5
Avaliação da Modularidade e Estabilidade das LP-SMA

"A lógica da validação permite nos movermos entre os limites do dogmatismo e do ceticismo." (Paul Ricoeur)

Neste capítulo são apresentados detalhes da avaliação da modularidade e estabilidade das duas LP-SMAs utilizadas como experimentos neste trabalho. Essas LP-SMAs foram desenvolvidas com o intuito de avaliar a modularidade e a estabilidade de diferentes técnicas de implementação no desenvolvimento e evolução. A primeira LP-SMA é a linha de produto do sistema *Expert Committee*, um sistema multi-agentes para o gerenciamento do processo de submissão e revisão de artigos de conferências e *workshops* (Seção 5.1). A segunda LP-SMA é o OLIS (*OnLine Intelligent Services*), uma aplicação *Web* que provê serviços para os usuários, tais como: anúncio de eventos e serviços de calendário (Seção 5.2). Ambas LP-SMAs seguiram a abordagem de desenvolvimento reativa de LPS, descrita no Capítulo 2. Essas LP-SMAs foram implementadas seguindo boas práticas de programação e largamente baseadas em padrões de projetos, a fim de atender requisitos de qualidade relevantes, como por exemplo, reusabilidade. O objetivo deste capítulo é fornecer comprovações empíricas da utilidade, dos benefícios e das armadilhas de se utilizar diferentes técnicas de implementação para o gerenciamento e modularização de *features* de agentes. Tais técnicas são: compilação condicional, arquivos de configuração e orientação a aspectos.

5.1
Primeiro Experimento

Esta seção descreve a LP-SMA do Expert Committee (EC). Inicialmente, na Seção 5.1.1, a LP-SMA do EC é descrita em termos do seu modelo de *features*, arquitetura, agentes e componentes que compõem o sistema. Depois, o formato do estudo experimental é apresentado na Seção 5.1.2. Por fim, os resultados da análise da estabilidade e modularidade são apresentados nas Seções 5.1.3 e 5.1.4. O objetivo deste primeiro experimento é comparar duas técnicas de implementação da LP-SMA do EC utilizando a plataforma JADE,

para implementação das *features* de agentes: (i) uma versão utilizando técnicas de compilação condicional; e (ii) outra versão utilizando técnicas de OA.

5.1.1 Linha de Produto de Sistemas Multi-Agentes do Expert Committee

O EC é um aplicação típica baseada na *Web*, cujo principal objetivo é o gerenciamento da submissão de artigos e o processo de revisão de conferências. O sistema EC fornece as funcionalidades necessárias para suportar o gerenciamento de conferências. A Tabela 5.1 lista todas as funcionalidades fornecidas pelo EC. Cada uma das funcionalidades oferecidas pelo EC é executada por um usuário específico do sistema, o qual pode assumir papéis específicos em uma conferência, tais como: (i) coordenador; (ii) chair da conferência; (iii) membro do comitê de programa; (iv) revisores; e (v) autores. Durante a evolução do EC, agentes de software foram introduzidos ao sistema com o objetivo de automatizar (ou semi-automatizar) funcionalidades existentes no sistema, como por exemplo: processo de revisão de artigos. A Figura 5.1 ilustra o modelo de *features* (Czarnecki 1998) do EC utilizando a ferramenta FMP (*Feature Modeling Plug-in*) (Antkiewicz and Czarnecki 2004).

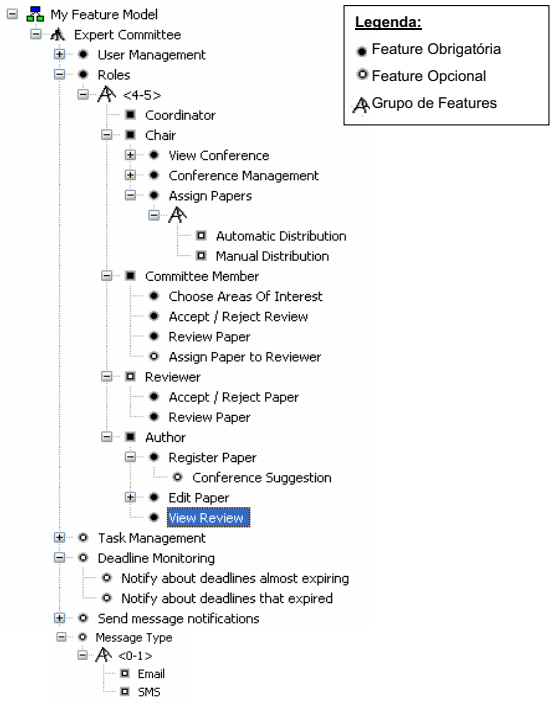


Figura 5.1: Modelo de Features Simplificado da LP-SMA do EC.

A arquitetura da LP-SMA do EC foi estruturada de acordo com o padrão em camadas (Buschmann et al. 1996), sendo composta das seguintes camadas:

Tabela 5.1: Funcionalidades do Expert Committee.

Papel	**Funcionalidade**
Coordenador	Criar Conferência
	Ver conferência
Chair	Definir os dados da conferência.
	Definir as áreas de interesse
	Definir os membros do comitê
	Definir os *deadlines*
	Distribuir artigos
	Notificar autores
	Ver conferência
Membro do comitê	Escolher as áreas de interesse
	Aceitar/Rejeitar artigo
	Distribuir artigo para um revisor
	Revisar artigo
Revisor	Aceita/Rejeita artigo
	Revisar artigo
Autor	Registrar artigo
	Editar artigo
	Ver revisão

(i) **Interface Gráfica do Usuário (GUI)** - esta camada é responsável por processar as requisições *Web* submetidas pelos usuários. Esta camada foi implementada usando o *framework* Struts (Foundation 2008);

(ii) **Negócios** - é responsável por estruturar e organizar os serviços de negócio fornecidos pelo sistema EC. O gerenciamento de transações dos serviços de negócio foi implementado usando os mecanismos de OA providos pelo *framework* Spring (Source 2008);

(iii) **Dados** - agrega as classes de acesso ao banco de dados do sistema, que são implementadas utilizando o padrão DAO (*Data Access Object*) (Alur et al. 2001). O *framework* Hibernate (Middleware 2006) foi utilizado para fazer a persistência dos objetos no banco de dados.

A Figura 5.2 ilustra a arquitetura da LP-SMA do EC destacando a arquitetura base. Na Figura 5.3 é detalhado o projeto OO da LP-SMA do EC. Para a implementação dos agentes de software, a plataforma JADE (JADE 2001) foi utilizada como plataforma base. Os agentes da LP são responsáveis por monitorar a execução de diferentes funcionalidades do núcleo do sistema EC para prover novas *features* de agentes. A integração entre a arquitetura *Web* e os agentes foi obtida através da introdução do padrão Observer (Gamma et al. 1995). Todos os serviços que fazem parte da camada de negócio são observáveis. Detalhes sobre cada agente contido na LP-SMA do EC são descritos abaixo:

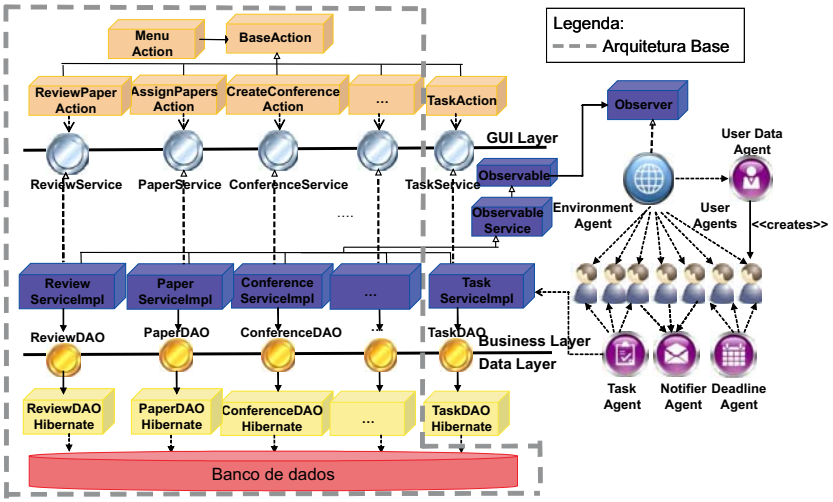


Figura 5.2: Arquitetura da LP-SMA do EC.

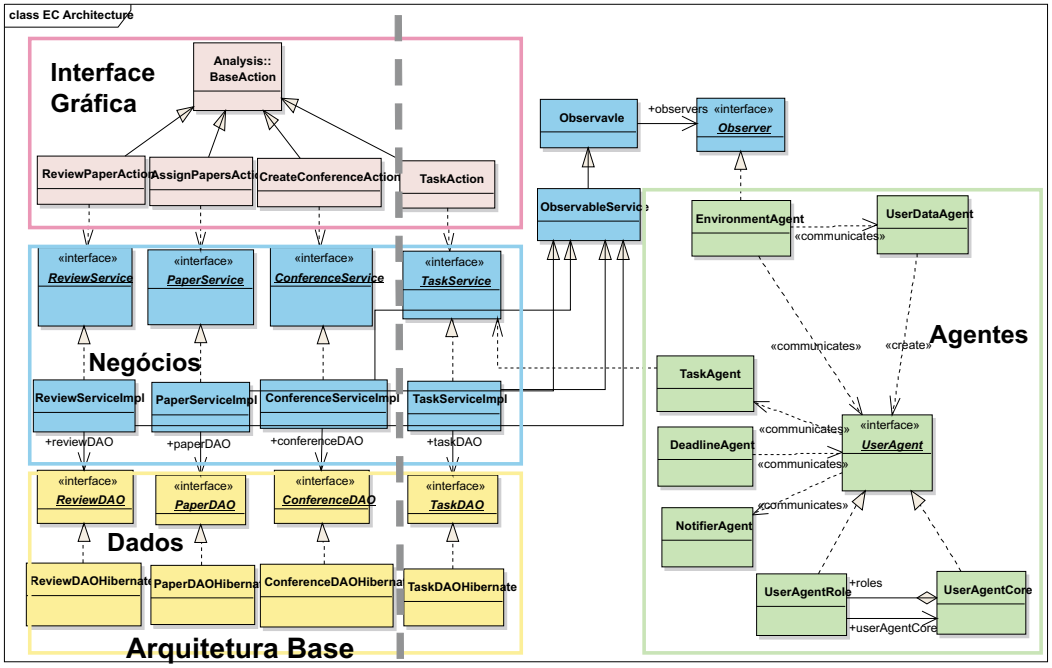


Figura 5.3: Arquitetura Detalhada da LP-SMA do EC.

- **Agente Ambiente (*Environment Agent*)** - este agente monitora o sistema EC por observar a execução de serviços de negócio específicos. Estes serviços monitorados do EC representam o ambiente no qual os agentes de usuário estão situados. O principal objetivo deste agente é notificar os outros agentes das mudanças ocorridas no sistema;
- **Agente de Dados do Usuário (*User Data Agent*)** - este agente recebe notificações quando novos usuários são criados no banco de dados. Quando isso acontece, este agente é responsável por criar um agente de usuário que será a representação do usuário no sistema;
- **Agente Usuário (*User Agent*)** - cada usuário armazenado no sistema tem um agente que o representa no sistema. A função deste agente é desenvolver ações que os usuários deveriam fazer. O agente usuário foi projetado de tal forma que novos papéis podem ser dinamicamente adicionados, tais como: chair, coordenador, autor. Cada papel do agente usuário é responsável por algumas ações específicas no sistema. Um exemplo deste tipo de comportamento autônomo é quando o *deadline* da submissão de artigos é encerrada. Após isso, o agente de usuário que possui o papel de chair da conferência distribuirá automaticamente os artigos para os membros do comitê. Além deste exemplo, a maioria dos agentes de usuário são responsáveis por analisar e descobrir pendências de tarefas baseada nos papéis que os usuários desempenham no sistema;
- **Agente de Prazos/Datas (*Deadline Agent*)** - este agente é responsável por monitorar os *deadlines* da conferência. Este monitoramento tem dois propósitos: (i) notificar os agentes de usuário quando o *deadline* está perto de se encerrar; e (ii) notificar os agentes de usuário quando o *deadline* já expirou;
- **Agente de Tarefas (*Task Agent*)** - este agente é responsável por gerenciar as tarefas dos usuários. Ele recebe pedidos para criar, remover e gerenciar a data de execução das tarefas. Esses pedidos são feitos pelos agentes de usuário;
- **Agente Notificador (*Notifier Agent*)** - este agente recebe notificações de outros agentes para enviar mensagens aos usuários do sistema. Na implementação atual, este envio de mensagens pode ser feito através de *e-mail* e SMS.

A maioria dos cenários de mudanças é relacionada à inclusão de comportamento autônomo, o que resulta na inclusão de agentes de software, de papéis ou comportamentos. A Tabela 5.2 apresenta as mudanças aplicadas em cada *release*. Basicamente, durante a evolução da LP-SMA do EC, três tipos

de variabilidade foram identificadas quando *features* opcionais e alternativas foram adicionadas:

(i) **Novas features para o gerenciamento de conferência**: a adição de novas funcionalidades relacionadas diretamente ao processo de gerenciamento da conferência resulta na inclusão de novas classes para processar a requisição do usuário. Um exemplo desse tipo de variabilidade é a inclusão da *feature* referente ao papel do revisor, a qual adiciona o suporte para que o membro do comitê delegue a revisão de um artigo para um revisor. Esse tipo de variabilidade é típico em LPS;

(ii) **Novos agentes de software**: diversos agentes de software foram incluídos na arquitetura da LP-SMA do EC (*releases* R3, R4, R5 e R7). Estes diversos agentes foram introduzidos com a finalidade de implementar comportamento autônomo relacionado a recomendações para pesquisadores (autores dos artigos), monitoramento dos *deadlines*, monitoramento de tarefas pendentes. A *feature* de gerenciamento de tarefas (*release* R7), por exemplo, implicou na inclusão de um novo agente no sistema com um conjunto de comportamentos associados, os quais podem estar presentes ou não, dependendo do produto a ser derivado;

(iii) **Novos comportamentos e papéis para o agente**: variabilidades internas são adicionadas aos agentes, tais como: novos comportamentos e papéis dos agentes. Estes tipos de *features* foram modularizadas como: (i) planos específicos a serem executados por um agente (*releases* R5 e R6); ou (ii) papéis específicos (*release* R3). A *feature* de sugestão de conferência (*release* R3) é um exemplo de tal *feature* opcional autônoma. O agente usuário ou mais especificamente o papel autor pode realizá-la. Quando um artigo é registrado na conferência, o papel autor do agente percebe o evento e envia sugestões de conferências relacionadas para o autor/autora que registrou o artigo.

5.1.2 Estrutura do Estudo Experimental

O estudo foi estruturado seguindo os princípios definidos por Wohlin et al. (Wohlin et al. 2000). O estudo foi estruturado da seguinte forma: definição do experimento, planejamento do experimento e operação do experimento.

Tabela 5.2: Cenários de Mudanças aplicados a LP-SMA do EC.

Releases	**Descrição**	**Tipo de Mudança**
R1	Arquitetura base do Expert Committee	
R2	Inclusão do papel Revisor.	Inclusão de *feature* opcional.
R3	Feature adicionada para incluir agentes de usuário (*User Agents*), incluindo o papel de autor. Nova feature para permitir a sugestão de conferências para os autores.	Inclusão de *features* opcionais.
R4	Inclusão do agente notificador (*Notifier Agent*) para enviar mensagens para os usuários do sistema através de e-mail e SMS.	Inclusão de *feature* opcional e alternativa.
R5	Inclusão do agente de *Deadlines* (*Deadline Agent*). Este agente é responsável por monitorar os *deadlines* da conferência	Inclusão de *feature* opcional.
R6	Inclusão da *feature* que permite o chair distribuir automaticamente os artigos para os membros do comitê. Extensão do agente de *deadlines* para permitir o lembrete dos deadlines.	Inclusão de *feature* opcional e extensão de *feature* opcional.
R7	Inclusão do agente de tarefas (*Task Agent*).	Inclusão de *feature* opcional.

Definição do Experimento

Inicialmente, seguindo o processo do experimento (Wohlin et al. 2000), foi elaborado a definição do experimento. O objetivo de realizar esta parte do processo é definir quais aspectos serão estudados neste experimento, antes do planejamento e execução. O modelo usado para formalizar a definição do experimento foi o GQM (*Goal-Question-Metric*)(Basili et al. 1986). Seguindo este modelo, os objetivos do experimento são:

Analisar as duas diferentes versões da LP-SMA do EC
Com o propósito de avaliar as técnicas de programação usando a plataforma JADE
Com respeito a modularidade e estabilidade
Do ponto de vista desenvolvedor/pesquisador
No contexto de estudantes da pós-graduação do Laboratório de Engenharia de Software da PUC-Rio.

Planejamento do Experimento

Seleção do Contexto e Seleção dos Participantes. O estudo foi executado de forma *off-line*, pois não é um estudo desenvolvido na indústria. Este estudo é classificado como quasi-experimento, pois apresenta ausência de randomização tanto na seleção dos objetos quanto na seleção dos participantes.

A realização do estudo experimental envolveu apenas uma estudante de mestrado da PUC-Rio.

Seleção das Variáveis: O próximo passo é a definição das variáveis do estudo, variáveis independentes e dependentes. As variáveis independentes são aquelas variáveis que podem ser controladas e mudadas no experimento. As variáveis dependentes são aquelas que precisam ser estudadas, mediante a manipulação das variáveis independentes, que neste caso são a modularidade e a estabilidade. No estudo do EC, as variáveis independentes são: (i) implementação OO com técnicas de compilação condicional usando JADE; e (ii) implementação OA usando JADE. Nesse estudo foram utilizadas um conjunto de métricas para quantificar a modularidade em termos dos seguintes atributos: separação de *concerns*, interação entre *concerns*, tamanho, coesão e acoplamento, que foram descritas no Capítulo 4. Esse estudo também compreendeu as seguintes métricas de impacto de mudanças (Yau and Collofello 1985). A proposta de usar estas métricas é avaliar os efeitos da propagação em termos de componentes, linhas de código e operações durante a inclusão de *features* de agentes na LP-SMA do EC. Estas métricas já foram empiricamente validadas em outros estudos, o que permite que o estudo tenha uma comparação mais objetiva com os resultados de estudos anteriores, como pode ser visto em outros trabalhos descritos no Capítulo 7.

Projeto do Experimento: Quando se vai projetar os experimentos, alguns fatores precisam ser considerados, que são os fatores e os tratamentos. O fator neste experimento é a LP-SMA do EC e os tratamentos são duas versões desenvolvidas da mesma LP-SMA, que são: JADE com técnicas de compilação condicional e JADE com técnicas de OA.

Operação do Experimento

O estudo foi dividido em quatro fases maiores: (i) projeto e implementação das *releases* da LP-SMA do EC usando as duas técnicas de implementação; (ii) avaliação da modularidade das *releases* da LP-SMA; (iii) medição dos impactos de mudanças; e (iv) análise dos dados.

A primeira versão desenvolvida para o sistema EC corresponde à implementação da arquitetura base da LPS. Tendo essa arquitetura base desenvolvida, foram aplicados um conjunto de cenários de mudanças, adicionando novas *features* à arquitetura da LPS, as quais são consideradas opcionais e alternativas. Assim, o sistema foi sendo incrementalmente evoluído, tendo como resultado a implementação de sete produtos (*releases*) da LP-SMA do EC. Cada novo produto da LP-SMA foi sempre implementado em duas versões diferentes: (i) uma codificada na linguagem Java utilizando técnicas de compi-

lação condicional; e (ii) a outra codificada em AspectJ (Laddad 2003). Cada nova *release* foi implementada baseada na versão anterior. Por exemplo, a *release* 2 OO representa a evolução da *release* 1. Sendo assim, essas diferentes versões do EC podem ser consideradas uma família de produtos, os quais compartilham características comuns e possuem alguns pontos de variação. Dessa forma, a evolução do EC transformou-se em uma LP-SMA, modelada de forma que é possível gerar os produtos das versões anteriores rapidamente.

As *releases* de ambas versões (OO e OA) foram implementadas de acordo com os mesmos princípios de reusabilidade e estabilidade, seguindo as mesmas decisões de projeto (Buschmann et al. 1996, Clements and Northrop 2002). O objetivo de garantir a padronização das versões OO e OA são: (i) garantir que ambas versões estejam implementando as mesmas funcionalidades; e (ii) manter o mesmo estilo de implementação de ambas versões.

A Figura 5.4 ilustra o diagrama de classes parcial da implementação OO da LP-SMA do EC. A Figura 5.5 mostra o diagrama parcial do projeto OA, ilustrando um subconjunto de módulos que foram afetados pelos aspectos durante a evolução da LP-SMA do EC. Porém, na implementação OA não houve alteração nos aspectos durante a evolução da LP-SMA, porque novos aspectos foram adicionados para implementar uma determinada *feature*. O estereótipo *«aspect»* na Figura 5.5 representa um aspecto do sistema. A seta de dependência representa que um aspecto afeta a estrutura das classes do sistema. As classes e aspectos foram marcados com uma seqüência de *Rs*. Isto indica se a classe ou aspecto foi adicionado ou modificado durante a implementação da *release* X.

5.1.3 Análise da Estabilidade

Esta seção descreve a análise da estabilidade relacionados aos impactos de mudanças das versões da LP-SMA do EC. O objetivo dessa seção é medir quantitativamente as propagações de mudanças ao longo das *releases*. A análise da estabilidade é baseada nas métricas definidas por (Yau and Collofello 1985): número de componentes adicionados e modificados (aspectos e classes); número de operações adicionadas e modificadas; e número de linhas código adicionadas e modificadas. A Tabela 5.3 ilustra a propagação das mudanças através das *releases* da LP-SMA do EC.

De acordo com a Tabela 5.3, para todas as *releases* o número de componentes adicionados na solução OA foi superior a implementação OO. Durante o desenvolvimento das *releases* R3, R4 e R7, existiu um aumento significante no número de componentes adicionados na versão OA quando

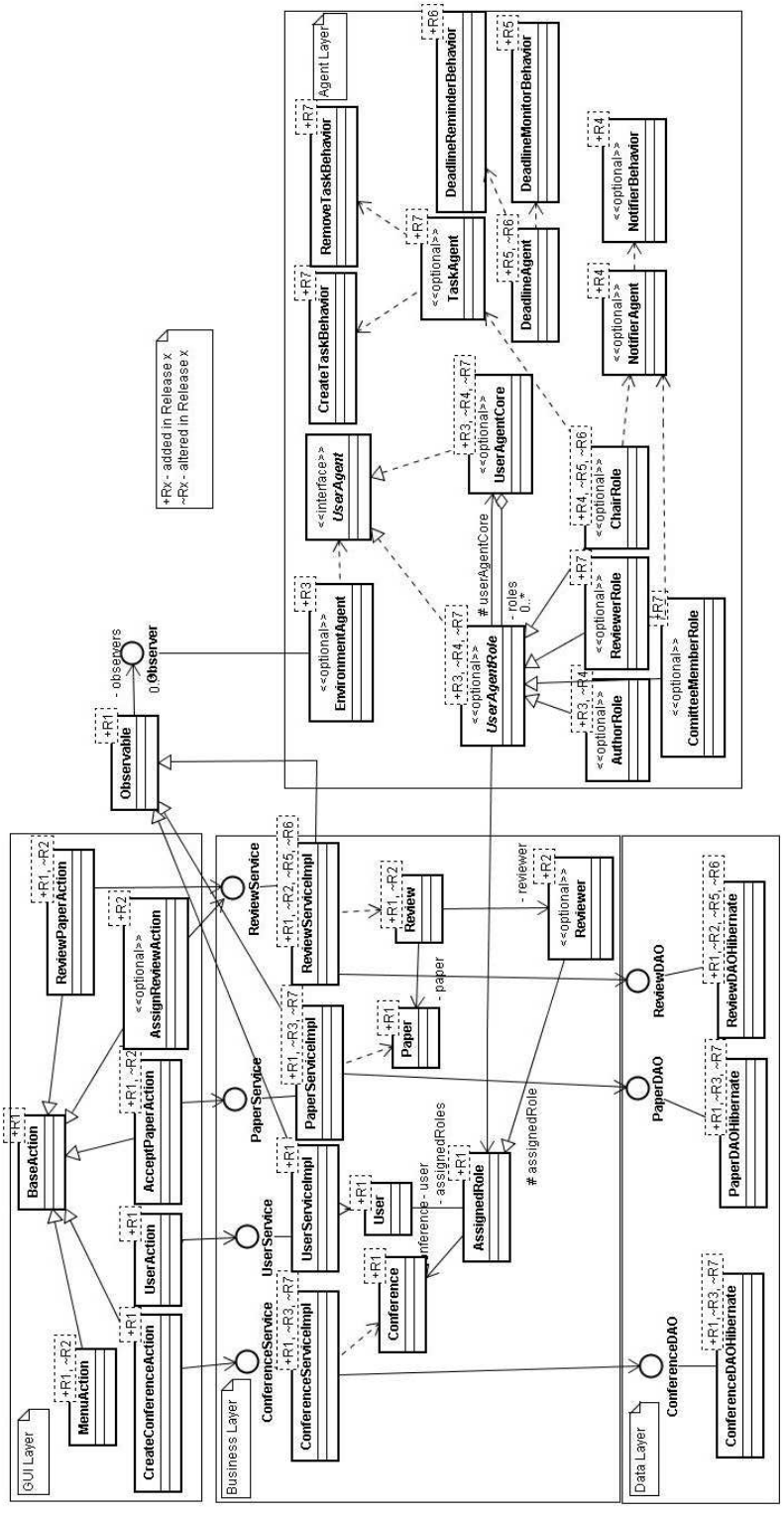


Figura 5.4: Projeto Simplificado OO da LP-SMA do EC.

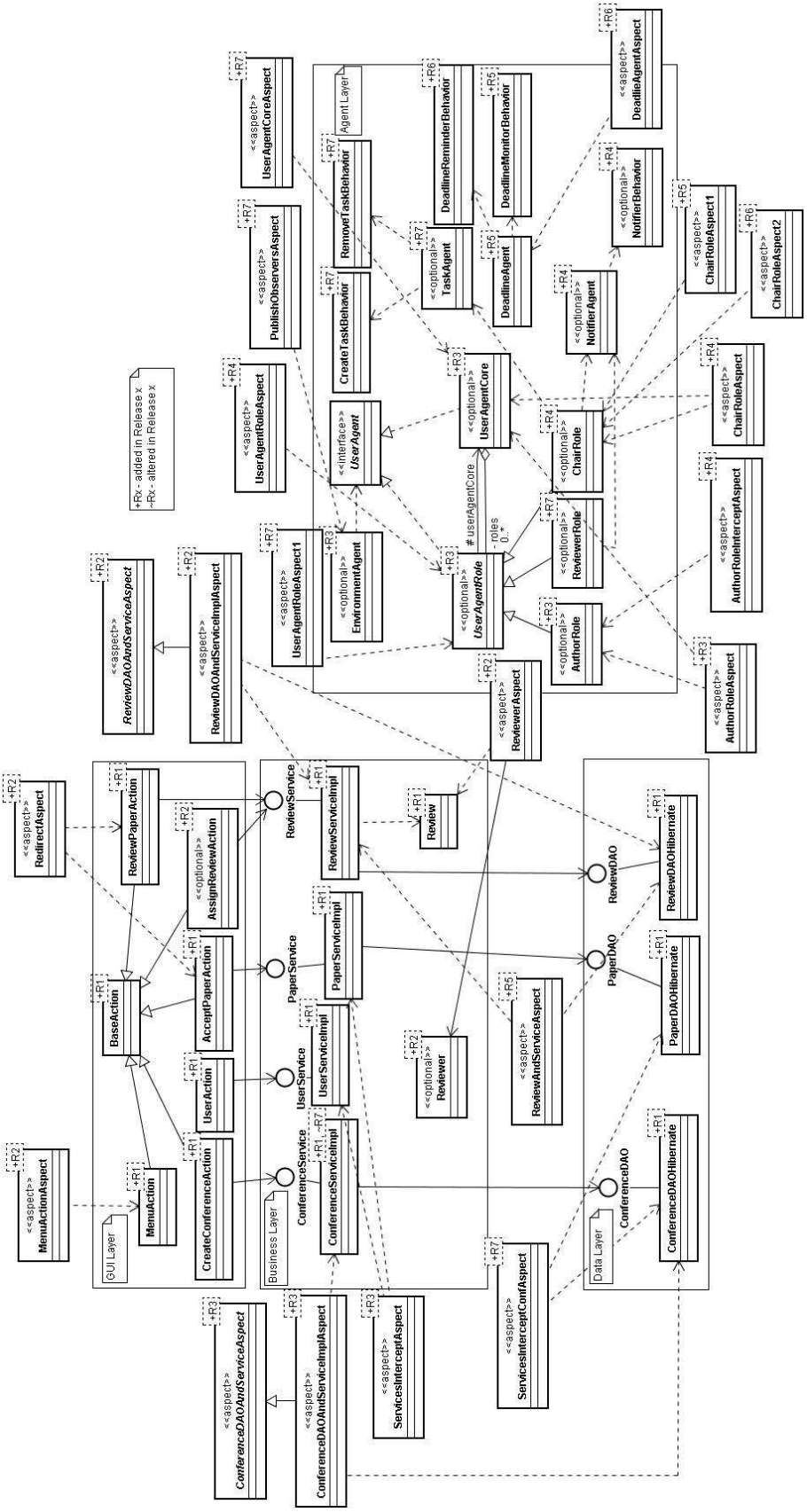


Figura 5.5: Projeto Simplificado OA da LP-SMA do EC.

Tabela 5.3: Propagação das mudanças nas releases do EC.

			R2	*R3*	*R4*	*R5*	*R6*	*R7*
Componentes	Adicionados	OO	3	27	11	3	3	26
		OA	9	35	20	6	8	34
	Modificados	OO	9	6	8	8	7	13
		OA	0	0	0	0	0	0
Operações	Adicionadas	OO	32	103	31	29	20	128
		OA	43	112	35	49	27	145
	Modificadas	OO	4	2	15	2	2	31
		OA	0	0	0	0	0	0
Pointcuts	Adicionados	OA	5	7	9	1	1	19
	Modificados	OA	0	0	0	0	0	0
Linhas de Código	Adicionadas	OO	418	1134	639	391	249	2203
		OA	511	1202	784	470	496	2166
	Modificadas	OO	0	0	0	0	0	0
		OA	0	0	0	0	0	0

comparada com a solução OO. Isto aconteceu, porque a partir da R3 toda a estrutura de agentes foi implementada, usando a plataforma JADE. Além da forma de programação especificada pelo JADE, na solução OA aspectos foram adicionadas para interceptar os componentes que implementam os agentes. Isto ocorreu para permitir a separação de *concerns* no nível de papéis e porque a inclusão do agente de tarefas traz impacto em diversos componentes. Assim, diversos aspectos precisaram ser criados para permitir a inclusão/remoção destas *features*, provendo assim um melhor gerenciamento de variabilidades. Na R7, o agente de tarefas foi adicionado, e junto com ele uma série de classes associadas para manipular os eventos específicos dos usuários na criação e remoção das tarefas e observação da datas de execução das tarefas.

É possível observar que para todas as *releases* na solução OA não houveram mudanças nos componentes (classes e aspectos) existentes. Isto ocorreu porque somente novos aspectos foram adicionados para afetar as classes existentes e implementar as novas *features* incluídas. Este é um fator positivo para preservar a arquitetura durante a evolução da LP-SMA do EC. Por outro lado, na solução OO houveram mudanças nos componentes existentes ao longo das *releases*. Isto ocorreu porque com o uso da técnica de compilação condicional, foram necessários a inclusão dos operadores *AND* and *OR* nos componentes existentes a fim de permitir a combinação das *features*. Por exemplo, na *release* R7, 13 componentes foram modificados na solução OO, enquanto que na solução OA não foram necessárias mudanças. Com relação ao número de operações adicionadas, a solução OA apresentou valores maiores em todas as *releases*. A razão disto é porque as mudanças que são feitas nos componentes pertencentes ao núcleo da LP-SMA são conseguidas através do

uso de declarações inter-tipos e *advices*. Como conseqüência de um número maior de operações e componentes incluídos na solução OA, o número de linhas de código foi sempre superior a solução OO em todas as *releases*, exceto na *release* R7.

Uma observação interessante nos dados colhidos é a ausência de linhas de código modificadas em ambas as versões, OO e OA. Basicamente, isto ocorreu porque a arquitetura proposta através da adoção de padrões de projeto para ambas as versões facilitaram a inclusão de novas *features* com mínimo impacto. Esta característica tornou o projeto das versões da LP-SMA do EC e implementação fácil de evoluir. Assim, linhas de código já implementadas anteriormente não precisam ser modificadas, somente novas linhas de código foram incluídas para implementar uma determinada *feature* opcional ou alternativa. Assim, os resultados colhidos para a métrica de estabilidade de linhas de código modificadas tem mostrado que o uso de boas práticas OO e OA e padrões de projeto podem beneficiar a estabilidade do projeto e facilitar o desenvolvimento e evolução de LP-SMA, pois reduz as mudanças no código fonte ao se incluir novas *features*.

Em relação aos *pointcuts* e operações modificados, não houve alterações na solução OA. Porém, na solução OO, algumas operações foram modificadas para permitir a implementação da *feature*. As mudanças não aplicadas aos componentes, operações e *pointcuts* na solução OA confirmam uma maior aderência ao princípio *Open-Closed* (Meyer 2000).

5.1.4 Análise da Modularidade Multi-Releases

Para a análise da modularidade, o estudo foi organizado da seguinte forma. Um conjunto de *features* foi selecionado para avaliar a modularidade das implementações das versões da LP-SMA do EC (OO e OA). Sendo assim, para cada *feature* selecionada, esta foi avaliada ao longo de todas as *releases* da LP-SMA. Esta seção apresenta os resultados obtidos com a aplicação das métricas referentes aos atributos de modularidade da LP-SMA do EC.

Separação de *Features* de Agentes

Esta subseção apresenta os resultados obtidos das métricas de *concerns* (SoC), que no contexto deste trabalho são as *features*. Para a coleta das métricas de *concern*, foram selecionadas as principais *features* que representam abstrações de SMA (papéis e agentes) que modularizam *features* de agentes da LP-SMA. Durante o desenvolvimento e evolução da LP-SMA do EC, o

comportamento destas *features* foi analisado ao longo das *releases* (Tabela 5.2).

A Figura 5.6 apresenta os resultados das métricas de *concern* para o papel Revisor, que é uma *feature* opcional adicionada na *R2* (Tabela 5.2). Os resultados mostram que o papel Revisor está espalhado em alguns componentes (classes e aspectos) e operações (métricas CDC e CDO) e entrelaçado com algumas *features* na implementação OA (métrica CDLOC). Isto indica que a implementação OA foi mais efetiva na modularização desta *feature* quando comparada com a implementação OO. Isto ocorreu porque na solução OA os trechos de código responsáveis por implementar este papel opcional é transferido das classes para um conjunto de classes dedicadas e um ou mais aspectos "cola". Na implementação OA de LPS, aspectos geralmente desempenham um excelente papel como código cola entre o código base e as *features* opcionais (Alves et al. 2005, Kulesza et al. 2006b). A técnica de compilação condicional, adotada na implementação OO, não possui esta habilidade porque tem um efeito um pouco intrusivo no código, devido a necessidade de adicionar cláusulas *#ifdef* e *#endif* em lugares onde há intersecção de *features*.

Na implementação OO, a *feature* Revisor está espalhada em um conjunto de classes, como por exemplo: `Reviewer`, `Review`, `ReviewPaperAction`, `AssignReviewAction` e `ReviewDAOHibernate` (Figura 5.4). Note, que na Figura 5.4, estas classes foram modificadas na R2, por causo disto elas estão marcadas com o símbolo R2. As mudanças nas classes mencionadas acima foram necessárias para introduzir código relacionado ao papel Revisor. Note também, que a classe `Review` possui uma associação direta com a classe `Reviewer`. A classe `Reviewer` foi adicionada na R2 e está totalmente dedicada a implementação do papel Revisor. Na implementação OA (Figura 5.5), parte do papel Revisor é implementado pela classe `Reviewer` e três aspectos: `ReviewerAspect`, `RedirectAspect` e `ReviewDAOAndServiceImplAspect`. Estes aspectos introduzem o comportamento do papel Revisor nas classes `Review`, `ReviewPaperAction`, `AssignReviewAction` e `ReviewDAOHibernate`, as quais não possuem qualquer código relacionado ao papel Revisor. Esta é a principal razão para a diminuição do grau de espalhamento e entrelaçamento na solução OA, refletido nas métricas de *concern*. Note que estas quatro classes não foram modificadas na R2 da implementação OA (Figura 5.5). O aspecto `ReviewerAspect` é responsável por adicionar o atributo *Reviewer* e alguns métodos usando construções inter-tipos da linguagem AspectJ. Este aspecto funciona como código "cola" entre as classes `Review` e `Reviewer`. O aspecto `ReviewDAOAndServiceImplAspect` é responsável por afetar os métodos de

acesso ao negócio e banco de dados que manipulam o objeto revisor. Este aspecto afeta métodos das classes `ReviewServiceImpl` e `ReviewDAOHibernate` para tratar as diferentes formas as revisões feitas por um revisor ou membro do comitê de programa. Está é a principal razão para a diminuição do grau de espalhamento e entrelaçamento na solução OA, refletidas das métricas de *concern*.

Por exemplo, o Código 5.1 mostra um trecho de código do aspecto `ReviewAndServiceAspect` que trata a aceitação da revisão de um artigo. Se a revisão é aceita por membro do comitê de programa, a execução procede normalmente de acordo com o método `acceptPaper(Review)` já definido na classe `ReviewDAOHibernate`. Caso contrário, se a revisão é aceita por um revisor, o método `acceptPaper(Review)` não é executado. Assim, o uso de OA não exigiu realizar quaisquer mudanças invasivas nas classes núcleo da LP-SMA para implementar a *feature* revisor. Por outro lado, na implementação OO com compilação condicional, o código do revisor foi diretamente inserido na classe `ReviewDAOHibernate` através de #ifdef/#endif, como pode ser visto no Código 5.2.

Código 5.1: Implementação da Aceitação da Revisão com POA.

```
1  public privileged aspect ReviewAndServiceAspect {
2      pointcut acceptPaper(...):
3              execution(*.ReviewDAOHibernate.acceptPaper(Review)) &&
4                              args(review) && target(...);
5      around(...) : acceptPaper(...) {
6          if (review.getCommiteeMemberAccepted() != null) {
7              proceed(review);
8          } else {
9              if (review.getReviewerAccepted() != null) {
10                 if (review.getReviewerAccepted().booleanValue()) {
11                     persistentReview.setReviewerAccepted(Boolean.
                           TRUE);
12                 }
13             }
14         }
15     }
16 }
```

Código 5.2: Implementação da Aceitação da Revisão com compilação condicional.

```
1  public void acceptPaper(...) throws RegisterException {
2      Review persistentReview = this.read(review.getId());
3      if (review.getCommiteeMemberAccepted() != null) {
4          if (review.getCommiteeMemberAccepted().booleanValue()) {
5              persistentReview.setCommiteeMemberAccepted(Boolean.
                   TRUE);
6          }
7      }
8      \#ifdef REVIEWER
9       if (review.getReviewerAccepted() != null) {
10         if (review.getReviewerAccepted().booleanValue()) {
11             peristentReview.setReviewerAccepted(Boolean.TRUE);
12         }
13     }
14     \#endif
15 }
```

Na Figura 5.6 é possível ver que o entrelaçamento da *feature* Revisor com outras *features* é muito maior na implementação OO (métrica CDLOC). Por outro lado, o nível de espalhamento da *feature* Revisor no que diz respeito as operações (métrica CDO) e componentes (métrica CDC) é um pouco maior na implementação OO. Isto ocorreu porque apesar dos aspectos modularizarem e isolarem a *feature* Revisor na implementação OA, esta *feature* não apresenta pedaços de código similares em diferentes classes, como por exemplo, o *concern Logging* exemplificado no Capítulo 3.4.

Nas implementações OO e OA da R7 houve um aumento significativo em todas as métricas porque a inclusão da *feature* Agente de Tarefas inclui diversas classes de eventos para se comunicar com a *feature* Revisor. Além disso, também houve mudanças em outras classes que representam os papéis e se comunicam com a *feature* Revisor, tais como: chair da conferência e membro do comitê de programa.

A Figura 5.7 mostra os resultados das métricas de *concern* para a *feature* Agentes de Usuário, a qual também é uma *feature* opcional. Para esta *feature*, os valores coletados na implementação OA não apresentaram melhores resultados quando comparados com a implementação OO em termos das métricas CDO e CDC. O número de operações para implementar a *feature* Agentes de Usuário aumentou ao longo da evolução da LP-SMA, porque novas operações foram adicionadas no código base da LP-SMA do EC com o intuito de habilitar os aspectos afetarem os *join points* da LP-SMA do EC. A Figura 5.4 mostra que na implementação OO a *feature* Agentes de Usuário

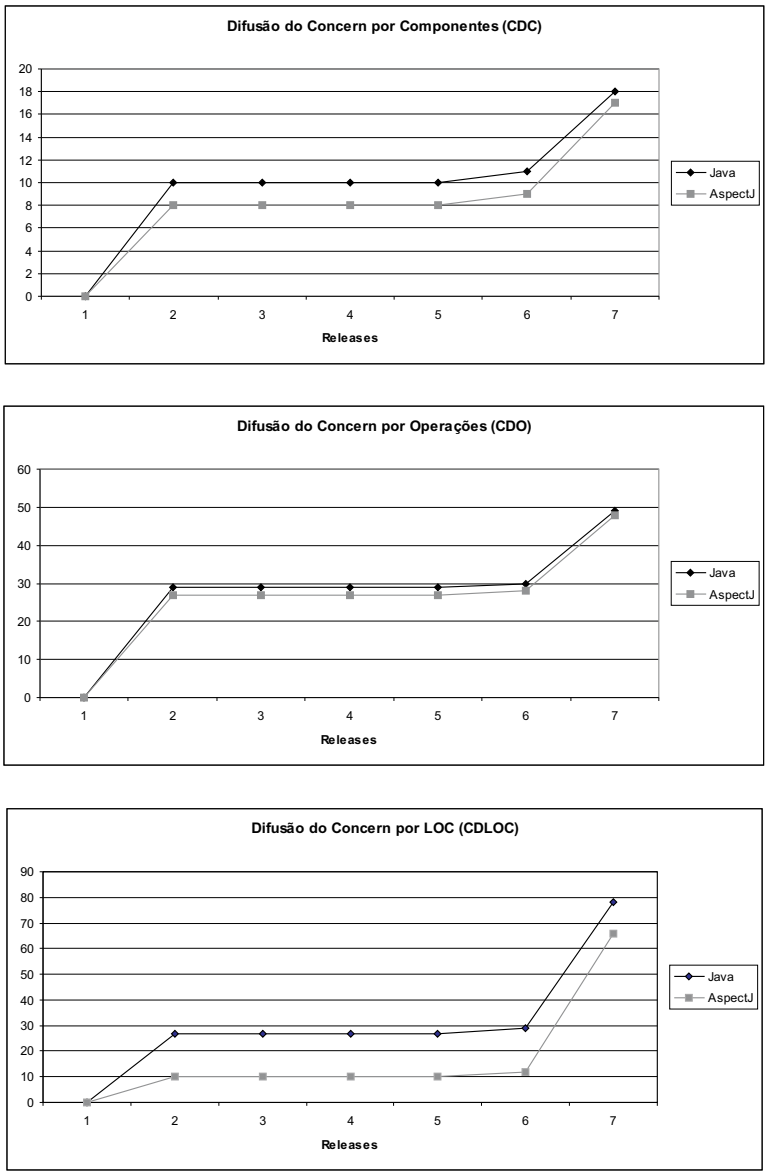


Figura 5.6: Métricas de concern para a feature Revisor.

está espalhada somente em alguns componentes (classes e/ou aspectos). Isto acontece porque na implementação OO com compilação condicional, somente é necessário adicionar as cláusulas *#ifdef* e *#endif* localmente em algumas classes. Assim, o grau de espalhamento apresenta valores baixos na solução OO. As classes `UserAgent`, `UserAgentCore` e `UserAgentRole` foram adicionadas na R3 e estão totalmente dedicadas a implementação da *feature* Agentes de Usuário.

Note que na Figura 5.4, as classes adicionadas na R3 são modificadas durante a evolução da LP-SMA. Tais mudanças nestas classes aumentam o nível de entrelaçamento (métrica CDLOC), como pode ser visto na Figura 5.7. Na implementação OA (Figura 5.5), parte da *feature* Agentes de Usuário é implementada pelas classes `UserAgent`, `UserAgentCore`, `UserAgentRole` e um conjunto de aspectos que afetam os papéis especificados: `AuthorRoleAspect`, `ChairRoleAspect`. Por causa disto, o grau de espalhamento é alto na solução OA (métrica CDC). Este número alto de aspectos pode ser visto como um ponto negativo para a solução OA, porque o entendimento do código da *feature* pode ser mais custoso, devido ao fato de existirem mais componentes. Note que as classes `UserAgentCore`, `UserAgentRole`, `AuthorRole` e `ChairRole` não são modificadas durante a evolução da LP-SMA do EC (Figura 5.5).

A Figura 5.8 mostra os resultados da métricas da *feature* Agente Notificador. Durante a evolução da LP-SMA, houve um aumento significativo no número de componentes na implementação OA a partir da *release* R5. Isto ocorreu porque na implementação OO o código de notificação dos eventos do sistema especificado foi codificado diretamente nas classes usando cláusulas *#ifdef/#endif*. Por outro lado, na implementação OA, diversos aspectos foram criados para interceptar as classes existentes. Esses aspectos foram criados porque cada um deles está relacionado a um papel específico do agente usuário. Sendo assim, estes aspectos precisam ser gerenciados separadamente para garantir a fácil inclusão/remoção da *feature* opcional que a representa. Portanto, na implementação OA o código foi modularizado em um conjunto de aspectos, tais como: `AuthorRoleInterceptAspect` e `ChairNotifierServices` (Figura 5.5). Como conseqüência desse alto número de componentes, o número de operações (métrica CDO) também é superior na solução OA. Entretanto, apesar desse alto número operações e componentes, o nível de entrelaçamento é menor na solução OA. Isto porque como cada aspecto está relacionado a um papel em específico, o nível de acoplamento é reduzido.

Os resultados da *feature* Agente de Prazos/Datas foram similares aos resultados da *feature* Agente Notificador, como pode ser visto na Figura 5.9.

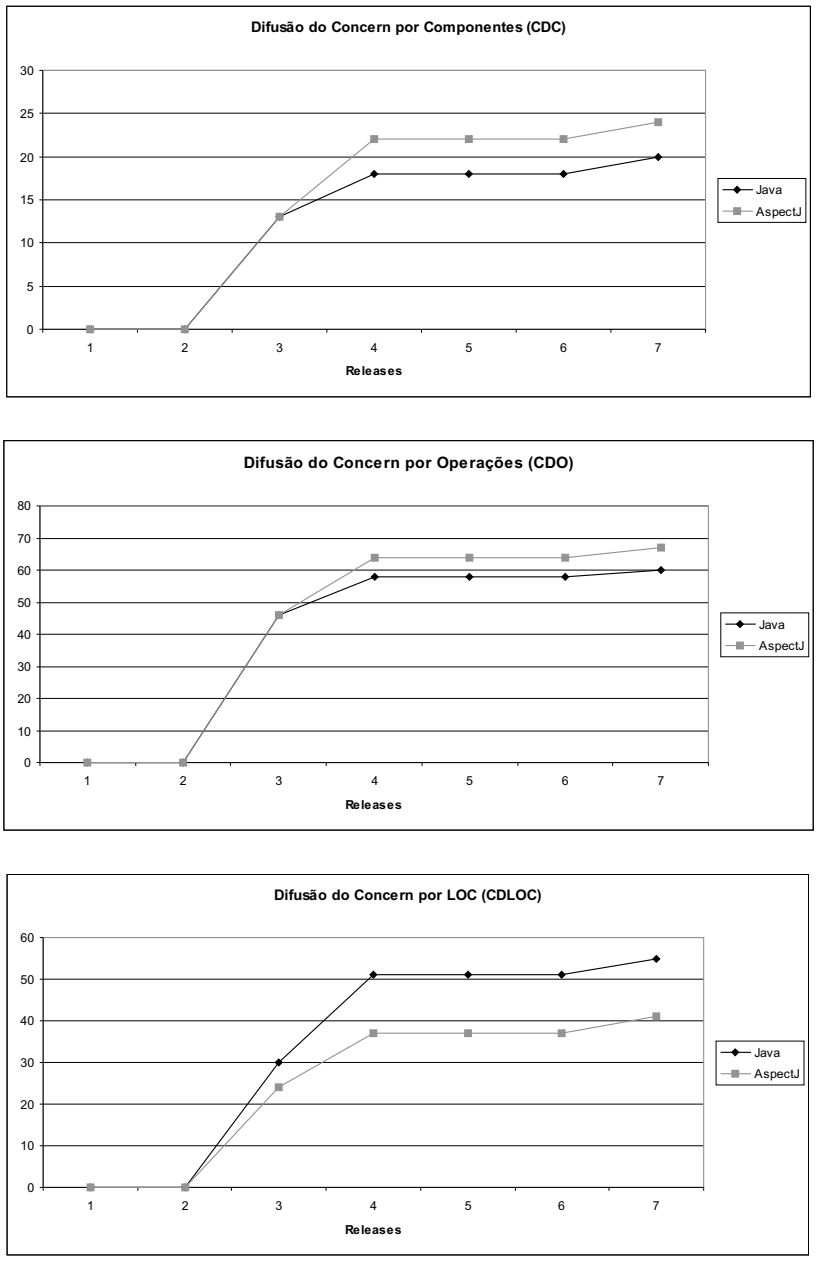


Figura 5.7: Métricas de concern para a feature Agentes de Usuário.

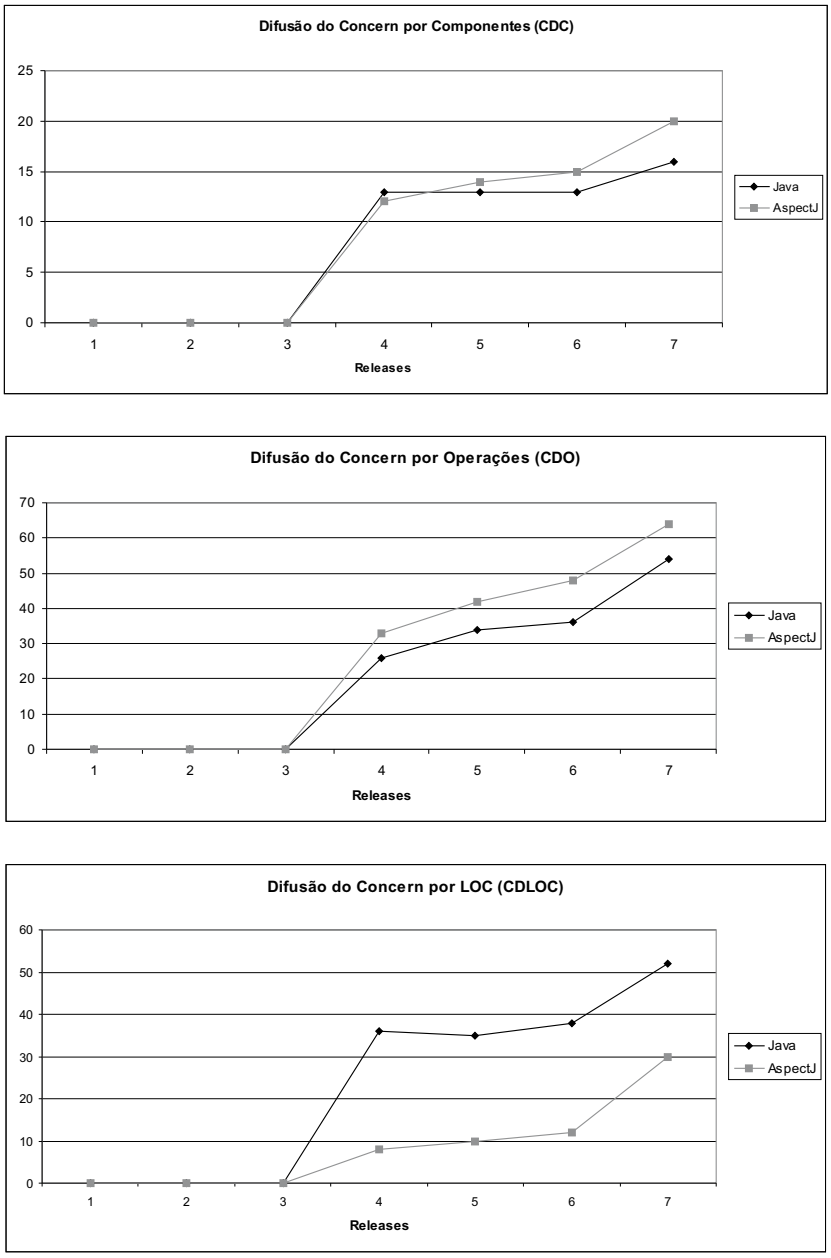


Figura 5.8: Métricas de concern para a feature Agente Notificador.

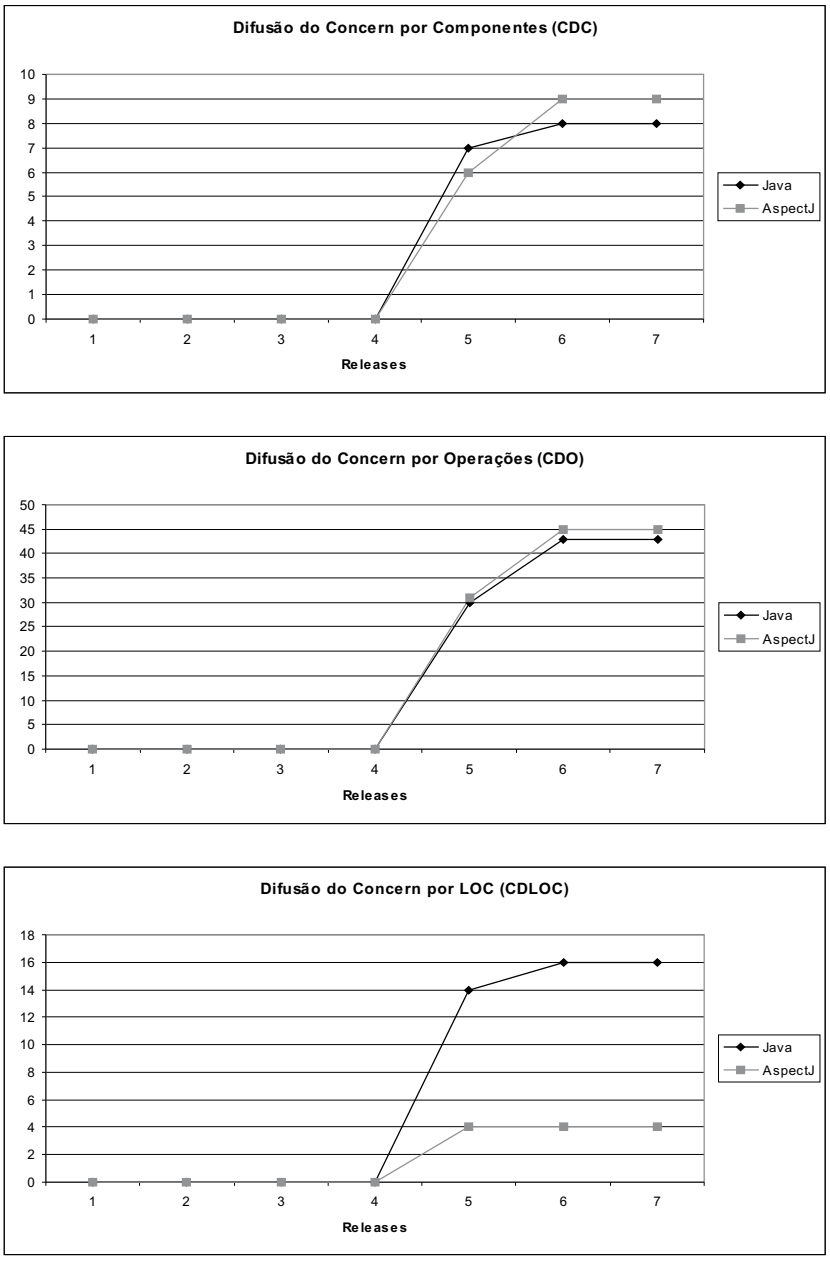


Figura 5.9: Métricas de concern para a feature Agente de Prazos/Datas.

Análise de Dependência de Features

As Figuras 5.10 e 5.11 mostram os resultados da métrica CIBC (*Component-level Interlacing Between Concerns*) (Sant'Anna et al. 2007) para as *features* Revisor e Agente de Usuário. O objetivo desta métrica é quantificar a interação de um determinado *concern* ou *feature* com outros *concerns*. De acordo com a Figura 5.10, a *feature* Revisor está entrelaçada com alguns *concerns* da LP-SMA do EC na implementação OO (Papéis: Autor, Chair, Membro do Comitê de Programa, Coordenador; *ACLMessage*, *Persistence*, *Review* e *MessageFactory*). Na implementação OA, esta *feature* se encontra entrelaçada com alguns *concerns*, porém os aspectos conseguiram minimizar este entrelaçamento. Isto ocorreu porque a implementação OA transferiu quase todos os elementos para implementar a *feature* Revisor das classes `Reviewer`, `Review`, `ReviewPaperAction`, `AssignReviewAction` e `ReviewDAOHibernate` (Figura 5.4) para os aspectos `ReviewerAspect`, `ReviewDAOAndServiceImplAspect` e `RedirectAspect` (Figura 5.5). Isto contribuiu para separar esta *feature* dos outros *concerns*. Na Figura 5.11 o grau de interação da *feature* Agente de Usuário com outros *concerns* também apresentou valores baixos na solução OA. Enquanto na solução OO a intersecção entre os *concerns* é obtida através da inclusão de *#ifdef/#endif*, na solução OA usamos declarações inter-tipos para conseguir o entrelaçamento no nível de componentes.

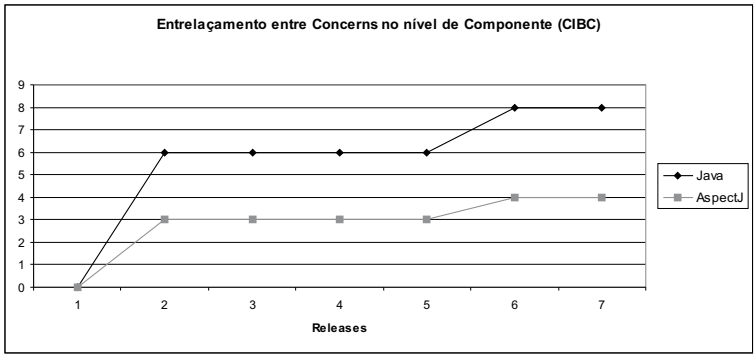


Figura 5.10: Métrica CIBC para a feature Revisor.

Acoplamento e Coesão

Na Figura 5.12 é exibido o resultado da métrica de acoplamento da LP-SMA do EC. A média de acoplamento na solução OA apresentou um melhor resultado que na solução OO. Observando o gráfico, a solução OA apresentou valores mais estáveis para o acoplamento. Apesar de muitos aspectos reduzirem o acoplamento das classes do sistema por modularizar as *features* ao longo da

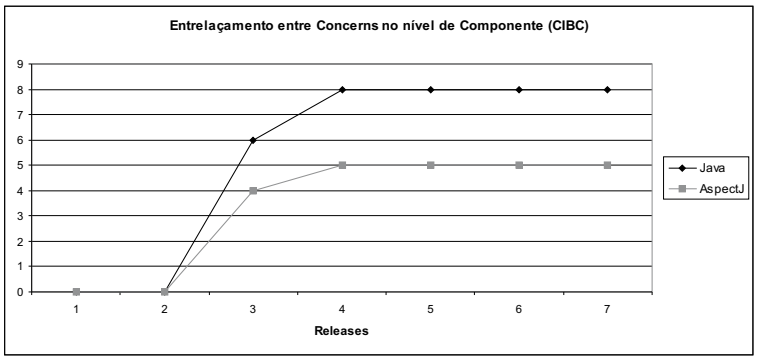


Figura 5.11: Métrica CIBC para a feature Agentes de Usuário.

evolução da LP-SMA, eles ainda precisam manter referências a algumas classes. Um dos motivos dessa referência é utilização de declarações inter-tipos para permitir o entrelaçamento entre as componentes (classes e aspectos). Porém, como a solução OA apresentou mais componentes (Figura 5.14(a)), essa maior quantidade de componentes proporcionou a produção de classes e aspectos mais desacoplados. Um dos exemplos de alto acoplamento é a inclusão do padrão role (Bäumer et al. 1997) no projeto OO. Na implementação do padrão role, cada papel acessa métodos de diversas classes, enquanto que na solução OA os papéis ficam melhor modularizados em aspectos, diminuindo assim o acoplamento.

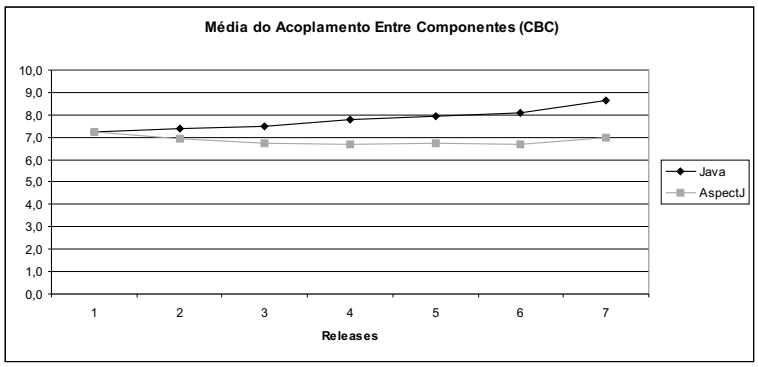


Figura 5.12: Média de Acoplamento do EC.

A Figura 5.13 apresenta a média de coesão da LP-SMA ao longo dos cenários de evolução. De acordo com o cálculo da métrica LCOO, apresentada no Capítulo 4, um valor baixo para LCOO indica uma alta coesão, enquanto que um valor alto indica uma falta de coesão. De acordo com a Figura 5.13, a coesão é melhor obtida na solução OA. Um dos fatores que contribuiu para esta alta coesão na solução OA foi a implementação do padrão Observer com aspectos (Hannemann and Kiczales 2002).

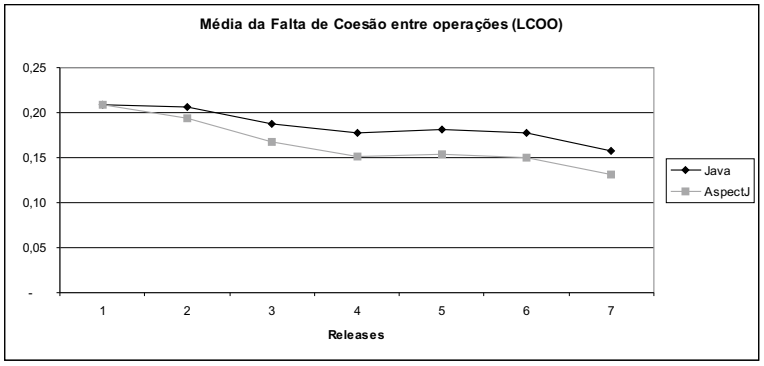


Figura 5.13: Média de Coesão do EC.

Por exemplo, quando um novo usuário é armazenado no sistema, o Agente Ambiente divulga essa informação, o Agente de Dados do Usuário percebe-a e cria um novo agente (Agente Usuário) para representar o usuário no sistema. Essa notificação é feita pelo aspecto `ServiceInterceptAspect`, que afeta o método `store()` da classe `UserServiceImpl` e cria um evento de criação do usuário para notificar o Agente Ambiente. Na Figura 5.5, é possível ver que o aspecto afeta algumas classes de implementações de serviços: `PaperServiceImpl` e `UserServiceImpl`. De acordo com este diagrama, o código do padrão Observer é inserido de uma forma transparente pelo aspecto `ServiceInterceptAspect`. Uma razão para os melhores resultados na solução OA em termos da métrica LCOO é que na implementação OO, as classes contêm métodos e atributos tanto da sua implementação original quanto relacionado ao padrão Observer. Além disso, a implementação OO está mais acoplada com as classes do padrão (CBC).

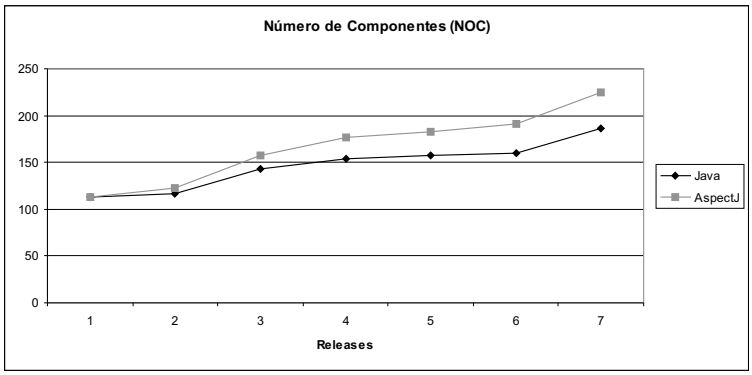
O Código 5.3 mostra um trecho do aspecto `ServiceInterceptAspect`, que concretiza o relacionamento observacional no Agente Ambiente. Primeiro, o *after advice* `createUser` é chamado para armazenar o usuário no banco de dados. Após a criação do usuário, papéis podem ser dinamicamente adicionados ao agente. Por exemplo, o papel autor é designado a um agente quando um usuário submete algum artigo na conferência. Assim, depois da submissão do artigo, o Agente Ambiente divulga esse evento e a classe *UserAgentCore* percebe e designa o papel de autor. Isto pode ser visto no Código 5.3 através do *after advice submitPaper*. Com este tipo de implementação usando POA é que os valores da métrica LCOO são mais baixos na solução OA.

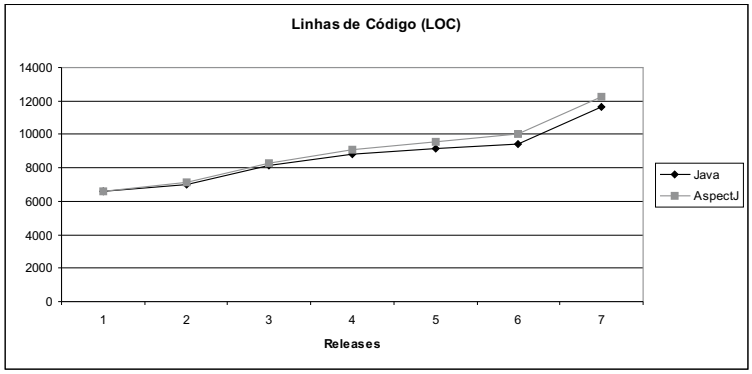
Código 5.3: Propagação de Eventos usando o padrão Observer.

```
1  public aspect ServicesInterceptAspect {
2      pointcut createUser(...): execution(* UserServiceImpl.store(
           User)) &&
3                  args(entity) && target(...);
4      pointcut submitPaper(...): execution(PaperServiceImpl.store(
           Paper)) &&
5                  args(entity) && target(...);
6      after(...): submitPaper(ent, paperService) {
7          Event event = new CreateEvent(EventTarget.PAPER,"paper",
               entity);
8          paperService.fireEventPerformed(event);
9      }
10     after(...): createUser(entity, userService) {
11         Event event = new CreateEvent(EventTarget.USER_DATA,"user"
               ,entity);
12         userService.fireEventPerformed(event);
13     }
14 }
```


Tamanho

A Figura 5.14 apresenta os resultados das seguintes métricas de tamanho da LP-SMA do EC: Linhas de Código (LOC), Número de Componentes (NOC) e Número de Operações (NOO). Os resultados mostraram que os valores coletados para a implementação OA foram maiores quando comparados com a implementação OO. Isto aconteceu principalmente porque a maioria dos aspectos criados durante a evolução são heterogêneos. Um aspecto heterogêneo é aquele que afeta múltiplas classes e *join points* de diferentes formas por introduzir diferentes comportamentos em cada um deles. Isto causa a criação de muitos aspectos (aumento dos valores coletados para a métrica NOC), em que cada um deles possui diferentes *advices* e *pointcuts* afetando as classes do sistema. Por este motivo, as métricas NOO e LOC são maiores na implementação OA. Na implementação OO, o uso de compilação condicional com a adição dos operadores *OR* e *AND* nas classes existentes foram suficientes para prover a combinação de determinadas *features*, tais como: Agentes de Usuário (*User Agents*) e Agente Notificador (*Notifier Agent*). A técnica de compilação condicional fica espalhada em todas as classes onde existe a intersecção entre as *features* e o código base da LP. Já a implementação OA exigiu a criação de novos aspectos heterogêneos para representar a combinação das *features*, como por exemplo, a criação desses dois aspectos: `AuthorRoleAspect` e `AuthorRoleInterceptAspect` (Figura 5.5).





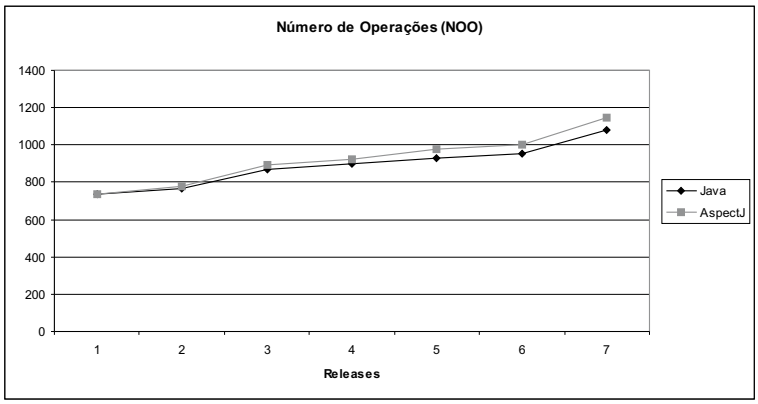


Figura 5.14: Métricas de Tamanho da LP-SMA do EC.

5.2 Segundo Experimento

Esta seção descreve o contexto e os resultados do segundo experimento. O objetivo deste segundo estudo é a comparação de duas plataformas para desenvolvimento de SMA (JADE e Jadex) e duas técnicas de implementação para gerenciamento das variabilidades: arquivos de configuração e aspectos. Na LP-SMA OLIS, o modelo de *features* e sua arquitetura em termos dos componentes e agentes que compõem o sistema são apresentados na Seção 5.2.1. O formato do segundo experimento é apresentado na Seção 5.2.2. Os resultados das métricas de análise da estabilidade e modularidade são apresentados nas Seções 5.2.3 e 5.2.4.

5.2.1 Linha de Produto de Sistemas Multi-Agentes OLIS

O OLIS (*Online Intelligent Services*) é uma LP-SMA para o domínio *Web* que provê diversos serviços pessoais para os usuários. O núcleo da arquitetura do OLIS é composta principalmente por dois serviços: anúncio de eventos e os serviços de calendário. O serviço de anúncio de eventos permite o usuário anunciar eventos para os outros usuários do sistema através de um quadro de eventos. O serviço de calendário permite o usuário agendar eventos no seu calendário. O OLIS foi projetado de tal forma a permitir que o sistema possa ser evoluído para incorporar novos serviços sem interferir nos serviços já existentes. O sistema pode gerenciar diversos tipos de eventos: eventos genéricos, eventos acadêmicos e eventos de viagem.

Durante o desenvolvimento e evolução do OLIS, nós aplicamos uma série de cenários de mudanças, adicionando *features* obrigatórias, alternativas e opcionais na arquitetura do OLIS. Estas novas *features* foram inseridas gradualmente no núcleo do sistema *Web*, caracterizando a LPS e habilitando sua customização: (i) trabalhar com diferentes tipos de eventos; e (ii) permitir que os agentes de usuário negociem eventos e ofereçam recomendações aos usuários. Estas mudanças foram originalmente pré-definidas pelos desenvolvedores. Após o desenvolvimento da primeira versão da aplicação *Web* do OLIS, foram identificadas que novas *features* com comportamento autônomo poderiam ser incluídas para automatizar algumas tarefas no sistema. O desenvolvimento da LP-SMA OLIS também seguiu a abordagem de desenvolvimento reativa, apresentada no Capítulo 2. A Figura 5.15 ilustra o modelo de *features* da LP-SMA OLIS.

A LP-SMA OLIS foi estruturada de acordo com o padrão arquitetural em camadas, nos mesmos moldes do EC, apresentado na Seção 5.1. As camadas

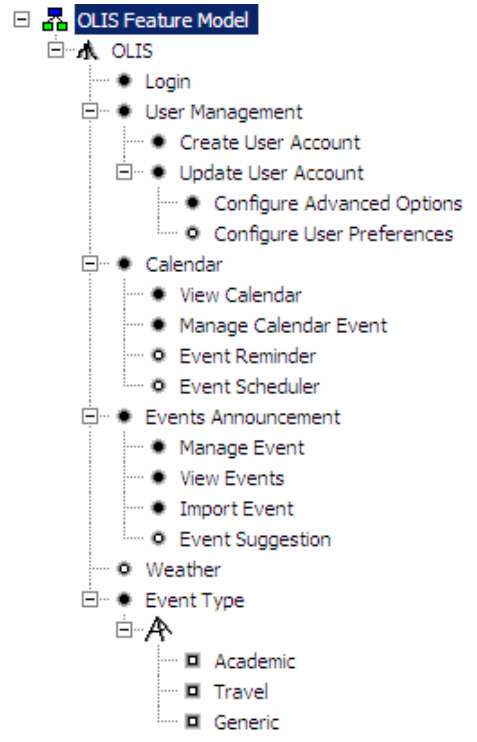


Figura 5.15: Modelo de features do OLIS.

e respectivas responsabilidades são exatamente as mesmas do EC - GUI, Negócios e Dados. A motivação de desenvolver as duas arquiteturas de forma semelhante foi permitir a derivação de um padrão arquitetural a partir dos dois estudos, que foi publicado em (Nunes et al. 2008b). A Figura 5.16 apresenta a arquitetura da LP-SMA OLIS, com uma linha delimitando quais componentes pertencem a arquitetura núcleo. As novas *features* incorporadas no núcleo do OLIS são:

(i) **Lembrete de Eventos** - o usuário configura o tempo (minutos) que ele quer ser lembrado antes dos eventos. Assim, o sistema envia mensagens para notificar o usuário sobre tais eventos;

(ii) **Agenda de Eventos** - quando o usuário adiciona um novo evento no seu calendário que involve mais participantes, o sistema verifica se existe conflito com eventos já existentes. Se sim, o sistema sugere uma nova data para o evento que seja apropriada de acordo com todos os horários dos participantes;

(iii) **Sugestão de Eventos** - quando um novo evento é anunciado, o sistema automaticamente recomenda o evento para outros usuários baseado nas suas preferências. Ele também considera se o tempo está apropriado de acordo com o tipo de lugar onde o evento vai acontecer;

(iv) **Tempo** - este é um novo serviço que provê informações sobre as condições de tempo atuais e as previsões do local. Este serviço é também usado pelo sistema para recomendar eventos de viagem anunciados.

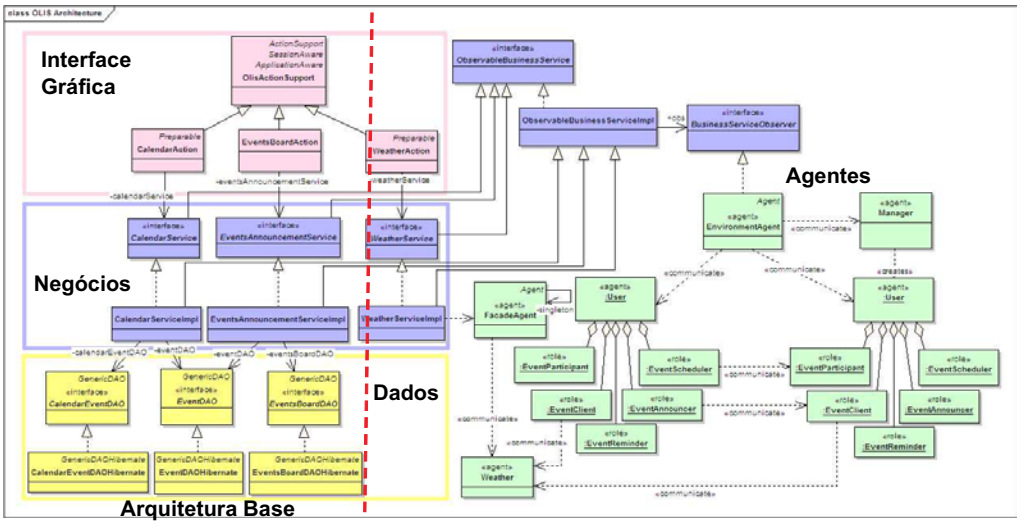


Figura 5.16: Arquitetura da LP-SMA OLIS.

A evolução da LP-SMA OLIS foi feita através da inclusão de agentes de software e seus respectivos papéis na arquitetura. Os agentes que compõem a LP-SMA OLIS são:

(i) **Agente Ambiente (Environment Agent)** - percebe mudanças no modelo de dados e propaga para os outros agentes. O comportamento deste agente é similar ao Agente Ambiente do EC;

(ii) **Agente Fachada (Facade Agent)** - recupera informações dos agentes para os serviços de negócio, é a fachada entre a aplicação *Web* e os agentes. Com este agente é possível omitir a existência dos outros agentes da aplicação *Web*. Assim, a aplicação só sabe da existência do *Facade Agent* para conseguir informações dos outros agentes;

(iii) **Agente de Tempo (Weather Agent)** - provê informações sobre o tempo e as previsões do tempo;

(iv) **Agente Gerenciador (Manager Agent)** - inicia e gerencia os agentes usuários quando o sistema inicia ou quando um novo usuário é inserido no sistema;

(v) **Agente Usuário (User Agent)** - cada usuário tem um agente que o representa no sistema. A função deste agente é desenvolver ações que o usuários deveriam fazer. Cada agente usuário é composto por papéis, os quais implementam *features* de agentes. Os papéis são:

(i) **Lembrete de Eventos** - lembra os usuários que o eventos está para acontecer;
(ii) **Agenda de Eventos** - convida outros usuários para um determinado evento em um horário compatível com a agenda de todos;
(iii) **Participante de Eventos** - aceita ou rejeita um convite de participação de um evento. No caso de uma rejeição, informa um horário apropriado de acordo com sua agenda;
(iv) **Anunciador de Eventos** - anuncia novos eventos para os outros agentes usuários;
(v) **Cliente de Eventos** - verifica se o evento anunciado é interessante de acordo com as preferências do usuário.

As Figuras 5.17, 5.18 e 5.19 apresentam os diagramas de classes simplificado das implementações JADE OO, JADE OA e Jadex da LP-SMA OLIS respectivamente. Essas Figuras também ilustram os principais componentes que foram afetados durante a evolução da arquitetura da LP-SMA. Os componentes foram marcados com uma seqüência de *Rs* para indicar se o componente foi adicionado ou modificado durante a implementação da *release* X.

Durante a evolução da LP-SMA OLIS, foram gerados oito produtos (*releases*). Na Tabela 5.4 são apresentadas as mudanças em cada *release*. Os cenários compreendem diferentes tipos de mudanças, inclusão de *features* obrigatórias, alternativas e opcionais. A maioria dos cenários de mudanças são relacionados a inclusão de *features* de agentes. Além das variabilidades presentes nas LPS e das já citadas na LP-SMA EC, que são: agentes e os papéis dos agentes, foi introduzida um novo tipo de variabilidade na LP-SMA OLIS, que é o conceito de capacidades (*capabilities*). Uma capacidade (Padgham and Lambrix 2000) é na verdade um conjunto de planos, isto é, um fragmento da base do conhecimento que é manipulado por aqueles planos e uma especificação da interface para a capacidade.

5.2.2 Estrutura do Estudo Experimental

O segundo estudo experimental também foi estruturado seguindo os princípios definidos por Wohlin et al. (Wohlin et al. 2000). A estrutura do estudo foi feita da seguinte forma: definição do experimento, planejamento do experimento e operação do experimento.

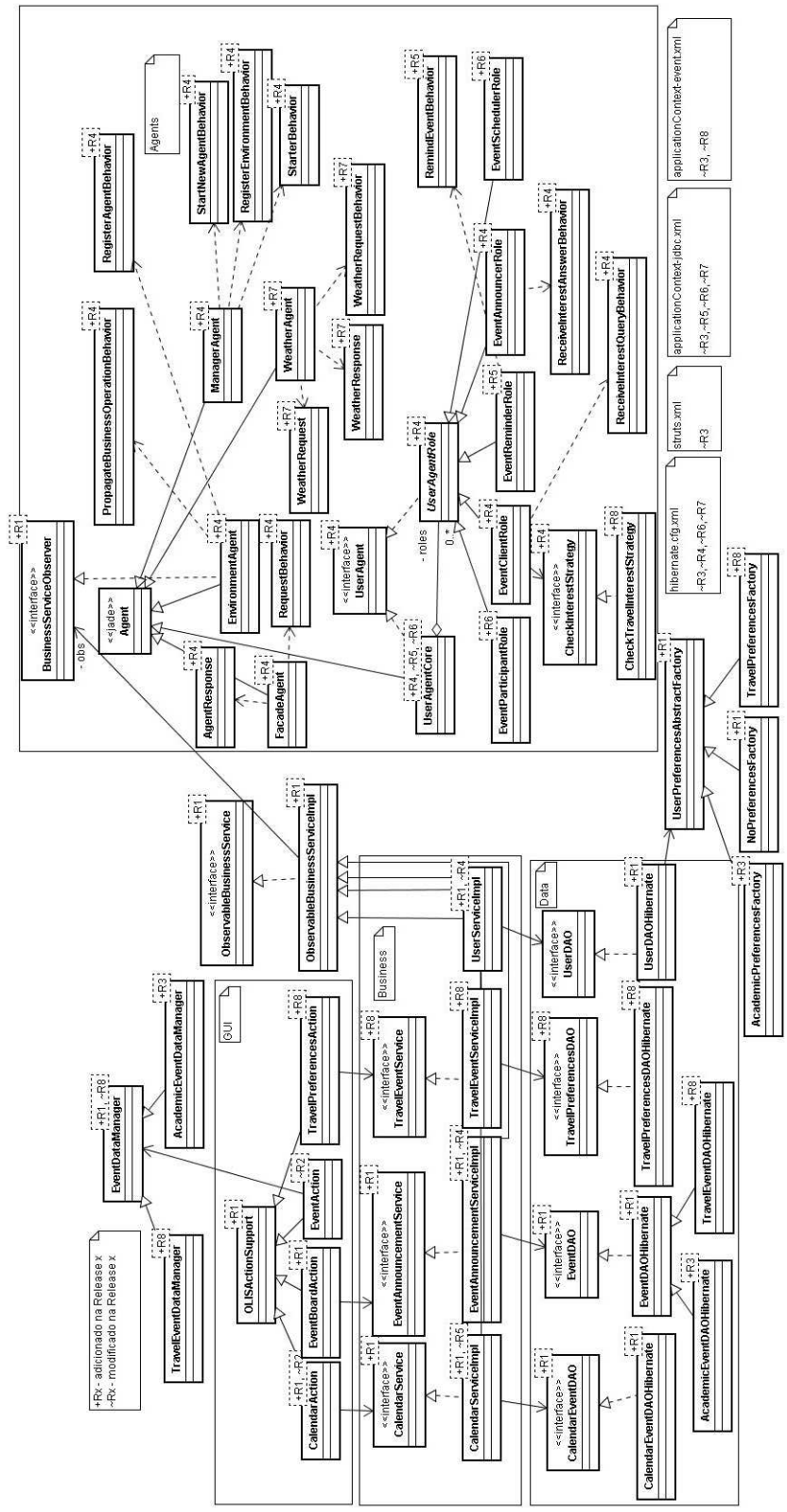


Figura 5.17: Arquitetura Simplificada da LP-SMA OLIS usando JADE.

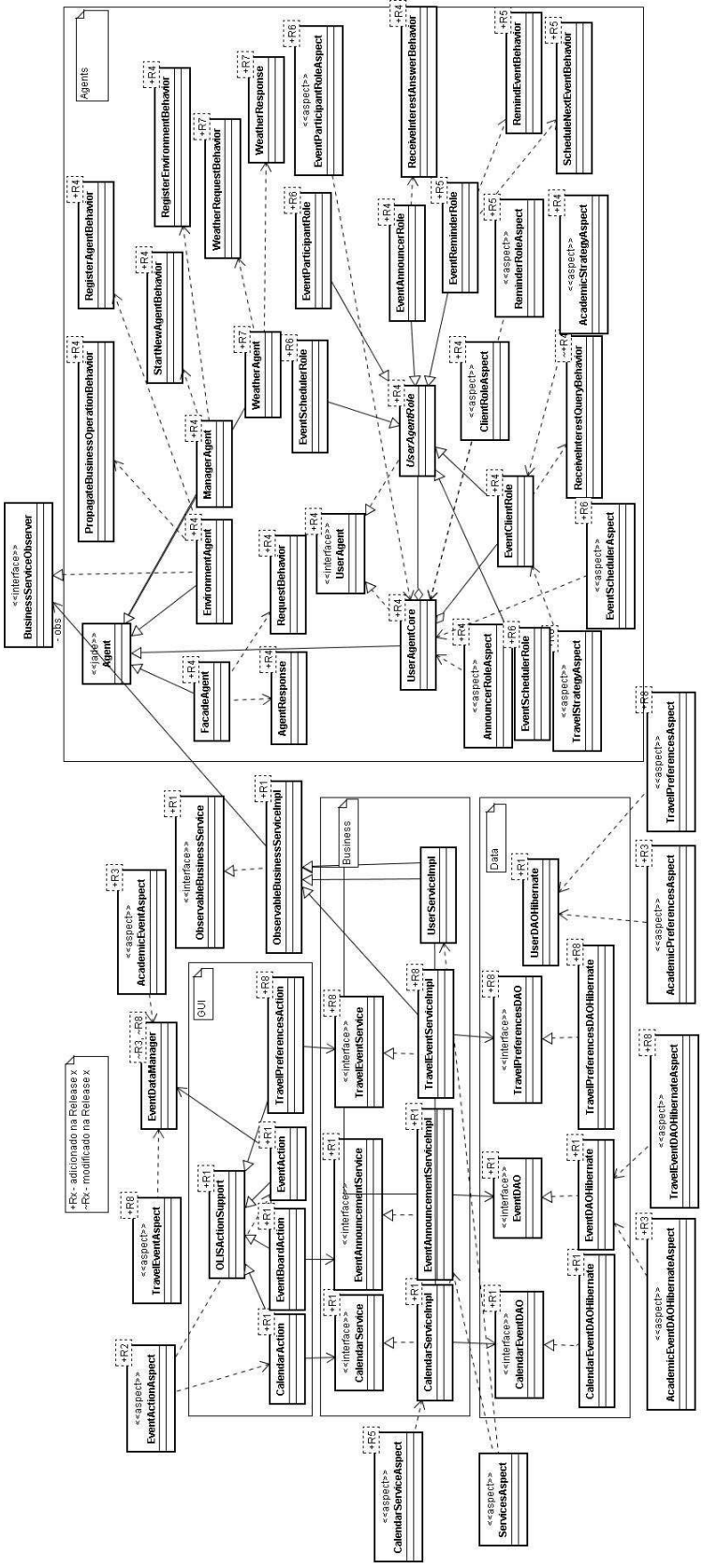


Figura 5.18: Arquitetura Simplificada da LP-SMA OLIS OA usando JADE.

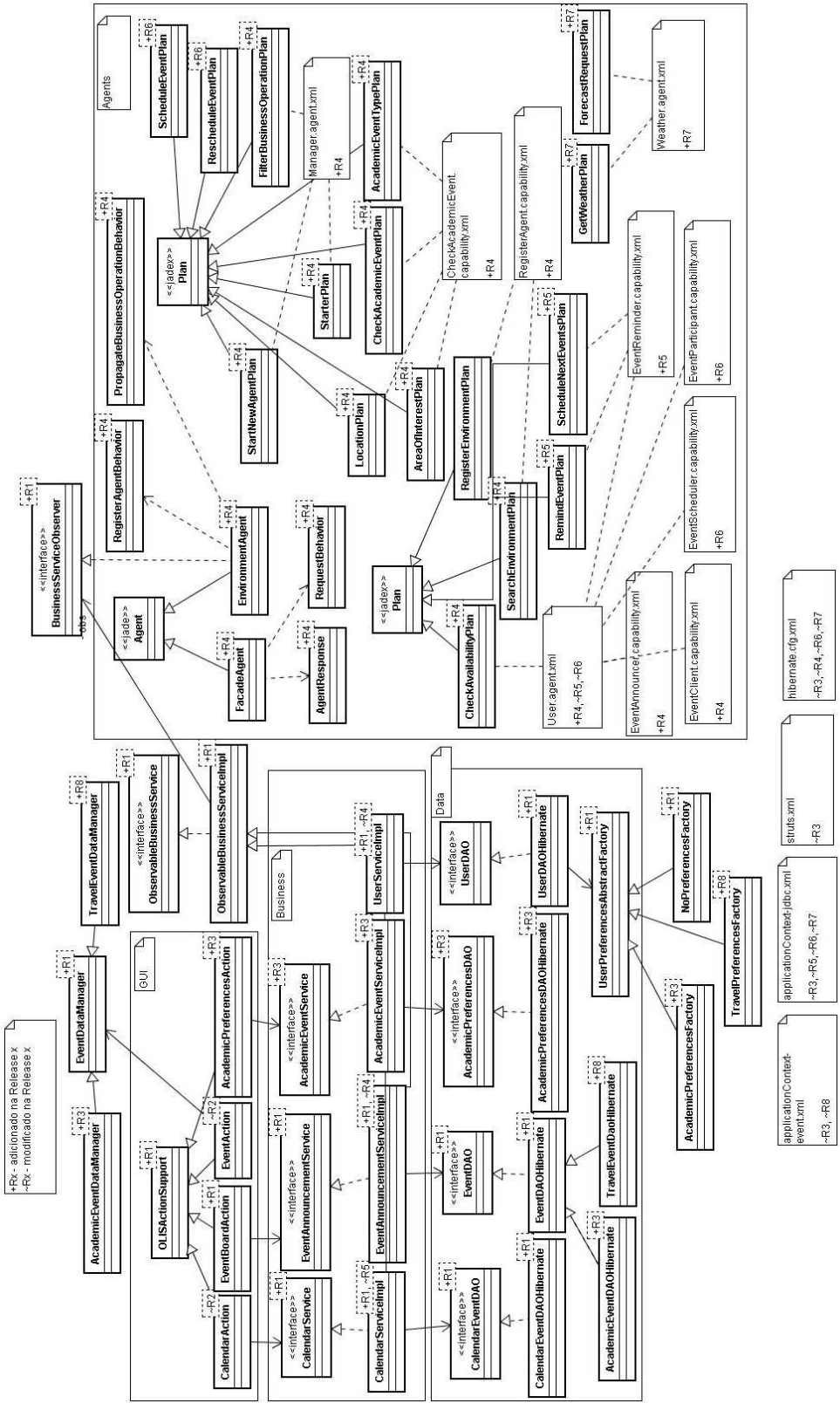


Figura 5.19: Arquitetura Simplificada da LP-SMA OLIS usando Jadex.

Tabela 5.4: Releases e Cenários de Mudanças aplicados a LP-SMA OLIS.

Releases	**Descrição**	**Tipo de Mudança**
R1	Núcleo do OLIS - Sistema *Web*	
R2	Nova *feature* adicionada para permitir o usuário importar eventos para o seu calendário.	Inclusão de *feature* obrigatória.
R3	*Feature* adicionada para permitir o suporte a eventos acadêmicos.	Inclusão de *feature* alternativa.
R4	Inclusão do papel de Sugestão de Eventos para sugerir eventos acadêmicos.	Inclusão de *feature* opcional.
R5	Inclusão do papel de lembrete de eventos.	Inclusão de *feature* opcional.
R6	Inclusão do papel de agenda de eventos.	Inclusão de *feature* opcional.
R7	Inclusão do agente de tempo.	Inclusão de *feature* opcional.
R8	*Feature* adicionada para permitir o suporte a eventos de viagem.	Inclusão de *feature* alternativa.

Definição do Experimento

Semelhante ao primeiro estudo da LP-SMA do EC, a definição do experimento também foi baseada no modelo GQM (*Goal-Question-Metric*)(Basili et al. 1986). Seguindo este modelo, os objetivos do experimento são:

Analisar as três diferentes versões da LP-SMA OLIS
Com o propósito de avaliar técnicas de programação utilizando as plataformas JADE e Jadex
Com respeito a modularidade e estabilidade
Do ponto de vista desenvolvedor/pesquisador
No contexto de estudantes da pós-graduação do Laboratório de Engenharia de Software da PUC-Rio.

Planejamento do Experimento

Seleção do Contexto e Seleção dos Participantes. O estudo foi executado de forma *off-line* e também é classificado como quasi-experimento. Os participantes envolvidos no desenvolvimento da LP-SMA OLIS foram duas estudantes de mestrado da PUC-Rio. Os participantes possuem boa experiência na área de Engenharia de Software, incluindo o projeto e desenvolvimento de SMA utilizando plataformas específicas. Porém, a realização do estudo experimental envolveu apenas uma estudante de mestrado da PUC-Rio.

Formulação das hipóteses: As hipóteses do estudo a serem testadas são: (i) Hipótese Nula, H0: Não existe diferença nos atributos de modularidade

e estabilidade das três implementações da LP-SMA OLIS (versões JADE, Jadex e JADE com OA); (ii) Hipótese Alternativa, H1: A versão implementada com Jadex é superior as outras versões e se comporta melhor em termos de atributos de modularidade e estabilidade; (iii) Hipótese Alternativa, H2: A versão implementada com JADE é superior as outras versões e se comporta melhor em termos de atributos de modularidade e estabilidade; e (iv) Hipótese Alternativa, H3: A versão implementada com JADE e técnicas de OA é superior as outras versões e se comporta melhor em termos de atributos de modularidade e estabilidade;

Seleção das Variáveis: No estudo do OLIS, as variáveis independentes são: (i) implementação OO baseada em JADE; (ii) implementação OO baseada em Jadex; e (iii) implementação OA baseada em JADE. Em todas as implementações, o principal objetivo foi prover diversas formas de se implementar e modularizar as novas *features* introduzidas na LP-SMA OLIS. As variáveis dependentes no experimento são a modularidade e estabilidade. Nesse estudo também foram utilizadas um conjunto de métricas para quantificar a modularidade em termos dos seguintes atributos: separação de *concerns*, interação entre *concerns*, tamanho, coesão e acoplamento (Capítulo 4). Além disso, também foram utilizadas as seguintes métricas de impacto de mudanças (Yau and Collofello 1985).

Projeto do Experimento: O fator neste experimento é a LP-SMA OLIS e os tratamentos são três versões desenvolvidas da mesma LP-SMA, que são: JADE com arquivos de configuração, Jadex com arquivos de configuração e JADE com técnicas de POA.

Operação do Experimento

O estudo foi dividido em quatro fases maiores: (i) projeto e implementação das *releases* da LP-SMA OLIS usando diferentes técnicas de implementação; (ii) avaliação da modularidade das releases da LP-SMA; (iii) medição dos impactos de mudanças; e (iv) análise dos dados.

Inicialmente, a versão OO baseada na plataforma Jadex e arquivos de configuração Spring foi implementada. Depois, a versão baseada na plataforma JADE e arquivos de configuração Spring. Os arquivos de configuração Spring permitem a injeção de dependências dentro de pontos de variáveis da arquitetura da LP-SMA OLIS. O uso desta técnica melhora a capacidade da LP-SMA para derivar produtos com diferentes configurações. Esta técnica foi usada em substituição a compilação condicional, utilizada na LP-SMA do EC (Seção 5.1). Por fim, a versão baseada na plataforma JADE com técnicas de OA foi implementada. A abordagem de OA foi utilizada para permitir uma melhor

modularização das *features* e ativação/desativação das *features* na derivação dos produtos.

Durante o desenvolvimento da LP-SMA OLIS, foram utilizados as mesmas práticas de projeto em todas as versões das releases do OLIS, tais como: adoção do padrão arquitetural e padrões de projeto comuns (Fowler 2002, Gamma et al. 1995) que refinam cada camada. Estas boas práticas de programação e atividades de validação foram conseguidas pelos participantes envolvidos no experimento, com o objetivo de avaliar o projeto da LP-SMA. Esta equivalência nos projetos das diferentes versões permitem a comparação justa e real.

5.2.3 Análise da Estabilidade

Nesta seção são descritos os resultados da análise quantitativa das implementações alternativas da LP-SMA OLIS. Também é explicado algumas das razões dos resultados através das diferenças observadas. A análise da estabilidade é baseada nas métricas definidas por (Yau and Collofello 1985): número de componentes adicionados ou modificados (aspectos, classes e arquivos XML), número de operações adicionadas ou modificadas e número de linhas código adicionadas ou modificadas. Como as implementações OO da LP-SMA OLIS foram baseadas em arquivos de configuração, estes também foram considerados como componentes a serem analisados durante a propagação das mudanças. A Tabela 5.5 ilustra a propagação das mudanças através das *releases* da LP-SMA OLIS.

Inclusão de Features Obrigatórias

A *release* 2 (R2) compreendeu a inclusão da *feature* obrigatória para permitir os usuários importarem eventos anunciados aos seus calendários. Neste caso, a análise (Tabela 5.5) apontou que a solução OA exigiu um componente adicional quando comparada com as soluções JADE e Jadex. Ambas as soluções JADE e Jadex (não-OA) exibiram os mesmos resultados, pois a estrutura dos agentes usando tais plataformas só é incluída a partir da R4. As soluções JADE e JADEX OO tiveram dois componentes e operações modificadas, enquanto a solução OA não necessitou de nenhuma modificação para implementar esta *feature*. O número de operações adicionados foi maior na solução OA, pois novos *pointcuts* foram criados para interceptar os métodos das classes existentes. Como uma conseqüência, isto implica mais linhas de código adicionadas na solução OA. A solução OA foi menos invasiva que as implementações não-OA, pois tiveram menos elementos de implementação

Tabela 5.5: Propagação das mudanças nas releases do OLIS.

			R2	*R3*	*R4*	*R5*	*R6*	*R7*	*R8*
Componentes	Adicionados	Jade	0	19	30	3	12	33	28
		Jadex	0	19	37	4	16	36	32
		OA	1	18	31	5	13	32	25
	Modificados	Jade	2	5	7	2	4	6	4
		Jadex	2	5	6	3	4	6	3
		OA	0	4	3	0	3	5	4
Operações	Adicionadas	Jade	2	97	154	13	49	264	112
		Jadex	2	97	85	4	41	252	101
		OA	4	96	163	16	52	264	111
	Modificadas	Jade	2	1	3	3	2	1	1
		Jadex	2	1	3	2	1	1	0
		OA	0	0	0	0	0	0	1
Pointcuts	Adicionados	OA	3	10	7	3	3	1	11
	Modificados	OA	0	0	0	0	0	0	0
Linhas de Código	Adicionadas	Jade	17	872	1656	226	575	2105	1265
		Jadex	17	872	1894	302	863	2258	1380
		OA	29	835	1652	255	623	2080	1189
	Modificadas	Jade	0	8	1	1	0	1	1
		Jadex	0	8	1	0	0	0	1
		OA	0	1	0	0	0	0	1

modificados (módulos existentes e *pointcuts*). Em nenhuma das soluções foram necessárias modificar linhas de código existentes para implementar tal *feature*. De acordo com o princípio *Open-Closed* (Meyer 2000), a solução OA apresentou uma melhor adesão a este princípio, pois não necessitou de mudanças em componentes, operações, *pointcuts* e linhas de código existentes.

Inclusão de Features Alternativas

A inclusão de *features* alternativas ocorreu nas *releases* R3 e R8. De acordo com os resultados apresentados na Tabela 5.5, a solução OA apresentou menos componentes e linhas de código que as soluções não OA. As soluções JADE e Jadex OO com arquivos de configuração possuem mais componentes para permitir a separação de *concerns* através do uso de herança e polimorfismo. A solução OA afetou mais componentes e adicionou novas funcionalidades interceptando pontos de junção específicos da LP-SMA OLIS (*pointcuts*). Na implementação da R3, a solução OA precisou modificar apenas uma linha de código, enquanto que as outras soluções precisaram de oito mudanças. Com relação as modificações em operações e *pointcuts*, a solução OA não precisou fazer modificações. A implementação da R8 com Jadex exigiu menos operações que as outras soluções. Esta diferença é explicada pelo fato dos elementos dos agentes serem especificados em arquivos XML no Jadex. Estas especifica-

ções incluem crenças, objetivos, planos, capacidades, eventos e expressões dos agentes. Porém, a implementação com Jadex apresentou mais componentes que as demais soluções, pois novos planos tiveram que ser adicionados. Além disso, novos arquivos XML também foram criados para representar os agentes, aumentando assim o número de componentes e linhas de código.

Inclusão de Features Opcionais

As *releases* R4 a R7 abrangem a inclusão das *features* opcionais. As R4, R5 e R6 precisaram de menos componentes na implementação JADE OO que as outras implementações. Porém, as *releases* R4, R6 e R7 exigiram um número maior ou igual de componentes modificados na implementação JADE OO, comparada com as outras soluções. A implementação Jadex precisou de menos operações que as outras versões nas *releases* R4 a R7. Isto aconteceu porque as *features* de agentes incluídas (agentes e seus respectivos papéis) são codificadas em XML. Como conseqüência, um resultado direto deste efeito é o aumento no número de linhas de código. De fato, a implementação Jadex apresentou valores maiores para linhas de código, considerando todas as *features* opcionais incluídas. Em termos de componentes modificados, todas as versões precisam de modificações em alguns componentes. Porém, a solução OA precisou de menos componentes modificados. Embora a solução OA tenha apresentado um número significativo de componentes adicionados em todos as *releases*, não foram necessárias mudanças em operações e *pointcuts* existentes. Além de novas operações criadas, novos advices foram incluídos. Estas inclusões implementam as novas features e permitem a integração com o núcleo da LP-SMA. Mais uma vez, de acordo com o princípio *Open-Closed* (Meyer 2000), a solução JADE com técnicas de OA se comportou melhor que as outras versões da LP-SMA OLIS.

5.2.4 Análise da Modularidade Multi-Releases

Esta seção apresenta os resultados do processo de medição baseada em um conjunto de métricas apresentadas na Seção 4. Nas subseções seguintes são apresentados os resultados e análise em termos dos atributos de modularidade. O interesse foi observar a estabilidade de tais atributos de modularidade para cada implementação da LP-SMA OLIS (JADE, Jadex e JADE com OA).

Separação de *Features* de Agentes

Nesta seção, cinco *features* da LP-SMA OLIS são analisadas segundo as métricas de *concern*. Esta análise compreende uma *feature* obrigatória,

uma *feature* alternativa e três *features* opcionais. A análise feita nesta seção considera os valores absolutos correspondente as métricas de *concern*. O objetivo é analisar como a modularidade de tais *features* se comporta durante as *releases* das três versões da LP-SMA OLIS. Para todas as métricas aplicadas, valores baixos são considerados ter melhores resultados, isto é, um melhor desempenho. Basicamente, as *features* selecionadas representam abstrações de SMA (e.g. papéis ou agentes) que capturam elementos de agentes fundamentais da LP-SMA.

A Figura 5.20 ilustra os resultados para a *feature* importar eventos, *feature* obrigatória adicionada na R2 (Tabela 5.4). Na inclusão da *feature* obrigatória os resultados mostram que a solução OA apresenta valores mais baixos, apresentando superior estabilidade em termos de entrelaçamento (CDLOC) e espalhamento (CDC). Por outro lado, o número de operações (CDO) necessário para se implementar esta *feature* é mesma em todas as versões. De acordo com os resultados apresentados, pode-se dizer que a modularização da solução OA é mais estável que as outras soluções. Os resultados mostram que ambas as implementações (JADE e Jadex) apresentaram os mesmos valores, pois os agentes ainda não foram incluídos nesta *release*. Isto indica que a implementação OA foi mais efetiva para modularizar esta *feature*. Esta *feature* nas implementações OO está espalhada em duas classes (`CalendarEventAction` and `EventAction`) e apresenta valores mais altos de entrelaçamento com outras *features* (CDLOC). Na Figura 5.17, as classes estão marcadas com o símbolo R2, indicando que as classes foram modificadas nesta *release*. Estas mudanças foi necessárias para introduzir código relacionado a *feature* em questão. Por outro lado, na solução OA, o aspecto `EventActionAspect` consegue afetar estas classes e injetar código relacionado a esta *feature*, não sendo necessário portanto, modificar as classes existentes (Figura 5.18).

A Figura 5.21 ilustra os resultados para a *feature* alternativa de eventos acadêmicos. Os valores foram os mesmos para todas as implementações em termos das métricas CDC, CDO e CDLOC. Isto ocorreu porque usando as soluções OO com a técnica de arquivos de configuração é necessário criar classes que estendem as classes pertencentes ao núcleo da LP-SMA. Isto é importante para permitir a configuração do produto e uma melhor modularização da *feature*. Por exemplo, nas soluções JADE e Jadex (Figuras 5.17 e 5.19), foi necessário a criação de algumas classes: `AcademicEventDataManager` é responsável por gerenciar os dados específicos dos eventos acadêmicos; `AcademicEventDAOHibernate` é responsável pelo acesso ao banco de dados dos objetos dos eventos acadêmicos; `AcademicPreferencesFactory` é responsável por instanciar as preferências acadêmicas do usuário. Na implementação

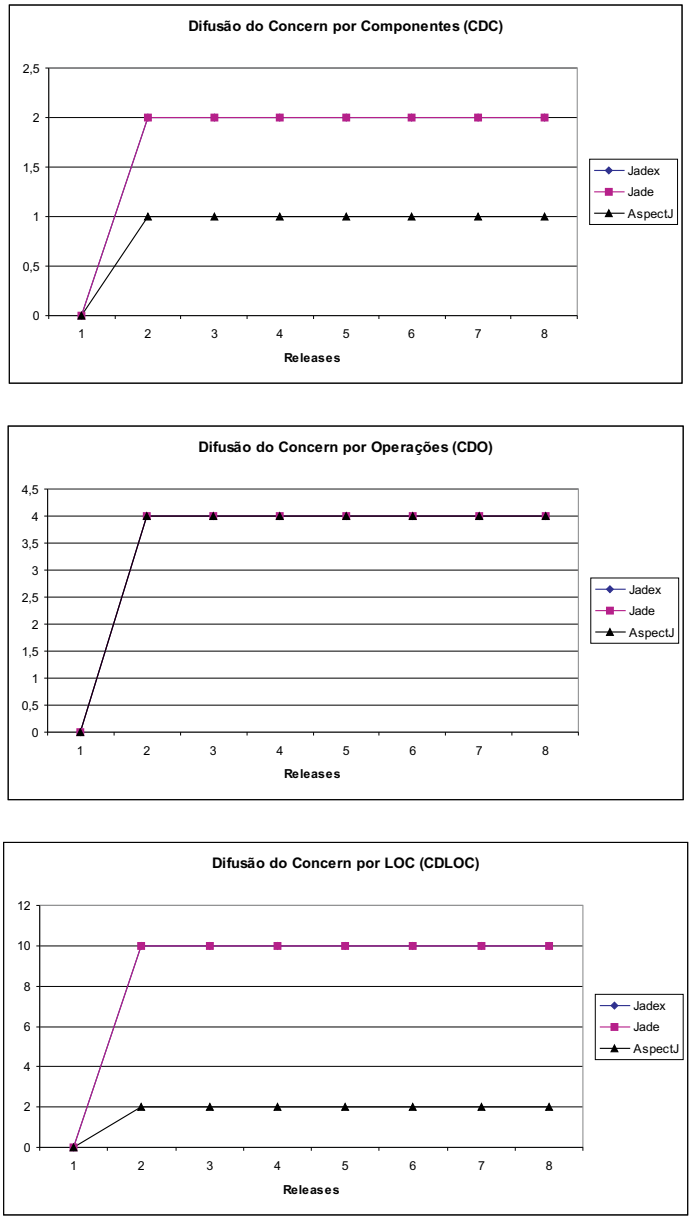


Figura 5.20: Métricas de concern para a feature obrigatória importar eventos.

OA (Figura 5.18), ao invés das classes, foram criados aspectos que afetam as classes existentes. Assim, para todas as implementações foram necessárias a criação da mesma quantidade de componentes. A diferença das versões está apenas na forma de implementação, na implementação OO foi utilizado herança, enquanto que na implementação OA, a abordagem de interceptação. Como conseqüência, a implementação desta *feature* alternativa apresentou resultados similares com respeito as métricas de *concern*.

A Figura 5.22 mostra os resultados para a *feature* opcional Sugestão de Eventos. Os resultados mostram que a implementação Jadex exibiu mais componentes (CDC) que as outras implementações. A partir desta *release*, a infra-estrutura de agentes foi incluída. Conseqüentemente, muitos componentes (classes dos agentes e/ou classes do Jadex e arquivos de configuração) foram adicionados para permitir a comunicação entre os agentes na implementação da *feature* de agentes. A implementação Jadex exigiu mais componentes porque foi necessário a criação de muitos planos (classes Java) e arquivos XML para gerenciar estes planos. Esses arquivos XML são necessários para poder especificar a interação entre estes planos. Por outro lado, estes componentes extras não traduziram em um alto número de operações (CDO). De acordo com o método de projeto proposto pela especificação do Jadex, para permitir uma melhor comunicação e separação dos planos, cada plano deve ter apenas a implementação de um método, no caso o método *body()* da classe `Plan`. O CDO foi o mesmo para as implementações JADE e OA. Apesar da solução OA está espalhada em menos componentes (CDC) na implementação desta *feature*, o número de operações (CDO) foi o mesmo que a implementação JADE. Além dos métodos das classes existentes, os aspectos introduziram métodos e *advices* que interceptam as classes existentes e que implementam esta *feature*. Porém, o entrelaçamento (CDLOC) desta *feature* com outras é menor na solução OA.

A Figura 5.23 mostra os resultados para a *feature* opcional Lembrete de Eventos. Para este *feature*, os valores coletados para a solução Jadex apresentaram melhores resultados quando comparados com as outras soluções, em termos das métricas CDC, CDO e número de atributos. Esta superioridade da implementação Jadex é relacionada ao fato de que a inclusão de um novo papel é feita através de um novo arquivo XML que contém todos os atributos e planos declarados. Na implementação JADE, os agentes contém comportamentos (*behaviors*) e estes contém muitos atributos. Na implementação JADE, como visto na Figura 5.17, uma parte da *feature* Lembrete de Eventos é implementada pelas classes `UserAgent`, `UserAgentCore`, `EventReminder` e `UserAgentRole`. É possível perceber que a classe `UserAgentCore` é modificada na R5. Portanto, o grau de espalhamento é mais alto na solução JADE (CDC

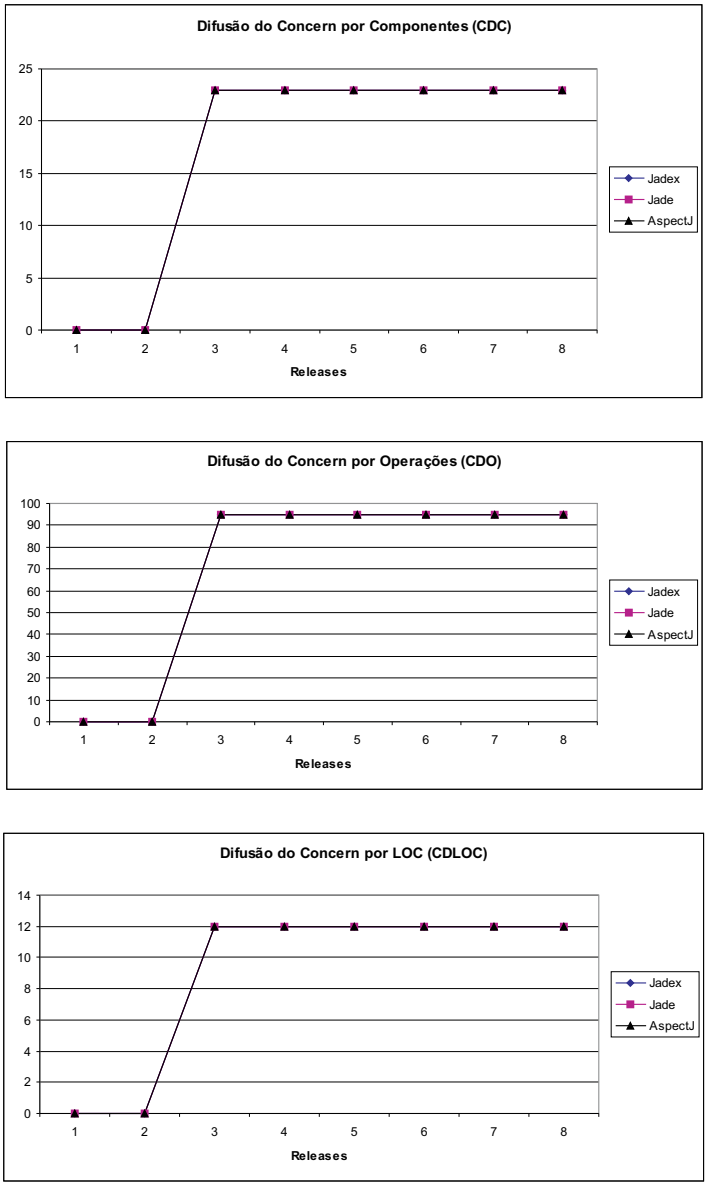


Figura 5.21: Métricas de concern para a feature alternativa eventos acadêmicos.

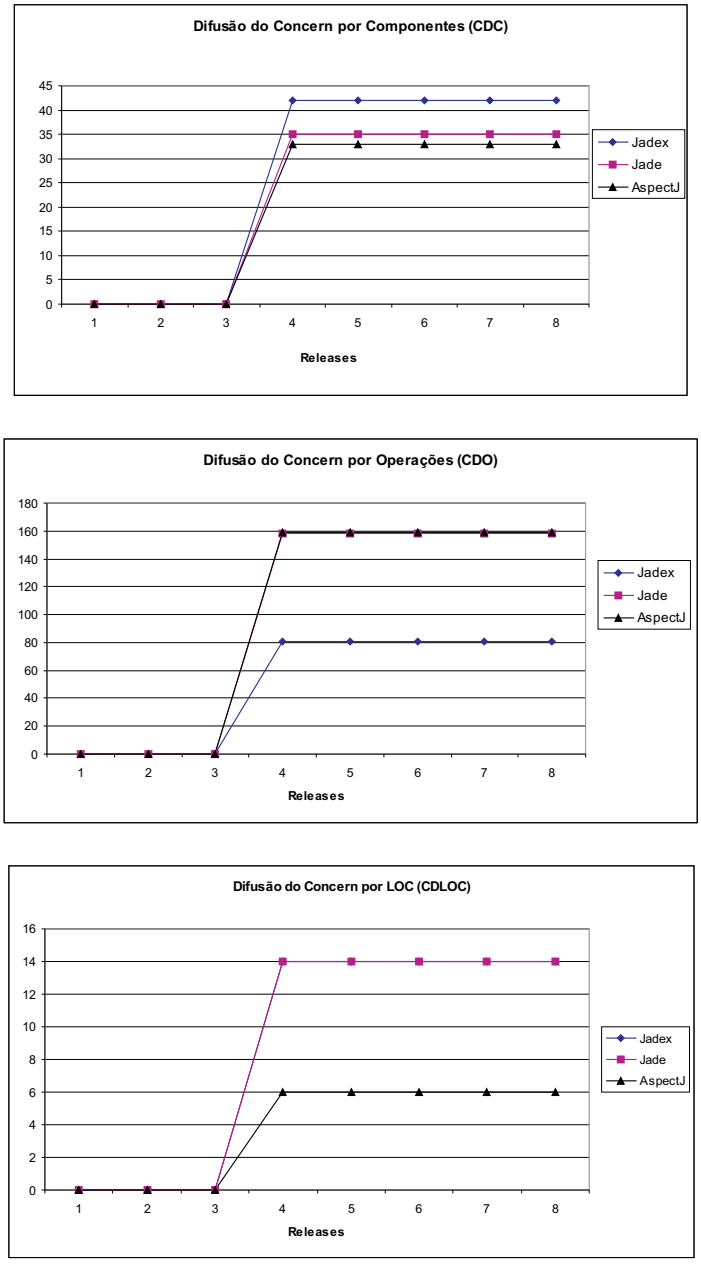
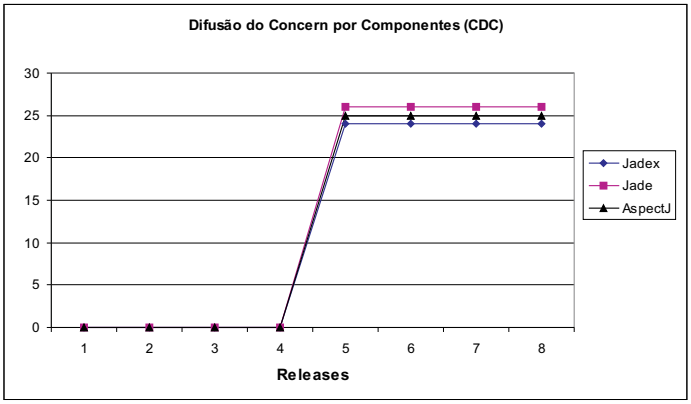


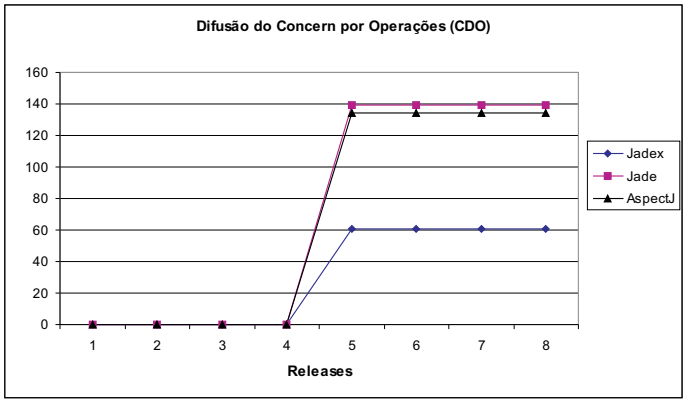
Figura 5.22: Métricas de concern para a feature opcional Sugestão de Eventos.

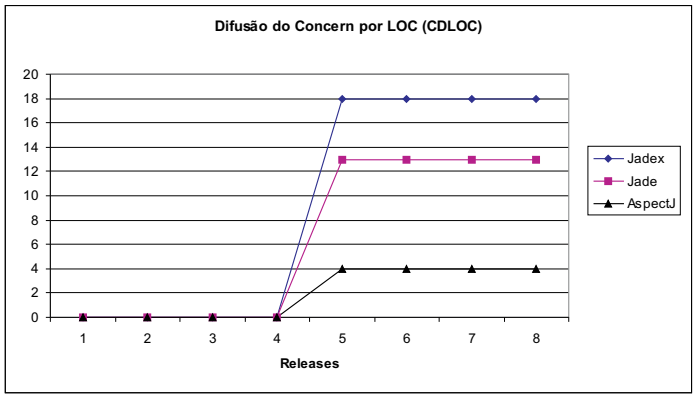
e CDO), mas menos entrelaçado na solução OA (CDLOC). Para esta *feature*, a solução Jadex foi mais efetiva na implementação desta *feature*. A mesma explicação vale para a *feature* opcional Agenda de Eventos incluída na R6 (Figura 5.24.

5.2.5 Acoplamento e Coesão

A Figura 5.25 ilustra os valores coletados para a métrica de acoplamento. De acordo com os resultados apresentados, é possível perceber que a métrica de acoplamento (CBC) na implementação OA se comportou melhor ao longo das *releases*. Isto se deve ao fato de que a maioria dos aspectos afetaram de maneira homogênea alguns componentes, adicionando código com o objetivo de permitir uma melhor modularização da *feature* e reduzir o espalhamento. Os valores da métrica de acoplamento aumentam ao longo da implementação das *releases*, pois muitos componentes são adicionados ao longo do processo de desenvolvimento reativo da LP-SMA OLIS. Assim como na implementação da LP-SMA do EC, na LP-SMA OLIS também foi utilizado o padrão Role (Bäumer et al. 1997) para implementação dos papéis dos agentes. Sendo assim, este também foi um dos motivos do alto acoplamento na implementação JADE. Na implementação JADE cada classe do papel acessa métodos de diversas classes, enquanto que na implementação OA cada papel é modularizado em um aspecto, contribuindo assim para redução do acoplamento.







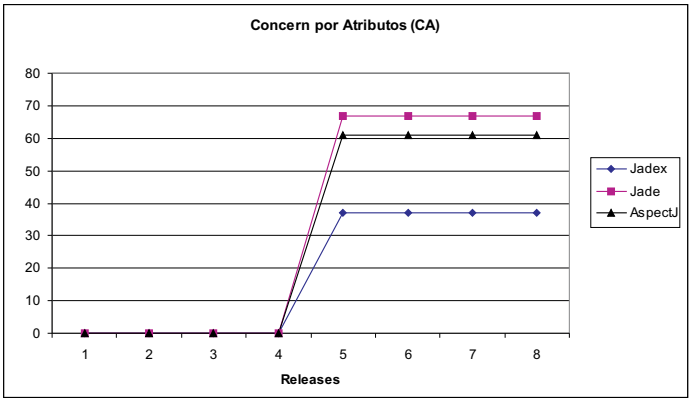


Figura 5.23: Métricas de concern para a feature opcional Lembrete de Eventos.

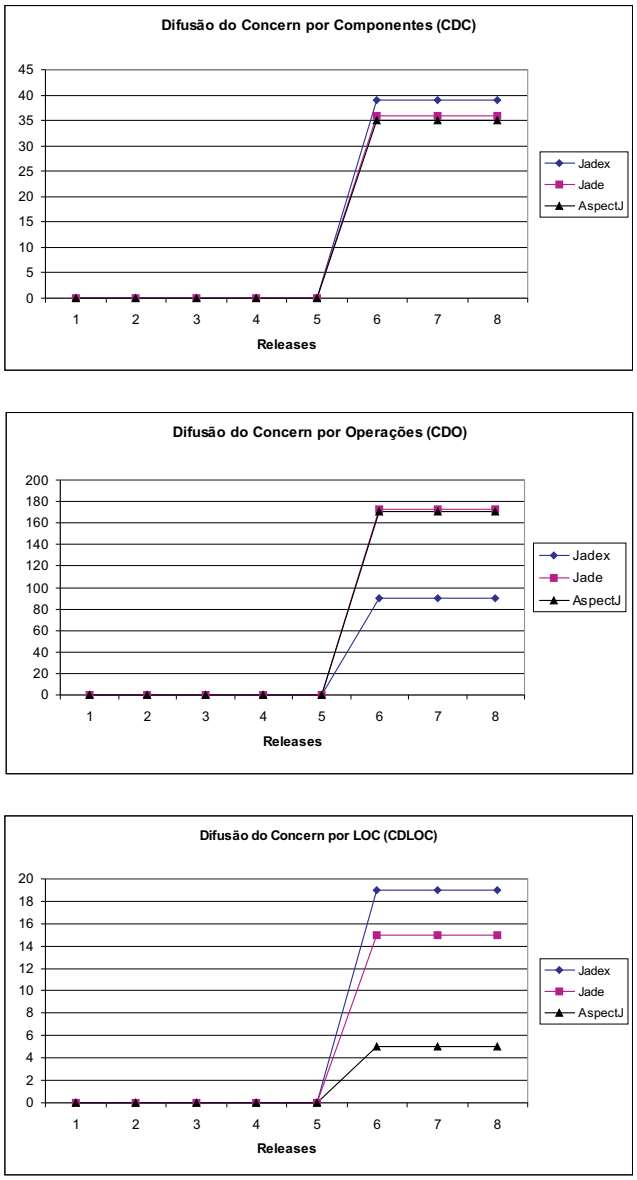


Figura 5.24: Métricas de concern para a feature opcional Agenda de Eventos.

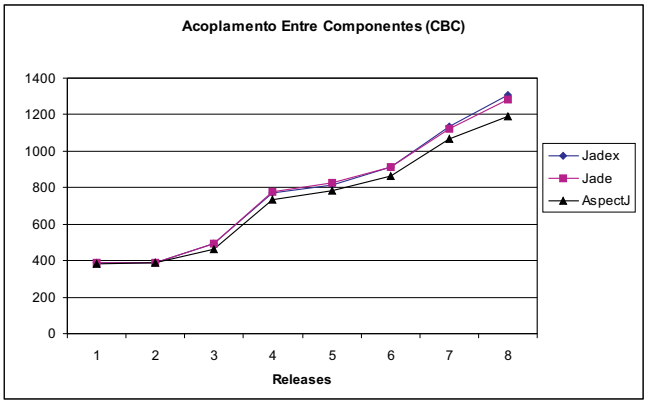


Figura 5.25: Métrica de acoplamento da LP-SMA OLIS.

Na Figura 5.26 é ilustrado os resultados para a métrica de coesão da LP-SMA OLIS. Os resultados mostram que a solução Jadex compreende um conjunto de classes altamente coesas ao longo das *releases*. Isto acontece porque na implementação Jadex, a interação entre os planos e a declaração de atributos são definidas em arquivos XML. Os planos na especificação Jadex representam o significado do agente para agir no ambiente. Estes planos são implementados como uma classe Java que estende a classe `Plan` provida pelo Jadex e precisa sobrescrever o método `body()`. Isto contribui para produzir planos altamente coesos. Adicionalmente, a implementação Jade indica uma falta de coesão. Uma razão que contribuiu para esta falta de coesão foi a implementação do padrão Observer (Gamma et al. 1995), que contribui para reduzir a coesão de cada classe que participa da implementação do padrão.

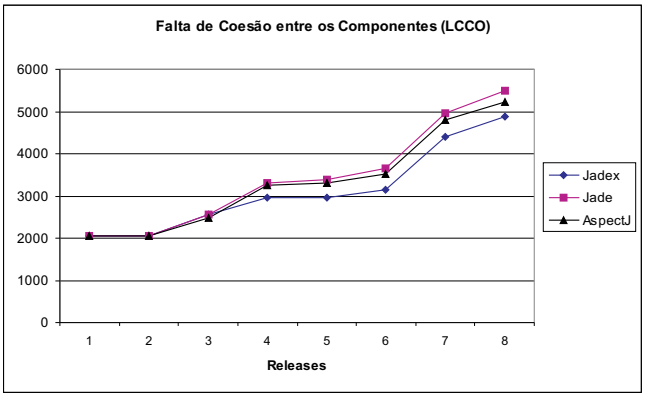


Figura 5.26: Métrica de coesão da LP-SMA OLIS.

5.2.6 Tamanho

Nesta subseção são apresentados os valores para as métricas de tamanho coletadas da LP-SMA OLIS (Figura 5.27). Durante o desenvolvimento e evolução da LP-SMA OLIS, mais componentes foram adicionados que modificados, como visto na análise da estabilidade (Seção 5.2.3). Este fator se deve a escolha das técnicas de implementação escolhidas (arquivos de configuração e aspectos) e a necessidade para o gerenciamento explícito das variabilidades na LP-SMA. De acordo com a Figura 5.27, a solução OA teve menos componentes (NOC) que as demais soluções. Na solução OA, o aspecto pode afetar mais que um componente, reduzindo o número de componentes e facilitando também des/plugabilidade das *features* de agentes. O número de componentes é maior na solução Jadex a partir do R4 porque cada plano do agente precisa ser separadamente modularizado e configurado em arquivos XML. Embora o número de componentes seja maior na solução Jadex, o número de operações não se comporta da mesma maneira, pois o número de operações (NOO) é menor que nas outras soluções. Isto é explicado pela presença de muitos componentes que são os planos dos agentes implementando um método específico. Esta propriedade é típica e satisfeita pelos frameworks OO. Cada ponto flexível nos frameworks OO são geralmente dedicados a codificar apenas alguns métodos específicos.

5.2.7 Interação entre Concerns

Esta subseção apresenta os resultados para o *concern* Agentes de Usuário. O *concern* agentes de usuário inclui os seguintes papéis: Sugestão, Lembrete, Agenda, Cliente e Participante. A métrica CIBC objetiva quantificar a interação entre os *concerns*, neste caso os *concerns* são os papéis dos agentes. A Figura 5.28 mostra os resultados da métrica CIBC para o *concern* Agentes de Usuário, quantificando o grau de entrelaçamento entre os *concerns* nos componentes da LP-SMA OLIS. O componente `UserAgentCore` na implementação Jade é responsável por inicializar os papéis dos agentes. Isto caracteriza o conceito de entrelaçamento do *concern*: os papeis são implementados por um conjunto de diferentes métodos, atributos e declarações nas classes compartilhadas pelos papéis dos agentes. Neste caso, não existe elemento comum implementando ambos (e.g sobreposição de *concerns*). Na plataforma JADE, os conceitos de papéis não estão presentes e, conseqüentemente, os papéis são especificados por um conjunto de classes: `UserAgentCore`, `UserAgent`, and `UserAgentRole` (Figure 5.17).

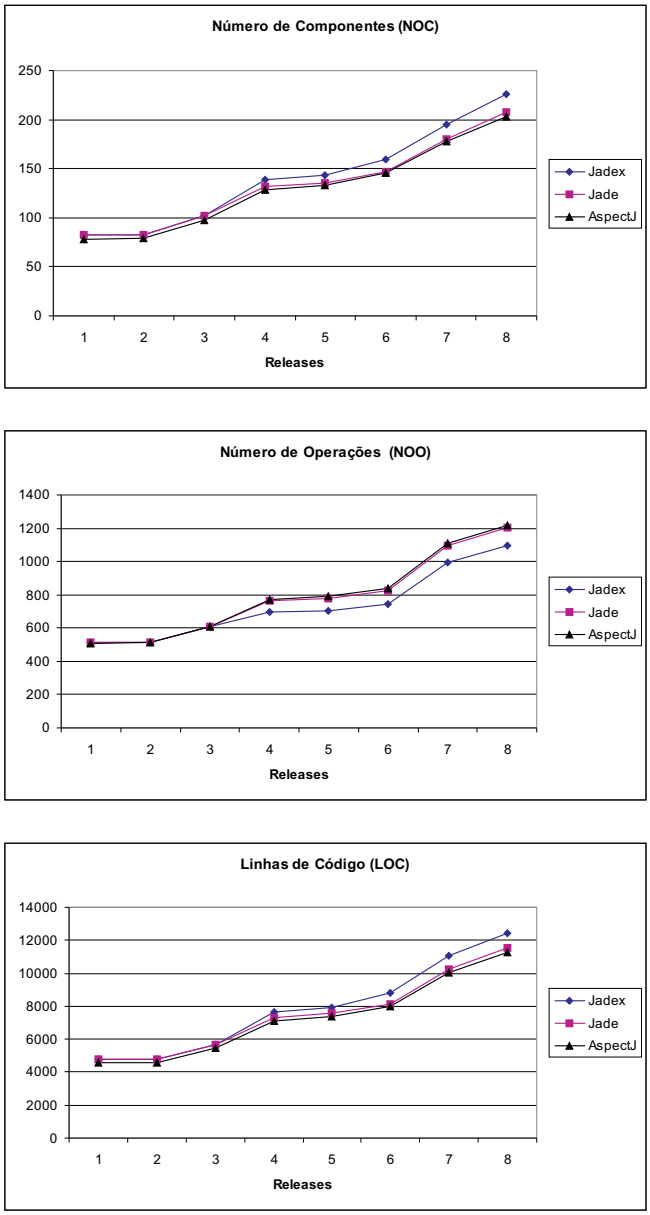


Figura 5.27: Métricas de tamanho da LP-SMA OLIS.

Na plataforma Jadex também não existe o conceitos de papéis, porém eles são implementados de maneira diferente da plataforma Jade. A especificação de papéis na plataforma Jadex é feita através de arquivos XML e são baseadas no conceito de capacidades (*capabilities*). Este conceito de capacidades não é implementado pelo JADE. Capacidades têm sido introduzidas em plataformas de desenvolvimento de SMA como um mecanismo da engenharia de software para suportar a modularidade e reusabilidade, além de permitir um certo nível de raciocínio. Estas capacidades são declaradas em um arquivo XML principal chamado *User.agent.xml*. O entrelaçamento de *concerns* também se manifesta nas implementações Jadex, porém tal entrelaçamento é observado através das *tags* dos arquivos XML. Apesar das diferentes implementações, foi observado o mesmo grau de entrelaçamento em ambas as implementações (JADE e Jadex). Por outro lado, todos os papéis são completamente modularizados em diferentes aspectos na solução OA. Portanto, o componente `UserAgentCore` não está entrelaçado com os papéis dos agentes. Neste caso, a solução OA foi mais efetiva para permitir a separação dos *concerns* de agentes.

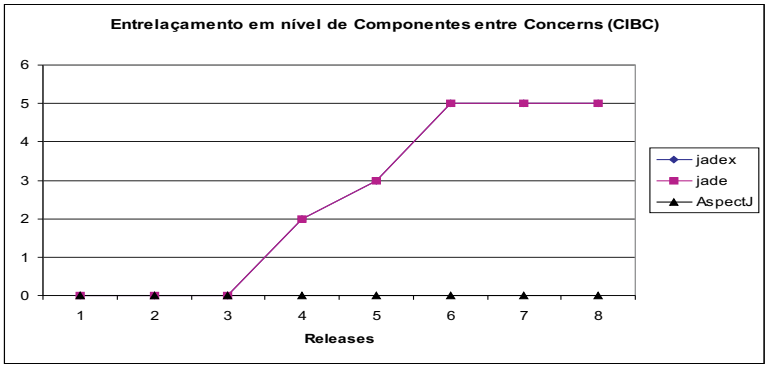


Figura 5.28: Métrica CIBC para a feature Agentes de Usuário.

5.2.8 Interpretação dos dados

Os resultados absolutos das métricas foram usados para testar as hipóteses formuladas, descritas na Seção 5.2.2. Primeiramente, para aplicação do teste estatístico, é necessário verificar se os valores possuem distribuição normal, aplicando o teste de Kolmogorov-Smirnov (Wohlin et al. 2000). Aplicando o teste estatístico, verificou-se que os dados não possuem distribuição normal. Após isso, foi aplicado o teste não parametrizado de Kruskal-Wallis (Wohlin et al. 2000). O teste foi aplicado com um nível de significância α = 0,025, o que significa que o resultado é tratado com um grau de confiança de 97,5%. O número dos graus de liberdade são f1 = a - 1 and f2 = N - a, onde a é a

quantidade das versões implementadas e N é a quantidade de valores coletados nestas versões. Portanto, tem-se f1 = a - 1 = 24 - 1 = 23 and f2 = 225 - 24 = 221. Baseado na tabela estatística descrita em (Wohlin et al. 2000) $F_{0,025,23,221}$ = 1,71. Como o resultado para o teste não parametrizado de Kruskal-Wallis foi H = 0,041, então tem-se que H < $F_{0,025,23,221}$, satisfazendo assim a hipótese nula. Assim, aplicando o teste estatístico não parametrizado, pode-se concluir que as três versões possuem o mesmo grau de modularidade e estabilidade.

5.3 Ameaças à Validação da Análise

Nesta seção são discutidas as ameaças à validade dos estudos experimentais das LP-SMA do EC e OLIS apresentados nas Seções 5.1 e 5.2.

Uma ameaça relevante a *validade de conclusão* é a o tamanho das LP-SMAs, isto é, talvez estas LP-SMAs não sejam suficientes grandes para se aplicar os testes estatísticos não-parametrizados (Wohlin et al. 2000). Porém, as LP-SMAs possuem um tamanho adequado quando comparado com outros estudos empíricos (Figueiredo et al. 2008, Garcia et al. 2003a, Lobato et al. 2008, Alves et al. 2007). Estas LP-SMAs envolveram sistemas baseados na *Web*, os quais foram implementados usando diversas tecnologias atuais e boas práticas de projeto. Além disso, tentou-se aplicar um conjunto de cenários de mudanças reais que pudessem ser aplicados em outros sistemas *Web* modernos, relacionados a inclusão de comportamentos autônomos relevantes para o domínio investigado.

A ameaça à *validade de construção* inclui o conjunto de métricas utilizadas para quantificar as propriedades da modularidade sensível a *concern* e métricas de impacto de mudanças. As métricas de *concerns* tiveram que ser calculadas manualmente, pois é necessário o sombreamento do código, identificando qual segmento de código contribui para implementação *feature* na LP-SMA. Este processo de coleta das métricas de *concerns* envolveu apenas um participante. Porém, foi sempre validado por mais dois pesquisadores. Por outro lado, para coletar as métricas de acoplamento, coesão, tamanho e métrica de impacto foi utilizado o *plugin* de métricas do Eclipse (SourceForge 2005) e a ferramenta Together (Corporation 2004). Apesar da coleta manual das métricas de *concerns*, visando evitar a fadiga e desinteresse, a participante envolvida na contagem das métricas dedicou poucas horas do dia a contagem das métricas durante um tempo. Portanto, as ameaças à *validade de construção* são mínimas.

As ameaças à *validade interna* são relacionadas as regras de alinhamento dos projetos das LP-SMA, em outras palavras, manter as mesmas práticas de

projeto em todas as *releases* das LP-SMA. Esta ameaça foi minimizada usando boas práticas para todas as versões de projeto, tais como, padrões de projeto e estilo arquitetural em camadas a fim de garantir um alto grau de modularidade e reusabilidade. Este alinhamento é necessário a fim de garantir a qualidade do projeto em todas as versões e tornar a comparação mais justa e igual. Além disso, os participantes envolvidos no projeto e desenvolvimento das LP-SMA possuem bons conhecimentos e experiência em Engenharia de Software e nas tecnologias utilizadas.

A principal ameaça à *validade externa* é a natureza dos experimentos escolhidos, pois estes experimentos foram desenvolvidos por duas desenvolvedoras no contexto da PUC-Rio. Para tentar minimizar esta ameaça, um número de pessoas experientes na área de Engenharia de Software Empírica foram envolvidas. Além disso, as decisões de projeto tomadas durante as implementações das arquiteturas das LP-SMAs foram sempre validadas pelos pesquisadores sênior, com bons conhecimentos e experiência na condução de estudos empíricos. Porém, é importante que mais experimentos sejam realizados, envolvendo outros tipos de LP-SMA, de natureza distinta, e participantes com diferentes experiências no projeto e implementação de LPS e SMA. Dessa forma, com uma grande quantidade de experimentos realizados é possível generalizar os resultados dos experimentos desenvolvidos.

5.4 Resumo

Neste capítulo foram apresentadas as duas LP-SMAs desenvolvidas e utilizadas como estudos experimentais. Ao longo do capítulo foram descritos detalhes do projeto e implementação dessas LP-SMAs, descrevendo as tecnologias que foram utilizadas no desenvolvimento, o modelo de *features*, a arquitetura e os agentes que compõem a arquitetura. Além disso, foram apresentados detalhes de como ocorreu a evolução das LP-SMAs, as quais foram desenvolvidas seguindo a abordagem reativa.

A primeira LP-SMA foi o EC, um sistema de gerenciamento de conferências. Para esta LP-SMA do EC foram implementadas duas versões utilizando a mesma plataforma para desenvolvimento de SMA, que é o JADE. Uma versão foi desenvolvida com técnicas de compilação condicional e a outra versão com técnicas de OA. O objetivo deste primeiro estudo foi a comparação dessas duas técnicas na implementação de *features* de agentes usando como base o *JADE*. A segunda LP-SMA foi o OLIS, que provê diversos serviços aos usuários. Para a LP-SMA OLIS foram implementadas três versões. Na primeira versão foi utilizada a técnica de arquivos de configuração usando a plataforma

Jadex. Na segunda segunda versão também foi utilizada a técnica de arquivos de configuração, só que usando a plataforma JADE. Por fim, na terceira versão foi utilizada a técnica de OA usando a plataforma JADE. O objetivo desta segunda LP-SMA é a comparação dessas duas técnicas de implementação (arquivos de configuração e aspectos) utilizando plataformas distintas para a implementação das *features* de agentes.

Na comparação das técnicas de implementação aliada as plataformas JADE e Jadex, foram utilizadas métricas de software para avaliar atributos de modularidade e estabilidade ao longo das *releases* que foram implementadas das LP-SMA. Além disso, foram aplicados testes estatísticos para validar ou não as hipóteses formuladas na concepção dos estudos experimentais.