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%.

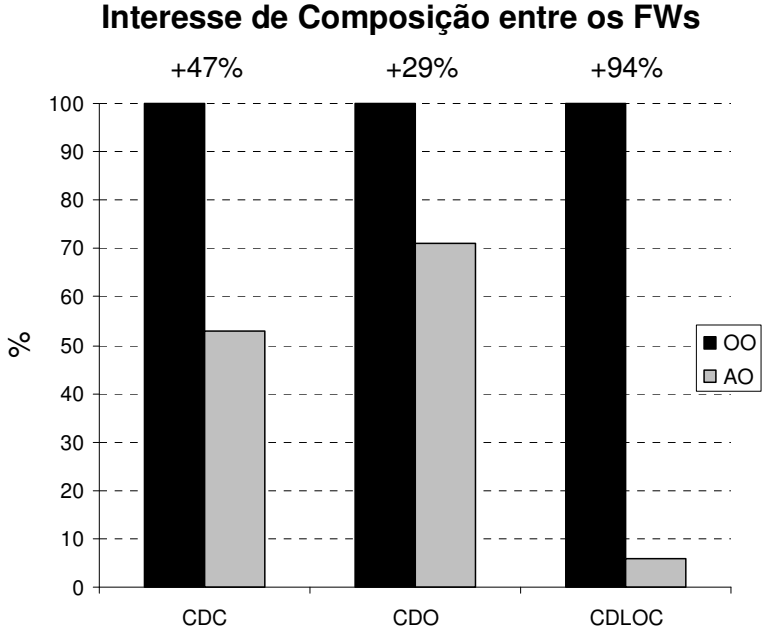

Figura 63. Métricas de Separação de Interesses

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).

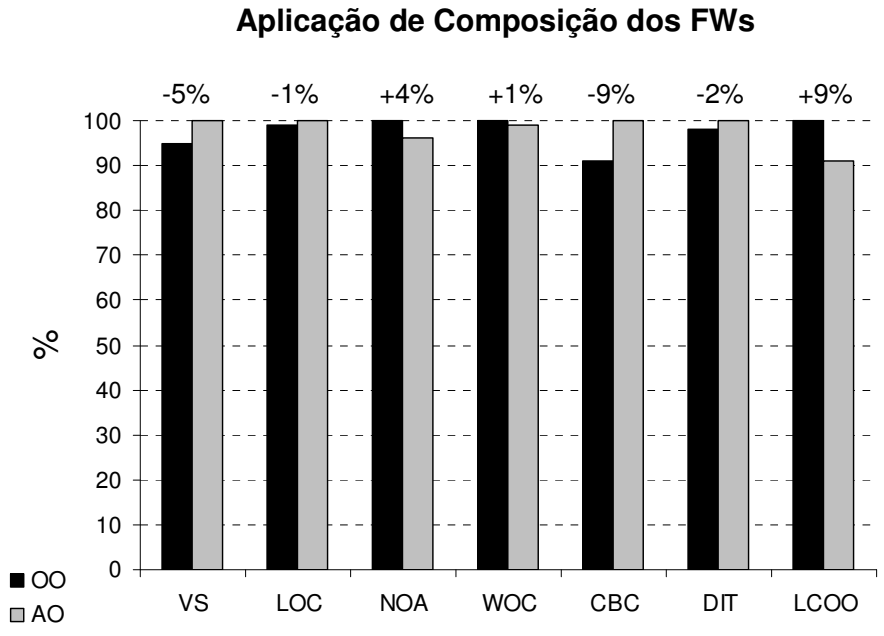

Figura 64. Métricas de Tamanho, Acoplamento e Coesão

[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.