6
Os Estudos Experimentais

O framework de avaliação proposto nesta dissertação foi usado no contexto de dois estudos experimentais de domínios distintos, com características, níveis de controle e níveis de complexidade diferentes. O primeiro estudo [42, 43] comparou uma abordagem orientada a objetos (baseada no uso de padrões de projeto) com uma abordagem orientada a aspectos para a realização do projeto e implementação de um sistema multi-agentes. O segundo estudo envolveu a aplicação do framework proposto para avaliar as implementações em Java e AspectJ dos padrões de projetos da Gangue dos Quatro (*GoF*) [44] propostas por Hannemann & Kiczales [5].

6.1.
O Primeiro Estudo Experimental

Esta seção descreve o contexto e os resultados do primeiro estudo experimental. O objetivo desse estudo foi o de servir como uma primeira avaliação do framework proposto. Esse estudo foi organizado para demonstrar a utilidade do conjunto de métricas e do modelo de qualidade no sentido de avaliar a manutenibilidade e a reusabilidade de sistemas de software. Foram coletados dados sobre o desenvolvimento de duas versões de um sistema multi-agentes, chamado de Portalware: uma versão orientada a aspectos (OA) e uma versão orientada a objetos (OO). A seleção desse estudo e a escolha do domínio de sistemas multi-agentes foram baseadas no fato de não ser óbvio quais entidades do problema deveriam ser projetadas como classes e quais deveriam ser projetadas com aspectos. Além disso, esse estudo de caso foi escolhido também por outras razões: (i) envolve tanto *concerns* específicos do domínio quanto *concerns* dependentes de aplicação, (ii) não é focado somente em *crosscutting concerns* tradicionais e triviais (como rastreamento e auditoria) e (iii) trata de *concerns* que não têm sido investigados na comunidade de DSOA.

6.1.1.
O Sistema Multi-Agentes

Esse estudo experimental foi derivado de um estudo de caso realizado pelo grupo SoCAgents/TecComm da PUC-Rio para o desenvolvimento do sistema multi-agentes Portalware. O sistema Portalware é um ambiente que apóia o desenvolvimento e gerenciamento de portais da Internet. Notações de UML [45] e a linguagem Java [17] foram usadas, respectivamente, para gerar o projeto e a implementação orientados a objetos. Uma extensão de UML para projeto orientado a aspectos [46] e a linguagem de programação AspectJ [16] foram usadas para gerar o projeto e a implementação orientada a aspectos.

Os *concerns* existentes no sistema Portalware são *concerns* típicos do domínio de sistemas multi-agentes reativos do mundo real. Esse sistema multi-agentes compreende vários *concerns* de agência, incluindo tipos de agentes, papéis, colaboração, interação, adaptação, autonomia e outros. O ambiente inclui alguns tipos de agentes para controlar portais e para coordenar e automatizar atividades repetitivas e que consomem muito tempo dos grupos de desenvolvimento de portais. O Portalware tem quatro tipos de agentes: (i) agentes mediadores, (ii) agentes de interface, (iii) agentes de usuário e (iv) agentes de informação. Os tipos de agentes implementam os *concerns* da parte fundamental do agente e mais alguns outros *concerns*. Nesse sistema, a parte fundamental é formada por três propriedades de agência: interação, autonomia e adaptação. Cada instância de agente tem diferentes propriedades e desempenha papéis distintos. A Figura 14 mostra os tipos de agentes existentes no Portalware e as propriedades e papéis associados a eles.

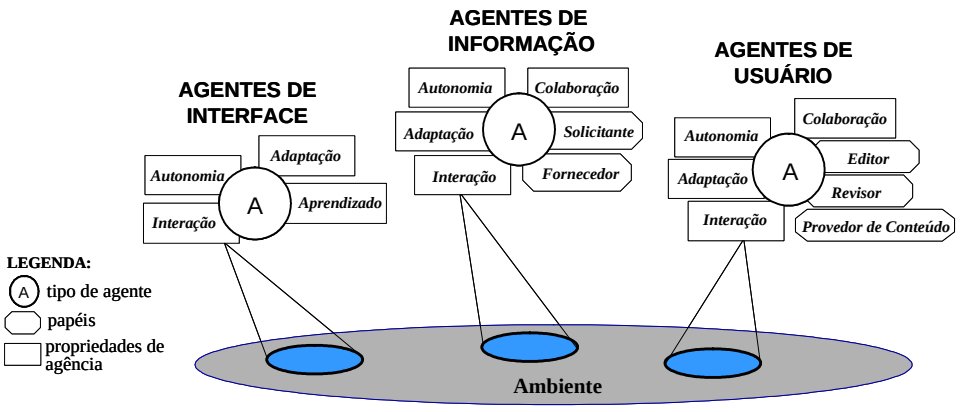

Figura 14 – Agentes do Portalware

Agentes mediadores fazem a mediação das conversações entre todos os agentes do sistema, provendo alguns serviços, tais como serviços de nome. Os agentes de interface monitoram a interface gráfica para melhorar a interação com os usuários. Eles aprendem a partir das preferências ou de instruções explícitas do usuário. Agentes de usuário representam os usuários do Portalware e existem para reduzir a necessidade de conversa direta entre eles. Uma vez que editores, revisores e provedores de conteúdo precisam se comunicar entre si para manter o portal, os agentes de usuários incorporam esses papéis e capacidades para automatizar e apoiar a colaboração em diferentes contextos. O agente, ao desempenhar o papel de editor, tem a responsabilidade de contactar os revisores e provedores de conteúdo e negociar com eles para usar seus serviços.

Os usuários do Portalware freqüentemente necessitam pesquisar informações que estão armazenadas em bancos de dados. Cada agente de informação controla um banco de dados e contém planos para a pesquisa de informação. Um plano alternativo de colaboração é usado quando um agente de informação não é capaz de encontrar a informação desejada em seu banco de dados. Durante a execução desse plano, o agente de informação desempenha o papel de solicitante para chamar os outros agentes de informação e requisitar a informação desejada. De forma similar, os agentes de informação chamados desempenham o papel de fornecedor para poderem receber a requisição e enviar o resultado da pesquisa.

6.1.2. O Formato do Primeiro Estudo Experimental

Os participantes desse estudo desenvolveram duas versões do sistema Portalware: uma versão baseada no desenvolvimento de software orientado a objetos e outra baseada no desenvolvimento de software orientado a aspectos (Capítulo 2). Uma vez que essas duas abordagens não foram concebidas com os *concerns* de sistemas multi-agentes em mente, dois métodos de apoio [4], especialmente adaptados para o desenvolvimento de sistemas multi-agentes, foram usados. Cada método está associado com uma das abordagens usadas e foi utilizado pelos participantes do estudo para aplicar a respectiva abordagem e suas abstrações. A Figura 15 apresenta um subconjunto do projeto orientado a objetos

do sistema Portalware. Ela mostra a combinação de diferentes padrões de projeto para tratar dos *concerns* de sistemas multi-agentes. Cada padrão está rodeado por uma linha pontilhada. A Figura 16 mostra um subconjunto do projeto orientado a aspectos. Nessa figura, o desenho de um losango é usado para representar os aspectos. Cada losango se relaciona com um ou mais retângulos, usados para representar classes. Esse relacionamento é expresso por uma linha do aspecto para a classe.

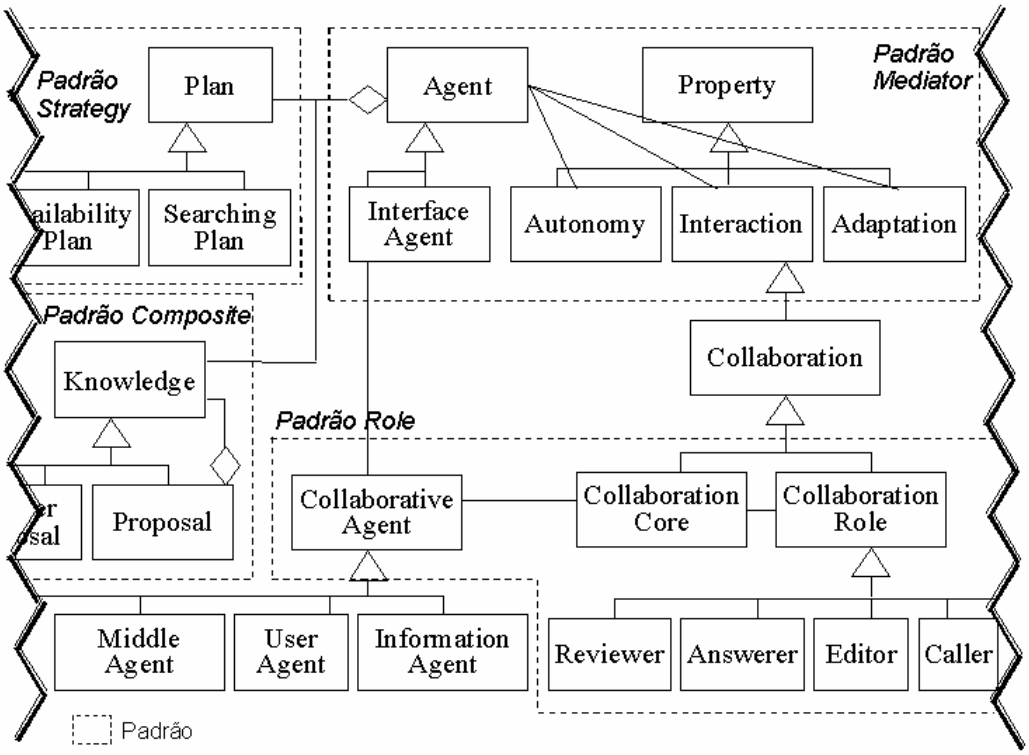

Figura 15 – Subconjunto do Projeto Orientado a Objetos

Quatro pessoas participaram desse estudo. Três eram estudantes de doutorado e uma estudante de mestrado da PUC-Rio. Todas elas participaram do desenvolvimento das duas versões do sistema, ou seja, participaram tanto do desenvolvimento do sistema orientado a aspectos como do sistema orientado a objetos. Todos eles tinham experiência em análise, projeto e implementação de software orientado a objetos. Os estudantes de doutorado também já haviam implementado grandes sistemas em Java (> 10 mil linhas de código). Entre eles, dois tinham experiência significativa em programação orientada a aspectos. O desenvolvimento das duas versões do sistema foi baseado nas mesmas

especificações de requisitos e procurou satisfazer o mesmo conjunto de funcionalidades.

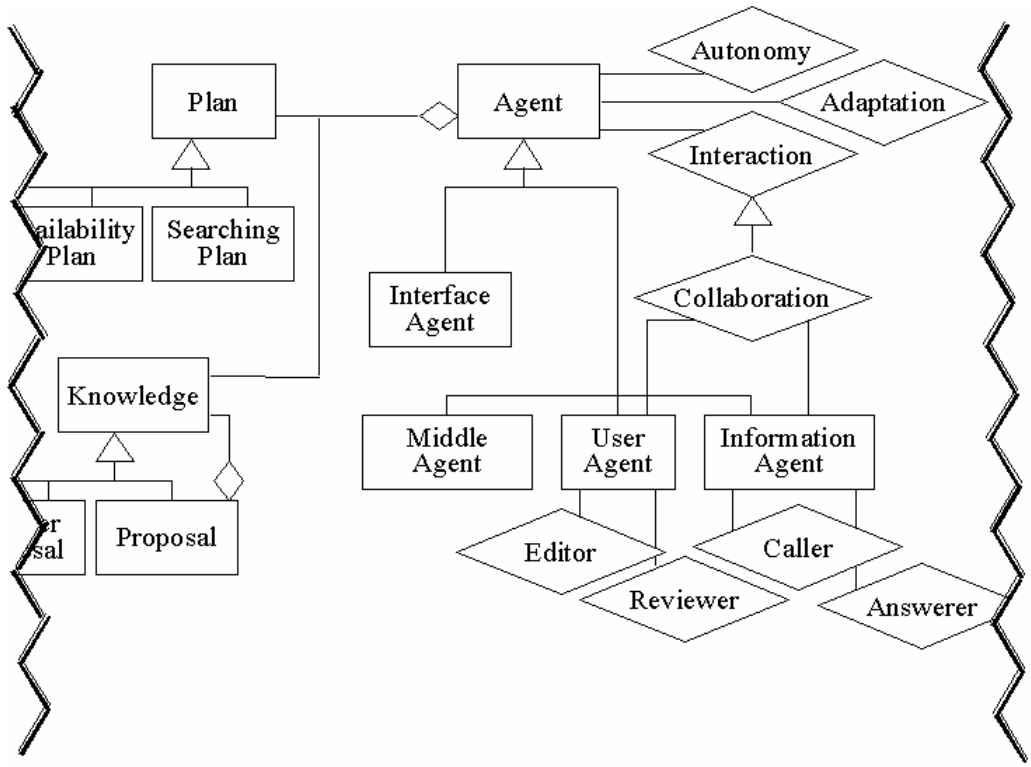

Figura 16 – Subconjunto do Projeto Orientado a Aspectos

O estudo foi dividido em duas fases: (1) a fase de construção e (2) a fase de evolução e reutilização. Na fase de construção, as duas versões do sistema foram desenvolvidas. Primeiramente, foi desenvolvida a versão orientada a aspectos e em seguida a versão orientada a objetos. Ainda na fase de construção, o conjunto de métricas do framework de avaliação foi aplicado sobre o projeto e código gerados em cada uma das versões. Antes que as métricas fossem aplicadas e os dados colhidos, foi realizada uma atividade de padronização do projeto e código das duas versões. Essa padronização teve dois objetivos: (i) garantir que os dois sistemas desenvolvidos implementassem as mesmas funcionalidades e (ii) remover problemas relacionados a diferenças de estilo de codificação.

A fase de evolução e reutilização envolveu o mesmo grupo de pessoas. O objetivo dessa fase foi confirmar os resultados da aplicação das métricas propostas como um mecanismo útil para predizer manutenibilidade e reusabilidade. Nessa fase, foram simuladas mudanças simples e complexas envolvendo os *concerns* de agência, tanto da solução OO quanto da solução OA, com o intuito de medir a

facilidade que se teria para evoluir e reutilizar seus componentes. Foram selecionados oito cenários de evolução e reutilização que são recorrentes em sistemas multi-agentes:

C1) Mudança dos papéis de um agente (evolução);

C2) Criação de um tipo de agente (evolução);

C3) Reutilização da parte fundamental do agente (reutilização);

C4) Inclusão da propriedade de colaboração em um tipo de agente (reutilização e evolução);

C5) Reutilização de papéis (reutilização);

C6) Criação de uma nova instância de agente (evolução);

C7) Mudança da definição da parte fundamental do agente (reutilização e evolução);

C8) Inclusão da propriedade de mobilidade em um tipo de agente (evolução).

Para cada cenário de evolução, a dificuldade para realizar as modificações foi medida em termos dos seguintes itens: (1) número de componentes (classes/aspectos) adicionados, (2) número de componentes alterados, (3) número de relacionamentos adicionados, (4) número de relacionamentos alterados, (5) número de operações (métodos/*advices*) adicionadas, (6) número de operações alteradas, (7) número de linhas de código adicionadas e (8) número de linhas de código alteradas. Para os cenários de reutilização, a dificuldade de reutilizar os componentes foi medida pelos mesmos itens dos cenários de evolução, mais os seguintes itens: (9) número de componentes copiados e (10) número de linhas de código copiadas.

6.1.3. Resultados da Fase de Construção

Esta seção apresenta os resultados obtidos com a aplicação das métricas do framework de avaliação (Capítulo 5) no projeto e código do sistema Portalware. Ela apresenta uma visão geral dos dados obtidos na fase de construção. Os dados dessa fase foram parcialmente coletados pela ferramenta CASE Together 6.0 [12]. Essa ferramenta apóia a aplicação de algumas métricas do framework em classes e interfaces: LOC, NOA, WOC (WMPC2 no Together), CBC (CBO no Together),

LCOO (LOCOM1 no Together) e DIT (DOIH no Together). Ainda não existe ferramenta que dê suporte a aplicação das métricas em aspectos, por isso todos os dados relativos aos aspectos foram colhidos manualmente, sem a ajuda de ferramentas específicas. Além disso, como as métricas de separação de *concerns* são métricas novas, também não existem ferramentas específicas para sua aplicação. Logo, elas foram aplicadas manualmente, com a ajuda apenas de um editor de texto para a realização do sombreamento do código.

A Figura 17 apresenta os resultados gerais das métricas de acoplamento, coesão e tamanho. Esses resultados são mostrados do ponto de vista do sistema como um todo. O número de componentes do sistema na versão OO foi 7% maior do que na versão OA (métrica VS). A quantidade de linhas de código (métrica LOC) e o número de atributos (métrica NOA) para o desenvolvimento do sistema multi-agentes OO foram respectivamente 12% e 9% maiores do que para o desenvolvimento do sistema OA. A versão OA do sistema também produziu melhores resultados em termos de peso de operações (6%), acoplamento entre componentes (9%) e coesão dos componentes (3%). O valor de DIT máximo foi cinco para a versão OO e três para a versão OA, ou seja, DIT máximo na versão OA foi 40% menor que não versão OO.

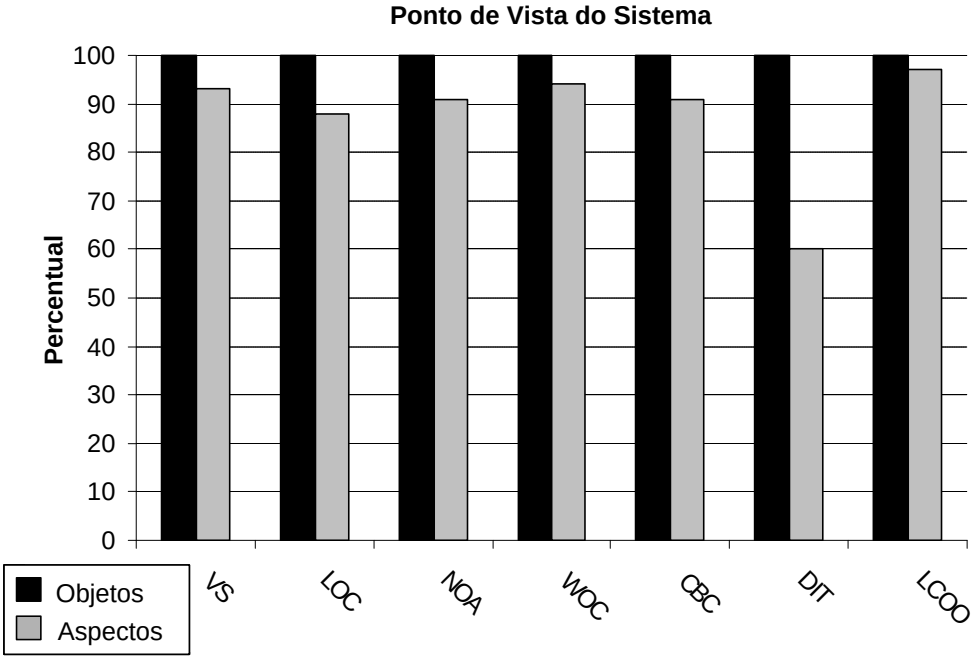

Figura 17 – Comparação dos resultados das duas versões

Os resultados absolutos obtidos do ponto de vista do sistema (Figura 17) mostram diferenças em favor da solução orientada a aspectos. Entretanto, é necessário analisar os dados mais detalhadamente para encontrar os pontos do projeto e do código em que o DSOA foi realmente vantajoso. As próximas subseções apresentam em detalhes os resultados das métricas de separação de *concerns*, acoplamento, coesão e tamanho. O Anexo A apresenta todos os dados obtidos com a aplicação do conjunto de métricas na fase de construção das duas versões do sistema.

6.1.3.1. Resultados das Métricas de Separação de *Concerns*

Difusão do *Concern* por Componentes (CDC). Todos os *concerns* de sistemas multi-agentes precisaram de mais componentes na definição da solução OO do que na definição da solução OA. Todos os papéis do sistema OO precisaram de mais de cinco classes para sua definição, ao passo que um único aspecto implementou cada papel do sistema OA. Por exemplo, o papel de provedor de conteúdo necessitou de seis componentes na versão OO contra apenas um na versão OA. As propriedades de agência também precisaram de mais componentes no projeto e implementação OO: adaptação (3 contra 1), autonomia (3 contra 2), colaboração (15 contra 6) e interação (7 contra 6). Finalmente, mais componentes também foram usados no projeto e implementação OO dos tipos de agentes e suas instâncias. Por exemplo, 37 contra 33 para tipo agente de usuário e 4 contra 2 para as instâncias do agente de usuário.

Difusão do *Concern* por Operações (CDO). Novamente todos os *concerns* precisaram de mais operações (métodos/*advices*) no sistema OO do que no sistema OA. A maioria dos *concerns* foi implementada no sistema OO com mais que o dobro de operações usadas no sistema OA. É o caso, por exemplo, das propriedades de adaptação e autonomia dos agentes. Há casos em que a diferença é ainda maior (por exemplos, os papéis dos agentes). Em poucos casos a diferença é pequena (por exemplo, a propriedade de interação).

Difusão do *Concern* por LOC (CDLOC). A medição aqui também mostrou que a solução OA foi mais eficaz para modularizar os *concerns* do sistema multi-agentes. Os resultado para a implementação dos tipos de agentes

não mostraram diferença. Com exceção do *concern* relativo a parte fundamental do agente, todos os outros *concerns* foram melhor modularizados no projeto e implementação OA. As diferenças detectadas foram bastante significativas para os seguintes *concerns*: (i) propriedades básicas dos agentes, (ii) papéis dos agentes, (iii) instâncias dos agentes e (iv) propriedade de colaboração. Com relação ao *concern* relativo a parte fundamental dos agentes, a fato do aspecto *Interaction* ter que especificar em sua definição que ele deve ser executado antes do aspecto *Collaboration* aumentou o número de pontos de transição, fazendo com que o valor de CDLOC para esse *concern* fosse um pouco maior na solução OA.

6.1.3.2. Resultados das Métricas de Acoplamento

Acoplamento entre Componentes (CBC). Existe uma diferença significativa entre os valores de CBC obtidos para as duas soluções da classe *Agent*. O valor de CBC é doze no sistema OO e nove no sistema OA. A causa dessa diferença foi a necessidade de serem mantidas referências explícitas na classe *Agent* para as classes que representam as propriedades básicas do agente. O inverso também foi necessário, ou seja, as classes *Autonomy*, *Interaction* e *Adaptation* requerem referências explícitas para o objeto da classe *Agent*. No sistema OA, somente os aspectos têm referências para a classe *Agent*.

Profundidade da Árvore de Herança (DIT). Existem muitos problemas no projeto OO em relação a essa métrica. O uso do padrão *Role* gerou uma hierarquia de cinco níveis para estruturar os papéis dos agentes, o que leva potencialmente a um acoplamento de herança alto [27]. Na solução OA, o valor de DIT foi igual a um, pois os papéis são encapsulados nos aspectos, que permeiam a hierarquia de tipo de agente. Além disso, o uso do padrão *Mediator* na solução OO acarretou numa hierarquia de três níveis para estruturar os tipos de agentes, enquanto que a mesma hierarquia na solução OA teve dois níveis apenas. Isso ocorreu pela necessidade de se criar um nível adicional (a classe *CollaborativeAgent*) na solução OO para separar os agentes colaborativos dos agentes não colaborativos. Na solução OA, essa separação é realizada transparentemente pelo aspecto *Collaboration* e, conseqüentemente, não afeta a profundidade da hierarquia de tipos de agentes.

6.1.3.3. Resultados da Métrica de Coesão

Falta de Coesão nas Operações (LCOO). Apesar dos resultados gerais do ponto de vista do sistema terem apresentado uma pequena diferença favorável ao sistema OA (Figura 17), muitos componentes importantes do sistema OO apresentaram melhores resultados de coesão em relação aos mesmos componentes da solução OA. Por exemplo, o valor de LCOO para a classe *Agent* foi 50 na solução OO e 57 na solução OA. O valor de LCOO para os componentes que formam o *concern* da propriedade de interação foi 29 na solução OO e 48 na solução OA.

6.1.3.4. Resultados das Métricas de Tamanho

Tamanho do Vocabulário (VS). O vocabulário externo do sistema multi-agentes OA é mais simples que o do sistema OO, pois a quantidade de componentes de projeto e implementação neste último (VS = 60) foi maior que no primeiro (VS = 56). A principal razão para esse resultado foi o fato dos padrões *Role* e *Mediator* requererem classes adicionais para tratar da decomposição e composição de múltiplos papéis de agentes e propriedades comportamentais, respectivamente.

Número de Atributos (NOA). O número de atributos dos componentes da solução OO foi maior que o número de atributos dos componentes da solução OA. Ou seja, o vocabulário interno dos componentes da solução OO é mais complexo que o dos componentes da solução OA. Isso ocorreu porque os objetos da classe *Agent* na solução OO precisam manter referência explícita para objetos que representam as três propriedades básicas de agentes e para objetos que representam os papéis. Por exemplo, a classe *Agent* (a classe que implementa a estrutura e o comportamento básico dos agentes) é composta por nove atributos na solução OO, mas só precisa de seis atributos na solução OA. Além disso, o *concern* de colaboração é modularizado diretamente pelos aspectos *Collaboration* e pelos aspectos de papéis na versão OA, ao passo que fica espalhado por quatro diferentes classes (*Collaboration*, *CollaborationCore*, *CollaborationRole*,

CollaborativeAgent) e mais as classes de papéis na versão OO. Conseqüentemente, isso leva a um aumento do valor da métrica NOA, pois essas classes precisam manter referências entre si.

Linhas de Código (LOC). O valor de LOC foi 1445 na implementação OO e 1271 na implementação OA, ou seja, o código OO tem 174 linhas a mais que o código OA. Em geral, a implementação de cada par <plano, papel> no código OO incluiu dez linhas a mais do que no código OA. Isso ocorreu porque a solução OO precisa de mais chamadas de métodos nas subclasses da classe *Role* para ativar e desativar papéis e para obter as referências para os objetos das subclasses de *Role*. Essas chamadas não são necessárias na solução OA, pois essa composição é especificada pelos *pointcuts* dos aspectos que implementam os papéis. Além disso, menos linhas de código foram necessárias para implementar o *concern* de colaboração no sistema OA, uma vez que, na solução OO, o padrão *Role* implementa classes adicionais, como já descrito anteriormente.

Peso de Operações por Componentes (WOC). Como apresentado anteriormente, a solução OA teve resultado melhor para essa métrica do ponto de vista do sistema. Porém, em muitos casos específicos, a solução OA apresentou valores mais altos para a métrica WOC. Isso aconteceu pelo fato de que a modularização de alguns *concerns* por meio dos aspectos exige que o contexto seja recapturado por meio dos *advices*. Alguns exemplos dos resultados de WOC são: (i) nove para a classe *Adaptation* na solução OO contra dez do aspecto *Adaptation* na solução OA, (ii) 12 para a classe *Autonomy* contra 14 para o aspecto *Autonomy* e (iii) 17 para a classe *Interaction* contra 25 para o aspecto *Interaction*. Isso aconteceu porque os aspectos *Interaction*, *Adaptation* e *Autonomy* recebem a referência ao objeto da classe *Agent* como parâmetro de seus *advices*, enquanto que tal referência é obtida em um atributo local nas classes correspondentes da solução OO.

6.1.4. Resultados da Fase de Evolução e Reutilização

Esta seção discute cada cenário de evolução e reutilização e os respectivos resultados tanto da versão orientada a aspectos quanto da versão orientada a objetos. Os resultados confirmaram os problemas identificados com o uso do

framework (Seção 6.1.3). A Tabela 2 dá uma visão geral dos resultados dos cenários de evolução e reutilização

	EVOLUÇÃO																			
	REUTILIZAÇÃO																			
	Comp. Alter.		Oper. Alter.		Comp. Adicion.		Oper. Adicion.		Relac. Alter.		Relac. Adicion.		LOCs Adicion.		LOCs Alter.		Comp. Copiad.		LOCs Copiad.	
	OO	**OA**	**OO**	**OA**	**OO**	**OA**	**OO**	**OA**	**OO**	**OA**	**OO**	**OA**	**OO**	**OA**	**OO**	**OA**	**OO**	**OA**	**OO**	**OA**
C1	1	1	3	3	5	5	2	3	0	0	15	15	101	98	1	1	-	-	-	-
C2	0	0	2	2	4	4	0	0	0	0	10	10	84	86	0	0	-	-	-	-
C3	0	0	2	2	4	4	0	0	0	0	10	10	84	86	0	0	0	0	0	0
C4	0	0	2	3	8	8	0	0	0	0	29	25	188	167	0	8	0	0	0	0
C5	1	1	2	1	0	0	1	1	0	0	4	2	16	14	0	0	0	0	6	6
C6	0	0	0	0	0	0	0	0	0	0	0	0	15	15	0	0	-	-	-	-
C7	5	1	0	0	0	0	0	0	5	2	1	1	0	0	5	1	0	0	40	0
C8	11	11	9	8	3	3	30	25	0	0	12	12	191	181	20	22	-	-	-	-

Tabela 2 – Resultados dos Cenários de Evolução e Reutilização

6.1.4.1. Cenário C1 – Mudança dos papéis de um agente

Um papel novo e seus respectivos planos foram incorporados ao sistema para encapsular o comportamento de um revisor e planos associados. Esse papel foi associado ao agente de usuário. Essa mudança resultou em impactos similares no sistema OA e no sistema OO. No entanto, foram necessárias algumas linhas de código a mais no sistema OO.

6.1.4.2. Cenário C2 – Criação de um tipo de agente

Os usuários do Portalware freqüentemente precisam procurar por informações armazenadas em múltiplos bancos de dados e disponíveis na rede para produzir o material requisitado pelos editores. Conseqüentemente, um novo tipo de agente, chamado de agente de informação (ainda não implementado na fase de construção) foi incluído no sistema para automatizar essa tarefa. Essa mudança resultou em impactos similares no sistema OA e no sistema OO. Contudo, esse cenário causou a necessidade de mais dois cenários (C3 e C4) que estão descritos imediatamente a seguir.

6.1.4.3.
Cenário C3 – Reutilização da parte fundamental do agente

Uma vez que todos os tipos de agente do sistema incorporam as características da parte fundamental do agente, a criação de um novo tipo de agente (cenário C2) requereu a reutilização dessas características. Como apresentado na Tabela 2, esse cenário também resultou em mudanças similares nas duas versões do sistema.

6.1.4.4.
Cenário C4 – Inclusão da propriedade de colaboração em um tipo de agente

A inclusão do agente de informação (cenário C2) também gerou a necessidade de reutilização da propriedade de colaboração, que já estava implementada no sistema. Isso porque agentes de informação colaboram entre si quando um agente de informação não encontra a informação solicitada em seu banco de dados. Sendo assim, os agentes de informação desempenham os papéis de solicitante e fornecedor. Desempenhando o papel de solicitante, ele solicita a informação para outro agente de informação. Como fornecedor, ele recebe solicitações de informação e envia a resposta para o solicitante. Portanto, esse cenário compreendeu duas tarefas: (i) a reutilização da propriedade de colaboração no contexto do agente de informação e (ii) a criação de dois papéis e a associação desses papéis ao agente de informação. Esse foi o cenário que resultou em maiores diferenças nas alterações feitas nas duas versões do sistema: (i) foi necessária a adição de 20 linhas de código a mais na solução OO do que na versão OA, (ii) mais relacionamentos foram adicionados no projeto OO e (iii) oito linhas foram removidas do código OA, enquanto nenhuma linha foi removida do código OO.

6.1.4.5.
Cenário C5 – Reutilização de papéis

A introdução de agentes de informação no sistema fez com que serviços de informação ficassem disponíveis para outros tipos de agente. Já que alguns agentes de usuário desempenham o papel de provedor de conteúdo, o sistema

pode automatizar a tarefa de selecionar informações relevantes para certos contextos em favor de seus usuários. Sendo assim, os agentes de usuário podem, desempenhando o papel de solicitante já definido no sistema, usar os serviços dos agentes de informação. Dessa forma, o papel de solicitante foi reutilizado e associado ao agente de usuário nas duas versões do sistema. Esse cenário de reutilização exigiu mais esforço na solução OO do que na solução OA de acordo com os seguintes dados: (i) número de operações alteradas (2 contra 1), (ii) número de relacionamentos adicionados (4 contra 2) e (iii) número de linhas de código adicionadas (16 contra 14).

6.1.4.6. Cenário C6 – Criação de uma nova instância de agente

Esse cenário investigou o impacto de se adicionar novas instâncias de agentes no sistema. Em particular, foi criada uma nova instância do agente de usuário para desempenhar o papel de provedor de conteúdo. Esse cenário de evolução exigiu as mesmas alterações nas duas versões do sistema. Quinze linhas de código foram adicionadas tanto no sistema OA quanto no sistema OO.

6.1.4.7. Cenário C7 – Mudança da definição da parte fundamental do agente

Esse cenário foi criado para simular uma mudança realmente drástica nas duas soluções. Com a inclusão do agente de informação no sistema, todos os outros tipos de agente poderiam ser capazes de usar os seus serviços, tendo para isso que colaborar com eles. Então a definição da parte fundamental teve que ser estendida para incluir a propriedade de colaboração. Dessa forma, todos os tipos de agente passaram a ser colaborativos. Para isso, a propriedade de colaboração foi removida da classe *UserAgent* e associada à classe *Agent*. Esse cenário reutilizou, portanto, a propriedade de colaboração em um novo contexto. A solução OA permitiu maior facilidade de evolução e reutilização nesse cenário: (i) cinco componentes foram alterados na versão OO e apenas um na versão OA, (ii) cinco relacionamentos foram alterados no projeto OO contra apenas dois no projeto OA e (iii) 40 linhas de código foram copiadas na implementação OO contra nenhuma na implementação OA.

6.1.4.8.
Cenário C8 – Inclusão da propriedade de mobilidade em um tipo de agente

Esse cenário tratou da evolução do agente de informação no sentido de torná-lo um agente móvel que pode se deslocar, através da rede, em busca da informação solicitada em outros ambientes, mais especificamente, em outras instâncias do sistema Portalware executadas em computadores espalhados pela rede. Os resultados desse cenário foram semelhantes em ambas soluções. As maiores diferenças, à favor da solução OA, ocorreram no número de linhas de código adicionadas (191 contra 181) e no número operações adicionadas (30 contra 25). Isso se deveu principalmente ao fato de que o agente de informação na solução OO mantém uma referência explícita para a classe que implementa propriedade de mobilidade e precisa iniciá-la e acioná-la explicitamente por meio de chamada de métodos. A descrição desse cenário é apresentada com detalhes em [47].

6.1.5.
Discussões sobre os Resultados do Primeiro Estudo Experimental

Em geral, os resultados da fase de construção (Seção 6.1.3) mostraram que a abordagem orientada a aspectos produziu um sistema com maior manutenibilidade e reusabilidade. Apesar do sistema orientado a aspectos ter sido desenvolvido antes do sistema orientado a objetos, o processo de medição guiado pelo framework de avaliação gerou melhores resultados para o primeiro sistema.

A abordagem orientada a aspectos produziu um sistema mais conciso em termos de linhas de código, quantidade de componentes e número de atributos dos componentes. O uso de padrões de projeto no sistema orientado a objetos levou a um aumento do número de classes. Essas conclusões são apoiadas por todas as métricas de tamanho, com exceção da métrica WOC.

A abordagem orientada a aspectos gerou operações mais complexas do que a abordagem orientada a objetos. Isso ocorreu porque a modularização dos *concerns* por aspectos requer que o contexto dos objetos afetados seja capturado pelos *advices*, o que faz com o número de parâmetros desses *advices* seja elevado. Essa conclusão foi apoiada pelos resultados da métrica WOC.

A utilização de padrões de projeto no sistema orientado a objetos levou também ao uso acentuado do mecanismo de herança entre classes, aumentando o acoplamento por meio de herança. Esse problema foi detectado pela métrica DIT. A abordagem orientada a objetos também produziu componentes mais altamente acoplados entre si, fato esse detectado pela métrica CBC.

Por outro lado, a métrica LCOO mostrou que a abordagem orientada a aspectos gerou, em muitos casos, componentes com coesão mais baixa que os componentes do sistema orientado a objetos. A falta de coesão dos aspectos se deveu ao fato de que os aspectos encapsulam comportamentos relativos a diferentes componentes afetados por ele, e que, muitas vezes, não são relacionados entre si.

A abordagem orientada a aspectos forneceu claramente um apoio melhor para a separação dos *concerns* do sistema multi-agentes. Essa conclusão é apoiada pelos resultados das três métricas de separação de *concerns*.

Os resultados obtidos na fase de evolução e reutilização (Seção 6.1.4) confirmaram o que já era previsto com os resultados da fase de construção. Isso dá substanciais evidências da utilidade das métricas propostas. Por exemplo, a inclusão da propriedade de colaboração em um tipo de agente (cenário C4) foi o cenário que resultou em maiores diferenças de mudanças necessárias em favor da abordagem orientada a aspectos. Esse resultado confirma o que já previam os resultados das métricas CBC, DIT, VS, NOA, WOC e das três métricas de separação de *concerns* obtidos na fase de construção. Essas métricas geraram resultados muito melhores em relação aos componentes da propriedade de colaboração no sistema orientado a aspectos. Portanto, nesse caso, o acoplamento entre componentes, o número de componentes, o número de atributos, a complexidade de operações e a separação de *concerns* influenciaram diretamente as atividades de reutilização e evolução do sistema.

6.2.
O Segundo Estudo Experimental

Esta seção descreve o segundo estudo experimental realizado com o uso do framework de avaliação proposto nesta dissertação. Da mesma forma que o

primeiro estudo (Seção 6.1), o objetivo desse segundo estudo foi avaliar o framework proposto. Ou seja, demonstrar a utilidade e usabilidade do conjunto de métricas e do modelo de qualidade. Nesse estudo, o framework de avaliação foi usado para comparar as implementações em Java e AspectJ de alguns padrões de projetos da *GoF* [44]. Essas implementações foram propostas por Hannemann & Kiczales [5]. Em [5], eles apresentam um estudo que compara qualitativamente implementações em Java dos 23 padrões da *GoF* com implementações em AspectJ dos mesmos padrões. Além de realizar uma análise qualitativa, eles baseiam essa análise em alguns atributos pouco difundidos na engenharia de software, como transparência de composição, localidade e facilidade de (des)ligamento[14].

A escolha desse contexto para a realização do segundo estudo experimental se deveu a alguns fatores: (i) a possibilidade de uso do framework em um domínio diferente do domínio do primeiro estudo, (ii) a aplicação das métricas em um código construído por outras pessoas e que usa algumas construções de Java e AspectJ que não tinham sido usadas na implementação do primeiro estudo. Além disso, esse estudo pode ser visto como uma complementação do estudo feito por Hannemann & Kiczales, pois compara as implementações dos padrões quantitativamente em termos de separação de *concerns*, acoplamento, coesão e tamanho, que são atributos bem conhecidos da engenharia de software.

6.2.1. O Estudo Realizado por Hannemann & Kiczales

Hannemann & Kiczales realizaram um estudo no qual eles desenvolveram e compararam implementações em Java [17] e AspectJ [16] dos 23 padrões de projeto da *GoF*. Eles afirmam que outros trabalhos mostraram que linguagens de programação afetam a implementação de padrões de projeto, por isso parece natural explorar o efeito de técnicas de programação orientadas a aspectos na implementação dos padrões da *GoF*. No seu estudo, Hannemann & Kiczales mantiveram o propósito e a aplicabilidade dos padrões, permitindo que fossem mudadas apenas a estrutura e a implementação da solução. Portanto, eles não criaram novos padrões, mas simplesmente verificaram como as implementações

[14] Tradução do autor para o termo *(un)pluggability* usado por Hannemann & Kiczales.

dos padrões da *GoF* podiam ser manipuladas usando outra linguagem. Os resultados do seu estudo mostraram que o uso de AspectJ melhorou a implementação de muitos padrões da *GoF*. Em alguns casos isso foi refletido numa nova estrutura da solução do padrão com menos ou diferentes participantes. Em outros casos, a estrutura permaneceu a mesma, e apenas a implementação foi mudada.

Para os 23 padrões da *GoF*, um pequeno exemplo que fazia uso do padrão foi criado e implementado em Java e AspectJ. O código está disponível e pode ser obtido em [48]. As implementações em Java correspondem as implementações em C++ do livro da *GoF* [44], com alguns ajustes devido as diferenças entre C++ e Java. As implementações em AspectJ foram desenvolvidas iterativamente. Tanto em Java quanto em AspectJ, alguns padrões tiveram algumas variantes e alternativas de implementação. Eles usaram a que lhes pareceu a mais genericamente aplicável.

Padrões atribuem papéis a seus participantes, por exemplo, os papéis de sujeito e observador no padrão *Observer*. Esses papéis definem a funcionalidade dos componentes participantes no contexto do padrão. Alguns padrões da *GoF* envolvem estruturas em que os papéis atravessam ou permeiam classes da instância do padrão. Por exemplo, no padrão *Observer* uma operação que muda o estado de qualquer sujeito tem que disparar notificações para seus observadores, em outras palavras, a ação de notificação permeia uma ou mais operações de cada classe que desempenha o papel de sujeito no padrão.

A Figura 18 (retirada de [5]) mostra um exemplo concreto do uso do padrão *Observer* no contexto de um sistema simples de elementos gráficos. Nesse caso, o padrão *Observer* é usado para atualizar a tela quando os elementos gráficos são alterados. Os métodos sombreadaos na figura contêm algum código necessário para implementar o padrão *Observer*. Isso mostra que o código necessário para implementar esse padrão fica espalhado pelas classes. Os participantes (isto é, as classes *Point* e *Line*) precisam saber do seu papel no padrão e, por isso, têm código do padrão em sua implementação. Para retirar ou incluir um papel em uma classe é necessário alterar a classe.

O estudo feito por Hannemann & Kiczales mostrou que os padrões cuja estrutura é caracterizada por papéis que permeiam as classes participantes foram os padrões que apresentaram uma melhora mais significativa com a

implementação em AspectJ. Eles comparam as implementações dos padrões em termos de quatro propriedades: localidade, reusabilidade, transparência de composição e facilidade de (des)ligamento.

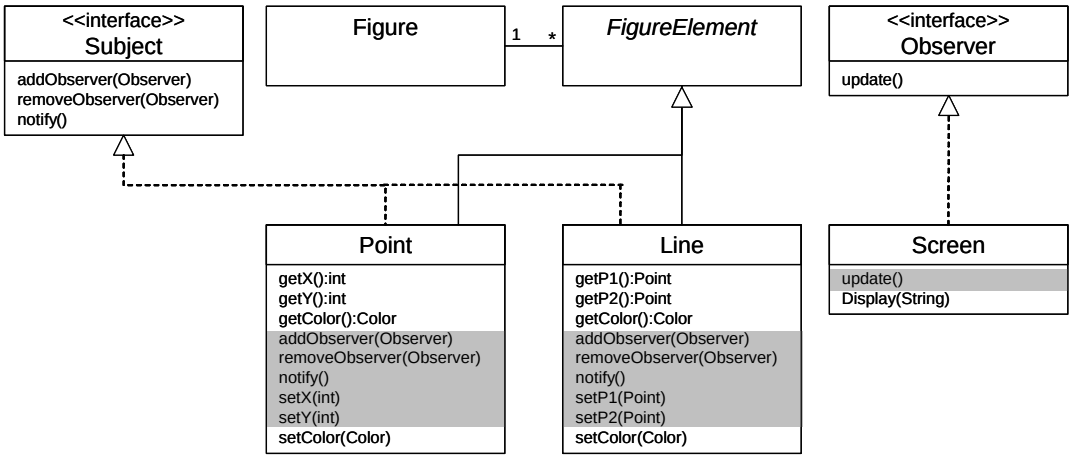

Figura 18 – Um sistema simples de elementos gráficos que usa o padrão *Observer* em Java

As implementações em AspectJ procuram eliminar o espalhamento do código do padrão pelas classes participantes de forma que eles não precisem saber do seu papel no padrão. Ou seja, todo o código que implementa uma instância do padrão fica localizado em um aspecto e não nas classes participantes. Esse aspecto introduz o papel nas classes participantes de forma transparente para elas. Dessa forma, qualquer alteração na instância do padrão fica confinada a apenas um lugar (localidade). Em algumas implementações orientadas a aspectos, uma parte do padrão foi implementada em um aspecto abstrato e pode, dessa forma, ser reutilizada por várias instâncias do mesmo padrão (reusabilidade). Pelo fato das classes participantes não possuírem código relativo ao padrão, se um deles participar de mais de uma instância do padrão, seu código não fica mais complicado e a instância do padrão não fica confusa (transparência de composição). Devido às classes participantes não serem cientes de seus papéis na instância do padrão, é possível alternar facilmente entre usar ou não o padrão no sistema (facilidade de (des)ligamento).

Em seu estudo Hannemann & Kiczales dividiram os 23 padrões da *GoF* em seis grupos de acordo com características em comum relativas a estrutura dos padrões ou suas implementações em AspectJ. Os padrões *Observer*, *Command*,

Mediator e *Chain of Responsibility* são do grupo de padrões implementados em AspectJ por um aspecto abstrato e mais um aspecto concreto para cada instância. Outro grupo é formado pelos padrões *Singleton*, *Prototype*, *Memento*, *Iterator* e *Flyweight*. Esses padrões administram o acesso a instâncias específicas de objetos. Todos eles oferecem métodos que servem como fábricas para clientes e compartilham uma estratégia criação por demanda. Em AspectJ, esse padrões também são implementados por um aspecto abstrato, com o código da fábrica dentro do aspecto. Os padrões *Adapter*, *Decorator*, *Strategy*, *Visitor* e *Proxy* compõem outro grupo. A implementação desses padrões praticamente desaparece, porque as construções da linguagem AspectJ os implementa diretamente. O quarto grupo é constituído pelos padrões *Abstract Factory*, *Factory Method*, *Template Method*, *Buider* e *Bridge*. Esses padrões são estruturalmente similares: herança é usada para distinguir implementações diferentes, mas relacionadas. Não é possível dar a esses padrões uma implementação mais reutilizável. Porém com AspectJ, é possível substituir as classes abstratas mencionadas na solução da *GoF* por interfaces sem perder a possibilidade de fornecer uma implementação padrão para seus métodos. Os padrões *State* e *Interpreter* formam o quinto grupo. Esses padrões têm um acoplamento muito estreito entre seus participantes, fazendo com que o seu código se espalhe por eles. Em AspectJ, partes do código espalhado pode ser modularizado. O último grupo é composto apenas pelo padrão *Facade*. Não existe diferença entre as implementações em AspectJ e em Java desse padrão.

De acordo com estudo realizado por Hannemann & Kiczales, as implementações em AspectJ de 17 dos 23 padrões de projeto ficaram bem localizadas em um ponto do código. Essa melhoria na localização permitiu que uma parte da implementação fosse abstraída em código reutilizável. Em 14 dos 17, foi observada a propriedade de transparência de composição nas instâncias do padrão, facilitando com que múltiplas instâncias dos padrões pudessem compartilhar participantes.

6.2.2. O Formato do Segundo Estudo Experimental

Diferentemente do primeiro estudo experimental, nesse estudo não houve uma fase em que o projeto e o código a serem avaliados tivessem que ser

desenvolvidos, pois o código analisado foi produzido por Hannemann & Kiczales em seu estudo (já descrito na seção anterior). Foram escolhidos seis padrões para serem avaliados, de forma a se ter um representante de cada grupo da divisão realizada por Hannemann & Kiczales, com exceção do grupo do padrão *Facade* cujas implementações em Java e AspectJ são idênticas.

Os padrões de projeto avaliados foram o *Observer*, o *Mediator*, o *Prototype*, o *Strategy*, o *State* e o *Abstract Factory*. As métricas de separação de *concerns*, acoplamento, coesão e tamanho foram aplicadas ao código em Java e ao código em AspectJ dos exemplos de uso desses padrões. Em seguida, cada exemplo foi modificado com a inclusão de mais classes participantes. Por exemplo, no caso do padrão *Observer* foram implementadas mais quatro classes que desempenhavam o papel de sujeito e mais quatro classes que desempenhavam o papel de observador. Depois disso, o conjunto de métricas do framework de avaliação foi aplicado sobre o código modificado, e os resultados comparados com os do código original.

Da mesma forma que o primeiro estudo, as métricas foram aplicadas nesse estudo com a ajuda parcial da ferramenta CASE Together 6.0 [12]. Os dados relativos aos aspectos foram obtidos investigando o projeto e o código sem a ajuda de ferramenta. As métricas de separação de *concerns* também foram aplicadas manualmente, com a ajuda apenas de um editor de texto para realizar o processo de sombreamento do código. Os *concerns* avaliados com as métricas de separação de *concerns* foram os *concerns* relativos aos papéis dos participantes dos padrões.

6.2.3. Resultados do Segundo Estudo Experimental

Esta seção apresenta os resultados da aplicação das métricas do framework de avaliação nos códigos dos exemplos de uso dos seguintes padrões de projeto: *Observer*, *Mediator*, *Prototype*, *Strategy*, *State* e *Abstract Factory*. Os resultados são relativos as implementações em Java e AspectJ, antes e depois da inclusão de novos componentes participantes. O Anexo B apresenta todos os dados obtidos com a aplicação do conjunto de métricas.

6.2.3.1. Resultados do Padrão *Observer*

O propósito do padrão *Observer* é definir uma dependência de um para muitos entre objetos de forma que, quando um objeto mudar o seu estado, todos os deus dependentes sejam notificados e atualizados automaticamente [44]. Os papéis principais dos participantes desse padrão são o **sujeito** e o **observador**.

O exemplo implementado em Java por Hannemann & Kiczales é formado pelas classes *Point* e *Screen* que implementam respectivamente as interfaces *Subject* e *Observer*. Além da classe *Main* que faz as instanciações e simula mudanças de estado das classes que são sujeito. A implementação em AspectJ é composta também pelas classes *Point* e *Screen*, pelo aspecto abstrato *ObserverProtocol* e pelos aspectos concretos *CoordinateObserver*, *ColorObserver*, *ScreenObserver*.

Os resultados totais, antes e depois da adição de novas classes participantes, são apresentados na Tabela 3. Depois da primeira coleta de dados, foram incluídas mais quatro classes para desempenhar o papel de sujeito e mais quatro classes para desempenhar o papel de observador em ambas implementações. A Tabela 3, assim como as tabelas das próximas seções deste capítulo, apresenta o somatório dos resultados de todas as classes e aspectos, com exceção da coluna da métrica DIT, que mostra o valor máximo obtido dentre os componentes.

	CBC		**DIT**		**LCOO**		**VS**		**LOC**		**NOA**		**WOC**	
	Antes	Depois	Antes	Depois	Antes	Depois	Antes	Depois	Antes	Depois	Antes	Depois	Antes	Depois
OO	13	45	2	2	16	80	5	13	109	363	6	26	38	134
OA	16	40	2	2	13	30	7	15	117	265	5	21	47	117

Tabela 3 – Resultados do padrão *Observer*: Acoplamento, Coesão e Tamanho

Pode-se notar com os resultados que, à medida que se aumentou o número de classes participantes, a solução OA passou a ter melhores resultados para quase todas as métricas, exceto para a DIT e VS. Isso aconteceu porque: (i) na solução OO, as classes que desempenham os papéis de observador e sujeito estão acopladas entre si pelo código do padrão (métrica CBC), (ii) na solução OO, cada classe participante tem mais linhas de código relativas à implementação do padrão (métrica LOC) e (iii) na solução OO, as classes participantes, além de métodos e atributos relativos a sua funcionalidade original, têm métodos e atributos relativos

ao papel que desempenham no padrão (métricas LCOO, NOA e WOC). A métrica VS é maior na solução OA, porque é necessário um aspecto concreto para cada instância do padrão, e nesse exemplo foram implementadas, ao mesmo tempo, três instâncias do padrão *Observer*.

Os resultados das métricas de separação de *concerns* também são favoráveis à solução OA depois da inclusão de novas classes participantes. Por exemplo, o *concern* relativo ao papel de observador ficou espalhado em 8 componentes e 43 operações na solução OO contra 5 componentes e 5 operações na solução OA (métricas CDC e CDO). Além disso o valor da métrica CDLOC para esse *concern* foi 70 na solução OO contra 10 na OA.

6.2.3.2. Resultados do Padrão *Mediator*

O padrão *Mediator* define um objeto que encapsula a maneira como um conjunto de objetos interage entre si [44]. Os papéis principais dos componentes participantes do padrão são o **mediador** e o **colega**.

O exemplo implementado em Java por Hannemann & Kiczales é composto pelas classes *Label* e *Button* que implementam respectivamente as interfaces *Mediator* e *Colleague*. A implementação em AspectJ é composta também pelas classes *Label* e *Button*, pelo aspecto abstrato *MediatorProtocol* e pelo aspecto concreto *MediatorImplementation*. Além da classe *Main*, que existe nas duas implementações.

Neste exemplo foram adicionadas mais quatro classes para desempenhar o papel de colega. A Tabela 4 mostra os resultados da aplicação das métricas de acoplamento, coesão e tamanho antes e depois da inclusão de novos participantes.

	CBC		DIT		LCOO		VS		LOC		NOA		WOC	
	Antes	Depois	Antes	Depois	Antes	Depois	Antes	Depois	Antes	Depois	Antes	Depois	Antes	Depois
OO	13	29	7	7	1	5	5	9	69	215	5	17	15	35
OA	11	23	7	7	1	1	5	9	87	212	5	13	20	32

Tabela 4 – Resultados do padrão *Mediator*: Acoplamento, Coesão e Tamanho

Os resultados desse padrão são semelhantes ao do padrão *Observer*. À medida que se aumentou o número de classes que desempenhavam o papel de colega, a solução OA passou a ter resultados melhores para as métricas CBC,

LCOO, LOC, NOA e WOC. Para a métrica CBC, isso ocorreu porque, na solução OO as classes que desempenham o papel de colega são acopladas a classe que desempenha o papel de mediador e vice-versa. O que não ocorre na solução OA. Na solução OO, as classes que desempenham o papel de colega têm um atributo de referência para a classe que desempenha o papel de mediador e um método para configurar essa referência. Isso fez com os valores das métricas NOA, WOC, LOC e LCOO para essas classes fossem maiores na solução OO do que na OA.

Os resultados das métricas de separação de *concerns* também foram favoráveis à solução OA depois da inclusão de novas classes para desempenhar o papel de colega. O *concern* relativo a esse papel ficou espalhado em 7 componentes e 12 operações na solução OO contra 3 componentes e 4 operações na solução OA (métricas CDC e CDO). Além disso o valor da métrica CDLOC para esse *concern* foi 36 na solução OO contra 6 na OA.

6.2.3.3. Resultados do Padrão *Prototype*

O propósito do padrão *Prototype* é especificar os tipos de objetos a serem criados a partir de uma instância protótipo, e criar novos objetos copiando esse protótipo [44]. O papel principal desse padrão é o papel de **protótipo**.

O exemplo desse padrão implementa a prototipagem de cadeias de caracteres. A implementação OO tem as classes *AnotherString, MyString* que desempenham o papel de protótipo e a classe *Main*. A implementação OA tem as classes *AnotherString*, *MyString* e *Main*, o aspecto abstrato *PrototypeProtocol* e o aspecto concreto *StringPrototypes*. Nesse exemplo foram adicionadas mais quatro classes para desempenhar o papel de protótipo. A Tabela 5 apresenta os resultados.

	CBC		DIT		LCOO		VS		LOC		NOA		WOC	
	Antes	Depois	Antes	Depois	Antes	Depois	Antes	Depois	Antes	Depois	Antes	Depois	Antes	Depois
OO	3	7	2	2	0	0	3	7	66	146	2	6	14	38
OA	5	13	2	2	0	0	5	9	86	158	2	6	19	39

Tabela 5 – Resultados do padrão *Prototype*: Acoplamento, Coesão e Tamanho

Os resultados para esse padrão já não foram tão favoráveis a implementação OA, pois, na solução OO, é pequena a quantidade de código referente ao papel de

protótipo que fica espalhado nas classes participantes. Mesmo com o aumento do número classes do papel protótipo, a métrica CBC foi maior para a solução OA, pois o aspecto *StringPrototypes* é acoplado com todas as classes que desempenham o papel de protótipo. Os resultados mostram também diferenças favoráveis a solução OO para as métricas LOC e WOC. Mas à medida que se aumenta o número de classes do papel protótipo, essas diferenças vão diminuindo lentamente, pois, na solução OO, essas classes têm apenas um método de três linhas de código relativo ao papel. Já as métricas de separação de *concerns* mostraram que o *concern* relativo ao papel de protótipo foi melhor modularizado na solução OA. Depois da inclusão de novos participantes a métrica CDO ficou com valor 7 para a solução OO e 5 para a solução OA, enquanto que CDLOC teve valor 30 para a solução OO e valor 18 para a solução OA.

6.2.3.4. Resultados do Padrão *Strategy*

O padrão *Strategy* define uma família de algoritmos, encapsula cada um deles e permite que eles sejam intercambiáveis [44]. Os papéis principais dos participantes desse padrão são o **contexto** e a **estratégia**.

Esse padrão é usado num exemplo que define algoritmos de ordenação de números. A implementação em Java compreende as classes *BubbleSort* e *LinearSort* que implementam a interface *SortingStrategy* e fazem o papel de estratégia, além da classe *Sorter*, que desempenha o papel de contexto, e a classe *Main*, responsável pelas instanciações e execução do exemplo. A implementação OA é composta pelas classes *BubbleSort*, *LinearSort*, *Sorter*, *Main* e pelos aspectos *Strategy Protocol* e *SortingStrategy*.

A evolução realizada nesse exemplo incluiu mais quatro classes para desempenhar o papel de contexto e usar as estratégias de ordenação. A Tabela 6 exibe os resultados das métricas de acoplamento, coesão e tamanho.

	CBC		DIT		LCOO		VS		LOC		NOA		WOC	
	Antes	Depois	Antes	Depois	Antes	Depois	Antes	Depois	Antes	Depois	Antes	Depois	Antes	Depois
OO	6	14	2	2	0	0	5	9	75	111	0	0	21	33
OA	10	18	2	2	0	0	6	10	93	137	1	1	26	34

Tabela 6 – Resultados do padrão *Strategy*: Acoplamento, Coesão e Tamanho

Os resultados da métrica CBC mostraram que o acoplamento foi sempre maior na solução OA, pois o aspecto *SortingStrategy* está acoplado a todas as classes que desempenham os papéis de contexto e estratégia. O valor da métrica LOC também foi sempre maior na solução OA, porque quando se aumentou o número de classes participantes, foi necessário aumentar também o número de linhas de código que tratam dessas classes dentro do aspecto *SortingStrategy*. E esse aumento de linhas de código na solução OA foi maior que aumento do código do padrão que fica espalhado nas classes participantes da solução OO. Já em relação à métrica WOC, percebe-se que a diferença favorável à solução OO foi diminuindo à medida que foram adicionadas mais classes do papel contexto. Isso ocorreu porque essas classes na solução OO precisam receber como parâmetro o objeto que representa a estratégia a ser usada.

Os resultados das métricas de separação de *concerns* foram favoráveis à solução OA quando se aumentou o número de classes participantes, pois, na solução OO, o código relativo ao papel *Context* fica espalhado pelas classes participantes e misturados com seu código, e, na solução OA, esse código fica isolado no aspecto. Depois da adição das novas classes, os resultados do *concern* relativo ao papel de contexto foram os seguintes: CDC, CDO e CDLOC tiveram os valores 6, 6 e 24 respectivamente na solução OO, contra 3, 4 e 8 na solução OA.

6.2.3.5. Resultados do Padrão *State*

O propósito do padrão *State* é permitir que um objeto altere seu comportamento quando seu estado interno muda [44]. Os participantes desse padrão desempenham os papéis de **contexto** e **estado**.

O exemplo usado por Hannemann & Kiczales implementa o controle de estados de uma fila. A solução OO é compostas pela classe *Queue* que implementa a interface *QueueContext* e pelas classes *QueueEmpty*, *QueueNormal* e *QueueFull* que implementam a interface *QueueState* e desempenham o papel de estado.

Durante o estudo, esse exemplo foi modificado com a inclusão de mais duas classes para desempenhar o papel estado, ou seja, mais dois estado para a fila. A Tabela 7 apresenta os resultados da aplicação das métricas.

	CBC		DIT		LCOO		VS		LOC		NOA		WOC	
	Antes	Depois	Antes	Depois	Antes	Depois	Antes	Depois	Antes	Depois	Antes	Depois	Antes	Depois
OO	9	15	2	2	0	66	7	9	127	295	6	8	51	128
OA	8	10	2	2	3	57	7	9	135	316	9	13	38	94

Tabela 7 – Resultados do padrão *State*: Acoplamento, Coesão e Tamanho

Os resultados da métrica CBC para esse padrão demonstraram que o acoplamento entre os componentes da solução OA foi sempre mais fraco do que entre os componentes da solução OO. Isso ocorreu porque, na solução OO, as classes que desempenham o papel de estado são acopladas entre si para realizar as mudanças de estado da fila. Além disso, os métodos dessas classes precisam receber como parâmetro um objeto do tipo *QueueContext* para poder realizar as transições de estado. Isso fez com que a métrica WOC, que considera também os parâmetros das operações, fosse maior na solução OO.

Na solução OA, todo o controle de transição de estado é feito pelo aspecto *QueueStateAspect*. Por outro lado, esse aspecto precisa manter uma referência para cada classe que desempenha o papel de estado e precisa implementar *advices* para realizar as transições de estado. Isso fez com que as métricas NOA e LOC gerassem valores mais altos para a solução OA. As métricas de separação de *concerns* apresentaram resultados similares para as duas implementações pois em ambas soluções não havia um espalhamento e mistura dos códigos dos papéis do padrão.

6.2.3.6. Resultados do Padrão *Abstract Factory*

O propósito do padrão *Abstract Factory* é prover uma interface para a criação de famílias de objetos relacionados ou dependentes entre si sem a especificação de suas classes concretas [44]. Os papéis principais dos participantes desse padrão são a **fábrica** e o **produto**.

O exemplo usado implementa uma fábrica para a criação de dois tipos botões e rótulos de interface gráfica. A implementação em Java é formada pelas

classes *RegularFactory* e *FramedFactory* que implementam a interface *AbstractFactory* e desempenham o papel de fábrica, e pelas classes *Display* e *Main*. A única diferença que a implementação em AspectJ tem para a implementação em Java é o aspecto *AbstractFactoryEnhancement*, que introduz comportamento padrão aos métodos da interface *AbstractFactory*. Por isso os resultados da aplicação das métricas de separação de *concerns*, acoplamento, coesão e tamanho foram semelhantes para as duas soluções. As únicas diferenças foram decorrentes do uso desse aspecto. Nesse estudo, as duas implementações desse exemplo foram modificadas com a adição de mais dois tipos fábricas de botões e rótulos gráficos. Veja alguns resultados na Tabela 8 logo a seguir.

	CBC		DIT		LCOO		VS		LOC		NOA		WOC	
	Antes	Depois	Antes	Depois	Antes	Depois	Antes	Depois	Antes	Depois	Antes	Depois	Antes	Depois
OO	23	31	7	7	1	1	5	7	145	203	4	6	19	27
OA	24	32	7	7	1	1	6	8	150	208	4	6	19	27

Tabela 8 – Resultados do padrão *Abstract Factory*: Acoplamento, Coesão e Tamanho

6.2.4. Discussões sobre os Resultados do Segundo Estudo Experimental

Os resultados desse estudo mostraram que as implementações em AspectJ trouxeram algum benefício em termos de separação de *concerns*, acoplamento, coesão e tamanho para muitos dos padrões de projetos estudados, e, conseqüentemente, melhoraram a manutenibilidade e a reusabilidade dos componentes dessas implementações. Os resultados relativos aos padrões *Observer* e *Mediator* foram os que indicaram as maiores diferenças favoráveis à implementação em AspectJ, pois, na solução em Java para esses padrões, uma quantidade maior de código relativo aos papéis do padrão fica espalhada pelas classes participantes da implementação do padrão.

Por outro lado, a métrica VS indicou que o número de componentes foi maior na maioria das implementações em AspectJ. Ou seja, o vocabulário externo ficou mais complexo nas implementações em AspectJ, devido à necessidade do uso de aspectos para implementar os padrões. Apesar de VS ser maior nas implementações orientadas a aspectos, os resultados das outras métricas de tamanho (LOC, NOA e WOC) foram, na sua maioria, melhores nas implementações em AspectJ, o que mostra que o código dos padrões, antes

espalhados pelas classes das soluções em Java, ficou localizado apenas nos aspectos das soluções em AspectJ.

A métrica LCOO não foi muito útil na análise da maioria dos padrões, mas, na análise do padrão *Observer*, ela indicou que o código dos papéis do padrão, espalhado pelas classes participantes, diminui a coesão dessas classes. A métrica DIT gerou sempre valores iguais para as duas soluções, pois as soluções em Java usam o mecanismo de implementação de interfaces, e as soluções em AspectJ usam o mecanismo de herança entre aspectos.

6.3. Ameaças à Validade dos Estudos Experimentais

Essa seção discute algumas questões sobre as ameaças à validade dos dois estudos experimentais apresentados neste capítulo. São questões relativas à validade de construção, à validade interna e à validade externa.

Validade de construção é o grau com o qual as variáveis dependentes e independentes medem realmente o que elas se propõem a medir [49]. O primeiro estudo experimental envolveu variáveis dependentes e independentes. As variáveis dependentes foram aquelas usadas na fase de evolução e reutilização (Seção 6.1.4) para medir o esforço de se realizar atividades de evolução e reutilização, como, por exemplo, a quantidade de linhas de código alteradas. As variáveis dependentes usadas nesse estudo foram baseadas num estudo anterior realizado por Li e Henry [9]. Em seu trabalho, o conceito de esforço para realizar manutenção é medido pelo número de linhas de código alteradas. Nós próximos estudos experimentais, podem ser usadas outras formas mais significativas de medir o esforço para realizar manutenção, tais como o tempo para entender, desenvolver e implementar modificações [34]. A análise da validade de construção das métricas propostas no framework, que são as variáveis independentes do primeiro estudo, está fora do escopo desta dissertação. No entanto, algumas dessas métricas são extensões das métricas de Chidamber & Kemerer, que são métricas já analisadas teoricamente em outros trabalhos [50, 51, 52].

Validade interna é o grau com o qual se pode confiar nas conclusões sobre o efeito das variáveis independentes nas variáveis dependentes [49]. Esse tipo de

validade também tem haver com o primeiro estudo, que envolveu variáveis dependentes e independentes. Esse estudo não pode ser considerado um experimento controlado. A principal ameaça interna a esse estudo se refere ao fato de que todas as pessoas participaram do desenvolvimento das duas versões do sistema. Contudo, apesar da influência do aprendizado dos participantes à medida que o estudo ia sendo realizado, o sistema orientado a aspectos, que foi desenvolvido primeiro, apresentou melhores resultados tanto na fase de construção quanto na fase de evolução e reutilização.

Validade externa é o grau com o qual os resultados do estudo podem ser generalizados para a população em estudo e para outros contextos [49]. O uso de estudantes no estudo, e o tamanho e complexidade limitados dos sistemas analisados, podem restringir a extrapolação dos resultados obtidos. Todavia, apesar dos resultados não poderem ser diretamente generalizados para desenvolvedores profissionais e sistemas do mundo real, esses estudos acadêmicos permitiram a realização de avaliações iniciais das métricas propostas, que indicaram que estudos mais profundos sobre elas merecem ser realizados.