Avaliação da adequação do uso de aspectos na implementação de variabilidades de linha de produto de software

Rodrigo de C. Brito[1], Thelma E. Colanzi[1]

[1]Departamento de Informática – Universidade Estadual de Maringá (UEM)
Maringá – PR – Brasil

rodrigocb@sanepar.com.br, thelma@din.uem.br

Abstract. *Software Product Line (SPL) is one of the several software development techniques that aim at the software reuse. In this regard, SPL is a systematic way to software construction based on a family of products. An example of SPL is LPS-BET, a SPL for the domain of urban transport, that was developed using black box components. The aspect-oriented programming (AOP) offers resources that allow to encapsulate concerns in a single module, improving the modularity of a system and, consequently, its maintainability. Thus, this study aimed to implement and analyze the replacement of some LPS-BET variabilities using AOP. The results achieved show that, in this context, the use of AOP to implement SPL variabilities does not exceed the implementation using components.*

Resumo. *Linha de Produto de Software (LPS) está entre as diversas técnicas que auxiliam o desenvolvimento de software visam à reusabilidade de software. Com este propósito, uma LPS consiste da construção sistemática de software baseada em uma família de produtos. Um exemplo de LPS é a LPS-BET para o domínio de transporte urbano, que foi desenvolvida utilizando componentes. A programação orientada à aspectos (POA) oferece recursos que permitem encapsular interesses em um único módulo, melhorando a modularidade de um sistema e, por consequência, sua manutenibilidade. Dessa forma, este trabalho teve como objetivo implementar e analisar a substituição de algumas variabilidades da LPS-BET utilizando POA. Os resultados alcançados constatam que, neste contexto, a utilização de POA não supera as vantagens oferecidas pelo uso de componentes.*

1. Introdução

Existem atualmente diversas técnicas de reuso de software, utilizando *frameworks*, componentes, geradores de aplicações, padrões de projeto, Linha de Produto de Software (LPS), dentre outros. Essas técnicas podem ser combinadas para alcançar benefícios como ganho de produtividade, melhoria da qualidade do produto, redução do *time-to-market*. Assim, o reuso sistemático de software consiste de uma mudança da abordagem de construção de sistemas únicos para o desenvolvimento de famílias de sistemas.

Desta forma, o objetivo da abordagem de LPS consiste da proposta de construção sistemática de software baseada em uma família de produtos. Nesta abordagem, um conjunto de características similares entre os vários produtos de um dado domínio

permite a definição de uma estrutura comum de itens que será compartilhada entre os produtos da LPS. Além disso, As características variáveis, denominadas de variabilidades, são parametrizadas possibilitando a diferenciação entre os produtos a serem gerados [van der Linden et al 2005].

A programação orientada a objetos (POO), não fornece abstrações necessárias para a modularização de interesses transversais – interesses cuja implementação encontra-se espalhada por vários módulos de um sistema - causando assim um espalhamento de código pelo sistema, repetindo a lógica desses interesses por diversos módulos. Por isso, os interesses transversais são um tanto quanto difíceis de implementar e manter.

Visando solucionar os problemas relacionados a espalhamento e entrelaçamento de código, nasceu, nos laboratórios da Xérox, a Programação Orientada a Aspectos (POA) [Kiczales et al 1997]. A POA permite encapsular em um único módulo um interesse transversal, melhorando a modularidade do software.

Segundo Donegan [2008 p.2], diversos trabalhos enfatizam a dificuldade de elicitar, representar e implementar variabilidades no contexto de uma LPS. Ela afirma ainda que diversos pesquisadores têm investigado essa questão e têm proposto outras soluções baseadas POA e em programação orientada a características [Mezini e Ostermann 2004; Apel e Batory 2006; Heo e Choi 2006; Lee et al., 2006; Anastasopoulos e Muthig 2004].

Desta forma, o objetivo deste trabalho é implementar as variabilidades de uma LPS para gestão de bilhetes eletrônicos de transporte (LPS-BET) [Donegan 2008], utilizando aspectos; e comparar os resultados obtidos em relação à implementação anterior da LPS-BET, a qual foi realizada utilizando componentes caixa-preta.

Para este trabalho foi desenvolvida uma solução baseada em aspectos para as variabilidades *Acesso Adicional e Integração Terminal*, com a intenção de avaliar resultados de uma solução baseada em componentes sobre uma solução baseada em aspectos.

Os resultados obtidos do desenvolvimento de variabilidades da LPS-BET com aspectos podem fornecer maiores indícios sobre quando é melhor utilizar aspectos na evolução de LPS, contribuindo para a melhoria do processo de desenvolvimento de LPS.

Este artigo está organizado da seguinte forma: na Seção 2 apresenta-se uma revisão bibliográfica sobre LPS, POA e o desenvolvimento da versão da LPS-BET projetada utilizando componentes. Na Seção 3 é aborda-se a solução elaborada para desenvolver as variabilidades *Acesso Adicional e Integração Terminal* utilizando POA. Na Seção 4 são analisados os resultados e na Seção 5 apresentam-se as considerações finais do trabalho.

2. Revisão bibliográfica

Nas próximas seções serão apresentados conceitos sobre Linhas de Produtos de Software, engenharia de domínio e de aplicação, Programação Orientada a Aspectos e o como foi desenvolvida a LPS-BET [Donegan 2008].

2.1 Linha de Produto de Software

Segundo Parnas [1979 apud Donegan 2008], uma coleção de sistemas que compartilham características comuns é chamada de “família de sistemas”. Hoje elas também são chamadas de Linha de Produto de Software. Uma LPS é constituída por um conjunto de aplicações similares de um domínio, que podem ser desenvolvidas a partir de uma arquitetura genérica comum, a arquitetura da LPS, e um conjunto de componentes que povoam a arquitetura [van der Linder et al 2005].

Segundo Clements [2006 apud Donegan 2008], a organização de uma LPS consiste em três atividades: desenvolvimento do núcleo de artefatos, desenvolvimento do produto e gerenciamento da LPS. A atividade de desenvolvimento do núcleo de artefatos pode ser chamada de engenharia de domínio, assim como a atividade de desenvolvimento do produto pode ser chamada de engenharia de aplicação. Nas próximas sub-seções são apresentadas as atividades de engenharia de domínio e de aplicação.

2.1.1 Engenharia de domínio

É nessa atividade que é feita a análise de domínio, análise esta que corresponde ao desenvolvimento de todas as características em comum aos produtos da LPS. Um núcleo é desenvolvido, como todas as características obrigatórias da LPS, para que posteriormente sejam implementadas as demais características. O núcleo de artefatos (do inglês *core assets)* pode incluir planos, requisitos, projetos, documentação, testes e código.

Na análise do domínio da aplicação são consideradas as funcionalidades que são comuns a todas as aplicações do domínio (núcleo da LPS), aquelas que são opcionais (existentes apenas para alguns produtos da LPS) e as alternativas (escolhidas a partir de um conjunto de possibilidades). Em uma LPS, as características constituem um conjunto de requisitos reutilizáveis e podem ser de um dos seguintes tipos:

- **Obrigatória:** característica principal do produto. São características compartilhadas por todos os produtos da LPS.

- **Opcional**: características fornecidas por somente alguns produtos da LPS.

- **Alternativa**: conjunto de características em que se deve fazer uma única escolha dentre as possíveis. Pode-se dizer que as opções são mutuamente exclusivas.

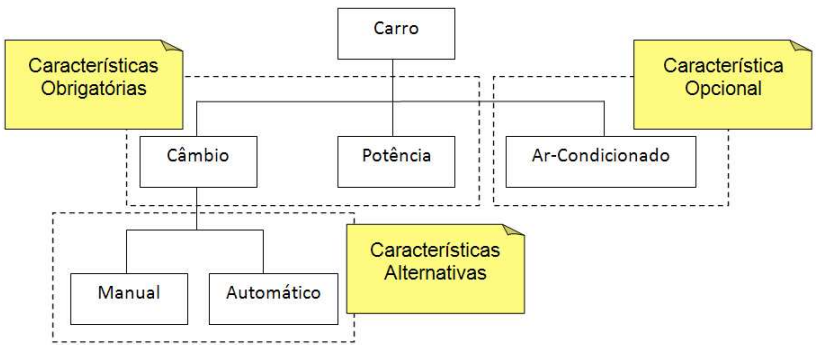

Figura 1. Modelo de Características [Adaptado de Kang et al 1990]

A Figura 1 exemplifica, para o domínio de carros, os tipos de características descritas anteriormente, onde as características obrigatórias que farão parte do núcleo

são “câmbio” e “potência”, as características “manual” e “automático” são alternativas e “Ar-condicionado” é opcional. As características alternativas pode-se imaginar como especializações, onde apenas uma característica pode implementada.

É no núcleo que todas as características do tipo obrigatórias estão implementadas, ou seja, todas aquelas que farão parte de todos os produtos da LPS. Ainda, para formar uma LPS é necessário representar as variabilidades que podem ocorrer em cada artefato que faz parte desta. Variabilidades são diferenças encontradas entre os produtos podendo ser reveladas e distribuídas entre os artefatos [van der Linden et al 2005]. Elas são descritas em termos de pontos de variação e variantes. Um ponto de variação é um lugar específico em um artefato da LPS ao qual uma decisão de projeto é conectada. Cada ponto de variação está associado a um conjunto de variantes que correspondem às alternativas de projeto para uma variabilidade [Heumans e Trigaux 2003 apud Oliveira Junior et al 2005].

A atividade de engenharia de domínio é realizada seguindo as etapas de Concepção, Elaboração, Construção e Transição correspondentes ao ciclo iterativo incremental [Donegan 2008]. Além disso, a engenharia de domínio poder ser incrementada utilizando-se de incrementos horizontais ou verticais. Utilizando os incrementos horizontais, o subgrupo de característica de uma aplicação-referência é implementada, produzindo no final uma aplicação-referência completa. Utilizando incrementos verticais, todas as variabilidades de um subgrupo de características são implementadas, de forma completa, porém não produzem necessariamente uma aplicação-referência completa. É na etapa de transição que são elaborados os gabaritos a serem aplicados na instanciação dos produtos.

2.1.2 Engenharia de aplicação

É a atividade em que cada aplicação é construída. É feita uma análise da aplicação-referência a ser produzida, e por meio dos resultados, são selecionadas as características elaboradas na engenharia de domínio.

A montagem de uma aplicação-referência pode ser realizada de duas formas: (i) **manual:** na qual o gabarito criado é seguido ao longo da engenharia de domínio para se gerar a aplicação; e (ii) **automatizada**, na qual**:** o mesmo gabarito utilizado na montagem do tipo manual pode ser processado por geradores automáticos a fim de gerar a aplicação ao final do processo.

A montagem da aplicação-referência, utilizando o método automatizado, pode ser feito por meio de geradores de aplicação. Segundo Donegan [2008], os geradores de aplicação são ferramentas de software que transformam informação de alto nível em implementação de baixo nível. O problema ou a tarefa a ser realizada por um programa consiste na informação de alto nível e é descrita por meio de uma especificação, que é usada pelo gerador de aplicação para automaticamente produzir um programa.

Geradores de aplicação configuráveis (GAC) são aqueles que podem ser adaptados para gerar aplicações de diversos domínios. Com essa idéia de que um gerador seja configurável para variados domínios, surgiu o Captor [Shimabukuro, 2006]. A Figura 2 demonstra a visão de um GAC, que consegue gerar os artefatos da aplicação por meio dos arquivos de configuração e dos gabaritos.

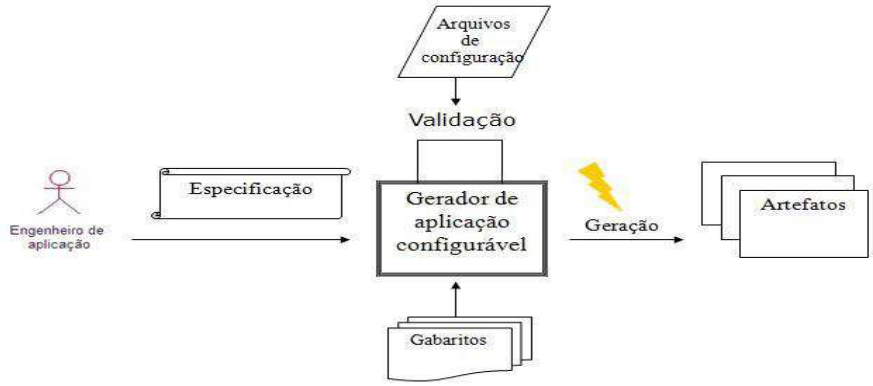

Figura 2. Gerador de Aplicação Configurável [SHIMABUKURO 2006]

2.2 Programação Orientada a Aspectos

As linguagens de POO possuem limitações, no projeto e na implementação de interesses que normalmente afetam diversas classes e/ou módulos, não conseguindo manter a separação de interesses. Um interesse representa uma característica relevante de uma aplicação, que pode ser definido como uma parte isolada de um domínio com o intuito de entender melhor cada parte isoladamente. Um interesse pode ser um requisito funcional ou não funcional [Meilsmith 2008]. Quando um mesmo interesse está presente em vários módulos do sistema, ele é chamado de interesse transversal.

Se o desenvolvimento de um software for realizado sem se preocupar com a separação de interesses, isso resultatá em código entrelaçado e espalhado. O código entrelaçado (*code tangling*) acontece quando a implementação de um módulo em um software interage simultaneamente com vários interesses, tais como *logging*, autenticação, *multi-threaded safety*, validações, entre outras. O código espalhado (*code scattering*) acontece quando um interesse é implementado em múltiplos módulos. Uma vez que interesses transversais, por definição, são espalhados por vários módulos, logo sua implementação também se espalha por esses módulos.

Diante desse problema Kiczales et al. [1997] propuseram a abordagem de POA para separar e encapsular esses interesses transversais e assim eliminar o espalhamento e entrelaçamento de código em módulos chamados de aspectos.

Os aspectos são compostos pelos elementos descritos a seguir:

• Pontos de junção : são locais bem definidos da execução de um programa, como, por exemplo, uma chamada a um método ou a ocorrência de uma exceção, dentre muitos outros, onde o aspecto pode ser aplicado [Kiczales et. al 1997].

• Pontos de atuação: são elementos do programa usados para definir um ponto de junção, como uma espécie de regra criada pelo programador para especificar eventos que serão atribuídos aos pontos de junção. Os pontos de atuação têm como objetivo criar regras genéricas para definir os eventos que serão considerados pontos de junção, sem precisar defini-los individualmente. Outra função dos pontos de atuação é apresentar dados do contexto de execução de cada ponto de junção, que serão utilizados pela rotina disparada pela ocorrência do ponto de junção mapeado no ponto de atuação.

• Adendos: são trechos da implementação de um aspecto executados nos pontos de junção. Os adendos são compostos de duas partes: a primeira delas é o ponto de

atuação, que define as regras de captura dos pontos de junção; a segunda é o código que será executado quando ocorrer o ponto de junção definido pela primeira parte. O adendo é um mecanismo bastante similar a um método (quando comparado com a programação OO), cuja função é declarar o código que deve ser executado a cada ponto de junção em um ponto de atuação, ou seja, um meio para alterar o comportamento dos pontos de junção. Há três tipos de adendos: (i) *before* que é executado antes do ponto de junção; (ii) *after*: que é executado depois do ponto de junção; e, (iii) *around* que é executado durante a execução do ponto de junção.

• Declarações inter-types: são declarações que adicionam novas funcionalidades a aplicação, por meio de declaração de novos atributos e métodos para uma classe ou conjunto de classes.

• Aspectos: encapsulam pontos de junção, pontos de atuação, adendos e declarações inter-types em uma unidade modular de implementação. São definidos de maneira semelhante às classes, enquanto essas encapsulam o código que se encaixa dentro delas, *aspectos* encapsulam o código que é ortogonal, transversal a essas classes**.**

A Figura 3 exemplifica cada um desses elementos. O aspecto *FaultHandler* consiste de uma *declaraçãointer-type* que introduz um atributo na classe *Server* (linha 03), apresenta ponto de atuação de nome *services* que ira capturar todas as chamadas de todos metodos de todas as classe (linha 13), é apresentado um adendo do tipo before (linhas 15-17), ou seja antes de executar os comandos contidos no ponto de atuação *services(s)*, serão exucutados os comandos contidos dentro do adendo (linha 16). Nas linhas de 19 – 22 é apresentado um adendo do tipo after, ou seja, os comandos contidos dentro do ponto de atuação *services(s)* serão executados depois da execução do contidos no adendo (linhas 20 - 21).

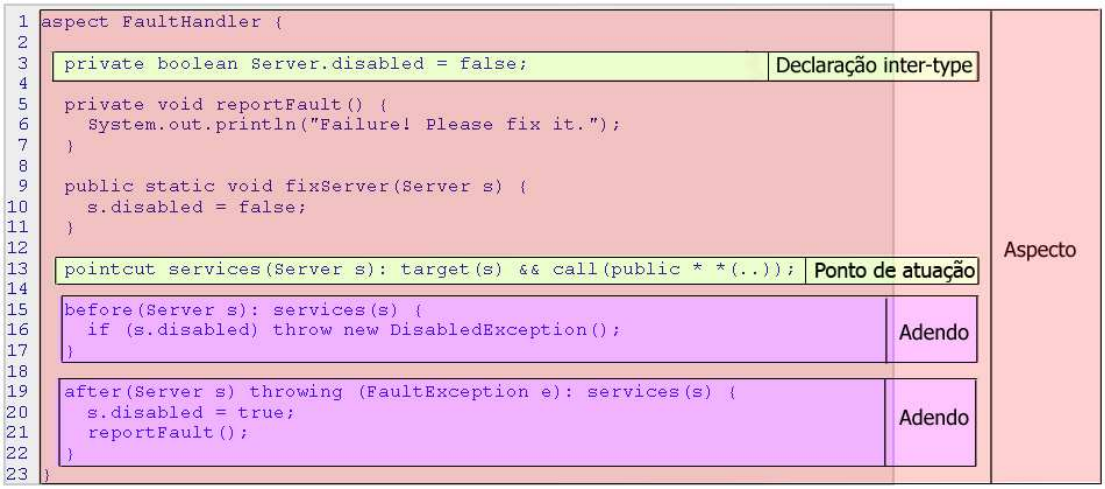

Figura 3. Elementos de um Aspecto [Adaptado de AspectJ Team 2009]

O aspecto adiciona uma funcionalidade ao código-base interceptando o fluxo de execução. No código-base não é preciso haver menção explícita ao aspecto. Um processo de combinação (weaving) é executado pelo compilador do aspecto (aspect weaver), para que o código-base e os aspectos sejam combinados, gerando ao final um produto completo, com todos interesses implementados. Portanto, ao utilizar a POA, os sistemas ficam mais legíveis, fáceis de entender, implementar, integrar, reusar, personalizar, evoluir e manter [Kiczales et al 1997].

2.3 Linha de Produto de Software para Bilhetes Eletrônicos de Transporte (LPS-BET)

LPS-BET [Donegan 2008] trata-se de uma LPS de gestão de bilhetes eletrônicos de transporte, que facilita o transporte urbano com o uso de um cartão eletrônico para pagar passagens e oferece várias outras funcionalidades para passageiros e companhias de ônibus.

A LPS-BET foi projetada e desenvolvida visando a geração de três produtos, a partir de uma análise feita em três sistemas reais de municípios brasileiros: os sistemas BET de São Carlos (SP), Fortaleza (CE) e Campo Grande (MS). A partir de uma análise detalhada das características semelhantes e das características particulares do domínio, foram projetados e implementados componentes do tipo caixa preta para a construção do núcleo e das variabilidades da LPS-BET. Componentes são unidades independentes, que encapsulam funcionalidades. Cada um deles oferecem serviços, por meio de interfaces bem definidas, para o meio externo [Brown 2000] .A diferença entre componente caixa-branca e componente caixa-preta é que no primeiro pode-se ter acesso ao código do componente, enquanto no segundo somente tem-se acesso à interface.

Para o desenvolvimento da LPS-BET, Donegan [2008] planejou quatro incrementos horizontais, sendo eles:

1º) Desenvolvimento dos casos de uso do núcleo do LPS-BET;

2º) Reuso do núcleo e desenvolvimento dos casos de uso de Fortaleza;

3º) Reuso do núcleo e de alguns casos de uso de Fortaleza e desenvolvimento dos casos de uso de Campo Grande;

4º) Reuso do núcleo, de alguns casos de uso de Fortaleza e de alguns casos de uso de Campo Grande, além do desenvolvimento dos casos de uso de São Carlos.

Por fim, o gerador de aplicações Captor [Shimabukuro 2006] foi utilizado na engenharia de aplicação para automatizar a geração dos produtos da LPS.

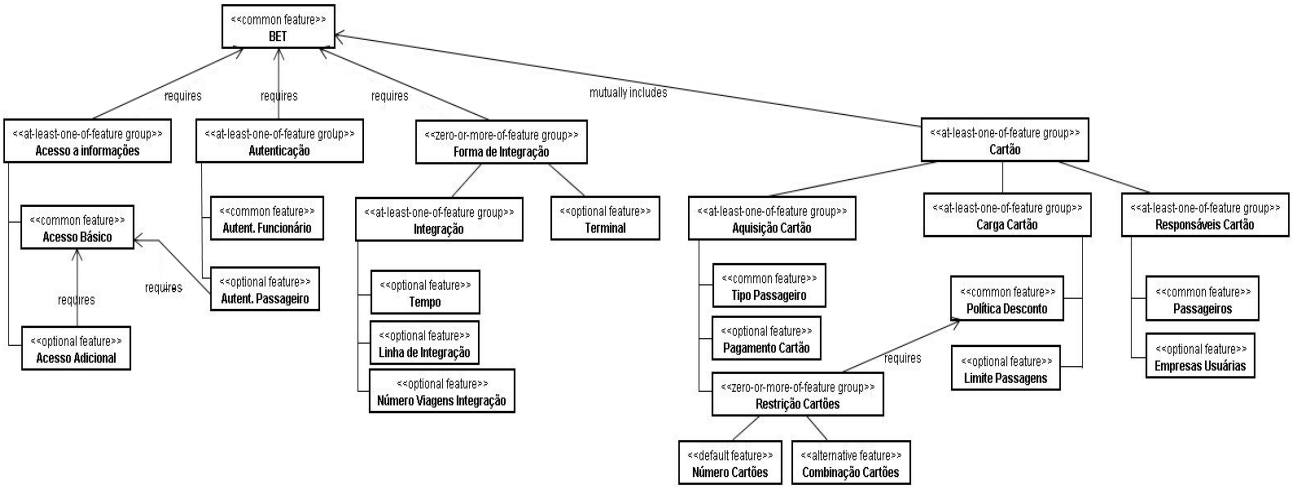

Figura 4. Diagrama de Características da LPS-BET [Donegan, 2008]

Na Figura 4 é apresentado o modelo de características da LPS-BET e, na Tabela 1, são mostradas as características (opcionais e alternativas) que devem ser incorporadas ao núcleo da LPS para obter cada um dos três produtos.

Tabela 1. Características opcionais ou alternativas para produtos da LPS-BET [Donegan 2008]

Característica	Fortaleza	Campo Grande	São Carlos
Acesso Adicional		**X**	**X**
Autenticação Passageiro			**X**
Forma de Integração - Terminal - Integração * Tempo * Linha de Integração * Número de Viagens de Integração	**X**	**X** **X** **X** **X**	**X** **X**
Pagamento de Cartão	**X**		
Restrição de Cartões - Número de Cartões - Combinação de Cartões		**X**	**X**
Empresas Usuárias	**X**	**X**	
Limite de Passagens			**X**

A implementação da LPS-BET inicialmente foi feita utilizando a arquitetura baseada em componentes. Decidiu-se utilizar POA para a implementação de requisitos não funcionais, como é demonstrado na subseção 2.3.1. Donegan [2008] fez ainda a implementação de um requisito funcional usando POA, para isso, os aspectos foram implementados visando interceptar as interfaces dos componentes. Este último caso é apresentado na sub-seção 2.3.2.

2.3.1 Implementação de requisito não funcional da LPS-BET usando POA

O requisito não funcional de autenticação foi projetado e implementado usando aspectos, caso contrário ele estaria espalhado em vários componentes (*Carga Cartão, Aquisição Cartão, GerênciaCtrl*), e cada um deles seriam responsáveis por autenticar e autorizar os usuários no sistema, levando a uma dificuldade de manutenção, devido ao espalhamento do código. Pode se observar na Figura 5 a presença do aspecto de autenticação, que está presente nos três produtos, e por isso faz parte do núcleo.

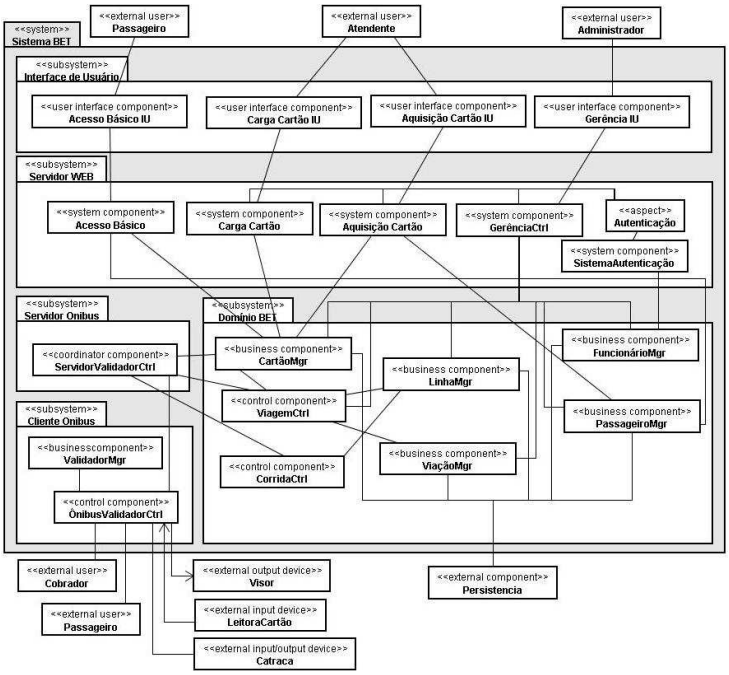

Figura 5. Arquitetura de componentes do núcleo da LPS-BET [Donegan 2008]

O aspecto de autenticação foi dividido em duas partes, uma para autenticar o usuário (Autenticação) e outra para autorizar as páginas da web que o usuário pode navegar (Autorização). Ambos possuem o mesmo ponto de junção, porém o de autenticação precede o de autorização. A Figura 6 apresenta os atributos, adendos e operações dos aspectos autenticação e autorização.

O aspecto autenticação requer algumas operações da interface ISistemaAutenticacao do componente SistemaAutenticacao: *atualizarSessao*, *estaAutenticado* e *estaExpirado* e seu adendo é do tipo *before*, pois a autenticação deve ocorrer antes do usuário acessar a página web.

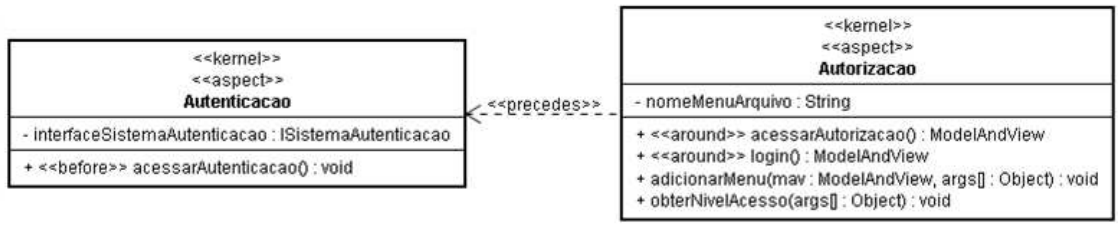

Figura 6. Especificação do aspecto Autenticação [Donegan 2008]

Os aspectos autenticação e autorização entrecortam as interfaces dos componentes *GerênciaCtrl*, *Aquisição Cartão* e *Carga Cartão*, conforme ilustrado na Figura 7. Esses são os 3 componentes nos quais o interesse de autenticação é necessário.

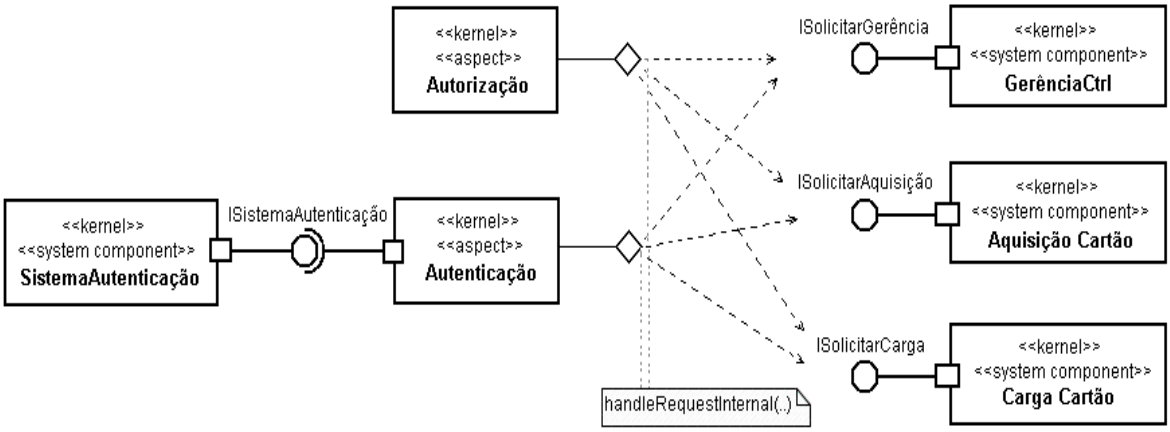

Figura 7. Arquitetura dos aspectos Autenticação e Autorização e dos componentes entrecortados [Donegan 2008]

Com a implementação desse aspecto, não houve retrabalho ao longo do desenvolvimento dos componentes da LPS-BET, pois este foi configurado para entrecortar todos os métodos handleRequestInternal (método usado especificamente por controladores do MVC [Spring 2009]) dos componentes que fazem parte do sub-sistema Servidor WEB, com exceção do acesso básico.

2.3.2. Implementação de requisito funcional da LPS-BET usando POA

A implementação inicial de variabilidades da LPS-BET foi elaborada utilizando apenas componentes. Porém, Donegan [2008] ilustrou o uso de POA para implementar requisitos não funcionais por meio do grupo de características de Integração (Tempo, Linha Integrada e número de viagens de integração). Esta solução foi projetada de tal forma que o aspecto interceptasse as interfaces dos componentes relacionados à característica de integração, por serem do tipo caixa-preta.

.O uso de linhas integradas pré-definidas, e/ou o uso de ônibus dentro de um intervalo de tempo, marcam a característica integração, podendo ainda haver um número máximo de integração dentro de um período de tempo.

Todas as características do grupo têm um comportamento semelhante, requerem alterações do mesmo componente (*viagemCtrl*), devendo ser capazes de processar a integração pela requisição das interfaces *IRegistrarViagem* e *IRegistrarArrecadação*. Eles diferem em relação ao componente de negócio que requerem, ou seja, solicitam interfaces diferentes, que significam que terão implementações diferentes para fazer a verificação da integração.

Um aspecto abstrato foi usado para generalizar a similaridade entre os três tipos de integração, cada variabilidade representa um aspecto e implementa o aspecto abstrato (*IntegracaoCtrl*), herdando o ponto de junção, um adendo e um método (ProcessarIntegração), além de duas interfaces comuns requeridas (Figura 8).

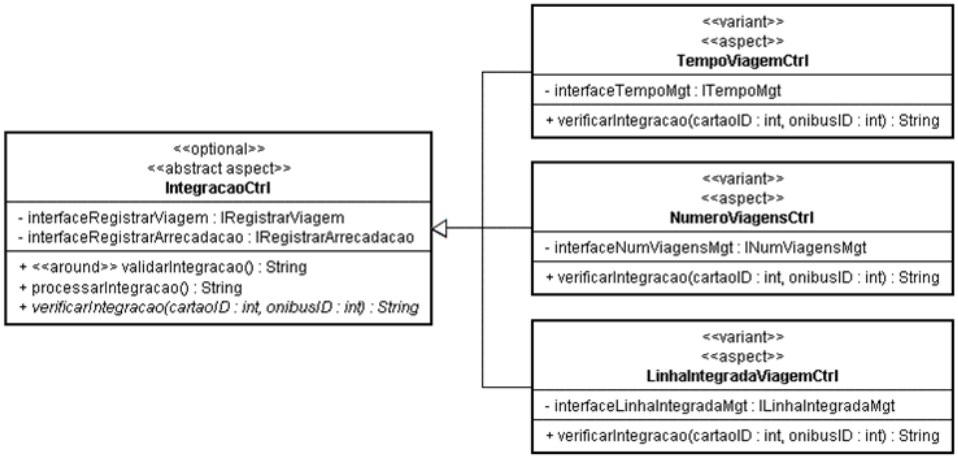

Figura 8. Aspecto abstrato e aspectos contratos para representar a característica Integração [Donegan 2008]

A implementação do aspecto abstrato *IntegraçãoCtrl* usando AspectJ [Kiczales et al 2001] é apresentada na Figura 9. As interfaces requeridas são definidas nas linhas 03 e 04. O ponto de junção entrecorta apenas a chamada dos métodos da interface *IProcessarViagem* (linha 06) e então o adendo do tipo around é executado (linhas 08-17). Se houver integração (estado recebe "INT-OK"), o adendo retorna o estado ao método da interface entrecortada, sem proceder a execução do método interceptado. Entretanto, caso não haja integração (estado é igual a "INT-NOK"), o método entrecortado procede com sua execução normal (linha 16).

O aspecto *IntegracaoCtrl* entrecorta a interface e é estendido pelo aspecto *TempoViagemCtrl* que requer as operações da interface *ITempoMgt* do componente de negócio *TempoMgr*.

```
1  public abstract aspect  IntegracaoCtrl {
2
3      ICartaoMgt interfaceCartaoMgt;
4      IRegistrarArrecadacao interfaceRegistrarArrecadacao;
5
6      pointcut validarIntegracao(): call(String lps.bet.interfaces.IProcessarViagem.*(..));
7
8      String around(): validarIntegracao(){
9          String estado="INT-NOK";
10         Object[] args = thisJoinPoint.getArgs();
11         if (args.length > 0){
12             int cartaoID = (Integer) args[0];
13             int onibusID = (Integer) args[1];
14             estado = verificarIntegracao(cartaoID, onibusID);
15         }
16         return !estado.equals("INT-OK") ? proceed():estado;
17         }
18
19     public String processarIntegracao(int onibusID, Viagem viagem){
20         interfaceRegistrarArrecadacao.registrarArrecadacao(onibusID, 0);
21         viagem.setNumViagens(viagem.getNumViagens()+1);
22         interfaceCartaoMgt.alterarViagem(viagem);
23         return "INT-OK";
24     }
25
26     public abstract String verificarIntegracao(int cartaoID, int onibusID);
27
28 }
```

Figura 9. Implementação do aspecto IntegraçãoCtrl [Donegan 2008]

A implementação do aspecto *TempoViagemCtrl* é exibida na Figura 10. Ele estende o aspecto abstrato *IntegraçãoCtrl* (linha 01) e, portanto, pode utilizar o método

processarIntegração (linha 16). O aspecto verifica se a integração se aplica comparando o tempo decorrido desde a última viagem (linha 14) com o tempo máximo de integração, obtido a partir do método buscarTempo da interface *ITempoMgt* (linha 10). Como esse aspecto estende o *IntegraçãoCtrl*, ele também apenas entrecorta a interface IProcessarViagem.

```
1  public aspect TempoViagemCtrl extends IntegracaoCtrl{
2
3      ITempoMgt interfaceTempoMgt;
4
5      public String verificarIntegracao(int cartaoID, int onibusID){
6
7      String estado = "INT-NOK";
8      long tempoDecorrido = Long.MAX_VALUE;
9      Viagem viagem = interfaceCartaoMgt.buscarUltimaViagem(cartaoID);
10     int tempoMaxIntegracao = interfaceCartaoMgt.buscarTempo();
11
12         if (viagem != null){
13             Calendar horaUltimaViagem = viagem.getHora();
14             tempoDecorrido = Calendar.getInstance().getTimeInMillis() - horaUltimaViagem.getTimeInMillis();
15         if (tempoDecorrido <= tempoMaxIntegracao)
16                 estado = processarIntegracao(onibusID, viagem);
17         }
18         return estado;
19     }
20 }
```

Figura 10. Implementação do aspecto TempoViagemCtrl [Donegan 2008]

A solução para adicionar a característica Tempo usando POA na arquitetura da LPS-BET, conforme ilustrado na Figura 11, não requer a alteração dos componentes, bastando configurar o contexto de aplicação do componente que implementa a interface requerida pelo componente básico. Assim, o componente básico não terá conhecimento da existência das variabilidades, pois serão utilizadas interfaces com nomes análogos.

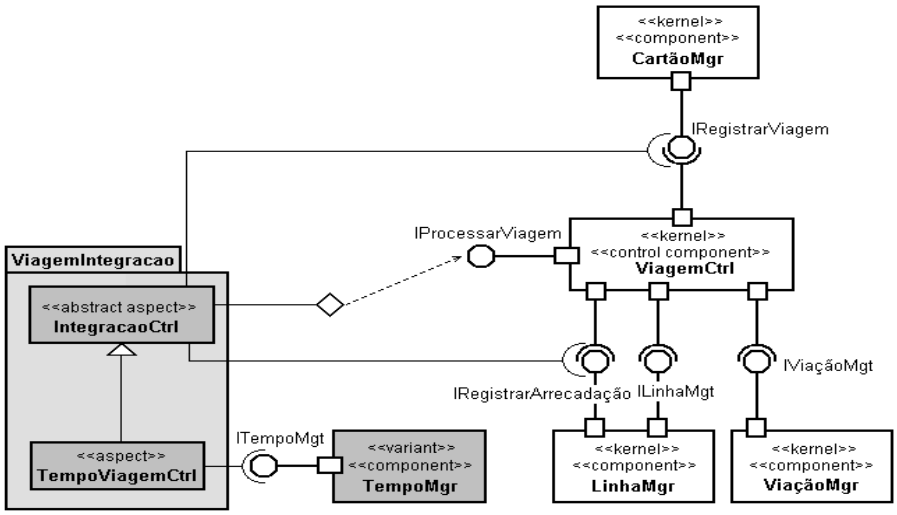

Figura 11. Uma solução usando aspectos para adicionar a característica Tempo na arquitetura [Donegan 2008]

3. Implementação das variabilidades da LPS-BET

Nesta seção são apresentadas as implementações referentes às variabilidades *Acesso Adicional* e *Integração -> Terminal Integrado* usando POA. A implementação da característica *Acesso Adicional* como um componente caixa preta havia sido desenvolvida por Donegan [2008]. Já a característica *Terminal Integrado* não havia sido desenvolvida. Para fins de comparação, primeiramente, foi desenvolvido o componente

caixa-preta referente à esta variabilidade, de acordo com o modelo de Donegan [2008] e, em seguida, foi desenvolvida uma versão equivalente utilizando aspectos.

A implementação da LPS-BET realizada por Donegan [2008] foi feita utilizando: (i) o ambiente Eclipse [Eclipse 2008] para o desenvolvimento de programas em Java, (ii) o Hibernate [Hibernate, 2008] para persistência dos dados e (iii) o banco de dados PostGreSQL [POSTGRESQL 2008]. Portanto, na realização deste projeto os mesmos recursos tecnológicos foram utilizados. Para a implementação dos aspectos que representam as variabilidades foi usada a linguagem para desenvolvimento de aspectos AspectJ [ASPECTJ 2009].

3.1 Acesso Adicional

Esta característica opcional aplica-se aos produtos de Campo Grande e São Carlos. Refere-se a um *acesso adicional* ao sistema, acesso do tipo Web, em que o passageiro tem a possibilidade de consultar viagens e imprimir extratos.

Como se pode observar na Figura 12, que demonstra parcialmente a implementação original em componentes da variabilidade *AcessoAdicional* requer uma interface *ICartaoMgt* que é implementada pelo componente *CartaoMgr*.

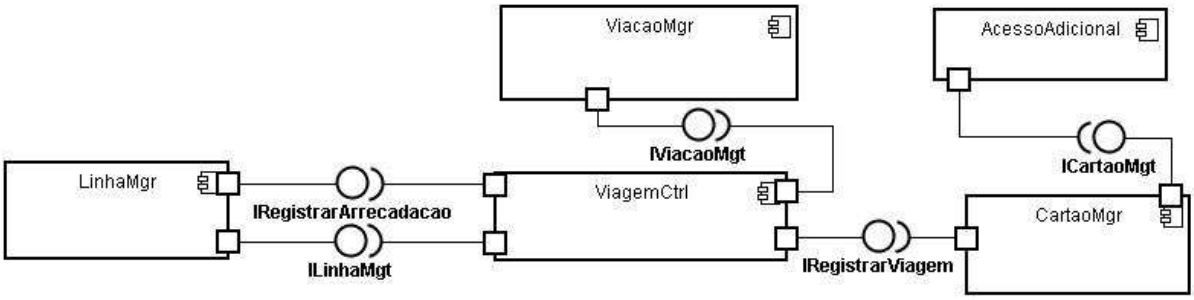

Figura 12. Arquitetura de Componentes para a Variabilidade Acesso Adicional

Utilizando-se como base a estrutura de componentes implementada, foi desenvolvida a versão desta variabilidade usando POA, que pode ser observada na Figura 13. O aspecto *AcessoAdicionalAspect* intercepta as operações da interface *ICartaoMgt*, utilizando-se do adendo do tipo around, pois ele direciona o fluxo de execução, que pode ser notado nas linhas 14 e 29 daFigura 14. A solução adotada neste trabalho foi elaborada com base no trabalho de Donegan [2008] que diz:

> Para soluções de projeto baseadas em componentes em que é necessário substituir um componente A' do núcleo por outro B' com a variabilidade a ser implementada, a solução usando aspectos requer que o aspecto entrecorte as operações das interfaces do componente A', não permitindo que as operações do componente A' sejam executadas. Pode ser feita uma solução em que o aspecto chama as operações do componente B' (componente de negócio) ou implementa as operações que estariam no componente B' (componente de controle ou de sistema). O aspecto executaria de modo a não permitir que a aplicação-referência utilize as operações do componente do núcleo (componente A'). Mesmo que o componente não seja utilizado, o componente se encontra conectado aos outros componentes e a arquitetura da aplicação-referência não corresponde à arquitetura de fato, podendo dificultar manutenções posteriores na aplicação. Nesse ponto a solução usando componentes possui vantagem sobre a solução usando aspectos, pois

a arquitetura dos componentes corresponde realmente à forma como a aplicação funciona. [Donegan 2008 p.99].

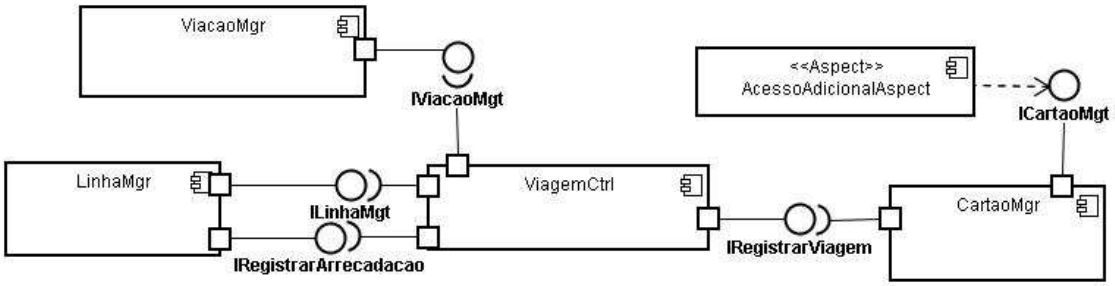

Figura 13. Variabilidade Acesso Adicional - Aspecto

Desta forma, foi implementada uma solução baseada no trabalho Donegan (2008), em que as operações da variabilidade Acesso Adicional fornecidas por meio da interface ICartaoMgt sejam entrecortadas pelo aspecto AcessoAdicionalAspect, fazendo com que os componentes da LPS-BET não consigam utilizá-las diretamente.

```
 1 package lps.bet.basico.cartaoMgr;
 2 import java.util.List;
 6 public aspect AcessoAdicionalAspect {
 7     public ICartaoMgt interfaceCartaoMgt;
 8     pointcut buscarViagensPorCartao(int cartaoID):
 9      call (List ICartaoMgt.buscarViagensPorCartao(int))
10      && args(cartaoID)
11      && !within(CartaoPgtoCartaoCtrl)
12      && !within(AcessoAdicionalAspect) ;
13     @SuppressWarnings("unchecked")
14     List around(int cartaoID): buscarViagensPorCartao(cartaoID){
15         try {
16             return interfaceCartaoMgt.buscarViagensPorCartao(cartaoID);
17         } catch (Exception e) {
18             e.printStackTrace();
19             List retorno = null;
20             return retorno;
21         }
22     }
23     pointcut buscarViagensTerminalPorCartao(int cartaoID):
24         call (List ICartaoMgt.buscarViagensTerminalPorCartao(int))
25         && args(cartaoID)
26         && !within(CartaoPgtoCartaoCtrl)
27         && !within(AcessoAdicionalAspect);
28     @SuppressWarnings("unchecked")
29     List around(int cartaoID): buscarViagensTerminalPorCartao(cartaoID){
30         try {
31             return interfaceCartaoMgt.buscarViagensTerminalPorCartao(cartaoID);
32         } catch (Exception e) {
33             e.printStackTrace();
34             List retorno = null;
35             return retorno;
36         }
37     }
38     public ICartaoMgt getInterfaceCartaoMgt() {
39         return interfaceCartaoMgt;
40     }
41     public void setInterfaceCartaoMgt(ICartaoMgt interfaceCartaoMgt) {
42         this.interfaceCartaoMgt = interfaceCartaoMgt;
43     }
44 }
```

Figura 14. Código fonte do Aspecto Acesso Adicional

Na Figura 15 o ponto de atuação buscarViagensPorCartao é apresentado nas linhas 8 a 12, criado com a denominação buscarViagensPorCartao. Ainda na Figura 15

observa-se nas linhas 14 a 22 a criação do adendo, que possui as funcionalidades tratadas pelo aspecto. A solução adotada na implementação do aspecto da Variabilidade *Acesso Adicional*, leva em conta a solução citada no trabalho de Donegan [2008], este é, o aspecto substitui o controlador que implementa as funcionalidades, e essas são tratadas nos adendos do aspecto. As operações que foram interceptadas pelo aspecto *AcessoAdicionalApect* foram as operações para buscar as viagens ocorridas nas linhas da BET e buscar as viagens que utilizaram terminais.

Na Figura 15 é apresentado o resultado da implementação da variabilidade *Acesso Adicional*, usando aspectos, em que são listadas as viagens realizadas por cartão, listando as linhas que o passageiro utilizou o cartão, bem como os terminais em que ele utilizou o cartão para ter acesso.

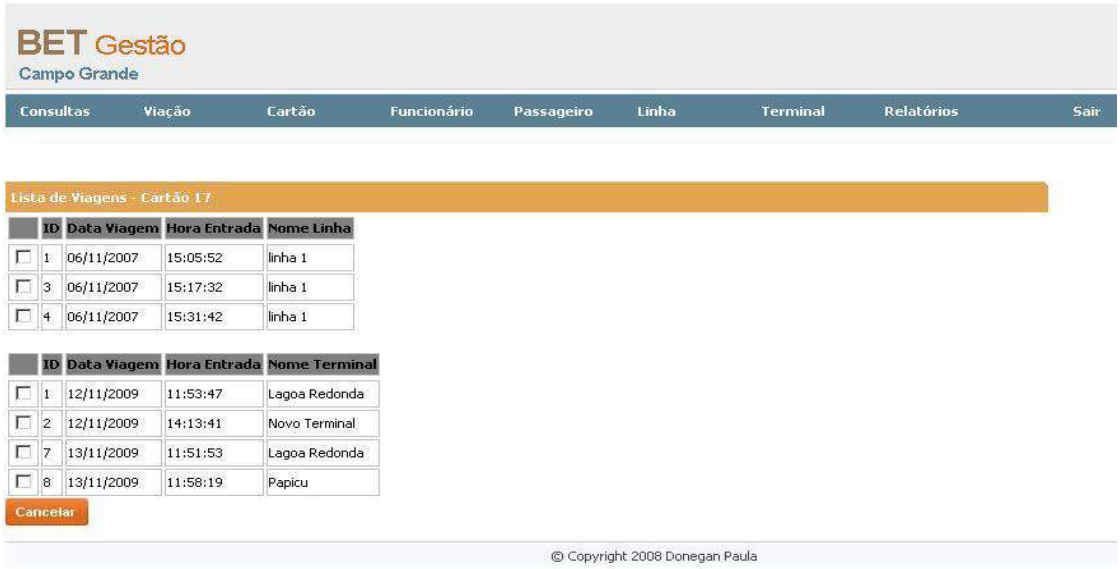

Figura 15. Tela de listagem de Viagens do passageiro

3.2 Terminal Integrado

Essa característica opcional faz parte dos produtos de Fortaleza e de Campo Grande. A integração do tipo terminal permite que o passageiro embarque e desembarque de terminais definidos na linha sem a necessidade de pagar uma nova passagem. Ou seja, a entrada nos terminais é feita através de validadores na entrada dos terminais, ou seja, catracas parecidas com as instaladas nos ônibus.

Como mencionado anteriormente, não havia implementação em componentes para esta característica. Então, foi implementada uma versão em componentes, com base na arquitetura em componentes ilustrada na Figura 16. Como se pode observar, foi criado um componente *TerminalIntegradoGUIMgr*, que implementa as operações exigidas pela interface *ITerminalIntegradoGUIMgt*.

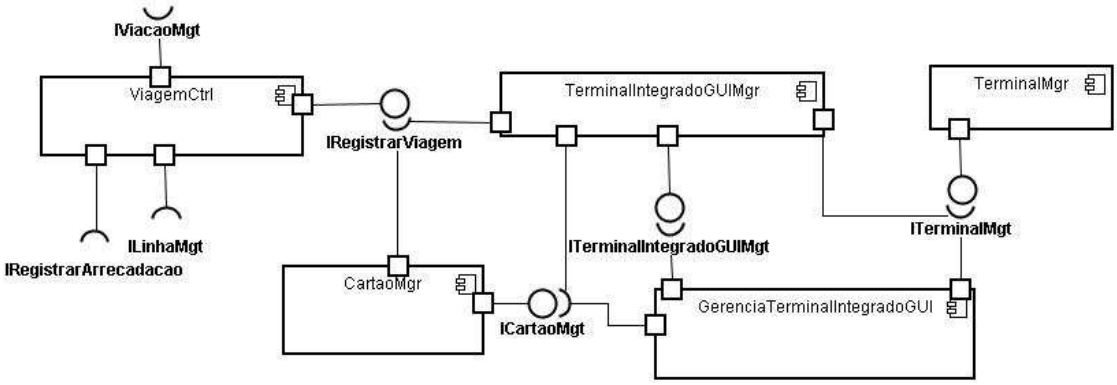

Figura 16. Diagrama de Componentes - Terminal Integrado

Na Figura 17 é ilustrado o diagrama de classes da referida variabilidade. Para o projeto dessa variabilidade as operações obrigatórias que devem ser implementadas foram definidas na interface. Essas operações foram implementadas pelo componente concreto *TerminalIntegradoGUIMgr*, que as fornece para a LPS-BET.

Pode-se observar Figura 17 que foi criada a operação tratarCartao, que tem a função de localizar o cartão do passageiro, verificar o tipo de passageiro, verificar a tarifa a ser debitada, registrando a viagem, liberando ou não a catraca para o embarque.

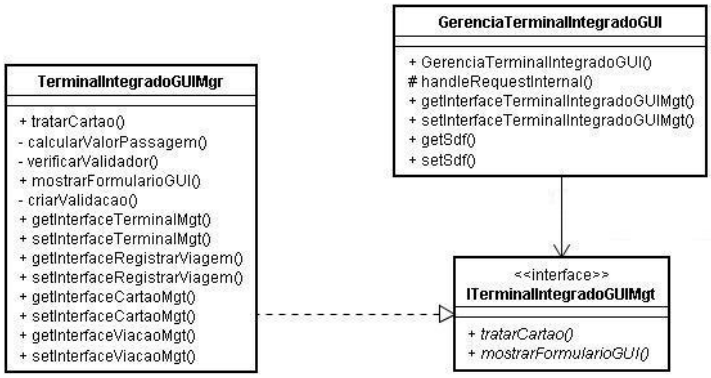

Figura 17. Diagrama de Classe - Integração terminal

Para essa variabilidade foi implementado ainda o componente controlador *GerenciaTerminalIntegradoGUI*. O resultado final da implementação dessa característica está ilustrado na Figura 18.

Figura 18. Tela que simula a catraca do Terminal

Com base na implementação baseada em componentes, foi implementada uma versão usando POA para essa variabilidade. O seu diagrama de componentes é ilustrado na Figura19, na qual é possível notar que o aspecto *GerenciaTerminalIntegradoAspect* intercepta as operações da interface *ITerminalIntegradoGUIMgt*. O código do aspecto é apresentado na Figura 20 a seguir.

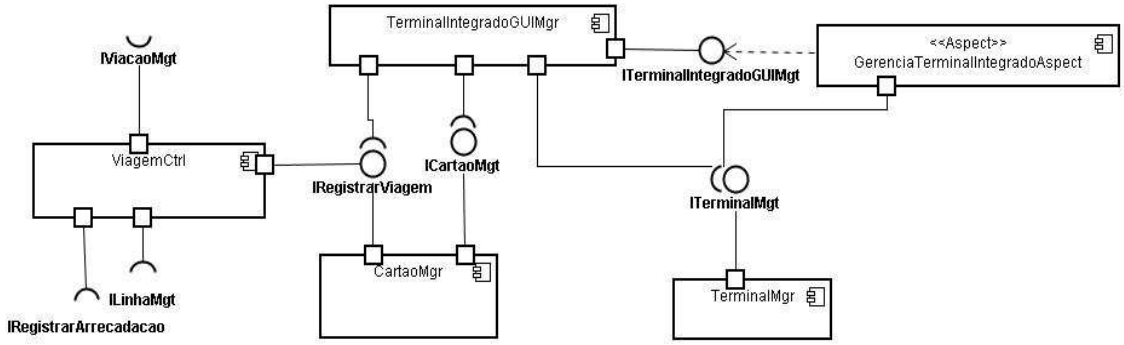

Figura19. Variabilidade Terminal Integrado Aspecto

```
1  package lps.bet.variabilidades;
2
3  import javax.servlet.http.HttpServletRequest;
4
5  import lps.bet.variabilidades.terminalIntegradoMgr.ITerminalIntegradoGUIMgt;
6  import org.springframework.web.servlet.ModelAndView;
7
8  public aspect GerenciaTerminalIntegradoAspect {
9      ITerminalIntegradoGUIMgt interfaceIterminalIntegradoGUIMgt;
10
11     pointcut tratarCartao(HttpServletRequest request, String dados):
12         call (ModelAndView ITerminalIntegradoGUIMgt.tratarCartao( HttpServletRequest , String ))
13         && args(request, dados)  && !within(GerenciaTerminalIntegradoAspect) ;
14
15     ModelAndView around(HttpServletRequest request, String dados):tratarCartao(request, dados){
16         return interfaceIterminalIntegradoGUIMgt.tratarCartao(request, dados);
17     }
18
19     pointcut mostrarForm(String erro, HttpServletRequest request ):
20         call (ModelAndView ITerminalIntegradoGUIMgt.mostrarFormularioGUI(  String, HttpServletRequest))
21         && args( erro, request)  && !within(GerenciaTerminalIntegradoAspect) ;
22
23     ModelAndView around( String erro,HttpServletRequest request):mostrarForm( erro,request){
24         return interfaceIterminalIntegradoGUIMgt.mostrarFormularioGUI(erro,request);
25     }
26
27     public ITerminalIntegradoGUIMgt getInterfaceIterminalIntegradoGUIMgt() {
28         return interfaceIterminalIntegradoGUIMgt;
29     }
30
31     public void setInterfaceIterminalIntegradoGUIMgt(
32             ITerminalIntegradoGUIMgt interfaceIterminalIntegradoGUIMgt) {
33         this.interfaceIterminalIntegradoGUIMgt = interfaceIterminalIntegradoGUIMgt;
34     }
35 }
```

Figura 20. Código do aspecto Terminal Integrado

4. Análise dos resultados

Para analisar os resultados obtidos neste trabalho, foram definidas algumas métricas. Na Tabela 2 são apresentadas as métricas obtidas da implementação da variabilidade *acesso adicional* e a métricas referentes à *integração terminal* são apresentadas na Tabela 3.

Tabela 2. Métricas relacionadas à Variabilidade Acesso Adicional

Métrica	**Valor**
Número de interfaces envolvidas com a variabilidade	1
Número de componentes envolvidos com a variabilidade	4
Número de aspectos envolvidos com a variabilidade	1
LOC do controlador	54
MLOC (Linhas de código dos métodos) buscarViagensPorCartao(cartaoID);	2
MLOC (Linhas de código dos métodos) buscarViagensTerminalPorCartao(cartaoID);	2
Número de interfaces interceptadas por aspectos	1
Número de adendos por aspecto	2
Número de operações de cada interface interceptada	2
Número de classes que referenciam a interface substituída por aspecto	4
ALOC (linhas de código dos adendos)	6
LOC do aspecto	58

Tabela 3. Métricas relacionadas à Variabilidade Terminal Integrado

Métrica	**Valor**
Número de interfaces envolvidas com a variabilidade	1
Número de componentes envolvidos com a variabilidade	5
Número de aspectos envolvidos com a variabilidade	1
LOC do controlador	40
MLOC (Linhas de código dos métodos) tratarCartao(HttpServletRequest request, String dados)	65
MLOC (Linhas de código dos métodos) mostrarFormularioGUI(String erro,HttpServletRequest request)	32
Número de interfaces interceptadas por aspectos	1
Número de adendos por aspecto	2
Número de operações de cada interface interceptada	2
Número de classes que referenciam a interface substituída por aspecto	3
ALOC (linhas de código dos adendos)	7
LOC do aspecto	36

Contudo, pode-se observar por meio das métricas apresentadas, que não houve muita diferença de implementação. Como citado na Seção 3.2, os aspectos foram implementados com base na sugestão de Donegan [2008], isto é, implementar os aspectos e interceptar as operações das interfaces requeridas utilizando os componentes de negócio existentes.

Analisando as métricas relacionadas à variabilidade *Acesso Adicional* (Tabela 2), observa-se que não há muita diferença na implementação utilizando-se aspectos. Por exemplo, entre o LOC do controlador e o LOC do aspecto observa-se uma diferença mínima de linhas de código. Observando ainda a soma do MLOC dos métodos e o ALOC dos adendos, os números resultantes são bem próximos, apontado assim pouca diferença de implementação. Para as métricas relacionadas à variabilidade Terminal Integrado, refletem uma situação muito similar.

Outro ponto a ser observado é o número de adendos por aspecto e o número de operações das interfaces interceptadas, que são iguais.

Dessa forma, a utilização de aspectos não trouxe benefícios visíveis para a LPS-BET na substituição das variabilidades propostas. Como nenhuma alteração na forma como a variabilidade funciona foi necessária, o aspecto apenas substituiu as funções dos componentes de negócio. Os componentes de negócio originais, por sua vez, não são mais utilizados por outras partes da arquitetura, embora ainda continuem conectados. Assim confirmou-se a afirmação de Donegan [2008 p. 99] de que "nesse ponto a solução usando componentes possui vantagem sobre a solução usando aspectos, pois a arquitetura dos componentes corresponde realmente à forma como a aplicação funciona". Isto é, constatou-se que não é vantajosa a substituição dos componentes de negócio da arquitetura da LPS-BET por aspectos.

5. Conclusão

Este trabalho teve por objetivo implementar uma solução baseada em aspectos para as variabilidades *Acesso Adicional* e *Integração Terminal* e avaliar os resultados alcançados em relação às implementações das mesmas variabilidades usando componentes. Constatou-se que a solução baseada em componentes possui vantagens sobre a solução usando POA. Isto porque o componente continua conectado a outros componentes, embora as operações sejam executadas a partir do aspecto, diferentemente da arquitetura projetada para a LPS-BET.

Assim, o objetivo do trabalho foi alcançado, além da experiência adquirida tanto no desenvolvimento e manutenção de uma arquitetura de LPS baseada em componentes e aspectos, quanto na configuração de um ambiente com os recursos tecnológicos necessários ao desenvolvimento deste trabalho.

Foram encontradas algumas dificuldades em relação ao projeto existente como, por exemplo, na documentação constava uma nomenclatura e no código fonte outro nome. Também foram encontradas dificuldades em relação à configuração do ambiente, sendo necessário estudar sobre as ferramentas utilizadas para alcançar a configuração adequada.

Pela experiência adquirida com o estudo dos conceitos de LPS, POA e da arquitetura da LPS-BET, pode-se afirmar que a implementação de novas variabilidades utilizando aspectos pode ser benéfica, desde que os aspectos não alterem a configuração da estrutura dos componentes da arquitetura ou ainda quando o seu uso tiver sido planejado desde o início do projeto da LPS.

Como trabalho futuro, pretende-se fazer novos estudos de casos para produzir evidências das situações nas quais o uso de POA para implementar variabilidades de LPS é vantajoso. Outro trabalho possível é fazer uma evolução da LPS-BET para adicionar novas variabilidades usando aspectos e usando componentes a fim de comparar resultados alcançados. Por fim, pesquisadores da USP/ São Carlos que fazem parte deste projeto, utilizarão uma versão do Captor implementada com POA para gerar os produtos com as variabilidades implementadas usando aspectos.

Referências

Anastasopoulos, M.; Gacek, C. (2001) Implementing Product Line Variabilities. ACM Sigsoft Software Engineering Notes, v. 23, n. 3, p. 109–117.

Aspectj Team (2009) The AspectJ Programming Guide. Acessado em maio, 2009. Disponível em: http://www.eclipse.org/aspectj/doc/released/progguide/

Bachmann, F.; Goedicke, M.; Leite, J.; Nord, R.; Pohl, K.; Ramesh, B.; Vilbig, A. (2003) A Meta-Model for Representing Variability in Product Family Development. In: Proceedings of the 5th International Workshop on Software Product-Family Engineering - PFE 2003, Springer, Siena, Italy, p. 66–80.

Becker, M.; Kaiserslautern, G. (2003) Towards a General Model of Variability in Product Families. In: Proceedings of the 1st Workshop of Software Variability Management at ICSE 2003, Groningen, The Netherlands, p. 9.

Bosch, J.; Florijn, G.; Greefhorst, D.; Kuusela, J.; Obbink, J.; Pohl, K(2002). Variability Issues in Software Product Lines. Software Product-Family Engineering: 4th International Workshop - PFE 2001, p. 11–19, Bilbao, Spain.

Brown, Alan W. (2000) Large-scale, component-based development. Upper-Saddle River : Prentice-Hall.

Clements, P.; Northrop, L. (2001) Software product lines: practices and patterns. 1. ed. Boston: Addison-Wesley

Donegan, P. M. (2008) Geração de Famílias de Produtos de Software com Arquitetura Baseada em Componentes. Dissertação (Mestrado em Ciência da Computação e Matemática Computacional). ICMC-USP/São Carlos. São Carlos, SP.

Eclipse IDE (2008). Eclipse Project. Disponível para acesso na URL: http://www.eclipse.org/, em junho de 2008.

Junior, E. A. O.; Gimenes, I. M. S.; Huzita, E. H. M.; Maldonado, J. C. A. (2005) Variability Management Process for Software Product Lines. In: Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research - Cascon, p. 225–241.

Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C. V.; Loingtier, J. M.; IRWIN, J (1997). Aspect-Oriented Programming. European Conference on Object-Oriented Programming (ECOOP), LNCS (1241), Springer-Verlag, Finland., June.

Hibernate (2008). Hibernate Java persistence framework project. Disponível para acesso na URL: http://www.hibernate.org/, em junho de 2008.

Kang, K.; Cohen, S.; Hess, J.; Novak, W.; Peterson S. (1990). Feature-Oriented Domain Analysis (FODA): Feasibility Study. CMU/SEI-90-TR-21, SEI, USA.

Linden, V. Der, F.; Pohl K.; Böckle G. (2005) Software Product Line Engineering, Foundations, Principles, and Techniques.

Meilsmith, G.A (2008). Uma investigação quanto ao uso de aspectos para implementação de variabilidades de Linha de Produto de Software. Trabalho de Graduação apresentado ao Curso de Ciência da Computação, do Centro de Tecnologia, da Universidade Estadual de Maringá..

Oliveira Junior, E. A. ; Gimenes, I. M. de S.; Huzita, E. H. M.; Maldonado, J. C. (2005) A Variability Management Process for Software Product Lines. Proceedings of the 15th Annual International Conference of Computer Science and Software Engineering (CASCON), 2005, Ontario - Toronto - Canada : IBM, v. 1. p. 30-44.

Postgresql (2008). PostgreSQL Global Development Group. Disponível em http://www.postgresql.org/. Acessado em junho, 2008.

Shimabukuro, E. K. J (2006). Um Gerador de Aplicações Configurável. Dissertação de Mestrado, Instituto de Ciências Matemáticas e de Computação - ICMC, Universidade de São Paulo (USP),.

Spring (2009). Spring Framework. Disponível para acesso na URL: http://www.springsource.org, em junho de 2009.

Weiss, D.; CHI TAU, R. L (1999). Software product-line engineering: a family-based software development process. Boston: Addison-Wesley,.