dos frameworks e da aplicação endereçada. A solução requer acesso e entendimento do código-fonte dos frameworks.

Tabela 8. Soluções OO para Composição de Fluxo de Controle de FWs

PROBLEMA: Lacuna entre Frameworks (*Framework Gap*)
Wrapping and Extending. Essa solução lida com o problema de lacuna entre frameworks quando frameworks do tipo *called* estão sendo usados. Ela propõe introduzir um módulo *wrapper* que agrega os frameworks e a extensão adicional requerida para preencher a lacuna dos requisitos não endereçados pela composição. O *wrapper* é também responsável por oferecer uma nova interface para os serviços resultantes da composição.
OO Mediator. Essa solução endereça o problema de lacuna quando frameworks do tipo *calling* estão sendo usados. Um *Mediator* é usado para controlar as interações entre os frameworks e as extensões adicionais que estão sendo compostas. As interações entre os frameworks e as extensões adicionais usualmente requer a modificação das classes internas do framework para notificar as demais de eventos de interesse.
Redesign and Extend. Essa solução consiste em modificar código-fonte de frameworks e extensões de forma a integrá-los em um novo framework. Ela pode ser adotada quando a composição entre o framework e demais extensões pretende ser reusada no desenvolvimento de futuras aplicações.

Tabela 9. Soluções OO para Lacuna entre Frameworks

Natureza transversal. A Tabela 11 mostra que 6 das 9 soluções OO investigadas possuem propriedades de entrelaçamento e espalhamento. Isso significa que a maioria das soluções OO requer a modificação de várias classes do framework. Em vários casos, o código de composição está disperso por diversas classes do framework e se entrelaça com o código já existente em tais classes. Isso atesta a natureza transversal do código de composição. Dessa forma, como conseqüência, o projeto e implementação de várias soluções de composição não são completamente modularizadas, e são usualmente afetadas pela replicação de elementos de projeto e código.

PROBLEMA: Composição de Funcionalidades de Entidades
Aggregation ou Herança Múltipla. Essas soluções propõem usar os mecanismos de agregação ou herança múltipla para compor as classes entidades de um framework com a funcionalidade adicional oferecida por um outro. Cada classe de composição agrega (ou herda) as classes de entidade específica de domínio e de funcionalidade adicional. O problema principal dessa solução é que mudanças ocorrendo nas classes entidade do framework específico de domínio não são necessariamente repassadas para a classe que representa a composição.
Observer. Essa solução é proposta para lidar com as desvantagens das soluções baseadas nos mecanismos de agregação ou herança múltipla. Ela propõe o uso do padrão *Observer* [45] para prover o mecanismo de notificação requerido entre os frameworks e as classes que implementam a composição. As classes entidades desempenham o papel *Subject* do padrão *Observer*, enquanto as classes de composição devem assumir o papel *Observer* do padrão.

Tabela 10. Soluções OO para Composição de Funcionalidades de Entidades

Mudanças Invasivas. Devido a natureza transversal do código de composição dos frameworks, muitas das soluções OO demandam mudanças invasivas no código de classes do framework. A solução *Mediator* proposta para resolver o problema de lacuna entre frameworks, por exemplo, requer a inclusão de código intrusivo de composição nas classes dos frameworks de forma a endereçar as interações de integração dos mesmos. Essas mudanças invasivas ocorrem em muitos casos porque o código de composição das classes dos frameworks requer a propagação de eventos internos entre os mesmos. A Tabela 11 reflete a necessidade de tais mudanças invasivas, através da propriedade de localidade. Ela mostra que 6 das 9 soluções não obtém sucesso na separação do código de composição. Umas das consequências negativas de tais mudanças invasivas é dificultar ou até mesmo impossibilitar a reutilização do framework em diferentes aplicações, uma vez que diversos trechos de código de composição inseridos farão parte apenas de uma ou poucas aplicações instanciadas.

Facilidade de Plugar/Desplugar. Muitas das soluções OO não podem ser facilmente plugadas ou desplugadas dos frameworks. Isso significa que desenvolvedores não podem adicionar ou remover o código de composição com o propósito de redefiní-lo ou adaptá-lo. A Tabela 11 mostra que 6 das 9 soluções não podem ser facilmente plugadas ou desplugadas. Exemplos de tais soluções são: (i) a solução *Wrapping* para o problema de fluxo de controle de frameworks; (ii) solução *Mediator* para o problema de lacuna entre frameworks; e (iii) solução *Observer* para o problema de composição de funcionalidade de entidades.

Complexidade Crescente. A complexidade de entender e modificar as soluções OO aumenta quando os desenvolvedores precisam lidar simultaneamente com diferentes composições entre frameworks. A propriedade de transparência de composição, apresentada na Tabela 11, representa a possibilidade de refletir sobre diferentes composições entre frameworks usadas simultaneamente. A Tabela 11 mostra que 6 das 9 soluções OO investigadas não podem ser compostas de forma transparente. Isso acontece porque muitas das soluções OO não são capazes de modularizar completamente o código relativo a cada composição entre frameworks. Como conseqüência, é difícil raciocinar independentemente sobre cada solução OO adotada.

	Propriedades de Modularidade				
Solução OO (Identificador do Problema de Composição)*	**Localidade**	**Transparência de Composição**	**Facilidade de (Des)Plugar**	**Espalhamento**	**Entrelaçamento**
1. Concurrency (I)#	não	não	não	Sim	sim
2. Wrapping (I)	não	não	não	Sim	sim
3. Remove and rewrite (I)	não	não	não	Sim	sim
4.Wrapping and Extending(II)	sim	sim	sim	Não	não
5. OO Mediator (II)	não	não	não	Sim	sim
6. Redesign and extend (II)	não	não	não	Sim	sim
7. Aggregation (III)	sim	sim	sim	Não	não
8. Multiple Inheritance (III)	sim	sim	sim	Não	não
9. Observer Pattern (III)	não	não	não	Sim	sim

* *O Identificador do Problema de Composição está atribuído da seguinte forma: (I) Composição de Fluxos de Controle; (II) Lacuna entre Frameworks; and (III) Composição de Funcionalidades de Entidades.*

\# *As propriedades da Solução Concurrency OO refere-se a definição de código de sincronização em classes específicas da aplicação.*

Tabela 11. Avaliação de Propriedades das Soluções OO

7.2.1.2. Análise Qualitativa das Soluções OA

Nosso estudo de composição propôs um conjunto de soluções OA para lidar com os problemas de composição de frameworks listados anteriormente. O objetivo do estudo foi analisar o impacto de programação orientada a aspectos na implementação do código de composição relacionado às soluções OO propostas em [90, 91]. Nessa seção, tais soluções são descritas em linhas gerais. Elas são organizadas baseadas no problema de composição endereçado. Exemplos específicos de algumas das soluções de composição propostas foram apresentados no capítulo 6 no estudo de caso do framework *Measurement*. Tais soluções serão referenciadas sempre que necessário. Referências para outros trabalhos, que exploram uma solução específica embora em diferentes contextos, são também apresentadas.

As Tabelas 12, 13 e 14 listam e descrevem novas soluções OA para cada um dos 3 problemas de composição de frameworks investigados no nosso estudo. Em alguns casos, programação orientada a aspectos é usada com o intuito de complementar a solução original, enquanto em outros casos ela substitui completamente a solução original proposta.

PROBLEMA: Composição do Fluxo de Controle de Frameworks
Concurrency + Synchronization Aspects[19]. POA pode ser usada de forma complementar a solução de concorrência proposta. Aspectos de Sincronização podem ser definidos para controlar a execução concorrente do código de aplicação comum de ambos os frameworks. Exemplos de controle de concorrência os quais podem ser implementados em AspectJ são sincronização de execução de métodos e bloqueio de fluxos de execução conflitantes [87, 112]. O uso de POA para especificar interesses de concorrência específicos tem sido endereçado por alguns trabalhos de pesquisa [87, 112]. Uma vez que apenas o código de sincronização é necessário ser codificado para essa solução, pode ser afirmado que aspectos são usados para implementar todo o código de

[19] O símbolo "+" indica uma solução complementar à solução OO proposta originalmente.

composição dos frameworks. Aspectos de sincronização podem também ser usados para complementar a solução *Observer Aspects* apresentada abaixo.
Wrapping >> Observer Aspects[20]. Nessa solução OO original, cada framework é encapsulado por um wrapper. Chamadas de métodos de entrada e saída de um dado framework são interceptados de forma a notificar o outro. Essa solução oferece a restrição de que eventos internos do framework não podem ser interceptados pelos *wrappers*. Usando POA, essa solução pode ser substituída pela implementação de aspectos *Observer* os quais podem não apenas interceptar chamadas de entrada e saída do framework, bem como eventos internos sempre que necessário. Entretanto, ao invés de definir um *wrapper* para cada framework, pode ser especificado um conjunto de aspectos os quais permitem interceptar diferentes eventos do framework, e notificar outros frameworks interessados.
Removal and Rewriting >> Observer Aspects. A solução OO "*Removal and Rewriting*" foi proposta por Mattsson et al para lidar com as restrições da solução *Wrapping* de não permitir a interceptação de eventos internos do framework. Dessa forma, a solução OA baseada na definição de um conjunto de aspectos *Observer* pode também ser adotada para substituir tal solução. A solução OA oferece uma melhor modularização para o código de composição, porque não é necessário realizar mudanças invasivas em classes internas do framework. Vale ressaltar que a solução OA apresentada no capítulo 6 (Seção 6.1.3.1) para composição dos frameworks Measurement e GUI, é baseada em aspectos Observer.

Tabela 12. Soluções OA para Composição de Fluxo de Controle de FWs

PROBLEMA: Lacuna entre Frameworks (*Framework Gap*)
Wrapping and Extending + AO Observer. Essa solução é proposta para resolver o problema de lacuna entre frameworks *called*. A solução OA complementa a solução OO original permitindo uma melhor integração entre os frameworks sendo compostos e as extensões que endereçam a lacuna entre os dois frameworks. Aspectos podem ser usados para definir diferentes tipos de interação entre os frameworks e as extensões, incluindo a manipulação e

[20] O símbolo “>>” indica uma solução que substitui a solução OO proposta originalmente.

notificação de eventos internos.
OO Mediator >> AO Mediator. A solução OO baseada no Mediator é proposta para endereçar o problema de lacuna entre frameworks *calling*. POA pode contribuir para a melhora de tal solução através da definição de um módulo *Mediator* na forma de um aspecto. O aspecto *Mediator* define o código de integração entre os frameworks e extensões adicionais de forma a endereçar os requisitos da aplicação. Durante a evolução independente de cada framework e extensões, mudanças podem ser introduzidas de forma modular e localizada dentro de cada um desses elementos. Além disso, o código de composição entre frameworks e extensões pode também evoluir de forma independente. Essa solução OA baseada no *Mediator* foi usada na composição entre os frameworks *Measurement*, GUI e de Estatística, apresentada no Capítulo 6 (Seção 6.1.3.2), o framework de Estatística é usado para preencher a lacuna de prover funcionalidades para cálculo de informações estatísticas que não são endereçadas pelos frameworks *Measurement* e GUI.
Redesign and Extend. As soluções OA baseadas no *Observer* e no *Mediator* apresentadas acima são soluções adequadas para resolver o problema de lacuna entre frameworks porque elas não demandam um alto acoplamento entre os mesmos e os componentes de extensão. Essas soluções evitam a necessidade de modificar a implementação das classes internas de cada framework. Dessa forma, sugere-se que a solução OO *Redesign and Extend* proposta seja usada apenas no caso em que: (i) existe uma conexão forte e estável entre os domínios dos frameworks e das extensões que motiva o desenvolvimento de um único framework; (ii) o novo framework será usado no desenvolvimento de várias aplicações; e (iii) é difícil compor os frameworks e extensões usando as solução OA baseadas no *Observer* e *Mediator*.

Tabela 13. Soluções OA para Lacuna entre Frameworks

PROBLEMA: Composição de Funcionalidades de Entidades
Entity-Functionality "Glue" Aspect. Uma única solução OA é proposta para lidar com as restrições impostas as soluções OO do problema de composição de funcionalidades de entidades. As soluções OO baseada em agregação e múltipla herança apresentam dificuldades para gerência de atualização de

estados nas classes de composição resultantes do uso de tais mecanismos. A solução OO baseada no Observer resolve tal restrição, mas por outro lado requer muitas mudanças invasivas nas classes do framework. A solução OA proposta define um aspecto que intercepta pontos de junção específicos nas classes do framework de domínio. Esse aspecto também especifica em que momento as funcionalidades adicionais do outro framework devem ser invocadas. Dessa forma, esse aspecto funciona como uma espécie de "cola" (*glue*) entre os dois frameworks. A solução OA para composição dos frameworks GUI e Estatística com o framework de persistência, apresentada no Capítulo 6 (Seção 6.1.3.3) pode ser vista como uma instanciação de solução OA baseada no "*Glue" Aspect*.

Tabela 14. Soluções OA para Composição de Funcionalidades de Entidades

Para cada um dos problemas de composição endereçados, foram observadas diversas melhorias quando usando soluções OA. Tais melhorias podem ser observadas principalmente em termos de modularização do código de composição. A Tabela 15 sumariza as melhorias obtidas para cada solução OA. De forma similar a comparação das soluções OO (Seção 7.2.1.1), foram usadas as propriedades de modularidade: localidade, composição, transparência e facilidade de plugar/desplugar. A propriedade complementar, apresentada na Tabela 15, indica se a solução OA adotada é usada em conjunção com alguma das soluções OO anteriormente propostas.

Um total de 5 novas soluções foram caracterizadas como alternativas as soluções OO anteriormente propostas. POA foi usada como uma solução complementar na implementação de 2 soluções OO. Todas as soluções OA trouxeram benefícios em relação às soluções OO (ver Tabela 15). Como mencionado anteriormente, muitas das soluções OA têm sido exploradas pela comunidade de pesquisa em DSOA, embora não no contexto de composição de frameworks.

Nosso estudo mostra que POA pode ser usado como uma tecnologia efetiva para modularizar completamente o código de composição entre frameworks. Todas as soluções OA alcançaram tal objetivo. Em geral, cada problema de composição entre frameworks demanda a integração de alguma das características

do framework, tais como: fluxos de controle, entidades ou funcionalidades. Boa parte das soluções OO define mudanças invasivas no código de classes do framework. Usando aspectos, podem ser definidos pontos específicos da execução de um framework os quais podem ser conectados a funcionalidade de outro framework de forma a endereçar sua composição. As soluções OA também evitam o espalhamento e entrelaçamento de código apresentado por muitas das soluções OO, apresentadas na seção 7.2.1.1.

	Propriedades de Modularidade			
Solução OA (Identificador do Problema de Composição)*	**Localidade**	**Transparência de Composição**	**Facilidade de (Des)Plugar**	**Complementaridade**
1. Concurrency + Synchronization Aspects (I)	sim	sim	sim	sim
2. AO Observer (I)	sim	sim	sim	não
3. OO Wrapper + AO Observer (II)	sim	sim	sim	sim
4. AO Mediator (II)	sim	sim	sim	não
5. "Glue" Aspect (III)	sim	sim	sim	não

**O Identificador do Problema de Composição está atribuído da seguinte forma: (I) Composição de Fluxos de Controle; (II) Lacuna entre Frameworks; and (III) Composição de Funcionalidades de Entidades.*

Tabela 15. Análise de Propriedades das Soluções OA

A completa modularização do código de composição do framework nas soluções OA também permite plugar e desplugar composições específicas, sempre que necessário. Uma vez que o código de composição é totalmente modularizado pelos aspectos, desenvolvedores podem adicionar ou remover o código de composição durante a implementação ou evolução de arquiteturas de software baseado em decisões de projeto específicas. Assim, essa propriedade de facilidade de (des)plugar traz mais flexibilidade quando decidindo pelo uso de composições específicas de frameworks.

A modularização do código de composição entre os frameworks também ajuda no entendimento das soluções adotadas. Cada aspecto expressa diretamente as decisões de projeto realizadas pelos desenvolvedores para implementar a composição entre os frameworks, podendo ser visto como uma documentação explícita das composições adotadas. Como conseqüência, essa documentação melhorada traz benefícios para o entendimento e manutenção do código de

composição. Em aplicações de software, onde desenvolvedores precisam especificar vários tipos de composição entre frameworks, essa documentação se torna ainda mais importante.

7.2.2. Estudo Quantitativo

Nosso estudo quantitativo envolveu a comparação das versões OO e OA da instância de composição do framework *Measurement* integrada com os frameworks de interface gráfica (GUI), estatística e persistência (Seção 6.1). No estudo, ambas as versões OO e OA da composição dos frameworks foram comparadas usando uma suíte de métricas [107] que permite quantificar os seguintes atributos de engenharia de software: (i) separação de interesses; (ii) acoplamento; (iii) tamanho; e (iv) coesão. Essa suíte foi desenvolvido por membros do grupo de pesquisa em DSOA do Laboratório de Engenharia de Software da PUC-Rio, e tem sido usado em diversos estudos comparativos de sistemas envolvendo diferentes tipos de interesses transversais, tais como, padrões de projeto [20, 51], distribuição [55, 83], persistência [55, 83], concorrência [55, 83], tratamento de exceções [40].

O objetivo central do estudo foi observar o comportamento de atributos bem conhecidos de engenharia de software, durante a refatoração do código de composição de frameworks usando técnicas orientadas a aspectos. O estudo foi organizado de acordo com os seguintes passos: (i) seleção das métricas de software; (ii) execução de procedimentos de avaliação; (iii) coleta dos valores das métricas; e (iv) análise dos valores obtidos. As subseções seguintes apresentam um breve resumo das métricas utilizadas, e os resultados e análise dos valores obtidos para as métricas para ambas as versões do sistema. A execução de procedimentos de avaliação consistiu basicamente no alinhamento do código das versões OO e OA da instância da composição dos frameworks, de forma a garantir que ambas as versões estavam utilizando um mesmo padrão de codificação.

A versão OO da composição dos frameworks foi obtida a partir da refatoração da versão OA. Isso garantiu que ambas as versões estavam implementando as mesmas funcionalidades. Foram criadas classes

correspondentes a cada aspecto de composição para atuar como mediadores das interações necessárias entre os frameworks. Essas classes agregam a funcionalidade atribuída aos aspectos de composição. Finalmente, várias chamadas foram inseridas ao longo das classes internas do framework que eram anteriormente interceptadas pelos aspectos. Essas classes repassam pedidos de serviços de um framework para o outro usando as classes de composição mencionadas acima.

7.2.2.1. As Métricas Utilizadas

O suíte de métricas utilizado no estudo quantitativo objetiva a avaliação dos atributos de separação de interesses, acoplamento, tamanho e coesão, no contexto de sistemas OO e OA. Ele foi definido como um refinamento de métricas OO existentes [25, 39]. A definição original de tais métricas foi estendida para contemplar a avaliação de sistemas OA levando em consideração as novas abstrações e mecanismos presentes em tal paradigma. O suíte também contempla a definição de novas métricas para avaliação da propriedade de separação de interesse. Essas métricas capturam o grau de espalhamento e entrelaçamento de um dado interesse do sistema ao longo de seus componentes (classes e aspectos), operações (métodos e adendos) e linhas de código.

As Tabelas 16, 17, 18 e 19 apresentam uma breve descrição das métricas, que estão organizadas de acordo com o atributo medido (tamanho, acoplamento, coesão e separação de interesses) pelas mesmas. Detalhes adicionais de tais métricas podem ser obtidos em [107].

No nosso estudo, as métricas de tamanho, acoplamento e tamanho foram calculadas automaticamente usando a ferramenta Together[21]. Apenas os aspectos da versão AO tiveram que ter os valores de tais métricas calculadas manualmente. As métricas de separação de interesses foram calculadas manualmente para ambas as versões. A coleta de tais métricas envolve inicialmente a seleção e sombreamento do código de classes que implementam parcialmente ou completamente um dado interesse. Nesse estudo quantitativo em particular foi sombreado todo o código de classes ou aspectos que implementam o interesse de

[21] Together Technologies. URL: http://www.borland.com/together/. 2007.

composição dos frameworks. Após o sombreamento de tais elementos de implementação, foi então feita a contagem do valor de tais métricas.

Métrica	Descrição
Coupling between Components (CBC)	Calcula o grau de acoplamento de um dado componente (classes, aspectos, interfaces) em relação aos demais componentes do sistema. São contados atributos, parâmetros, tipos de retorno, declaração de throws, variáveis locais, etc. No caso de aspectos são também contados declarações de tipos dentro de construções do tipo intertipo, ponto de corte, adendo, etc.
Depth of Inheritance Tree (DIT)	Mede a profundidade de uma dada subclasse (ou subaspecto) para a classe (ou aspecto) raiz da sua árvore de herança.

Tabela 16. Métricas de Acoplamento

Métrica	Descrição
Vocabulary Size (VS)	Calcula o número de componentes (classes e aspectos) existentes no sistema.
Lines of Code (LOC)	Determina o número de linhas de código de um dado componente do sistema. Comentários não devem ser considerados na contagem. O cálculo dessa métrica exige a adoção de convenções de codificação em todos os componentes do sistema.
Number of Attributes (NOA)	Conta o número de atributos presentes em um dado componente (classe ou aspecto).
Weighted Operations per Component (WOC)	Determina a complexidade de um componente (classe ou aspecto) em função das suas operações. Ela é calculada baseada na quantidade de parâmetros existentes em todos os métodos (e advices) de um dado componente.

Tabela 17. Métricas de Tamanho

Métrica	Descrição
Concern Diffusion over Components (CDC)	Esta métrica calcula a quantidade de componentes (classes e aspectos) necessários para implementar um dado interesse do sistema.
Concern Diffusion over Operations (CDO)	Calcula a quantidade de métodos e adendos existentes nos componentes responsáveis por implementar um dado interesse do sistema.
Concern Diffusion over LOC (CDLOC)	Conta o número de pontos de transição para cada interesse ao longo das linhas de código do sistema. Esta métrica captura com precisão o grau de entrelaçamento e espalhamento de um dado interesse ao longo dos componentes do sistema.

Tabela 18. Métricas de Separação de Interesses (SoI)

Métrica	Descrição
Lack of Cohesion in Operations (LCOO)	Esta métrica mostra a proximidade de relação entre seus componentes internos. Ela é calculada baseada na manipulação dos atributos de uma classe (ou aspecto) por seus métodos (ou adendos).

Tabela 19. Métrica de Coesão

7.2.2.2. Análise e Discussão dos Resultados

Nessa seção são apresentados e discutidos os resultados do processo de medição sobre as versões OO e OA da composição dos frameworks. As Tabelas 20 e 21 apresentam os valores absolutos obtidos. As Figuras 63 e 64 apresentam gráficos comparativos dos resultados obtidos para as versões OO e OA do sistema investigado. Os valores apresentados para as métricas nas tabelas e gráficos se referem aqueles coletados considerando todas as classes e aspectos de cada versão do sistema. Nos gráficos, o eixo Y apresenta a porcentagem relativa ao valor absoluto de cada métrica. Cada par de barras do gráfico é associado a um valor percentual, o qual representa a diferença entre os resultados obtidos para ambas as versões OO e OA do sistema. Uma porcentagem positiva indica que a versão OA foi superior, enquanto uma porcentagem negativa indica que a versão OA foi inferior.

Métrica	CDC	CDO	CDLOC
Versão OO	17	63	17
Versão AO	9	45	1

Tabela 20. Valores Absolutos para Métricas de Separação de Interesses

Métrica	Acoplamento		Coesão	Tamanho			
	CBC	DIT	LCOO	VS	LOC	NOA	WOC
Versão OO	148	128	331	62	1548	71	393
Versão AO	163	131	302	65	1563	66	388

Tabela 21. Valores para Métricas de Acoplamento, Coesão e Tamanho

As métricas de separação de interesses (SoI) foram usadas em nosso estudo para quantificar a efetividade da separação do código de composição entre os frameworks. Como o principal objetivo da versão OA, era permitir uma melhor modularização dos interesses transversais relacionados a composição, eram esperados valores superiores das métricas de SoI para tal versão. O gráfico apresentado na Figura 63 confirma nossa hipótese: a versão OA apresenta valores superiores para todas as métricas de SoI. Os valores obtidos sempre apresentaram valores favoráveis da versão OA da ordem de pelo menos 29%.

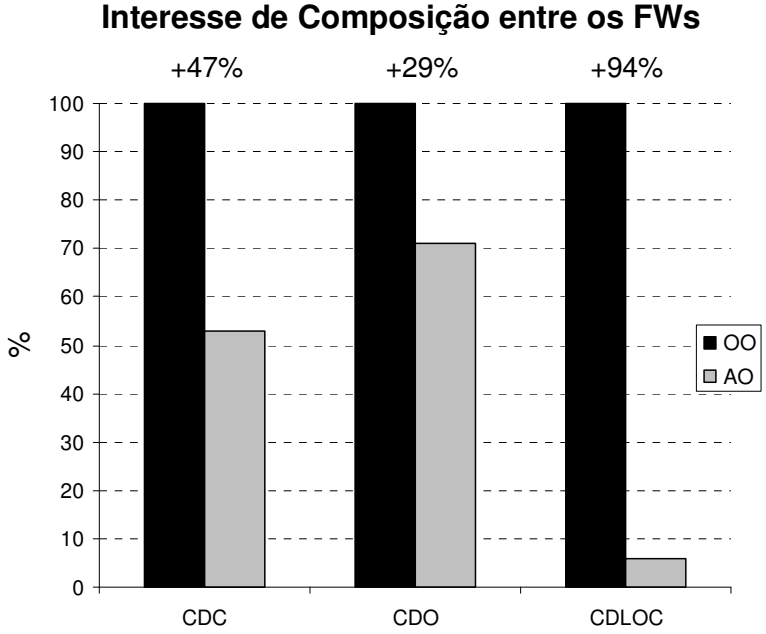

Figura 63. Métricas de Separação de Interesses

Os valores obtidos para a métrica CDC mostra que o interesse de composição entre os frameworks na versão OO se espalha por 17 diferentes classes (Tabela 20), enquanto na versão OA ele é modularizado por um total de 9 elementos de implementação, entre classes e aspectos. A Figura 63 mostra essa diferença percentual nos valores obtidos para tal métrica. A métrica CDO mostra que na versão OO foram necessários 63 operações para implementar o interesse de composição, em contraposição a versão OA necessitou implementar um total de 45 operações (métodos ou adendos). Finalmente, os resultados também demonstram através da métrica CDLOC, o quão o interesse de composição se

apresenta mais entrelaçado com outros interesses dos frameworks na versão OO em comparação com a versão OA. A versão OO apresenta um total de 17 alternâncias de interesse ao longo do código, enquanto que na versão OA não existe alternância entre interesses (valor igual a 1 obtido para a métrica CDLOC). Isso significa que o código de composição foi totalmente isolado em um conjunto de aspectos e classes totalmente dedicados a implementação do interesse de composição dos frameworks.

Os resultados coletados para as métricas de tamanho, acoplamento e coesão no estudo quantitativo, demonstrou um certo equilíbrio entre as versões OO e OA para a aplicação de composição dos frameworks, conforme pode ser visto no gráfico da Figura 64[22]. A versão OA exibiu melhores resultados para as métricas de número de atributos (NOA) e coesão (LCOO). Já a versão OO levou uma boa vantagem considerando as métricas de número total de componentes (VS) e acoplamento (CBC). Houve um certo equilíbrio entre ambas versões do sistema no que se refere as métricas de linhas de código (LOC), profundidade de árvores de herança (DIT) e complexidade de operações do sistema (WOC).

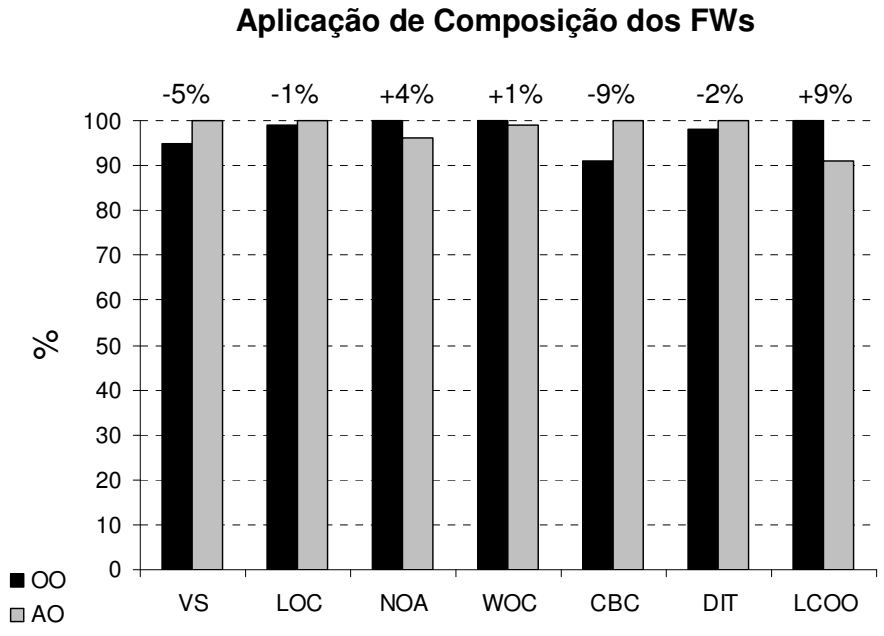

Figura 64. Métricas de Tamanho, Acoplamento e Coesão

[22] Os valores obtidos para cada uma das classes e aspectos das versões OO e OA do estudo de composição são apresentadas no Apêndice II.

As diferenças nos valores das métricas do número de componentes (VS) e acoplamento (CBC) desfavoráveis para a versão OA são decorrentes principalmente da criação dos três aspectos EJPs para os frameworks *Measurement*, GUI e de estatística. Para expor um conjunto de pontos de junção de extensão, cada aspecto EJP é acoplado explicitamente às classes ao qual ele intercepta, aumentando dessa forma os valores obtidos para a métrica CBC. Embora apresente valores superiores para a métrica de acoplamento, a versão OA do sistema oferece a vantagem de permitir plugar/desplugar facilmente o código associado a cada composição entre os frameworks. A versão OO por sua vez requer a introdução de diversas mudanças invasivas em suas classes internas que dificultam a gerência do código tanto do framework quanto da composição propriamente dita. Assim, levando em consideração a natureza de cada solução, a versão OO apresenta um forte acoplamento entre os frameworks, embora tenha obtido um valor menor para a métrica de CBC quando comparado a versão OA.

A versão OA supera a versão OO no que se refere às métricas de número de atributos (NOA) e falta de coesão (LCOO), em 4% e 9%, respectivamente, conforme pode ser visto na Figura 64. O valor superior da métrica NOA na versão OO, ocorre porque foram criadas para essa versão classes de composição para substituir os aspectos de integração existentes na versão OA. Essas classes funcionam como a "cola" necessária para endereçar as composições transversais entre os frameworks. Assim, na versão OO algumas das classes internas de cada framework se comunicam com tais classes de composição que por sua vez repassam tal pedido para outros frameworks. Cada classe de composição é definida como um *Singleton* [45], e dessa forma cada uma declara explicitamente um atributo estático para referenciar a única instância que poderá existir daquela classe. Tais atributos estáticos contribuem para a diferença nos valores da métrica NOA. O valor superior da métrica de falta de coesão (LCOO) na versão OO também é causado por esses atributos *Singleton*, criados nas classes de composição da versão OO. Como tais atributos não são usados por vários métodos daquelas classes, eles contribuem para o aumento no valor de tal métrica.

Houve um grande equilíbrio entre ambas as versões do sistema considerando as métricas de linhas de código (LOC), profundidade de árvores de herança (DIT) e complexidade de operações do sistema (WOC). Havendo uma diferença de no máximo 2% quando comparando os valores coletados para cada

uma das versões, conforme pode ser visto no gráfico da Figura 64. As pequenas diferenças no valor das métricas DIT e LOC são também causadas pela criação dos três aspectos EJPs. Já a pequena diferença favorável a métrica WOC na versão OA é causada pelos métodos do padrão *Singleton* das classes de composição da versão OO.

Como conclusões gerais, nosso estudo quantitativo validou os benefícios trazidos pelas soluções OA de composição em termos de separação de interesses, complementando dessa forma o estudo qualitativo apresentado na seção 7.2.1. Além disso, o estudo também permitiu observar que outros atributos internos de tamanho, coesão e acoplamento permaneceram estáveis e equilibrados para ambas as versões do sistema.

7.3. Discussões e Lições Aprendidas

7.3.1. Composição e Interação de Aspectos de Extensão

Durante a definição de diferentes aspectos para um mesmo framework ou aplicação, podem ser identificadas interdependências entre os mesmos. Nesse caso, pode ser caracterizada a ocorrência de uma interação entre aspectos. Sanen et al [106] apresentam uma classificação de diferentes tipos de interação que podem existir entre aspectos, sendo eles: (i) exclusão mútua – nessa situação existe dois ou mais aspectos que não podem co-existir em uma mesma aplicação, porque provavelmente possuem o mesmo propósito, embora sejam implementados de forma distinta; (ii) dependência – está relacionado com a situação em que um dado aspecto necessita explicitamente da implementação oferecida por um outro aspecto, e como conseqüência, depende do mesmo; (iii) reforço – é caracterizado pela situação em que um dado aspecto influencia o correto funcionamento de um outro, oferecendo alguma informação ou funcionalidade útil; e (iv) conflito – define uma situação de interferência semântica entre os aspectos, isso significa que um aspecto que funciona corretamente quando isolado podem trazer problemas quando compostos com outros aspectos.

Para cada diferente tipo de interação entre aspectos, diferentes ações devem ser tomadas para endereçar a sua resolução. A exclusão mútua e a dependência entre aspectos de extensão, por exemplo, podem ser resolvidas através da instanciação conjunta (dependência) ou não (exclusão mútua) de aspectos do framework. O modelo generativo orientado a aspectos (apresentado no Capítulo 5) pode ajudar na resolução de tais tipos de interação, através da definição de restrições (*constraints*) `<<excludes>>` e `<<requires>>` no modelo de característica que auxilia o processo de instanciação. Como o reforço também pode ser visto como um tipo específico de dependência entre aspectos, ele também pode ser resolvido com a inclusão conjunta dos aspectos que reforçam um ao outro durante a instanciação dos aspectos do framework.

Quando vários aspectos de extensão são definidos para um mesmo framework existe também grande chance que eles atuem no mesmo ponto de junção de um dado EJP. Nesse caso, é importante observar se durante tal interação existe algum conflito entre os aspectos envolvidos. Nos casos de conflitos entre aspectos, deve-se verificar se os mesmos podem ser resolvidos, através do uso dos mecanismos de composição entre aspectos. Em AspectJ, a ordem de combinação de aspectos sobre pontos de junção do sistema pode ser definidas através da especificação de relações de precedência entre os aspectos. A construção `declare precedence` de AspectJ, permite definir ordens de precedência entre aspectos. Isso garante a ordem de execução de adendos baseado nas regras de precedência de AspectJ[23].

De forma geral, interações entre aspectos do tipo dependência, reforço e conflito devem ser projetadas e implementadas cuidadosamente. Nossa abordagem contribui de duas formas para lidar com situações de interações entre aspectos: (i) a especificação de EJPs na nossa abordagem torna explícito os pontos de junção que aspectos podem interagir, apoiando assim a atividade de identificação de possíveis pontos de interação entre aspectos; e (ii) a definição explícita de relações transversais válidas entre características `<<crosscutting>>` e `<<join point>>` no nosso modelo de configuração permite restringir interações indesejáveis entre aspectos.

[23] The Aspectj Programming Guide. URL: http://eclipse.org/aspectj/, A. Team, Editor. 2007..

7.3.2. Documentação de EJPs

Um ponto importante a ser considerado quando desenvolvendo frameworks usando nossa abordagem é como documentar os EJPs. A forma como os mesmos são documentados pode auxiliar desenvolvedores a implementar mais facilmente extensões aos seus respectivos frameworks. Diferentes formas de documentação podem ser usadas de forma complementar. Em particular uma combinação de documentação baseada em linguagem de programação, textual e visual pode auxiliar a tornar mais explícitos os EJPs.

A documentação dos EJPs de um framework usando código-fonte, ilustrada nos Capítulos 4 e 6, é considerada essencial, porque além de documentá-los formalmente usando construções de linguagens de programação, pode também ser usada pelos desenvolvedores durante a codificação dos aspectos de extensão. Tal documentação foi inspirada na proposta de especificação de interfaces transversais (XPIs) apresentada em [56]. Também a codificação de contratos a serem respeitados pelos aspectos de extensão auxilia a controlar interações indesejadas entre o framework e os aspectos de extensão. Sullivan et al [115] também apresentam uma representação textual para documentar XPIs, a qual pode também ser usada para documentar os EJPs. Embora, as propostas para documentação de XPIs sejam úteis para documentar EJPs, novas propriedades devem ser consideradas durante sua documentação. EJPs que podem ser especializados devem ser indicados explicitamente, para facilitar a distinção entre aqueles EJPs que afetam apenas classes internas do framework daqueles que podem estender diretamente implementação concretas de pontos flexíveis de uma instância do framework. A documentação baseada em código fonte, por exemplo, pode apresentar exemplos de como EJPs que afetam diretamente pontos flexíveis, podem ser adaptados para afetar apenas instâncias específicas de tais pontos flexíveis.

Muitas técnicas de documentação de frameworks OO (tais como, *cookbooks* [37]) enfatizam o uso de exemplos para mostrar como instanciar corretamente os pontos de extensão do framework. No caso de EJPs é também fundamental apresentar exemplos de como os mesmos podem ser usados para incorporar alguma funcionalidade adicional no framework. Exemplos de como estender os

EJPs para implementar aspectos que desempenham papéis de padrões de projeto clássicos, tais como, *Observer*, *Mediator* e *Decorator*, são também fundamentais para exemplificar os diferentes tipos de aspectos que podem ser escritos para um dado EJP. Finalmente, a documentação de EJPs pode também oferecer alguma representação visual para torná-los mais explícitos na documentação da arquitetura e projeto detalhado do sistema. A seção 4.1.1 apresentou uma proposta simples de documentação das EJPs usando estereótipos UML. Abordagem baseadas em UML [23, 24] ou ADL [12, 47] para representação explícita e mais detalhada de EJPs merecem também ser exploradas.

7.3.3. Casos de Uso de Extensão

Casos de uso [61] é uma técnica amplamente adotada para especificação de requisitos. Ela tem sido adotada por muitos processos de desenvolvimento de software modernos. Um caso de uso pode ser definido como uma seqüência de ações executadas por um sistema que traz algum resultado de valor observável para um usuário em particular. A técnica de caso de uso oferece um mecanismo de extensão, o qual pode ser usado para descrever comportamento (obrigatório ou opcional) extra. O relacionamento `<<extend>>` pode ser definido entre casos de uso para alcançar tal propósito de extensão. Um caso de uso de extensão pode adicionar comportamento em conjuntos de pontos de extensão específicos de outros casos de uso. Jacobson [60] argumenta que o mecanismo de extensão de casos de uso pode ser usado para modelar aspectos durante a etapa de especificação de requisitos. Na abordagem proposta por Jacobson, casos de uso de extensão modelam aspectos e os pontos de extensão de casos de uso representam pontos de junção no nível de requisitos.

Existe uma forte sinergia entre os EJPs propostos em nossa abordagem e os pontos de extensão de casos de uso propostos por Jacobson [60]. Esses últimos são candidatos naturais a serem modelados e implementados como EJPs. Da mesma forma, casos de uso de extensão podem ser implementados como aspectos de extensão da nossa abordagem. Nosso trabalho também apresenta um conjunto de regras de mapeamento entre tipos de características e elementos de implementação existentes na nossa abordagem OA para o desenvolvimento de

frameworks. A existência de regras de mapeamento entre elementos existentes em modelos de características, de casos de uso e de implementação definidos pela nossa abordagem, pode auxiliar-nos a definir e manter ligações de rastreamento (*traceability links*) [63] entre diferentes artefatos produzidos nas etapas de requisitos, projeto arquitetural e implementação. Tais ligações de rastreamento podem auxiliar no desenvolvimento e evolução de frameworks e linhas de produto oferecendo suporte para atividades de engenharia de software, tais como, análise de impacto de mudanças e gerenciamento consistente de variabilidades. Assim, a definição explítica de mapeamentos entre os elementos de implementação propostos na nossa abordagem, tipos de características e casos de uso, pode ser útil para oferecer um melhor suporte ao gerenciamento de ligações de rastreamento existentes entre tais artefatos.

7.3.4. Modelagem e Estabilidade de EJPs

Uma das principais dificuldades no desenvolvimento de frameworks OO é encontrar seus pontos flexíveis (*hot-spots*). Métodos de análise de domínio [33, 101] e experiência no desenvolvimento de aplicações para um mesmo domínio são técnicas usualmente adotadas para encontrar funcionalidades comuns e variáveis existentes em frameworks e linhas de produto. EJPs são também considerados pontos flexíveis (*hot-spots*) de frameworks ou linhas de produto. Eles representam pontos específicos na execução de cenários específicos do framework e linhas de produto, os quais podem ter uma extensão transversal inserida. EJPs são modelados como eventos ou estados de transição ocorrendo durante a execução de funcionalidades do framework. Esses eventos ou estados são dependentes do domínio e funcionalidades sendo endereçadas pelo framework.

Enquanto algumas heurísticas para encontrar EJPs podem ajudar, tal como a identificação de eventos relevantes e estados de transição na execução do framework, é fundamental estender métodos e técnicas existentes de análise de domínio para endereçar a modelagem de relações transversais entre características em fases preliminares de desenvolvimento (especificação e modelagem de requisitos e da arquitetura do sistema). As regras de mapeamento definidas entre

tipos de características, casos de uso e elementos de implementação apresentadas nesse trabalho (Seção 7.3.3) representam um relevante passo nesse sentido. Elas podem auxiliar na modelagem antecipada de EJPs. De forma geral, um EJP pode ser modelado como um ponto de integração de relação transversal entre duas ou mais características.

O entendimento e modelagem de EJPs em fases preliminares do desenvolvimento pode também contribuir para o aumento da estabilidade de sua implementação. Os EJPs especificam não apenas um conjunto de pontos de junção de extensão no qual um framework ou uma linha de produto pode ser estendida, mas também representam interfaces existentes entre as classes do framework e os aspectos de extensão. Nesse sentido, eles têm o mesmo propósito das XPIs, propostos por Griswold et al [56]. Conseqüentemente, as EJPs oferecem como benefício possibilitar a evolução das classes do framework sem quebrar os aspectos que estendem a sua funcionalidade. Entretanto, para alcançar esse benefício é fundamental que pontos de junção expostos por EJPs e seus respectivos contratos permaneçam funcionando quando refatorando as classes do framework. EJPs devem também evoluir para acomodar novos requisitos requeridos pelos aspectos de extensão, tais como, expor novos pontos de junção do framework argumentos adicionais nos pontos de junção existentes. Todos os contratos definidos por EJPs devem ser revalidados e se necessário modificados durante a refatoração e evolução de EJPs devido a mudanças nas classes do núcleo ou novas demandas dos aspectos de extensão.

7.3.5. Estratégias de Adoção de Linhas de Produto de Software

Diferentes estratégias de adoção [71] podem ser usadas para desenvolver linhas de produto de software (LPSs), são elas: proativa, extrativa e reativa. A abordagem proativa motiva o desenvolvimento de uma LPS considerando todos os produtos existentes em um dado escopo especificado. O conjunto completo de artefatos é desenvolvido partindo do zero. Na abordagem extrativa, uma LPS é desenvolvida a partir de sistemas de software existentes. Características comuns e variáveis são extraídas desses sistemas para derivar uma versão inicial da linha de produto. Finalmente, a abordagem reativa motiva o desenvolvimento incremental

de LPSs. Os artefatos de uma LPS endereçam inicialmente apenas um conjunto restrito de produtos. Quando existem novas demandas para incorporar novos requisitos ou produtos, os artefatos que implementam as características comuns e variáveis são incrementalmente estendidos para endereçar tais demandas.

Embora nossa abordagem tenha sido usada principalmente em refatorações OA de frameworks e linhas de produtos OO existentes, as diretrizes de implementação de características transversais usando aspectos (Capítulo 4) podem também ser usadas em conjunção com diferentes estratégias de adoção de LPSs. Também as regras de mapeamento apresentadas neste trabalho (Capítulo 5) desempenham um papel importante nas estratégias de adoção, uma vez que mostram claramente as relações entre tipos de características e elementos de implementação. Na abordagem extrativa, nossas diretrizes e regras de mapeamento são úteis para determinar quais características opcionais, alternativas e de integração podem ser refatoradas usando aspectos. A abordagem reativa pode se beneficiar da especificação prévia de EJPs para introduzir novas demandas de características opcionais e de integração transversais que agem sobre o núcleo do framework ou linha de produto. Os EJPs também promovem um fraco acoplamento entre o núcleo da LPS e suas respectivas extensões transversais. Esse fraco acoplamento pode auxiliar diretamente no desenvolvimento incremental ou evolução de LPSs simplificando a complexidade de entendimento do núcleo e permitindo plugar/ desplugar determinadas características. Finalmente, a abordagem proativa pode se beneficiar do mapeamento entre casos de uso de extensão, características e aspectos de extensão (Subseção 7.6). Processos de desenvolvimento de LPSs existentes podem usar tais regras de mapeamento para apoiar a análise e identificação de aspectos candidatos para implementar determinadas características.

7.4. Sumário

Neste capítulo foram discutidos os benefícios e lições aprendidas a partir do uso da abordagem na realização dos três estudos de caso. Em particular, foi apresentado um estudo qualitativo e um estudo quantitativo que reforçam os benefícios trazidos pela abordagem em termos de modularidade do framework. O

estudo quantitativo também mostrou que a modularização do código de composição usando aspectos no estudo de caso do framework *Measurement*, não penalizou consideravelmente atributos internos de tamanho, coesão e acoplamento. O capítulo foi finalizado com a discussão de diversas lições aprendidas e discussões sobre a abordagem no que se refere: (i) à lidar com problemas de interação entre aspectos; (ii) a questões acerca da modelagem e estabilidade de EJPs; (iii) ao potencial de integração com os casos de uso de extensão propostos por Jacobson et al [60]; (iv) aos benefícios trazidos pela abordagem para estratégias de adoção de linhas de produto de software; e (v) à necessidade de oferecer documentação explícita para os EJPs.

8
Trabalhos Relacionados

Neste capítulo a abordagem OA para desenvolvimento de frameworks é comparada com diversos trabalhos desenvolvidos pela comunidade. A abordagem é confrontada com: (i) abordagens para implementação de famílias de software que buscam uma melhor modularização de características; (ii) abordagens para instanciação de linhas de produto; e (iii) abordagem baseada em interfaces transversais.

8.1.
Interfaces Transversais (XPIs)

Nosso conceito de EJPs é inspirado no trabalho proposto por Sullivan et al [115] de especificação de interfaces transversais (XPIs). XPIs permitem abstrair comportamento transversal, isolando o projeto de aspectos do código base, e vice-versa. A proposta inicial de XPIs foi documentar *design rules*, através de um formato descritivo contendo os seguintes elementos: nome, razão da existência, dependências, escopo do código afetado, um conjunto de contratos que são oferecidos ou requeridos do núcleo da aplicação ou aspectos. O uso das XPIs é ilustrado por tais pesquisadores, através de um *middleware* de redes sobrepostas, e comparada com a abordagem baseada no princípio *Obliviousness*. Como continuidade do trabalho, Griswold et al [56] mostram como representar XPIs como construções sintáticas em AspectJ.

EJPs podem ser vistos como a especialização do conceito das XPIs para o contexto de desenvolvimento de frameworks. EJPs estão para XPIs, assim como pontos flexíveis (*hot-spots*) OO de frameworks estão para classes abstratas e interfaces. O objetivo central das EJPs é expor um conjunto de eventos/estados do framework que podem ser facilmente estendidos ou observados por aspectos de extensão. Nossa proposta para especificação sintática de EJPs em AspectJ é uma extensão a aquela proposta em [56], com a introdução do conceito de especialização. Para a especificação da parte semântica de EJPs, entretanto, uma

metodologia foi definida a qual explicita diferentes tipos de contratos que podem ser estabelecidos para regular as interações entre o núcleo do framework, os EJPs e os aspectos de extensão.

8.2. Abordagens para Implementação de Famílias de Software

8.2.1. Programação Orientada a Características

Programação orientada a características (FOP – *feature oriented programming*) foi proposta [15, 111] para lidar com o encapsulamento de características que podem ser usados para estender a funcionalidade de programas base existentes. A idéia central proposta por FOP é permitir a geração de aplicações através da composição de características. Uma característica pode ser vista como um incremento na funcionalidade do programa. FOP propõe uma abordagem de refinamento incremental onde características representadas por unidades de implementação modular são gradativamente aplicadas a uma base inicial de código-fonte. *Mixin-Layers* [111] é uma das tecnologias que podem ser usadas para implementar características em FOP. Um módulo *Mixin-Layer* permite o encapsulamento de refinamentos de diversas classes.

Batory et al [14] argumentam as vantagens que abordagens de FOP têm em relação a frameworks OO para projetar e implementar linhas de produto. O módulo *Layer* proposto por abordagens de FOP, permite encapsular várias abstrações que refinam uma base pré-definida, juntas tais abstrações determinam a implementação de uma característica dentro de uma única unidade modular. Tal mecanismo permite uma melhor gerência de características em frameworks, através da adição, remoção ou modificação das diferentes *Layers* definidas para a linha de produto. Mezini & Osterman [94] identificam, entretanto, que FOP é uma abordagem capaz apenas de prover suporte para modularização de características hierárquicas, não sendo capaz de endereçar o projeto e implementação de características transversais. Em especial, FOP não oferece mecanismos para suportar modularização transversal: (i) estática – existindo casos em que os refinamentos a serem definidos para uma dada característica não possuem correspondência direta para estruturas modulares do código base; e (ii) dinâmica –

abordagens FOP só permitem interceptar chamadas de métodos, não havendo suporte em tais abordagens para expressar/especificar um conjunto de pontos de junção do código base a serem afetados por um dado refinamento definido por uma *Layer*.

Recentes estudos [8, 88, 94] têm explorado o uso integrado das abordagens de POA e FOP. Mezini & Osterman [93] propõem CaesarJ, uma linguagem orientada a aspectos que incorpora conceitos de ambas as abordagens visando lidar com as deficiência de ambas. CaesarJ propõe o conceito de *family class* para agregar um conjunto de refinamentos a serem aplicados a uma dada base existente com o objetivo de implementar uma dada característica. Cada refinamento é definido por uma *virtual class*, a qual por sua vez pode definir extensões para serem aplicadas à classes existentes, como por exemplo, redefinir ou sobrepor métodos. Pontos de corte e adendos podem também ser definidos dentro do escopo de uma *family class*, permitindo assim que mecanismos OA possam ser usados na implementação de características sempre que necessário. Mecanismos para facilitar a reutilização assim como a instalação dinâmica de implementações de características estão também disponíveis em CaesarJ.

Apel et al [8] apresentam a abordagem *Aspectual Mixin Layers* (AMLs), a qual também integra os conceitos de POA e FOP, propondo mecanismos no nível de linguagem de programação que estendem a abstração de *Mixin Layers* [111], e para permitir o encapsulamento de *mixins* e aspectos. Cada AML pode implementar papéis de extensão a uma classe, assim como aspectos que contribuem para a implementação de uma característica do sistema. Os autores propõem a estruturação de arquiteturas OA em 2 níveis: (i) aspectos devem ser usados no nível de classes para implementar características transversais; (ii) enquanto *mixins* são responsáveis pela organização da arquitetura como um todo oferecendo mecanismos para estruturar aspectos e classes em módulos de mais alto nível que endereçam determinadas características do sistema. Apel & Batory [7] apresentam os benefícios e potencial da abordagem AMLs numa linha de produto de redes sobrepostas. Os resultados de tal estudo de caso demostram: (i) a utilidade de FOP para endereçar extensões localizadas, tais como, a adição de novas classes e novos membros em classes existentes; e (ii) a utilidade da adoção de aspectos para melhorar a modularidade transversal de determinadas características e para reduzir a quantidade de código redundante.

Nossa abordagem propõe a modularização de características transversais encontradas em frameworks usando programação orientada a aspectos. AspectJ foi a linguagem adotada usada em nossos estudos de caso, por ser a de maior amadurecimento até o momento. De forma distinta aos trabalhos apresentados de combinação de uso de FOP e aspectos, nossa abordagem utiliza EJPs como um mecanismo que permite isolar o projeto do núcleo do framework e de seus aspectos de extensão, dando dessa forma mais disciplina ao desenvolvimento do framework ou arquitetura de família de sistemas. Nos estudos de caso apresentados nesta tese, os mecanismos de POA em combinação com a técnica de frameworks OO foram suficientes para endereçar a modularização adequada de suas características. Explorar o potencial do uso combinado de FOP e POA na implementação de características encontradas em arquiteturas de famílias de sistemas é um tópico interessante para explorar, de forma a derivar diretrizes para potenciais cenários de aplicação de cada uma das tecnologias. Também o uso de técnicas generativas de instanciação baseada em modelos de características ou DSLs em combinação com técnicas de FOP também merece ser investigado.

8.2.2. Método de Decomposição Horizontal

Zhang e Jacobsen [118] propõem o método de Decomposição Horizontal, um conjunto de princípios que auxiliam na modularização de arquiteturas orientadas a aspectos, compostas de um núcleo e um conjunto de customizações do núcleo. Tais customizações são implementadas usando programação orientada a aspectos. Os seguintes princípios norteiam o método: (i) reconhecimento da relatividade de aspectos – que indica que a semântica de aspectos é determinado pela funcionalidade principal da aplicação; (ii) estabelecer a decomposição coerente do núcleo – a base de uma decomposição OA é o estabelecimento da funcionalidade de um núcleo coerente; (iii) definir a semântica de um aspecto de acordo com a decomposição do núcleo – usando o núcleo como referência, uma funcionalidade é considerada transversal se ambos sua semântica e implementação não estão localizados em um único componente do núcleo; (iv) manter uma arquitetura *class-directional* – aspectos devem referenciar/interceptar as classes do núcleo, mas não o contrário; e (v) aplicar refatorações incrementais –

reconhece a dificuldade de estabelecimento de um núcleo e propõe a realização de refatorações incrementais ao longo da evolução do sistema.

O método proposto por Zhang & Jacobsen organiza um conjunto de boas práticas utilizadas para desenvolvimento de arquiteturas OA e o combina com a técnica de refatoração orientadas a aspectos [96] de forma a facilitar a evolução e customização do núcleo. O método foi aplicado com sucesso no domínio de *middlewares* para sistemas distribuídos [118] e também ao sistema de banco de dados Prevayler [52]. Os principais benefícios do método em tais estudos foram: (i) obter uma versão mais simples e reduzida do núcleo do sistema sendo refatorado; e (ii) aumento do grau de configurabilidade e adaptabilidade do sistema, através da resolução de problemas de combinação e conflito entre características.

Nossa abordagem para desenvolvimento de frameworks também motiva a modularização de sua arquitetura através da definição de um núcleo central contendo a funcionalidade obrigatória a todas as instâncias do framework, e um conjunto de extensões OA que são usadas para implementar características transversais opcionais ou alternativas a esse núcleo. Nossa abordagem também é centrada no princípio *class-directional*, proposto inicialmente por Kerten & Murphy [67]. Ao contrário do método de Decomposição Horizontal, o qual adota o princípio de Inconsciência (*Obliviousness*) [43], nossa abordagem é centrada na definição de EJPs, os quais buscam uma diminuição do acoplamento entre o núcleo e as extensões do framework. O uso de EJPs no desenvolvimento de frameworks ou arquiteturas flexíveis OA traz os seguintes benefícios adicionais: (i) facilidade de entendimento da forma como cada extensão OA afeta o núcleo; (ii) possibilita a identificação de conflitos entre extensões OA agindo sobre os mesmos conjuntos de pontos de junção; e (iii) maior estabilidade do núcleo e dos aspectos de extensão durante a evolução do sistema.

8.2.3. Frameworks Transversais

Camargo & Masiero [21, 22] propõem uma classificação para Frameworks Orientados a Aspectos (FOAs). De acordo com tais pesquisadores, um FOA é definido como um conjunto de classes e aspectos que formam uma estrutura semi-

completa que necessita ser completada com detalhes de uma aplicação específica. Os FOAs são classificados como sendo de dois tipos: (i) Framework Transversal (FTs) – que são os FOAs que endereçam apenas um interesse transversal; e (ii) Frameworks de Aplicação Orientado a Aspectos (FAOAs) – que são aqueles FOAs que definem uma arquitetura genérica para um dado domínio de aplicação e cuja instanciação produzem uma aplicação. A arquitetura dos FAOAs é projetada usando as abstrações de classes e aspectos. O foco principal do trabalho desses pesquisadores é no desenvolvimento de Frameworks Transversais. Eles propõem uma arquitetura de referência para o projeto de FTs e apresentam três famílias de FTS que endereçam os seguintes interesses transversais: persistência de dados, segurança e regras de negócio. Finalmente, um processo para desenvolvimento de software apoitado pelo uso de FTs também é proposto.

Uma das diferenças centrais de nosso trabalho com o proposto por tais autores, é que o foco principal dessa tese é nos Frameworks de Aplicação Orientado a Aspectos, em contraposição aos Frameworks Transversais. Essa tese propõe diretrizes explícitas para a modularização de características transversais em FAOAs. Nossa abordagem é também centrada na exposição de EJPs dos FAOAs na forma de interfaces transversais, não sendo baseada no princípio de Inconsciência (*Obliviousness*). Os aspectos do núcleo e de extensão (variabilidade e integração) da nossa abordagem podem também a princípio serem projetados na forma de FTs. Um ponto importante de discussão na categorização de FOAs proposta por tais autores é identificar as diferenças entre bibliotecas de aspectos e FTs. Uma biblioteca de aspectos [70] define um conjunto de aspectos abstratos que modularizam um dado interesse transversal e que podem ser especializados para serem usados no contexto específico de uma aplicação. Diversas linguagens OA têm disponibilizado bibliotecas de aspectos [66]. A diferença central entre ambos é sutil, e indica que os FTs precisam não apenas ser compostos com a aplicação (através da concretização de pontos de corte abstratos), mas também oferece variabilidades funcionais a serem instanciadas (através da implementação de métodos abstratos dos aspectos ou da agregação de uma diferente implementação para um dado tipo abstrato - classe ou interface).

8.2.4. Aspectos de Especialização

Santos et al [108] propõem o conceito de *Aspectos de Especialização* com o objetivo de possibilitar a modularização de pontos flexíveis (*hot-spots*) de frameworks. De acordo com tais pesquisadores, a instanciação de frameworks ocasiona o entrelaçamento e espalhamento de código relacionado diretamente a cada um dos pontos de extensão do framework. Os autores propõem a criação de diversos aspectos abstratos de especialização internos ao framework, os quais são especializados por aspectos da aplicação, durante a instanciação do framework. Diversos benefícios são relatados a partir do uso da abordagem em um estudo de caso, tais como, aumento da configurabilidade e coesão de cada característica relacionada aos pontos flexíveis, redução da complexidade de uso do framework, facilidade de reúso e evolução. Uma das principais desvantagens da abordagem é a criação de uma série de aspectos de especialização, a qual cresce de forma proporcional à quantidade de pontos flexíveis definidos pelo framework.

A principal diferença entre nossa abordagem e a proposta por tais autores é que embora ambas busquem a melhoria da configurabilidade do framework através da gerência de suas variabilidades, nossa abordagem endereça a modularização de características variáveis internas ao framework, enquanto os aspectos de especialização endereçam características oriundas da especialização de cada ponto flexível do framework, e que ocorrem durante a sua instanciação. Assim, as abordagens podem ser usadas de forma complementar, embora isso possa ocasionar a codificação de uma quantidade excessiva (e talvez desnecessária) de aspectos. Nossa abordagem propõe um modelo generativo que habilita a instanciação automática dos pontos flexíveis até mesmo na presença de entrelaçamento e espalhamento de código relacionados à instanciação. Estratégias de uso de modelos de característica ou linguagens específicas de domínio vêm sendo adotadas por várias ferramentas disponíveis na indústria.

8.2.5. Framed Aspects

Frame [11] é uma tecnologia que permite particionar o código-fonte de um sistema ou componente em uma hierarquia de unidades menores, denominadas

frames. Cada *frame* especifica a funcionalidade comum e variável de um dado elemento de implementação. Desenvolvedores podem então definir uma especificação (conjunto de parâmetros) a ser manipulada por um processador de *frames* que então se responsabiliza pela customização da parte varíavel dos frames.

Framed Aspects [89] é uma abordagem que combina as tecnologias de programação orientada a aspectos e *frames* para facilitar a customização automática de aplicações. Aspectos são usados para modularizar interesses transversais e *frames* habilitam a parametrização e configuração de variabilidades encontradas em aspectos. O papel desempenhado pela tecnologia de frames na abordagem *Framed Aspects* é endereçado na nossa abordagem pelo uso de templates de aspectos. Tais templates podem ser customizados baseado em informação coletada por modelos de características.

Outra diferença fundamental entre as abordagens, é que *Framed Aspects* define vários passos de decisão do processo de instanciação no código de templates de *frames* por meio de meta-tags. Na abordagem proposta nesse trabalho, boa parte dessas decisões é especificada separadamente no modelo de configuração. Isso facilita a adaptação ou evolução das decisões relacionadas a customização da arquitetura. Finalmente, nossa abordagem também propõe um conjunto explícito de diretrizes para a modularização de características transversais em frameworks ou arquiteturas de famílias de aplicações os quais não são endereçados pela abordagem *Framed Aspects*.

8.2.6. Abordagem Extrativa de Linha de Produtos

Alves et al [2, 4, 5] definem um método sistemático para a criação e evolução de linhas de produto de software (LPSs). A abordagem propõe a extração de LPSs a partir de produtos existentes, usando refatorações. Essas refatorações são usadas para expor variabilidades requeridas por uma dada LPS. Durante sua evolução, a LPS é então estendida para endereçar novas variabilidades e/ou produtos. O conjunto de refatorações proposto pode ser aplicado tanto no nível de código quanto no modelo de característica da LP.

Aspectos são usados para modularizar características transversais encontradas na implementação de uma LPS.

Nossa abordagem é complementar àquela proposta por Alves et al [2, 4, 5]. As diretrizes apresentadas para modularização de características transversais podem ser usadas durante a extração ou evolução de LPSs. Os EJPs podem também auxiliar os desenvolvedores durante a evolução da LPS, oferecendo um conjunto de pontos de junção candidatos a serem estendidos. Nossa abordagem também oferece uma proposta para especificação do conhecimento de configuração, o qual não é plenamente endereçado por aquele trabalho. Por outro lado a abordagem proposta por tais autores pode trazer mais sistematização a refatoração e evolução de frameworks e linhas de produto, permitindo acompanhar e sincronizar simultaneamente as mudanças aplicadas aos modelos de arquitetura, característica e configuração.

8.3. Abordagens para Instanciação de Frameworks e Linhas de Produto

8.3.1. Abordagem para Instanciação de Frameworks OO

Filho et al [41] propõem uma abordagem para instanciação de frameworks centrada no uso do modelo de característica e da linguagem UML. A abordagem é estruturada em três passos:

(i) *processo de documentação do framework* – um diagrama de classes do framework é relacionado com o seu respectivo modelo de característica. Elementos de projeto (classes, métodos, atributos) que representam pontos de extensão do framework são explicitamente anotados no diagrama de classes representando o framework. Tais elementos de projeto foram definidos como extensões do meta-modelo de UML. Finalmente, relações de dependência UML são definidas para relacionar elementos de ambos os modelos;

(ii) *processo de geração de script de instanciação* – nesse primeiro passo da instanciação do framework, desenvolvedores selecionam as características desejadas para a aplicação que irão criar, e uma ferramenta de posse do modelo de característica e do diagrama de classes representando o framework, gera um script

na linguagem RDL (*Reuse Description Language*) [97] contendo os passos necessários para instanciar a aplicação; e

(iii) *instanciação do framework* – no último passo da instanciação, o script RDL gerado pela atividade anterior é processado para estender o diagrama de classes do projeto original do framework de forma a inserir as classes, métodos e atributos que implementam a concretização dos seus pontos de extensão. Esse processo é semi-automático, intervenções humanas podem ocorrer de forma a solicitar dos desenvolvedores informações adicionais, tais como, nomes de classes, métodos e atributos. Trabalhos recentes têm estendido a abordagem proposta para lidar com a instanciação de frameworks orientados a aspectos [99].

Existem similaridades e diferenças entre nosso modelo generativo para instanciação de frameworks e a abordagem proposta por Filho et al [41]. Ambas as abordagens são centradas na definição de três modelos: de característica, de configuração (ou dependência) e de arquitetura. Nosso modelo de característica, entretanto, incorpora duas propriedades (`<<crosscutting>>` e `<<join point>>`) de forma a oferecer uma representação de aspectos do espaço de solução no espaço de problema. O modelo de arquitetura usado por tais autores é o diagrama de classes UML, enquanto na nossa abordagem tal modelo é uma representação dos artefatos de implementação. As relações de dependência definidas em nosso modelo de configuração têm um propósito similar àquelas propostas por tais autores. Nosso modelo de configuração incorpora informações adicionais, tais como, restrições específicas para definição de relações transversais entre características, assim como especifica o mapeamento de features `<<join point>>` para pontos de junção concretos no espaço de solução. Outra diferença fundamental entre as abordagens, é que são usados templates na nossa abordagem para expressar variabilidades ocorrendo em elementos de implementação da arquitetura, enquanto tais autores usam anotações explícitas em modelos de projeto para representar os pontos de extensão.

8.3.2. Pure::Variants

Existem algumas ferramentas, tais como pure::variants[24] e Gears[25], que foram desenvolvidas para lidar com a derivação e instanciação de famílias de sistemas usando como base modelos de característica. Pure::variants é uma das principais ferramentas disponíveis no mercado. Ela permite especificar linhas de produto ou famílias de sistemas, através da definição de: (i) um modelo de característica; (ii) um modelo da família – que é responsável pela representação dos componentes da arquitetura do sistema; e (iii) modelos do espaço de configuração – definem configurações e transformações, através do modelo de descrição de variantes, que serão utilizadas na instanciação da linha de produto. A ferramenta propõe um modelo que é independente tanto de tecnologias de implementação da arquitetura da linha de produto quanto das técnicas de transformação e geração de código utilizadas.

Existem similaridades e diferenças entre o nosso modelo generativo e a ferramenta pure::variants. Ambos se propõem a instanciar linhas de produto ou famílias de sistemas a partir do modelo de característica. A especificação de relações de dependência entre elementos de implementação e características na ferramenta pure::variants é definida no próprio modelo da família, através de um conjunto de regras de restrição lógica, enquanto na nossa abordagem essas relações são especificadas separadamente no modelo de configuração. Essa especificação separada permite adaptar ou modificar o conhecimento de configuração de forma independente dos modelos de característica e arquitetura. Também é possível que 2 (ou mais) linhas de produto que usem o mesmo conjunto de artefatos (expresso pelo modelo de arquitetura) sejam especificada cada uma com o seu próprio modelo de configuração. Nosso modelo generativo também propõe dois elementos adicionais não contemplados em pure::variants que são as relações válidas entre características to tipo `<<crosscutting>>` e `<<join point>>`, além do mapeamento de características `<<join point>>` para pontos de junção concretos. Tais elementos são diretamente responsáveis pela instanciação das variabilidades OA.

[24] Pure::Variants. URL: http://www.pure-systems.com/, 2007.
[25] Big Lever. Gears. URL: http://www.biglever.com. 2007.

9
Conclusões e Trabalhos Futuros

Nesse capítulo são apresentadas as conclusões gerais do trabalho com uma sintetização das contribuições, além de uma lista de trabalhos futuros.

Esse trabalho discutiu problemas de modularização de características transversais encontradas em frameworks OO. Uma abordagem OA para o desenvolvimento de frameworks foi apresentada. Ela é composta por: (i) um conjunto de diretrizes que auxiliam na modularização de características transversais encontradas em frameworks usando aspectos; e (ii) um modelo generativo que endereça a instanciação automática de variabilidades OO e OA presentes no framework. A abordagem é fundamentada na definição de pontos de junção de extensão (EJPs), os quais expõem eventos e estados relevantes do framework. Cada EJP é um potencial candidato a ser estendido para implementação de características transversais opcionais, alternativas ou de integração. Um ponto importante da abordagem é que os EJPs não precisam ser todos descobertos antecipadamente, eles podem ser especificados ao longo do ciclo de desenvolvimento e evolução de frameworks, sendo úteis não apenas em abordagens proativas, mas também naquelas extrativas ou reativas [71] baseadas em técnicas de refatoração ou reengenharia.

O trabalho demonstra a complementaridade entre os paradigmas OO e OA. A técnica de frameworks OO, uma das mais utilizadas pela indústria para o projeto e implementação de arquiteturas flexíveis, desempenha seu papel de modularização de características essenciais que devem estar presentes em todas as instâncias do framework. Aspectos são usados para modularizar características transversais opcionais, alternativas ou de integração que precisam ser incorporadas ao núcleo do framework. EJPs mediam a forma como o núcleo do framework pode ser estendido pelos aspectos, exposto um conjunto de pontos de junção de classes do framework e especificando um conjunto de contratos que devem ser respeitados tanto pelo núcleo quanto pelos aspectos de extensão. O trabalho também ilustra como dois trabalhos estado da arte na área de DSOA,

podem ser usados no desenvolvimento de arquiteturas flexíveis, sendo eles: (i) interfaces transversais [56, 115] – as quais permitem o uso sistemático e controlado de aspectos. EJPs podem ser vistos como especializações das interfaces transversais; e (ii) implementação de padrões de projeto com aspectos [58] – aspectos de extensão representam um exemplo prático do e útil uso de aspectos para modularizar padrões de projeto clássicos, tais como, *Observer*, *Mediator* e *Decorator*.

9.1. Contribuições

As seguintes contribuições são resultados diretos dessa tese:

- **Uma Abordagem OA para Desenvolvimento de Frameworks** (Capítulo 3). Definição de uma abordagem sistemática para a implementação de características tranversais encontradas em frameworks OO, através do uso de técnicas orientadas a aspectos. Essa abordagem é fundamentada na definição de um conjunto de pontos de junção de extensão (EJPs) existentes no framework;
- **Diretrizes para Implementação de EJPs em AspectJ** (Capítulo 4). Desenvolvimento de um conjunto de diretrizes para a implementação de EJPs em AspectJ, incluindo o endereçamento de seus contratos e suas respectivas especializações durante a definição de aspectos de extensão;
- **Categorização de Contratos entre Frameworks e Extensões** (Capítulo 4). Proposta de uma categorização de diferentes tipos de contratos existentes entre os elementos de nossa abordagem. Essa categorização envolve: (i) contratos internos entre o núcleo do framework e os EJPs - que buscam garantir que manutenções no framework não invalidarão os EJPs; e (ii) contratos externos entre os EJPs e os aspectos de extensão - que visam garantir que cada aspecto de extensão respeita restrições e invariantes do framework;
- **Modelo Generativo Orientado a Aspectos** (Capítulo 5). Proposta de um modelo generativo OA que endereça a instanciação de frameworks ou arquiteturas de linhas de produto implementadas com o uso de linguagens de programação orientadas a aspectos;

- **Estudos de Caso** (Capítulo 6). Desenvolvimento de 3 estudos de caso de frameworks de diferentes domínios, que permitiram fazer uma avaliação inicial da abordagem. Os estudos de caso ilustram o caratér genérico da abordagem e a possibilidade de aplicá-la no desenvolvimento de frameworks de aplicação OO.

9.2. Trabalhos em Andamento e Futuros

Diversas pesquisas estão em andamento e podem ser endereçadas futuramente como continuidade desse trabalho de doutorado, dentre elas:

- **Implementação de uma Ferramenta de Instanciação de Arquiteturas de Famílias de Sistemas**. Uma ferramenta [26] vem sendo desenvolvida como continuidade das pesquisas dessa tese a qual contempla o modelo generativo OA apresentado. A plataforma Eclipse [109] assim como plugins disponíveis para a modelagem de características [6], manipulação de modelos e geração de código [18] estão sendo usados no desenvolvimento de tal ferramenta;
- **Desenvolvimentos de Novos Estudos de Caso**. Embora alguns estudos de caso tenham sido apresentados nessa tese, novos frameworks e famílias de sistemas de larga escala precisam ser desenvolvidos ou refatorados de forma a validar e evoluir a abordagem. Exemplos de frameworks que já vêm sendo desenvolvidos usando a abordagem são: um framework de um servidor de multimídia adaptativo [72] e um framework de análise e monitoramento de processos de negócio em sistemas web [36];
- **Realização de Avaliações Quantitativas**. Dentro do contexto de desenvolvimento de novos estudos de caso, é também importante a realização de novos estudos quantitativos e qualitativos, com o objetivo de comparar a abordagem proposta com outras abordagens que buscam a modularização de características transversais em famílias de sistemas. Um dos estudos que pode ser realizado é a refatoração de uma versão do Prevayler [52] que foi originalmente implementado em OO e posteriormente refatorada usando o método de Decomposição Horizontal [118];
- **Elaboração de um Método para Desenvolvimento de Frameworks/Linhas de Produto**. Nossa abordagem endereça atualmente as etapas de projeto detalhado, implementação e instanciação de frameworks. Um de

nossos objetivos é refiná-la para endereçar mais sistematicamente o desenvolvimento de família de sistemas e linhas de produto (e não apenas frameworks), e contemplar as etapas de análise e especificação de requisitos, modelagem arquitetural e testes. Integração com métodos e abordagens existentes é outra atividade fundamental na definição de tal método. Uma das primeiras abordagens que se pretende integrar é a abordagem proposta por Jacobson et al [62];

- **Endereçamento Sistemático de Interações entre Aspectos**. A abordagem proposta nessa tese não apresenta diretrizes detalhadas para a resolução de interações entre aspectos, sobretudo no que refere a seu projeto e implementação. Diretrizes preliminares foram apresentadas (Seção 7.3.1), mas precisam ser refinadas baseado na experiência oriunda de novos estudos de caso;
- **Desenvolvimento de uma Abordagem de Teste Complementar**. Esse trabalho apresentou uma categorização de contratos os quais podem ser usados para garantir que o núcleo do framework e os aspectos de extensão respeitam determinadas restrições impostas. Uma abordagem de teste, complementar as diretrizes de projeto e implementação apresentadas na tese, está sendo desenvolvida [29, 30] a qual combina a especificação de contratos com a definição de estratégias de teste de unidade e integração;
- **Realização da Abordagem de EJPs em Tecnologias de Implementação**. Essa tese apresentou uma concretização da abordagem baseada em EJPs para a linguagem AspectJ. Pretende-se explorar a realização da abordagem no contexto de outras linguagens de desenvolvimento e modelos de componentes, tais como, Spring [65] e OSGi[26].

[26] OSGi Service Platform: The OSGi Alliance, URL: http://www.osgi.org/.

Referências

1. ALUR, D.; MALKS, D.; and CRUPI, J. **Core J2EE Patterns: Best Practices and Design Strategies**. Second Edition ed. 2003: Prentice Hall PTR.
2. ALVES, V. Implementing Software Product Line Adoption Strategies, Phd Thesis. March 2007, Centro de Informática, Universidade Federal de Pernambuco.
3. ALVES, V.; CARDIM, I.; VITAL, H.; SAMPAIO, P. H. M.; DAMASCENO, A. L. G.; BORBA, P.; and RAMALHO, G. Comparative Analysis of Porting Strategies in J2me Games. **21st IEEE International Conference on Software Maintenance (ICSM 2005)**. Budapest, Hungary, September 2005, IEEE Computer Society 2005, pp. 123-132.
4. ALVES, V.; GHEYI, R.; MASSONI, T.; KULESZA, U.; BORBA, P.; and LUCENA, C. Refactoring Product Lines. **Proceedings of the 5th international conference on Generative programming and component engineering**. Portland, Oregon, USA, 2006, ACM Press, pp. 201-210.
5. ALVES, V.; MATOS, P.; COLE, L.; BORBA, P.; and RAMALHO, G. Extracting and Evolving Mobile Games Product Lines. **Proceedings of Software Product Line Conference (SPLC´2005)**, 2005, Springer-Verlag, pp. 70-81.
6. ANTKIEWICZ, M. and CZARNECKI, K. Featureplugin: Feature Modeling Plug-in for Eclipse. **Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange**. Vancouver, British Columbia, Canada, 2004, ACM Press, pp. 67-72.
7. APEL, S. and BATORY, D. When to Use Features and Aspects?: A Case Study. **Proceedings of the 5th international conference on Generative programming and component engineering**. Portland, Oregon, USA, 2006, ACM Press, pp. 59-68.
8. APEL, S.; LEICH, T.; and SAAKE, G. Aspectual Mixin Layers: Aspects and Features in Concert. **Proceeding of the 28th international conference on Software engineering**. Shanghai, China, 2006, ACM Press, pp. 122-131.
9. BALDWIN, C. Y. and CLARK, K. B. **Design Rules: The Power of Modularity**. 2000, Cambridge, MA: MIT Press.
10. BANIASSAD, E. Discovering Early Aspects. **IEEE Software, Special Issue on Aspect-Oriented Programming**, 2006. 23(1): pp. 61-70.
11. BASSETT, P. G. **Framing Software Reuse: Lessons from the Real World**. 1996: Prentice Hall.
12. BATISTA, T.; CHAVEZ, C.; GARCIA, A.; KULESZA, U.;

SANT'ANNA, C.; and LUCENA, C. Aspectual Connectors: Supporting the Seamless Integration of Aspects and Adls. **XX Simpósio Brasileiro de Engenharia de Software (SBES'2006)**. Florianópolis, 2006,

13. BATISTA, T.; CHAVEZ, C.; GARCIA, A.; RASHID, A.; SANT'ANNA, C.; KULESZA, U.; and FILHO, F. C. Reflections on Architectural Connection: Seven Issues on Aspects and Adls. **Proceedings of the 2006 International Workshop on Early Aspects at ICSE**. Shanghai, China, 2006, ACM Press, pp. 3-10.
14. BATORY, D.; CARDONE, R.; and SMARAGDAKIS, Y. Object-Oriented Frameworks and Product-Lines. **1st Software Product-Line Conference (SPLC'2000)**. Denver, 1999, pp. 227-248.
15. BATORY, D.; SARVELA, J. N.; and RAUSCHMAYER, A. Scaling Step-Wise Refinement. **Proceedings of the 25th International Conference on Software Engineering**. Portland, Oregon, 2003, IEEE Computer Society, pp. 187-197.
16. BOOCH, G.; JACOBSON, I.; and RUMBAUGH, J. **Unified Modeling Language - User's Guide**. 1999: Addison-Wesley.
17. BOSCH, J. Design of an Object-Oriented Framework for Measurement Systems, **Domain-Specific Application Frameworks**. 1999, John Wiley. pp. 177-205.
18. BUDINSKY, F.; STEINBERG, D.; MERKS, E.; ELLERSICK, R.; and GROSE, T. **Eclipse Modeling Framework**. 2003: Addison-Wesley.
19. BUSCHMANN, F.; MEUNIER, R.; ROHNERT, H.; SOMMERLAD, P.; and STAL, M. **Pattern-Oriented Software Architecture, Volume 1: A System of Patterns**. 1996: Wiley.
20. CACHO, N.; SANT'ANNA, C.; FIGUEIREDO, E.; GARCIA, A.; BATISTA, T.; and LUCENA, C. Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming. **Proceedings of the 5th International Conference on Aspect-oriented Software Development**. Bonn, Germany, 2006, ACM Press, pp. 109-121.
21. CAMARGO, V. and MASIERO, P. Frameworks Orientado a Aspectos. **IX Simpósio Brasileiro de Engenharia de Software (SBES'2005)**. Uberlândia, 2005,
22. CAMARGO, V. V. D. Frameworks Transversais: Definições, Classificações, Arquitetura E Utilização Em Um Processo De Desenvolvimento De Software, Phd Thesis. September 2006, Instituto de Ciências Matemáticas e de Computação (ICMC), Universidade de São Paulo.
23. CHAVEZ, C.; GARCIA, A.; KULESZA, U.; SANT'ANNA, C.; and LUCENA, C. Crosscutting Interfaces for Aspect-Oriented Modeling. **Journal of the Brazilian Computer Society**, 2006. 11(3): pp. 43-58.
24. CHAVEZ, C. V. F. A Model-Driven Approach to Aspect-Oriented Design, Phd Thesis, **Computer Science Department**. 2004, PUC-Rio.
25. CHIDAMBER, S. and KEMERER, C. A Metrics Suite for Oo Design. **IEEE Transations on Software Engineering**, June 1994. 20(6): pp. 476-

493.

26. CIRILO, E.; KULESZA, U.; and LUCENA, C. Genarch – a Model-Based Product Derivation Tool. **Proceedings of Simpósio Brasileiro de Componentes, Arquitetura e Reutilização de Software (SBCARS´2007)**. Campinas, Brazil, August 2007, pp. 31-46.
27. CLEMENTS, P. and NORTHROP, L. **Software Product Lines: Practices and Patterns**. 2001: Addison-Wesley Professional.
28. CODENIE, W.; HONDT, K. D.; STEYAERT, P.; and VERCAMMEN, A. From Custom Applications to Domain-Specific Frameworks. **Commun. ACM**, 1997. 40(10): pp. 70-77.
29. COELHO, R.; ALVES, V.; KULESZA, U.; NETO, A. C.; GARCIA, A.; STAA, A. V.; LUCENA, C.; and BORBA, P. On Testing Crosscutting Features Using Extension Join Points. **International 3rd Workshop on Software Product Line Testing, 3rd (SPLiT 2006), in conjunction with 10th International Software Product Line Conference (SPLC 2006),**. Baltimore, Maryland, 2006, pp. 23-30.
30. COELHO, R. and STAA, A. V. Using Interfaces to Support the Testing of Crosscutting Features, **Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications**. 2006, ACM Press: Portland, Oregon, USA.
31. COLYER, A. **Eclipse Aspectj: Aspect-Oriented Programming with Aspectj and the Eclipse Aspectj Development Tools**. 2004: Addison-Wesley.
32. CZARNECKI, K. Overview of Generative Software Development. **Proceedings of the European Commission and US National Science Foundation Strategic Research - Workshop on Unconventional Programming Paradigms**. Mont Saint-Michel, France, 2004,
33. CZARNECKI, K. and EISENECKER, U. W. **Generative Programming: Methods, Tools, and Applications**. 2000: ACM Press/Addison-Wesley Publishing Co. 832.
34. CZARNECKI, K. and HELSEN, S. Feature-Based Survey of Model Transformation Approaches. **IBM Systems Journal**, 2006. 45(3): pp. 621-640.
35. CZARNECKI, K.; HELSEN, S.; and EISENECKER, U. Staged Configuration Using Feature Models. **3rd Software Product-Line Conference (SPLC'2004)**, September 2004,
36. DIAS, K. L. Um Framework Orientado a Aspectos Para Monitoramento E Análise De Processos De Negócio, Proposta De Dissertação De Mestrado, **Departamento de Informática**. March 2007, PUC-Rio.
37. FAYAD, M.; SCHMIDT, D.; and JOHNSON, R. **Building Application Frameworks: Object-Oriented Foundations of Framework Design**. 1999: John Wiley & Sons.
38. FAYAD, M. and SCHMIDT, D. C. Object-Oriented Application Frameworks. **Commun. ACM**, 1997. 40(10): pp. 32-38.

39. FENTON, N. and PFLEEGER, S. **Software Metrics: A Rigorous Practical Approach**. 1997: London: PWS.

40. FILHO, F. C.; CACHO, N.; FIGUEIREDO, E.; MARANHÃO, R.; GARCIA, A.; and RUBIRA, C. M. F. Exceptions and Aspects: The Devil Is in the Details, **Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering**. 2006, ACM Press: Portland, Oregon, USA.

41. FILHO, I. M.; OLIVEIRA, T. C. D.; and LUCENA, C. J. P. D. A Framework Instantiation Approach Based on the Features Model. **Journal of Systems and Software**, 2004. 73(2): pp. 333-349

42. FILMAN, R.; ELRAD, T.; CLARKE, S.; and AKSIT, M. **Aspect-Oriented Software Development**. 2005: Addison-Wesley.

43. FILMAN, R. and FRIEDMAN, D. Aspect-Oriented Programming Is Quantification and Obliviousness, **Aspect-Oriented Software Development**. 2005, Addison-Wesley. pp. 21-35.

44. FOWLER, A. **A Swing Architecture Overview, Sun Developer Network, Url: [http://java.sun.com/products/jfc/tsc/articles/architecture/].** December 2005.

45. GAMMA, E.; HELM, R.; JOHNSON, R.; and VLISSIDES, J. **Design Patterns: Elements of Reusable Object-Oriented Software**. 1995: Addison-Wesley Longman Publishing Co., Inc. 395.

46. GARCIA, A. From Objects to Agents: An Aspect-Oriented Approach, Phd Thesis, **Computer Science Department**. 2004, PUC-Rio.

47. GARCIA, A.; CHAVEZ, C.; BATISTA, T. V.; SANT'ANNA, C.; KULESZA, U.; RASHID, A.; and LUCENA:, C. J. P. D. On the Modular Representation of Architectural Aspects. **Third European Workshop on Software Architecture, EWSA 2006**. Nantes, France, September, 2006, Springer, pp. 82-97.

48. GARCIA, A.; KULESZA, U.; SANT'ANNA, C.; and LUCENA., C. The Mobility Aspect Pattern. **Proceedings of the Fourth Latin American Conference on Pattern Languages of Programming (SugarLoafPLoP'04)**. Fortaleza, Brazil, August 2004,

49. GARCIA, A.; KULESZA, U.; SARDINHA, J.; MILIDIÚ, R.; and LUCENA., C. The Learning Aspect Pattern. **The 11th Conference on Pattern Languages of Programs (PLoP2004)**. Monticello, Illinios, USA, September 2004,

50. GARCIA, A.; LUCENA, C.; and COWAN, D. Agents in Object-Oriented Software Engineering. **Software: Practice and Experience**, May 2004. 34(5): pp. 1-33.

51. GARCIA, A.; SANT'ANNA, C.; FIGUEIREDO, E.; KULESZA, U.; LUCENA, C.; and STAA, A. V. Modularizing Design Patterns with Aspects: A Quantitative Study. **Proceedings of the 4th international conference on Aspect-oriented software development**. Chicago, Illinois, 2005, ACM Press, pp. 3-14.

52. GODIL, I. and JACOBSEN, H.-A. Horizontal Decomposition of Prevayler. **Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative Research**. Toronto, Ontario, Canada, October 2005, IBM 2005, pp. 83-100.

53. GREENFIELD, J. and SHORT, K. **Software Factories: Assembling Applications with Patterns, Frameworks, Models and Tools**. 2005: John Wiley and Sons.

54. GREENFIELD, J.; SHORT, K.; COOK, S.; and KENT, S. **Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools**. 2004: Wiley.

55. GREENWOOD, P.; BARTOLOMEI, T.; FIGUEIREDO, E.; DOSEA, M.; GARCIA, A.; CACHO, N.; SANT'ANNA, C.; SOARES, S.; BORBA, P.; KULESZA, U.; and RASHID, A. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. **European Conference of Object-Oriented Programming (ECOOP'07)**, 2007, Springer-Verlag, pp. 176-200.

56. GRISWOLD, W.; SULLIVAN, K. J.; SONG, Y.; SHONLE, M.; TEWARI, N.; CAI, Y.; and RAJAN, H. Modular Software Design with Crosscutting Interfaces. **IEEE Software, Special Issue on Aspect-Oriented Programming**, 2006. 23(1): pp. 51-60.

57. HANENBERG, S.; SCHMIDMEIER, A.; and UNLAND, R. Aspectj Idioms for Aspect-Oriented Software Construction. **Proceedinges of 8th European Conference on Pattern Languages of Programs (EuroPLoP'2003)**, 2003,

58. HANNEMANN, J. and KICZALES, G. Design Pattern Implementation in Java and Aspectj. **Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications**. Seattle, Washington, USA, 2002, ACM Press, pp. 161-173.

59. JACKSON, M. and ZAVE, P. Distributed Feature Composition: A Virtual Architecture for Telecommunications Services. **IEEE Transactions on Software Engineering**, October 1998.

60. JACOBSON, I. Use Cases and Aspects – Working Seamlessly Together. **Journal of Object Technology**, August 2003. 2(4): pp. 7-28.

61. JACOBSON, I.; CHRISTERSON, M.; JONSSON, P.; and OVERGAARD, G. **Object-Oriented Software Engineering: A Use Case Driven Approach**. 1992: Addison-Wesley.

62. JACOBSON, I. and NG, P.-W. **Aspect-Oriented Software Development with Use Cases.** Object Technology Series. 2004: The Addison-Wesley

63. JARKE, M. Requirements Tracing. **Commun. ACM**, 1998. 41(12): pp. 32-36.

64. JOHNSON, R. and FOOTE, B. Designing Reusable Classes. **Journal of Object-Oriented Programming**, 1988. 1: pp. 22-35.

65. JOHNSON, R.; HOELLER, J.; ARENDSEN, A.; RISBERG, T.; and SAMPALEANU, C. **Professional Java Development with the Spring**

Framework. 2005: Wrox.

66. KERSTEN, M. Aop Tools Comparison: Development Environments, **IBM Developers Work**. 2005.

67. KERSTEN, M. and MURPHY, G. C. Atlas: A Case Study in Building a Web-Based Learning Environment Using Aspect-Oriented Programming. **Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications**. Denver, Colorado, United States, 1999, ACM Press, pp. 340-352.

68. KICZALES, G. Aspect-Oriented Programming. **European Conference of Object-Oriented Programming (ECOOP'97)**, 1997, Springer-Verlag, pp. 220-242.

69. KICZALES, G.; HILSDALE, E.; HUGUNIN, J.; KERSTEN, M.; PALM, J.; and GRISWOLD, W. Getting Started with Aspectj. **Commun. ACM**, 2001. 44(10): pp. 59-65.

70. KICZALES, G.; HILSDALE, E.; HUGUNIN, J.; KERSTEN, M.; PALM, J.; and GRISWOLD, W. An Overview of Aspectj. **Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)**, 2001,

71. KRUEGER, C. Easing the Transition to Software Mass Customization. **4th International Workshop on Software Product-Family Engineering (PFE'2001)**, 2001, pp. 282-293.

72. KULESZA, R.; KULESZA, U.; and BRESSAN, G. Implementing an Adaptation Layer for Multimedia Servers Using Aspect-Oriented Programming. **Proceedings of the 12th Brazilian Symposium on Multimedia and the Web (WebMedia 2006)**. Natal, Rio Grande do Norte, Brazil, November 2006, ACM 2006, pp. 293-302.

73. KULESZA, U.; ALVES, V.; GARCIA, A.; LUCENA, C. J. P. D.; and BORBA, P. Improving Extensibility of Object-Oriented Frameworks with Aspect-Oriented Programming, **Proceedings of 9th International Conference on Software Reuse, Icsr 2006 Turin, Italy, June 12-15, 2006. Lecture Notes in Computer Science: Reuse of Off-the-Shelf Components**. 2006, Springer-Verlag. pp. 231-245.

74. KULESZA, U.; ALVES, V.; GARCIA, A.; NETO, A. C.; CIRILO, E.; LUCENA, C.; and BORBA, P. Mapping Features to Aspects: A Model-Based Generative Approach. **Early Aspects 2007 Workshop, AOSD'2007**. Vancouver, Canada, 2007, Springer-Verlag, pp. 155-174.

75. KULESZA, U.; COELHO, R.; ALVES, V.; NETO, A. C.; GARCIA, A.; LUCENA, C.; STAA, A. V.; and BORBA, P. Implementing Framework Crosscutting Extensions with Xpis and Aspectj. **Proceedings of XX Simpósio Brasileiro de Engenharia de Software (SBES'2006)**. Florianópolis, 2006, pp. 177-192.

76. KULESZA, U.; GARCIA, A.; BLEASBY, F.; and LUCENA, C. Instantiating and Customizing Product Line Architectures Using Aspects and Crosscutting Feature Models. **Workshop on Early Aspects, OOPSLA'2005**. San Diego, EUA, 2005,

77. KULESZA, U.; GARCIA, A.; and LUCENA, C. An Aspect-Oriented Generative Approach. **Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications**. Vancouver, BC, CANADA, 2004, ACM Press, pp. 166-167.

78. KULESZA, U.; GARCIA, A.; and LUCENA, C. Composing Object-Oriented Frameworks with Aspect-Oriented Programming, **Monografias em Informática**. 2006, Computer Science Department, PUC-Rio: Brazil.

79. KULESZA, U.; GARCIA, A.; LUCENA, C.; and ALENCAR, P. A Generative Approach for Multi-Agent System Development, **Software Engineering for Multi-Agent Systems Iii**. 2004, Springer-Verlag. pp. 52-69.

80. KULESZA, U.; GARCIA, A.; LUCENA, C.; and STAA, A. V. Integrating Generative and Aspect-Oriented Technologies. **Proceedings of XVIII Simpósio Brasileiro de Engenharia de Software (SBES'2004)**. Brasília, Brazil, October 2004, pp. 130-146.

81. KULESZA, U.; LUCENA, C.; ALENCAR, P.; and GARCIA, A. Customizing Aspect-Oriented Variabilites Using Generative Techniques. **Proceedings of International Conference on Software Engineering and Knowledge Engineering (SEKE'06)**. San Francisco, July 2006, pp. 17-22.

82. KULESZA, U. and LUCENA, C. J. P. An Aspect-Oriented Approach to Framework Development. **Proceedings of 4th Software Product Line Doctoral Symposium, In conjunction with the 10th Software Product Lines International Conference - SPLC, Technical Report.** Baltimore, MD, USA August, 2006, Fraunhover IESE, pp. 67-79.

83. KULESZA, U.; SANT'ANNA, C.; GARCIA, A.; COELHO, R.; STAA, A. V.; and LUCENA, C. J. P. D. Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study. **Proceedings of 22nd IEEE International Conference on Software Maintenance (ICSM 2006)**. Philadelphia, Pennsylvania, USA, September 2006, IEEE Computer Society pp. 223-233.

84. KULESZA, U.; SANT'ANNA, C.; and LUCENA, C. Refactoring the Junit Framework Using Aspect-Oriented Programming. **Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications**. San Diego, CA, USA, 2005, ACM Press, pp. 136-137.

85. LADDAD, R. **Aspectj in Action: Practical Aspect-Oriented Programming**. 2003: Manning Publications.

86. LAVENDER, R. and SCHMIDT, D. Active Object: An Object Behavioral Pattern for Concurrent Programming, **Pattern Languages of Program Design**. 1996, Addison-Wesley.

87. LOPES, C. D: A Language Framework for Distributed Programming, Phd Thesis, 1997, Northeastern University.

88. LOPEZ-HERREJON, R. E.; BATORY, D.; and COOK, W. Evaluating Support for Features in Advanced Modularization Technologies.

Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP'05), 2005, Springer-Verlag, pp. 169-194.

89. LOUGRAN, N. and RASHID, A. Framed Aspects: Supporting Variability and Configurability for Aop, **Proceedings of 8th International Conference on Software Reuse, Icsr 2004**. 2004, Springer-Verlag. pp. 127-140.

90. MATTSSON, M. and BOSCH, J. Framework Composition: Problems, Causes, and Solutions, **Building Application Frameworks: Object-Oriented Foundations of Framework Design**. 1999, John Wiley & Sons. pp. 467-487.

91. MATTSSON, M.; BOSCH, J.; and FAYAD, M. E. Framework Integration Problems, Causes, Solutions. **Commun. ACM**, 1999. 42(10): pp. 80-87.

92. MEYER, B. **Object-Oriented Software Construction**. 2nd Edition ed. 2000: Prentice Hall PTR.

93. MEZINI, M. and OSTERMANN, K. Conquering Aspects with Caesar. **Proceedings of the 2nd International Conference on Aspect-oriented Software Development**. Boston, Massachusetts, 2003, ACM Press, pp. 90-99.

94. MEZINI, M. and OSTERMANN, K. Variability Management with Feature-Oriented Programming and Aspects. **Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering**. Newport Beach, CA, USA, 2004, ACM Press, pp. 127-136.

95. MITCHELL, T. **Machine Learning**. 1997: McGraw Hill, New York.

96. MONTEIRO, M. P. and FERNANDES, J. M. Towards a Catalog of Aspect-Oriented Refactorings. **Proceedings of the 4th International Conference on Aspect-oriented Software Development**. Chicago, Illinois, 2005, ACM Press, pp. 111-122.

97. OLIVEIRA, T. C. D.; ALENCAR, P. S. C.; FILHO, I. M.; LUCENA, C. J. P. D.; and COWAN, D. D. Software Process Representation and Analysis for Framework Instantiation. **IEEE Trans. Software Eng.** , 2004. 30(3): pp. 145-159.

98. PARNAS, D. L. On the Design and Development of Program Families. **IEEE Transactions on Software Engineering (TSE)**, 1976. 2(1): pp. 1-9.

99. PENCZEK, L. and OLIVEIRA, T. Rdl+Aspects: Uma Linguagem De Processo Para Sistematizar O Reúso De Frameworks Orientados a Aspectos. **III Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos (WASP´2006)**. Florianópolis, 2006,

100. POHL, K.; BÖCKLE, G.; and LINDEN, F. J. V. D. **Software Product Line Engineering: Foundations, Principles and Techniques**. 2005: Springer.

101. PRIETO-DIAZ, R. and ARANGO, G. **Domain Analysis and Software Systems Modeling**. 1991: IEEE Computer Society Press.

102. RASHID, A. and CHITCHYAN, R. Persistence as an Aspect.

Proceedings of the 2nd International Conference on Aspect-oriented Software Development. Boston, Massachusetts, 2003, ACM Press, pp. 120-129.

103. RASHID, A.; MOREIRA, A.; and ARAÚJO, J. Modularisation and Composition of Aspectual Requirements, **Proceedings of the 2nd International Conference on Aspect-oriented Software Development**. 2003, ACM Press: Boston, Massachusetts.

104. RIEHLE, D. and GROSS, T. Role Model Based Framework Design and Integration. **Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications**. Vancouver, British Columbia, Canada, 1998, ACM Press, pp. 117-133.

105. RUSSELL, S. and NORVIG, P. **Artificial Intelligence: A Modern Approach**. 2nd Edition ed. 2002: Prentice Hall.

106. SANEN, F. Classifying and Documenting Aspect Interactions. **5th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS) at AOSD 2006**. Bonn, Germany, 2006,

107. SANT'ANNA, C.; GARCIA, A.; CHAVEZ, C.; LUCENA, C.; and STAA, A. V. On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. **Proceedings of the XVII Brazilian Symposium on Software Engineering**. Manaus, Brazil, October 2003,

108. SANTOS, A. L.; LOPES, A.; and KOSKIMIES, K. Framework Specialization Aspects. **Proceedings of the 6th International Conference on Aspect-oriented Software Development**. Vancouver, British Columbia, Canada, 2007, ACM Press, pp. 14-24.

109. SHAVOR, S.; D'ANJOU, J.; FAIRBROTHER, S.; KEHN, D.; KELLERMAN, J.; and MCCARTHY, P. **The Java Developer's Guide to Eclipse**. 2003: Addison-Wesley Professional.

110. SHONLE, M.; LIEBERHERR, K.; and SHAH, A. Xaspects: An Extensible System for Domain-Specific Aspect Languages. **Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications**. Anaheim, CA, USA, 2003, ACM Press, pp. 28-37.

111. SMARAGDAKIS, Y. and BATORY, D. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. **ACM Trans. Softw. Eng. Methodol.**, 2002. 11(2): pp. 215-255.

112. SOARES, S. An Aspect-Oriented Implementation Method, Phd Thesis, **Computer Science Department**. 2004, Federal University of Pernambuco.

113. SOARES, S.; LAUREANO, E.; and BORBA, P. Implementing Distribution and Persistence Aspects with Aspectj. **Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications**. Seattle, Washington, USA, 2002, ACM Press, pp. 174-190.

114. STAHL, T. and VOELTER, M. **Model-Driven Software Development: Technology, Engineering, Management**. 2006: Wiley.

115. SULLIVAN, K.; GRISWOLD, W. G.; SONG, Y.; CAI, Y.; SHONLE, M.; TEWARI, N.; and RAJAN, H. Information Hiding Interfaces for Aspect-Oriented Design. **Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering**. Lisbon, Portugal, 2005, ACM Press, pp. 166-175.

116. UBAYASHI, N. and TAMAI, T. Separation of Concerns in Mobile Agent Applications. **Proceedings of the 3rd International Conference Reflection 2001**. Kyoto, Japan, September 2001, Springer, pp. 89-109.

117. WEISS, D. and LAI, C. **Software Product-Line Engineering: A Family-Based Software Development Process**. 1999: Addison-Wesley Professional.

118. ZHANG, C. and JACOBSEN, H.-A. Resolving Feature Convolution in Middleware Systems. **Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications**. Vancouver, BC, Canada, 2004, ACM Press, pp. 188-205.

Apêndice I
Tradução de Termos

Abaixo é apresentada uma tabela contendo a tradução do inglês para o português de termos que foram utilizados nessa tese.

Termo em Inglês	Tradução para o Português
Advice	Adendo
Aspect	Aspecto
Concern	Interesse
Crosscut	Entrecortar
Crosscutting	Transversal
Crosscutting concern	Interesse transversal
Extension Join Point (EJP)	Ponto de Junção de Extensão
Feature	Característica
Generative	Generative(o)
Inter-type declaration	Declaração intertipo
Join point	Ponto de junção
Obliviousness	Inconsciência
Pointcut	Ponto de corte
Scattering	Espalhamento
Tangling	Entrelaçamento

Apêndice II
Estudo Quantitativo

Esse apêndice apresenta os valores coletados para as versões OO e OA do estudo de composição dos frameworks *Measurement*, GUI, de estatística e persistência.

Classes	CBC	DIT	LOC	LCOO	NOA	WOC
AbstractSensor	0	3	5	-	0	1
ActiveObject	2	3	19	1	1	3
ActuationStrategy	0	1	4	-	0	2
Actuator	1	1	13	0	1	4
ActuatorTableModel	2	4	29	-	0	10
BCActuationStrategy	7	2	32	-	0	2
BCClock	0	1	5	-	0	1
BCMICalculationStrategy	5	2	21	-	0	2
BeerPictureTableModel	1	4	29	-	0	10
BeerProcessingTimeAlgorithm	5	1	36	-	0	2
BeerProcessingTimeStatistic	0	2	15	-	0	3
BeerQualityAlgorithm	2	1	9	-	0	2
BeerQualityStatistic	0	2	15	-	0	3
BlockingQueue	1	1	15	0	1	4
CalculationStrategy	0	1	4	-	0	2
CalibrationStrategy	0	2	5	-	0	2
ClienteUpdateStrategy	0	2	7	-	0	2
ConcreteCalculationStrategy	0	2	7	-	0	2
ConcretePersistence	2	2	19	3	1	3
ConcreteSensor	1	3	6	-	0	1
ConcurrencyStrategy	0	1	4	-	0	2
ConcurrencyStrategyException	0	4	9	-	0	3
DataTableModel	0	3	35	37	2	19
GUIAbstractTableModel	3	3	53	79	2	31
GUIFramework	10	7	97	8	16	6
HWBCActuator	1	2	7	-	0	1
HWBCCamera	6	2	37	1	2	3
HWBCMeasurementGUI	7	2	41	15	1	9
HWBCTrigger	4	2	29	3	2	8
ItemFactory	2	1	38	4	3	10
ItemProcessingTime	1	1	27	3	2	8
MICalculationStrategy	2	2	11	-	0	2
Main	13	1	57	-	0	2
MeasurementApplication	6	8	33	3	1	4
MeasurementEntity	1	1	10	0	1	3
MeasurementGUIFacade	8	1	81	31	2	18

MeasurementGUIStatisticFacade	11	1	86	43	3	14
MeasurementItem	6	2	52	18	4	17
MeasurementValue	1	2	34	10	3	12
ObjectDAO	1	1	9	-	0	8
OnChangeUpdateStrategy	1	2	10	-	0	2
PeriodicUpdateStrategy	0	2	7	-	0	2
PersistenceFacade	4	1	23	0	1	7
PersistencyFrameworkFacade	3	1	52	24	2	19
PhysicalActuator	0	1	4	-	0	1
PhysicalSensor	1	1	12	4	1	5
Picture	1	1	23	0	1	7
ProcessedItemDAO	2	2	20	-	0	8
ProcessingTimeStatisticDAO	2	2	20	-	0	8
ProcessingTimeTableModel	0	4	17	-	0	4
Sensor	0	1	23	3	2	10
Sensor	2	2	26	11	3	10
SensorTableModel	1	4	31	-	0	10
Statistic	2	1	20	9	1	9
StatisticAlgorithm	2	1	5	-	0	2
StatisticFrameworkFacade	9	1	110	21	6	23
ThreadId	0	1	3	-	0	0
ThreadPerRequestStrategy	0	1	8	-	0	3
ThreadPoolStrategy	2	3	41	0	5	6
Trigger	2	4	15	0	1	4
TriggerTableModel	2	4	29	-	0	10
UpdateStrategy	0	1	4	-	0	2
Total	148	128	1548	331	71	393

Tabela 22. Valores Coletados para Métricas – Versão OO

Classes	CBC	DIT	LOC	LCOO	NOA	WOC
AbstractSensor	0	3	5	-	0	1
ActiveObject	2	3	19	1	1	3
ActuationStrategy	0	1	4	-	0	2
Actuator	1	1	13	0	1	4
ActuatorTableModel	2	4	29	-	0	10
BCActuationStrategy	6	2	30	-	0	2
BCClock	0	1	5	-	0	1
BCMICalculationStrategy	5	2	21	-	0	2
BeerPictureTableModel	1	4	29	-	0	10
BeerProcessingTimeAlgorithm	5	1	36	-	0	2
BeerProcessingTimeStatistic	0	2	15	-	0	3
BeerQualityAlgorithm	2	1	9	-	0	2
BeerQualityStatistic	0	2	15	-	0	3
BlockingQueue	1	1	15	0	1	4
CalculationStrategy	0	1	4	-	0	2
CalibrationStrategy	0	2	5	-	0	2
ClienteUpdateStrategy	0	2	7	-	0	2
ConcreteCalculationStrategy	0	2	7	-	0	2
ConcreteSensor	1	3	6	-	0	1
ConcurrencyStrategy	0	1	4	-	0	2
ConcurrencyStrategyException	0	4	9	-	0	3
DataTableModel	0	3	35	37	2	19
GUIAbstractTableModel	1	3	51	79	2	31
GUIFramework	10	7	97	8	16	6
HWBCActuator	1	2	7	-	0	1
HWBCCamera	5	2	35	1	2	3
HWBCTrigger	3	2	27	3	2	8
ItemFactory	2	1	38	4	3	10
ItemProcessingTime	1	1	27	3	2	8
MICalculationStrategy	2	2	11	-	0	2
Main	11	1	53	-	0	2
MeasurementApplication	4	8	29	3	1	4
MeasurementEntity	1	1	10	0	1	3
MeasurementItem	6	2	52	18	4	17
MeasurementValue	1	2	34	10	3	12
ObjectDAO	1	1	9	-	0	8
OnChangeUpdateStrategy	1	2	10	-	0	2
PeriodicUpdateStrategy	0	2	7	-	0	2
PersistencyFrameworkFacade	3	1	52	24	2	19
PhysicalActuator	0	1	4	-	0	1
PhysicalSensor	1	1	12	4	1	5
Picture	1	1	23	0	1	7
ProcessedItemDAO	2	2	20	-	0	8
ProcessingTimeStatisticDAO	2	2	20	-	0	8
ProcessingTimeTableModel	0	4	17	-	0	4
Sensor	0	1	23	3	2	10
Sensor	2	2	26	11	3	10
SensorTableModel	1	4	31	-	0	10
Statistic	2	1	20	9	1	9
StatisticAlgorithm	2	1	5	-	0	2

StatisticFrameworkFacade	7	1	108	21	6	23
ThreadId	0	1	3	-	0	0
ThreadPerRequestStrategy	0	1	8	-	0	3
ThreadPoolStrategy	2	3	41	0	5	6
Trigger	1	4	13	0	1	4
TriggerTableModel	2	4	29	-	0	10
UpdateStrategy	0	1	4	-	0	2
Aspectos	**CBC**	**DIT**	**LOC**	**LCOO**	**NOA**	**WOC**
BeerCanMeasurementGUIAspect	9	2	36	6	0	7
MeasurementGUIAspect	14	1	87	31	2	19
MeasurementGUIStatisticAspect	16	1	82	26	0	12
GUIEvents	4	1	15	-	0	0
MeasurementEvents	4	1	16	-	0	0
PersistenceAspect	6	1	30	0	1	7
BeerCanPersistenceAspect	4	2	12	-	0	1
StatisticEvents	2	1	7	-	0	0
Total	163	131	1563	302	66	388

Tabela 23. Valores Coletados para Métricas – Versão OA

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo