
Geração de Famílias de Produtos de
Software com Arquitetura Baseada em
Componentes

Paula Marques Donegan

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito: 23 de junho de 2008

Assinatura: _____

Geração de Famílias de Produtos de Software com Arquitetura Baseada em Componentes

Paula Marques Donegan

Orientador: *Prof. Dr. Paulo Cesar Masiero*

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação — ICMC/USP como parte dos requisitos para obtenção do título de Mestre em Ciências de Computação e Matemática Computacional.

USP - São Carlos
Junho/2008

Agradecimentos

Ao longo destes anos de estudos tenho muito a agradecer e a muitas pessoas que me ajudaram direta e indiretamente.

Agradeço antes de tudo a Deus, por me dar oportunidade, capacidade e vontade para realizar este trabalho.

Agradeço ao professor Paulo Cesar Masiero que me ensinou muito nos últimos anos e acreditou no meu potencial. Obrigada pelas diversas sugestões desde o assunto a ser pesquisado, artigos e livros a serem lidos, até as vírgulas e palavras a serem corrigidas.

Agradeço aos professores do mestrado que contribuíram para minha formação e aos funcionários do ICMC que me deram constante auxílio.

Agradeço ao Otávio que me ajudou muito durante o mestrado. Obrigada por me mostrar as diversas tecnologias, ensinar-me muita coisa e participar de discussões referentes à minha pesquisa. Obrigada pela paciência, pelo companheirismo e por ser você.

Agradeço à minha linda família. Palavras não são suficientes para expressar como vocês me ajudaram, o quanto amo, admiro e tenho orgulho de vocês. Thanks dad and uncle John for reviewing my English and being interested in my research. Dad, thanks for always giving me good advice and telling me to try and try again whenever I don't succeed.

Agradeço às famílias pg-sce 2006, Labes e de amigos cearenses: Abe, Ades, Alê, André Endo, André Freire, André Maluquinho, Antonielly, Aretha, Bira, Camila, Carlos, Dalcimar, Débora, Edson, Eduardo, Erika, Ellen, Elievam, Etienne, Fabiano, Gabriel, Gabriela, Heráclito, Ivan, Jarbas, KLB, Kika, Jaú, Lúcio, Luíza, Maldonado, Marcão, Marcella, Marcelo, Marília, Marllos, Mateus, Matrix, Maycon, Meltje, Mel, Otávio, Paula, Rosana, Rosely, Resina, Simone, Sofia, Stanley, Taty, Tott, Valdecir, Valter, Van, Vânia, Vasco e Viviane.

Agradeço a todas as pessoas que contribuíram de alguma forma para a realização deste trabalho. Obrigada a meus amigos que mesmo à distância não deixaram de me apoiar.

Agradeço ao Giovano por ter me colocado na prática da Engenharia de Software, pois me encontrei e me encantei.

Obrigada Belchior pelas palavras de incentivo para fazer esse mestrado e pela companhia na montagem e na revisão da minha monografia e de artigos dia e noite. O senhor foi um professor exemplar e faz muita falta entre nós. Obrigada por ter sido um professor, um orientador, um conselheiro e um amigo.

Finalmente, agradeço à FAPESP pelo apoio financeiro.

*You gain strength, courage and confidence by every experience
in which you really stop to look fear in the face. (...)
You must do the thing you think you cannot do.
(Eleanor Roosevelt)*

Uma adaptação de um processo de software específico para linhas de produtos de software é proposta. Esse processo tem o objetivo de ser ágil, de apoiar as atividades de projetar e desenvolver características com re-trabalho mínimo e de facilitar a engenharia de aplicações. A fase de engenharia de domínio é iterativa e incremental e propõe uma arquitetura baseada em componentes. As aplicações são geradas por um gerador de aplicações configurável a partir de uma linguagem de modelagem de aplicações baseada no modelo de características. Adicionalmente, é apresentado um estudo detalhado de alternativas para projeto de componentes em uma linha de produtos, considerando componentes do tipo caixa-preta e caixa-branca visando a facilitar a composição e o reúso de componentes. Uma linha de produtos para controle de Bilhetes Eletrônicos em Transporte urbano (BET) foi projetada e implementada usando o processo proposto. Alternativas para a implementação baseada em aspectos de requisitos transversais e de variabilidades da linha de produtos BET, bem como sua geração automática, são apresentadas e discutidas.

Abstract

Adaptation of a specific software product line process is described. The adapted process aims to be agile, minimising rework for feature design and development activities and facilitating applications engineering. The domain engineering phase is iterative and incremental, using a component-based architecture. Applications are generated by an application generator configurable using an application modeling language based on the features diagram. Additionally, we present a detailed study of alternatives for design of product line components, considering white-box and black-box components, aiming to facilitate component composition and reuse. A product line for control of Electronic Transport Cards (ETC) was designed and developed using the proposed process. We present and discuss implementation alternatives based on aspect-oriented development to represent crosscutting and variability requirements of the ETC product line, as well as the automated generation of these requirements.

Sumário

Resumo	9
Abstract	11
1 Introdução	1
1.1 Contextualização	1
1.2 Motivação	2
1.3 Objetivos	3
1.4 Organização	4
2 Reúso de Software	5
2.1 Considerações Iniciais	5
2.2 Reúso de Software	6
2.3 Linhas de Produtos de Software	8
2.3.1 Processos de Engenharia de Linhas de Produtos de Software	9
2.4 Componentes e Frameworks de Componentes	15
2.4.1 Modelos de Componentes de Software	15
2.4.2 Frameworks de Componentes de Software	18
2.5 Desenvolvimento Baseado em Componentes	19
2.5.1 Catalysis	19
2.5.2 KobrA	20
2.5.3 <i>UML Components</i>	21
2.6 Geradores de Aplicação	22
2.6.1 Gerador de Aplicação Configurável	27
2.7 Considerações Finais	31
3 Arquitetura de Software, Componentes e Aspectos	33
3.1 Considerações Iniciais	33
3.2 Programação Orientada a Aspectos	34
3.2.1 A Linguagem AspectJ	35
3.2.2 A Linguagem FuseJ	36
3.3 Arquitetura de Software	39
3.3.1 A Linguagem Acme	41
3.3.2 O Desenvolvimento Orientado a Aspectos e as Linguagens de Descrição de Arquitetura	42

3.4	Desenvolvimento de Software Baseado em Componentes e em Aspectos	43
3.4.1	Abordagens de Componentes Aspectuais	44
3.5	Considerações Finais	46
4	Desenvolvimento de Linhas de Produtos de Software	47
4.1	Considerações Iniciais	47
4.2	Princípios Adotados para o Desenvolvimento da LPS	48
4.3	Adaptações Propostas ao Processo ESPLEP	48
4.3.1	Engenharia de Domínio	50
4.3.2	Engenharia de Aplicação	54
4.4	Processo ESPLEP Adaptado e Instanciado para a LPS-BET	55
4.4.1	Ciclo de Desenvolvimento do Núcleo	58
4.4.2	Ciclo de Desenvolvimento da Aplicação-Referência de Fortaleza	65
4.4.3	Ciclo de Desenvolvimento da Aplicação-Referência de Campo Grande	68
4.4.4	Ciclo de Desenvolvimento da Aplicação-Referência de São Carlos	70
4.5	Informações sobre a Construção da LPS-BET	73
4.6	Considerações Finais	77
5	Decisões de Projeto da LPS-BET	79
5.1	Considerações Iniciais	79
5.2	Projeto Baseado em Componentes	79
5.2.1	Modelagem com novas classes	80
5.2.2	Modelagem com novas subclasses	84
5.3	Uso de Aspectos no Projeto da LPS-BET	92
5.3.1	Aspectos para Implementar Requisitos Não-Funcionais	92
5.3.2	Aspectos para Implementar Variabilidades da LPS-BET	95
5.4	Considerações Finais	100
6	Uso do Gerador Captor para a Engenharia de Aplicações da LPS-BET	101
6.1	Considerações Iniciais	101
6.2	Configuração do Captor para o Domínio BET	101
6.3	Engenharia de Aplicações da LPS-BET	106
6.3.1	Engenharia da Aplicação BET de Campo Grande	108
6.3.2	Engenharia de Outras Aplicações	116
6.4	Considerações Finais	119
7	Conclusão	123
7.1	Considerações Iniciais	123
7.2	Contribuições	123
7.3	Trabalhos Futuros	124
A	Documento de Requisitos do Sistema BET de Fortaleza	137
B	Documento de Requisitos do Sistema BET de Campo Grande	143
C	Documento de Requisitos do Sistema BET de São Carlos	149
D	Alguns Artefatos da LPS-BET	155

Lista de Figuras

2.1	Processo de Desenvolvimento de Linhas de Produtos de Software no ESPLEP (Gomaa, 2004)	10
2.2	O Processo de Engenharia da Linha de Produtos no ESPLEP (Gomaa, 2004)	11
2.3	Processo da Engenharia de Aplicação no método FAST (Weiss e Lai, 1999)	14
2.4	Processo de Desenvolvimento do método <i>UML Components</i> (Cheesman e Daniels, 2001)	22
2.5	Exemplo de uma especificação de uma bóia náutica	25
2.6	Hierarquia de módulos das bóias náuticas (Weiss e Lai, 1999)	26
2.7	Gabarito do módulo de Monitor do Sensor	26
2.8	Estratégia de geração de bóias náuticas por composição (Weiss e Lai, 1999)	27
2.9	Arquitetura do Captor	29
2.10	Etapa do fluxo de execução do Captor para geração de artefatos (Shimabukuro, 2006)	30
2.11	Configuração do Captor para um determinado domínio (Shimabukuro, 2006)	30
3.1	Exemplo de aspecto em AspectJ	37
3.2	Estrutura geral da entidade de configuração do FuseJ (Suvée <i>et al.</i> , 2006)	38
3.3	Exemplo de codificação de interação orientada a aspecto entre os componentes TransferNetCe Logger(Suvée <i>et al.</i> , 2006)	39
3.4	Interação entre componentes TransferNetCe Logger(Suvée <i>et al.</i> , 2006)	39
3.5	Diagrama de um sistema simples cliente-servidor na Acme (Garlan <i>et al.</i> , 2000)	42
3.6	Sistema simples de cliente-servidor na Acme (Garlan <i>et al.</i> , 2000)	42
4.1	Incrementos verticais e horizontais	50
4.2	Incrementos de desenvolvimento de uma LPS	52
4.3	Ciclos de desenvolvimento da LPS e as suas fases	52
4.4	Visão geral do processo de desenvolvimento da LPS-BET	57
4.5	Diagrama de Características do núcleo da LPS-BET	59
4.6	Parte do diagrama de casos de uso da LPS-BET	61
4.7	Arquitetura da LPS-BET	62
4.8	Arquitetura de componentes do núcleo da LPS-BET	63
4.9	Parte dos componentes e interfaces do núcleo da LPS-BET	64
4.10	<i>Bean</i> para injeção de dependências do componente ViagemCtrl	65
4.11	Arquitetura de Componentes para a aplicação-referência de Fortaleza	67
4.12	Arquitetura de Componentes para a aplicação-referência de Campo Grande	69
4.13	Arquitetura de Componentes para a aplicação-referência de São Carlos	72

4.14	Organização da implementação da LPS-BET	74
4.15	Linhas de Código (LOC) resultantes dos ciclos de desenvolvimento da LPS-BET	76
4.16	Porcentagem de reuso e desenvolvimento das aplicações-referência da LPS-BET	77
5.1	Parte do diagrama de características relacionadas a <i>Terminal</i>	80
5.2	Parte do modelo de classes relacionado à característica <i>Terminal</i>	81
5.3	Arquitetura de componentes parcial incluindo a característica <i>Terminal</i>	82
5.4	Parte do diagrama de características relacionadas à característica <i>Formas de Integração</i>	83
5.5	A característica <i>LinhaIntegrada</i> no modelo de classes	84
5.6	Componentes para implementar a característica <i>Linha Integrada</i>	84
5.7	As características <i>Tempo</i> e <i>Número de Viagens de Integração</i> no modelo de classes	85
5.8	SistemaViarioUrbanocomo uma classe parametrizada com pontos internos de variação	85
5.9	Classes em um componente caixa-branca, incluindo as características <i>Tempo</i> e <i>Número de Viagens de Integração</i>	86
5.10	Componentes caixa-preta de negócio para o grupo de características de <i>Integração</i>	86
5.11	Operações da Interface ITempoMgt	87
5.12	Solução <i>b</i> para integrar a característica <i>Tempo</i> na arquitetura da LPS-BET	87
5.13	Solução <i>c</i> para integrar a característica <i>Tempo</i> na arquitetura da LPS-BET	88
5.14	Interface IProcessarViagem	89
5.15	Classe TempoViagemCtrl do componente de mesmo nome com implementação da interface IProcessarViagem	89
5.16	<i>Beans</i> relacionados ao componente TempoViagemCtrl	90
5.17	Uma solução para integrar as características <i>Tempo</i> e <i>Número de Viagens</i> na arquitetura da LPS-BET	91
5.18	Adendos, operações e atributos dos aspectos Autenticaçãoe Autorização	93
5.19	Arquitetura dos aspectos Autenticaçãoe Autorizaçãoe dos componentes entrecortados	94
5.20	Implementação do aspecto Autenticação	94
5.21	Aspecto abstrato e aspectos contratos para representar a característica <i>Integração</i>	96
5.22	Implementação do aspecto abstrato IntegraçãoCtrl	97
5.23	Uma solução usando aspectos para adicionar a característica <i>Tempo</i> na arquitetura	97
5.24	Implementação do aspecto ViagemTempoCtrl	98
5.25	<i>Beans</i> relacionados ao aspecto TempoViagemCtrl	99
6.1	Árvore de formulários definidos no Captor para a LPS-BET	102
6.2	Estrutura hierárquica dos formulários para a LPS-BET	103
6.3	Definição do elemento variante Integração	104
6.4	Exemplo de Estrutura XML da Especificação do Captor	105
6.5	Parte do gabarito bet-servlet.xml.xsl	106
6.6	Parte do gabarito menu.xml.xsl	107
6.7	Arquivo de Mapeamento de Gabaritos para a LPS-BET	107
6.8	Diagrama de Características para a aplicação de Campo Grande	110
6.9	Criação de um novo projeto para geração de uma aplicação pelo Captor	111
6.10	Definição do elemento Nome da Aplicação para a aplicação de Campo Grande	111
6.11	Definição dos valores dos elementos para a aplicação de Campo Grande	112
6.12	Captor gera arquivo XML com valores dos elementos variantes dos formulários	112
6.13	Estrutura XML da Especificação de Campo Grande no Captor	113

6.14	Captor gera arquivo XML com valores dos elementos variantes dos formulários . . .	113
6.15	Parte do gabarito bet-servlet.xml.xsl que identifica características de Campo Grande	114
6.16	Fragmento do arquivo de configuração gerado pelo Captor para Campo Grande . . .	115
6.17	Aplicação web do sistema BET de Campo Grande	116
6.18	Simulador do validador da LPS-BET	116
6.19	Aplicação web do núcleo da LPS-BET	117
6.20	Exemplo de Estrutura XML da Especificação de uma combinação aleatória de ca- racterísticas no Captor	119
6.21	Fragmento do arquivo de configuração gerado pelo Captor para aplicação com combinação de características da LPS-BET	120
6.22	Aplicação web para uma combinação aleatória de características da LPS-BET . . .	121
D.1	Diagrama de Casos de Uso da LPS-BET	156
D.2	Diagrama de Características para a LPS-BET	157
D.3	Diagrama de Classes da LPS-BET	158
D.4	Diagrama de Estados para os componentes do Ônibus	159
D.5	Diagrama de Estados para os componentes relacionados ao ValidadorServidorCtrl	160

Lista de Tabelas

2.1	Subatividades e artefatos produzidos para as atividades iniciais do processo ESPLEP	12
2.2	Variabilidades das Bóias Náuticas	25
4.1	Características opcionais ou alternativas para aplicações da LPS-BET	60
4.2	Casos de uso necessário para os incrementos da LPS-BET	60
4.3	Casos de uso da análise e do projeto parcial no ciclo de desenvolvimento do núcleo	64
4.4	Características e casos de uso opcionais da aplicação-referência de Fortaleza	66
4.5	Características e casos de uso opcionais da aplicação-referência de Campo Grande	68
4.6	Características e casos de uso opcionais da aplicação-referência de Campo Grande	71
4.7	Métricas gerais da LPS-BET	75
4.8	Métricas relacionadas ao núcleo e às variabilidades da LPS-BET	75
4.9	Médias de classes, atributos, métodos e LOC por componente para o núcleo, as variabilidades e a LPS-BET	75
4.10	Porcentagem de Reúso do Núcleo da LPS-BET	76
5.1	Vantagens e desvantagens no uso de componentes do tipo caixa-branca ou caixa-preta	82
5.2	Diferentes opções de projeto de variabilidades que requerem novos atributos e operações em uma LPS	86
5.3	Soluções de junção de variabilidades por meio de controladores na arquitetura baseada em componentes caixa-preta	91
6.1	Disposição dos elementos variantes nos formulários	103

Lista de Siglas

Sigla	Significado
ACBSE	Aspect Component Based Software Engineering
ADL	Architecture Description Language
AOCE	Aspect Oriented Component Engineering
BET	Bilhetes Eletrônicos de Transporte municipal
Captor	Configurable Application Generator
CCM	Corba Component Model
CIDL	Component Implementation Definition Language
COOL	Coordination Aspect Language
COTS	Commercial Off The Shelf
DLL	Dynamic Link Libraries
DSBC	Desenvolvimento de Software Baseado em Componentes
DSBC/A	Desenvolvimento de Software Baseado em Componentes e Aspectos
DSOA	Desenvolvimento de Software Orientado a Aspectos
EJB	Enterprise Java Beans
ESPLEP	Evolutionary Software Product Line Engineering Process
FAC	Fractal Aspect Component
FAST	Family-oriented, Abstraction, Specification, and Translation
FODA	Feature-Oriented Domain Analysis
FORM	Feature-Oriented Reuse Method
FWS	Floating Weather Station
GREN	Gerenciamento de Recursos de Negócio
GUI	Graphical User Interface
IDL	Interface Description Language
LMA	Linguagem de Modelagem de Aplicação
LOC	Lines of Code
LPS	Linha de Produtos de Software
MLOC	Method Lines of Code
MSIL	Microsoft Intermediate Language
MTL	Mapping Transformation Language
MVC	Model View Controller

OMG	Object Management Group
OSGi	Open Service Gateway Initiative
PASTA	Process and Artifact State Transition Abstraction
PLUS	Product Line UML-based Software engineering
POA	Programação Orientada a Aspectos
PU	Processo Unificado
PuLSE	Product Line Software Engineering
RIDL	Remote Interface Aspect Language
UC	Use Case
UML	Unified Modeling Language
XML	eXtensible Markup Language
XSL	eXtensible Stylesheet Language
XSLT	eXtensible Stylesheet Language Transformations

Introdução

1.1 Contextualização

Características (*features*) comunicam requisitos ou funções em termos de abstrações funcionais distintamente identificáveis que necessitam ser implementadas, testadas e mantidas (Kang *et al.*, 1998). Uma coleção de sistemas que compartilham características comuns foi chamada por Parnas (1979) de “família de sistemas”. Atualmente, uma família de sistemas é mais conhecida como uma Linha de Produtos de Software (LPS) ou uma família de produtos de software. Uma LPS consiste de um conjunto de sistemas de software compartilhando características comuns e gerenciadas que satisfazem a necessidades específicas de um segmento particular de mercado ou de negócio que são desenvolvidos a partir de um conjunto comum de ativos centrais de forma sistemática (Clements e Northrop, 2001). É vantajoso desenvolver uma linha de produtos quando existe mais a ser ganho ao se analisar os sistemas de forma coletiva, ao invés de analisá-los separadamente, ou seja, quando os sistemas apresentam mais características em comum do que características que os distinguem (Parnas, 1979). Nesse caso, o processo de desenvolvimento da LPS é dividido em engenharia de domínio, com as atividades genéricas de desenvolvimento dos artefatos da linha, e em engenharia de aplicações, com as atividades de desenvolvimento de aplicações. Caso contrário, são desenvolvidos os sistemas tradicionais únicos (“*single systems*”).

Características variantes, que podem estar presentes em apenas alguns dos produtos de uma linha ou família, diferenciam um produto de outros. O projeto de uma LPS pode usar diversas técnicas de projeto de software para facilitar o reúso, como frameworks orientados a objetos, componentes, geradores de código, padrões de projeto, diagramas de características e linguagens

orientadas a aspectos. Essas técnicas podem ser usadas isoladamente ou em diferentes combinações.

Os componentes correspondem ao principal conceito preferido por vários autores para projetar LPS (Gomaa, 2004; Atkinson *et al.*, 2001; Pohl *et al.*, 2005), enquanto abordagens de geração são preferidas por outros autores (Weiss e Lai, 1999). O Desenvolvimento de Software Baseado em Componentes (DSBC) surgiu como uma perspectiva de desenvolvimento de software caracterizada pela composição de partes já existentes. Os componentes podem ser implementados como caixa-branca ou caixa-preta. Se os componentes forem caixas-brancas, a combinação de partes já existentes pode ser feita acessando e alterando partes internas do componente, ou seja, o código do componente. Isso é possível se o componente tiver sido desenvolvido internamente à empresa. Caso sejam caixas-pretas, são feitas apenas combinações de componentes por meio das interfaces e criação de novos componentes sem alterar código interno dos componentes, como é o caso de COTS (do inglês *Commercial Off-The-Shelf*) que são geralmente obtidos de terceiros, mas podem também ser desenvolvidos internamente.

Diversos artigos enfatizam a dificuldade de elicitar, representar e implementar variabilidades no contexto de uma LPS (Bachmann *et al.*, 2003; Becker e Kaiserslautern, 2003; Bosch *et al.*, 2002; Junior *et al.*, 2005; Anastasopoulos e Gacek, 2001). Características variantes são difíceis de implementar, pois elas podem se espalhar por diversas unidades de decomposição (como classes e componentes, dependendo da tecnologia utilizada) e são geralmente implementadas usando padrões de projeto, classes parametrizadas e outras técnicas de baixo nível. Diversos pesquisadores têm investigado essa questão e têm proposto outras soluções baseadas em linguagens de programação com programação orientada a aspectos e programação orientada a características (Mezini e Ostermann, 2004; Apel e Batory, 2006; Heo e Choi, 2006; Lee *et al.*, 2006; Anastasopoulos e Muthig, 2004). A combinação de aspectos, componentes e frameworks é proposta por Griss (2000) para implementar LPS, mas sem apresentar uma proposta particular em relação a como fazer a combinação.

1.2 Motivação

Estudos têm indicado problemas a serem investigados, como aspectos, arquiteturas, métodos ágeis, componentes de software e LPS (Finkelstein e Kramer, 2000; Osterweil, 2007; Taylor e van der Hoek, 2007). Percebe-se que a maioria das pesquisas relacionadas a LPS são divididas em duas vertentes: uma foca mais na engenharia de domínio e outra na engenharia de aplicação.

A vertente que foca mais na engenharia de domínio desenvolve um domínio geralmente tomando como base algumas aplicações-referência da LPS. Normalmente, para essa vertente são consideradas as características relacionadas à linha e a partir disso elabora-se o projeto da linha. De modo geral, o projeto é feito baseado em componentes, pois componentes têm o conceito de reuso bem definido e facilitam a troca de um pelo outro para obter aplicações da linha. Como

exemplos de autores que seguem essa linha tem-se Gomaa (2004) e Clements e Northrop (2001). Gomaa usa componentes no processo de desenvolvimento de LPS, porém, não detalha como identificar e implementar os componentes. Na fase de engenharia de aplicação, não entra em detalhes, apenas sugere a montagem das aplicações usando os componentes desenvolvidos anteriormente.

Outra linha tem como foco a engenharia de aplicação e o objetivo é a automação dessa fase, sendo mais ágil a engenharia de aplicações. Por outro lado essa abordagem de forma geral é menos flexível, pois para realizar mudanças deve-se mudar a forma de gerar as aplicações. Como forma de melhorar o processo de desenvolvimento de aplicações da LPS, essa linha considera o uso de linguagens de modelagem de aplicações e o uso delas em geradores de aplicações. Um exemplo de autores que seguem essa abordagem é Weiss e Lai (1999).

Além disso, o desenvolvimento de uma LPS não é apresentada de forma completa e detalhada em publicações. Os artigos, que são mais práticos, normalmente tratam superficialmente o desenvolvimento da LPS, não entrando em detalhes do projeto usando componentes e em sua implementação. Os livros mostram mais detalhes do processo seguido, mas ficam geralmente restritos a exemplos não aplicados realmente na prática e acabam sendo superficiais em alguns pontos, como no projeto de componentes e na sua implementação.

1.3 Objetivos

Um dos objetivos desta dissertação é definir um processo de desenvolvimento de LPS que use de forma integrada um projeto baseado em componentes para a linha e use geradores de código para gerar os produtos da linha, convergindo assim as duas vertentes discutidas na seção 1.2. O objetivo também é que o processo de desenvolvimento da LPS seja ágil, desenvolvendo-a em incrementos que produzam resultados rapidamente e que forneçam *feedback* em relação à eficiência, ou não, do que está sendo feito.

Outro objetivo desta dissertação é investigar questões relacionadas ao projeto de LPS com uma arquitetura baseada em componentes, ou seja, pesquisar diferentes alternativas de projetos de componentes para projetar e implementar variabilidades de LPS. Em especial, pretende-se investigar diferenças entre soluções de arquitetura de uma LPS usando componentes caixa-branca e caixa-preta. Adicionalmente, tem-se o objetivo de pesquisar o uso de aspectos em uma LPS com uma arquitetura baseada em componentes. Para tanto pretende-se analisar o uso de aspectos para representar requisitos e variabilidades funcionais e não-funcionais de uma LPS. Além disso, pretende-se investigar a geração de produtos de uma linha de produtos de software que tenha uma arquitetura baseada em componentes. Para isso, deve ser definida uma linguagem de modelagem de aplicações para o domínio em questão baseada nas características da linha. Essa linguagem é utilizada pelo gerador para automatizar o processo de engenharia de aplicações do domínio.

Como meio para analisar essas questões e validar as propostas a serem feitas, traçou-se o objetivo de desenvolver uma LPS completa e não trivial para validar os processos e as técnicas

definidas e para apoiar estudos posteriores. O desenvolvimento da LPS serve então como prova de conceito para os objetivos da dissertação e possui um detalhamento dificilmente encontrado em publicações. Com isto, pretende-se dominar o ciclo de desenvolvimento de LPSs e ter um software que possa apoiar outras pesquisas.

1.4 Organização

O capítulo 2 é dedicado ao reúso de software. A noção de reúso é definida e são apresentadas e discutidas diferentes abordagens, técnicas e processos de reúso. É apresentada também uma introdução aos conceitos de linhas de produto de software, componentes de software, desenvolvimento baseado em componentes e geradores de aplicação. Trabalhos relevantes publicados sobre esses assuntos são revisados e discutidos.

No capítulo 3 são apresentados assuntos considerados importantes para o desenvolvimento desta dissertação, como a programação orientada a aspectos, dando ênfase para as linguagens AspectJ e FuseJ; a arquitetura de software, destacando-se as linguagens de descrição de arquitetura; e propostas de desenvolvimento integrando componentes e aspectos.

No capítulo 4 é descrito um processo de desenvolvimento da LPS escolhida para gerenciamento de Bilhetes Eletrônicos de Transporte municipal (LPS-BET), apresentando-se em detalhes os ciclos de desenvolvimento incrementais da engenharia de domínio da linha e as atividades realizadas em cada ciclo. Além disso, é descrito sucintamente o processo seguido para a engenharia de aplicações, reusando ativos centrais da linha.

No capítulo 5 são discutidas questões relacionadas ao projeto da LPS desenvolvida com arquitetura baseada em componentes, comparando soluções e as abordagens caixa-preta e caixa-branca para a implementação de variabilidades. Além disso, são considerados os aspectos como alternativas para implementação de variabilidades em uma arquitetura baseada em componentes.

No capítulo 6 é apresentada a configuração do gerador de aplicações configurável para o domínio da LPS desenvolvida e o detalhamento da engenharia de aplicações ao usar esse gerador para obter as aplicações desejadas.

Finalmente, no capítulo 7 é concluída a dissertação de mestrado com as considerações finais, os trabalhos futuros relacionados à dissertação e alguns pontos considerados importantes no desenvolvimento da LPS usando uma arquitetura baseada em componentes e em aspectos e tendo aplicações produzidas por um gerador de aplicações.

Reúso de Software

2.1 Considerações Iniciais

A proposta de geração de famílias de produtos com arquitetura baseada em componentes tem como base conceitual o reúso de software, que pode ser obtido na engenharia de linhas de produtos de software. O desenvolvimento da linha tem como base uma arquitetura de componentes de software, sendo importante conhecer os modelos de componentes e os métodos para o desenvolvimento baseado em componentes. Após o desenvolvimento, a engenharia de produtos de uma família pode ser feita com o apoio de geradores de aplicação.

Portanto, neste capítulo são apresentadas algumas abordagens de reúso de software relevantes para a proposta de trabalho apresentada nesta dissertação. Na Seção 2.2 é apresentada uma visão geral de reúso de software. Na Seção 2.3 são apresentadas definições de Linhas de Produto de Software e dois métodos para o seu desenvolvimento – o PLUS e o FAST. Em seguida, na Seção 2.4 são introduzidos os componentes de software e os frameworks formados por componentes. Também são apresentadas as características dos modelos de componentes JavaBeans, Corba e .NET. Na Seção 2.5 é discutido brevemente como são desenvolvidos sistemas baseados em componentes, que são exemplificados por meio dos métodos Catalysis, Kobra e UML Components. Na Seção 2.6 o conceito de um gerador de aplicação é discutido e apresentado um gerador do tipo configurável. Por fim, na Seção 2.7 são apresentadas as considerações finais do capítulo.

2.2 Réuso de Software

Segundo Krueger (1992), réuso de software é o processo que tem o objetivo de criar sistemas de software a partir de software existente, ao invés de construí-los a partir do zero. Freeman (1987) apresenta uma definição muito parecida, mas complementa que o réuso consiste do uso de artefatos já existentes e exemplifica os tipos de artefatos que podem ser reusados: estruturas de projeto, partes de código-fonte, módulos de implementação, especificações, documentação e transformações. Biggerstaff e Perlis (1989) definem o réuso de software como a reutilização de qualquer tipo de conhecimento sobre um sistema em outros sistemas similares, com o objetivo de reduzir o esforço de desenvolvimento e a manutenção nesses novos sistemas. Dessa forma, evidenciam a importância do conhecimento ser reusado entre os sistemas.

Quando não há o réuso tem-se a possibilidade do compartilhamento de código por indivíduos ou por pequenos grupos, mas as pessoas trabalham de forma independente em projetos não-relacionados e não há uma comunicação relacionada ao código de cada desenvolvedor. Em um primeiro nível começa a ser feito um réuso informal de código (réuso *ad-hoc*). De modo não-sistemático, alguns desenvolvedores confiam no código um do outro e copiam partes de código em um sistema ou de um sistema para outro. Com isso pode haver uma redução de tempo de desenvolvimento no projeto, porém, como o réuso não é sistemático, podem ocorrer problemas, como manutenção do código copiado. Um nível superior de réuso consiste do réuso de instâncias de código sem realizar mudanças nelas. Assim, tem-se uma estratégia de réuso de código como uma caixa-preta. Há a definição de um processo bem simples de réuso, que consiste do teste, da documentação e do encapsulamento dessas instâncias de código para o posterior réuso (Griss, 2001a). As bibliotecas de código são um exemplo de uma técnica que realiza esse tipo de réuso.

As técnicas mais sofisticadas de réuso realizam um réuso sistemático, baseando-se em um processo repetível e preocupado principalmente com o réuso de artefatos de mais alto nível, como requisitos, projetos e subsistemas, e focam no domínio¹ (Frakes e Isoda, 1994). Assim, o réuso sistemático de software consiste de uma mudança da abordagem de construção de sistemas únicos para o desenvolvimento de famílias de sistemas ou de sistemas relacionados. Portanto, o réuso sistemático necessita da engenharia de domínio, que tem duas fases: a análise de domínio e a implementação de domínio.

A análise de domínio é o processo de descobrir e registrar as similaridades e as variabilidades dos sistemas de um domínio (Prieto-Diaz e Arango, 1991) e é muito importante para haver um réuso eficiente. A implementação de domínio consiste no uso das informações descobertas com a análise de domínio para a criação dos artefatos reusáveis. A modelagem e a implementação do domínio podem ser feitas de diversas maneiras, diferenciando as técnicas de réuso sistemático, tais como os frameworks de software orientados a objetos, o desenvolvimento de software baseado em componentes e os geradores de aplicação. Essas técnicas apresentam uma fase de análise de

¹área de uma aplicação ou, de maneira mais formal, um conjunto de sistemas que compartilham decisões de projeto (Frakes e Isoda, 1994)

domínio bastante semelhante, enquanto na fase de implementação de domínio elas apresentam características mais específicas.

Após a engenharia de domínio podem ser desenvolvidas famílias de sistemas usando os artefatos reusáveis obtidos na implementação do domínio e, de acordo com os requisitos do sistema, podem ser feitas algumas adaptações nos artefatos. Isso também implica em uma variação das características das técnicas de reúso para a engenharia de aplicações usando os artefatos reusáveis implementados.

Diversas técnicas estão disponíveis para o desenvolvedor obter múltiplas formas de reúso de artefatos de software e durante todas as fases do processo de construção de sistemas (Frakes e Kang, 2005). Mesmo com essa grande diversidade de técnicas de engenharia de software, existem similaridades entre as técnicas de reúso usadas, como abstração, seleção, especialização e integração de artefatos de software (Biggerstaff e Perlis, 1989).

Qualquer abordagem de reúso utiliza alguma forma de abstração para os artefatos de software, pois é uma característica essencial para qualquer técnica de reúso (Krueger, 1992). A maioria das abordagens de reúso auxilia os desenvolvedores a selecionar os artefatos de software reusáveis, após localizar esses artefatos e compará-los para poder selecionar os mais apropriados. Os artefatos similares são unidos em um único artefato genérico e, após selecionar o artefato generalizado para reúso, o desenvolvedor especializa-o por meio de parâmetros, transformações, restrições ou outra forma de refinamento. As tecnologias de reúso têm tipicamente um arcabouço de integração, usado para combinar um conjunto de artefatos selecionados e especializados.

Ao longo do ciclo de vida de um sistema, o reúso pode provêr diversas vantagens em relação ao desenvolvimento tradicional (Zand e Samadzadeh, 1994):

1. Diminuição de duração dos ciclos de desenvolvimento e diminuição dos custos de produção em esforços futuros de desenvolvimento.
2. Melhoria de confiabilidade do sistema pelo reúso de componentes que já foram provados como corretos (ao menos operacionalmente) e redução da necessidade de testes sistêmicos.
3. Redução do custo de manutenção do ciclo de vida.
4. Possibilidade da organização desenvolvedora se recuperar do investimento realizado em sistemas existentes ao produzir novos programas e conduzir novos esforços de projeto.

Favaro *et al.* (1998) classificam as vantagens em duas categorias: (i) benefícios operacionais, nos quais seriam incluídas as três primeiras vantagens citadas acima e (ii) benefícios estratégicos, nos quais a quarta vantagem se encaixa, assim como, os benefícios da adoção do reúso em termos da oportunidade de entrar em novos mercados, a flexibilidade de responder a forças competitivas e mudar condições de mercado.

A complexidade dos métodos e das técnicas de reúso aumenta ao passo que se move do nível de especificação para o nível de codificação. Entretanto, em compensação, o resultado da aplicação

das técnicas em termos de eficiência de tempo e espaço também aumenta nessa mesma direção (Zand e Samadzadeh, 1994). Quanto mais específico o domínio para o reúso, mais fácil fica para obter o resultado desejado.

2.3 Linhas de Produtos de Software

Uma coleção de sistemas que compartilham características comuns foi chamada por Parnas (1979) de “família de sistemas”. Atualmente, uma família de sistemas é mais conhecida como uma Linha de Produtos de Software (LPS) ou uma família de produtos de software. Existem variados exemplos de linhas de produtos, como as pirâmides no Egito, as linhas de aeronaves na indústria aeronáutica e os modelos de telefones celulares de determinados fabricantes. É vantajoso desenvolver uma linha de produtos quando existe mais a ser ganho ao se analisar os sistemas de forma coletiva, ao invés de analisá-los separadamente, ou seja, quando os sistemas apresentam mais características em comum do que características que os distinguem (Parnas, 1979). Diferentemente, quando um conjunto de produtos possui diversas similaridades e diversas diferenças trata-se de populações de produtos (van Ommering, 2002). As similaridades compartilhadas pelos produtos de software podem ser exploradas para alcançar economias na produção e assim os produtos podem ser construídos a partir de artefatos comuns (Clements e Northrop, 2001).

Segundo Atkinson *et al.* (2001), a engenharia de LPS visa criar artefatos genéricos de software que sejam reusáveis em uma família de produtos-alvo. Griss (2001b) evidencia a importância também da existência de variações entre os produtos e define LPS como um grupo de produtos que compartilham um conjunto comum de requisitos, porém também possuem variabilidades significativas nos requisitos. De forma semelhante, Weiss e Lai (1999) definem uma linha de produtos como um processo projetado para se obter vantagens das características comuns e das variabilidades previsíveis de uma família de produtos.

Com a intenção de deixar claro a forma como os sistemas de uma LPS são desenvolvidos, Clements e Northrop (2001) definem-na como um conjunto de sistemas de software que compartilham um conjunto de características comuns e gerenciadas, satisfazendo a necessidades específicas de um segmento de mercado ou de negócio e que são desenvolvidas de maneira pré-definida a partir de um conjunto comum de ativos centrais.

Quando diferentes aplicações são analisadas na mesma LPS ou domínio de problema, elas são frequentemente comparadas com base em suas características (*features*). Ao desenvolver LPS e componentes para reúso é importante entender as características que devem ser fornecidas por um conjunto de recursos reusáveis. Griss (2001b) define *feature* como uma característica que usuários e clientes vêem como sendo de importância na descrição e distinção de membros de uma linha de produtos. Uma característica pode ser um requisito específico ou um conjunto de requisitos específicos e uma seleção entre requisitos opcionais ou alternativos. Ela também pode

ser relacionada a propriedades específicas de determinado produto e relacionada a propriedades de implementação.

As técnicas usadas para análise de domínio usando características variam na forma como identificam o domínio e representam as características e seus relacionamentos. O *Software Engineering Institute* (SEI) utiliza o FODA (*Feature-Oriented Domain Analysis*) (Kang *et al.*, 1990) e o FORM (*Feature-Oriented Reuse Method*) (Kang *et al.*, 1998). O segundo é uma extensão do primeiro, especificamente para linhas de produto. Outra técnica é a FeatureRSEB (Griss *et al.*, 1998), que se baseou no FODA para integrar a modelagem de características na engenharia de software orientada a reúso. Além disso, diversos métodos de desenvolvimento e implementação de LPS utilizam diagramas de características, tais como o PLUS (*Product Line UML-Based Software Engineering*) (Gomaa, 2004) e o FAST (*Family-Oriented, Abstraction, Specification, and Translation*) (Weiss e Lai, 1999).

2.3.1 Processos de Engenharia de Linhas de Produtos de Software

Em geral, recomenda-se que uma organização com o objetivo de desenvolver uma LPS tenha desenvolvido ao menos três aplicações similares que pertençam ao mesmo domínio (Roberts *et al.*, 1997; Weiss e Lai, 1999). A evolução de uma LPS pode ser pró-ativa, reativa ou extrativa (Krueger, 2002). Na abordagem pró-ativa, a organização já visa inicialmente desenvolver a LPS para cobrir todo o escopo de produtos. Diferentemente, a abordagem reativa apenas desenvolve produtos da LPS sob demanda. Uma abordagem intermediária, chamada extrativa, ocorre quando partes de código de produtos existentes são generalizados para uma LPS de tal forma que possam ser reusadas para outros produtos.

No caso de uma evolução pró-ativa, a organização pode usar um processo baseado em engenharia reversa ou em engenharia avante que diferem basicamente na sua primeira fase. No processo baseado em engenharia avante uma nova LPS é desenvolvida e as funcionalidades comuns podem ser determinadas antes das funcionalidades variáveis. Na engenharia reversa existem sistemas que estão disponíveis para análise e modelagem e são candidatos para modernização e inclusão em uma LPS (Gomaa, 2004).

Vários autores organizam o processo de desenvolvimento de engenharia de LPS em duas fases principais (Gomaa, 2004; Weiss e Lai, 1999; Bayer *et al.*, 1999; Clements e Northrop, 2001). Essas fases têm a finalidade de separar as atividades genéricas de domínio das atividades de desenvolvimento de aplicações. Na primeira fase, o engenheiro analisa o domínio, verifica os aspectos similares e variáveis das aplicações, projeta e codifica os artefatos que apoiarão o processo de desenvolvimento de aplicações. Na segunda fase, o engenheiro elicita os requisitos da aplicação de um determinado domínio e desenvolve a aplicação usando os artefatos desenvolvidos durante a primeira fase. Apesar dessas fases semelhantes, cada processo fornece destaque para as atividades das fases de forma diferente e também pode possuir fases adicionais.

Método PLUS

PLUS (Gomaa, 2004) é um método de projeto de LPS baseado na notação da UML (Fowler, 2004). Ele pode ser integrado a outros métodos e processos de software tradicionais para dar suporte ao desenvolvimento de linhas de produto. O método PLUS é composto pelo processo ESPLEP (do inglês *Evolutionary Software Product Line Engineering Process*) (Gomaa, 2004) que apresenta uma perspectiva de desenvolvimento de LPS. Ele é iterativo, orientado a objetos e compatível com o PU (Processo Unificado) (Jacobson *et al.*, 1999) e o modelo de desenvolvimento em espiral (Boehm, 1986). O ESPLEP é composto por dois (sub)processos (ou ciclos de vida): a engenharia da LPS (chamada também de engenharia de domínio) e a engenharia da aplicação de software, ilustrados na Figura 2.1.

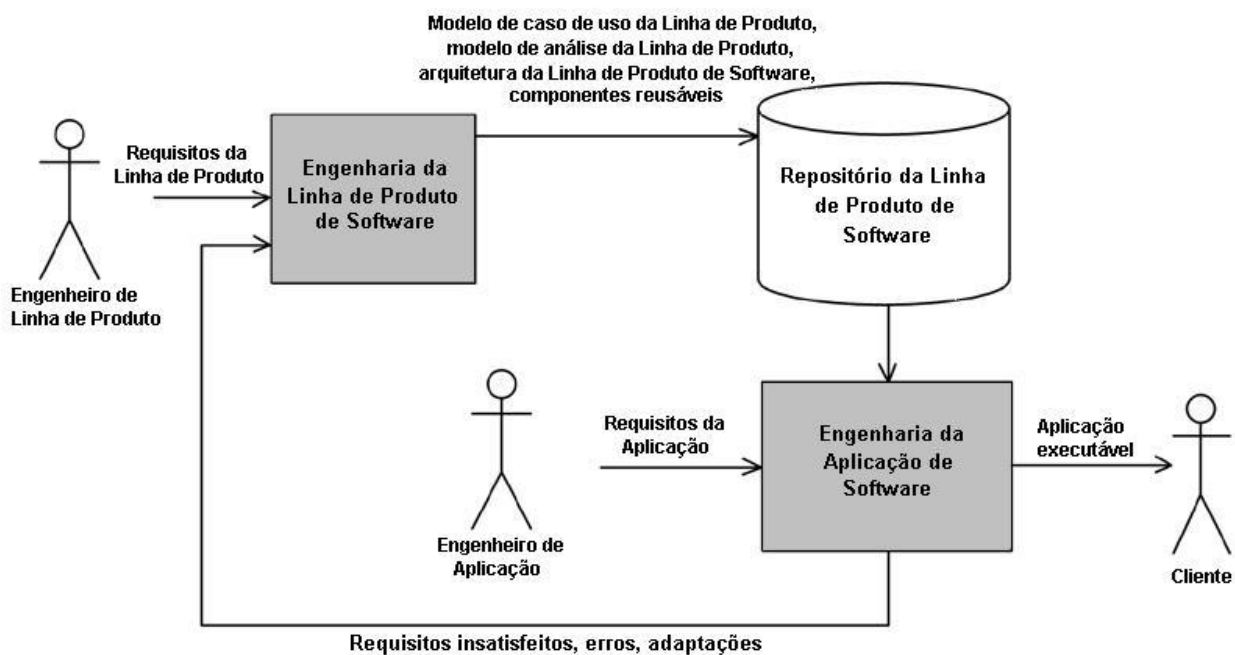


Figura 2.1: Processo de Desenvolvimento de Linhas de Produtos de Software no ESPLEP (Gomaa, 2004)

O ESPLEP é um processo de desenvolvimento de software baseado no conceito de casos de uso (UC do inglês *Use Case*). A engenharia da linha de produtos consiste do desenvolvimento de modelos de casos de uso, de análise e de arquitetura da linha de produtos, assim como de componentes reusáveis, que serão armazenados em um repositório da linha de produtos, visto também na Figura 2.1. Na modelagem de requisitos, os requisitos da LPS são definidos em termos de atores e UCs. Durante a modelagem de análise, cada caso de uso da linha de produtos passa a ser descrito por objetos que participam no caso de uso e por suas interações. Em seguida, a arquitetura da LPS baseada em componentes é desenvolvida. O próximo passo é a implementação incremental de componentes, que consiste do projeto detalhado dos componentes, sua codificação e de seus testes unitários. Finalmente são realizados os testes de integração e os testes sistêmicos da LPS. O

ciclo de vida da engenharia da linha de produtos é mostrado na Figura 2.2. Durante a engenharia de aplicação, uma aplicação individual, que seja membro da LPS, é desenvolvida. Dados os requisitos da aplicação, os modelos da linha de produtos são adaptados para derivar os modelos da aplicação específica. A partir da arquitetura da aplicação e dos componentes apropriados do repositório da LPS, obtém-se a aplicação executável. O repositório da LPS contém os ativos centrais, que são artefatos já validados da LPS.

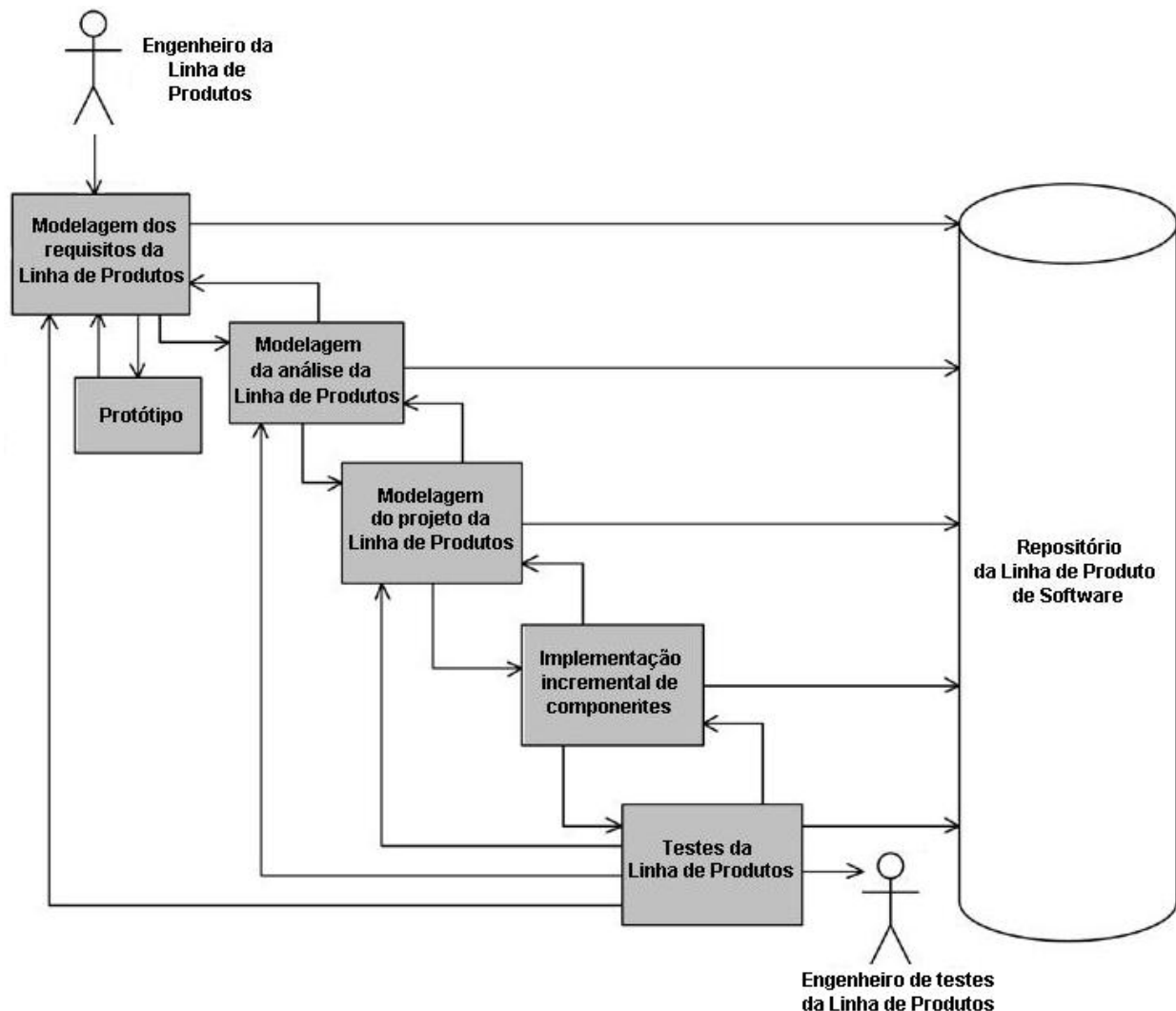


Figura 2.2: O Processo de Engenharia da Linha de Produtos no ESPLEP (Gomaa, 2004)

O ESPLEP é uma adaptação do PU feita com o objetivo de que cada atividade do PU (mostradas na Figura 2.2) corresponda a uma disciplina do PU. As atividades do ESPLEP possuem os mesmos nomes das disciplinas do PU. Todas as fases do PU – Concepção, Elaboração, Construção e Transição – são iterativas e envolvem diversas disciplinas. A seguir será discutido como o PLUS considera que deve ser feita a integração do seu processo com o PU.

A fase de Concepção consiste de um estudo de viabilidade da linha de produtos e da definição do seu escopo, do seu tamanho e do seu grau de similaridades e variabilidades. Essa fase produz

informações suficientes sobre a linha de produtos para que possa ser tomada a decisão de seguir, ou não, no desenvolvimento. UCs iniciais são identificados para cada membro em potencial da LPS e a partir desse ponto são identificados os UCs básicos. Desenvolve-se também um modelo de características inicial para determinar características comuns, opcionais e alternativas.

A Elaboração é dividida em duas iterações: uma para o núcleo da LPS e outra para a evolução da LPS. Na primeira iteração o modelo de UCs é revisado e desenvolvido em maiores detalhes. A análise dos UCs leva à análise das características. É necessário desenvolver em seguida modelos de análise e projeto do núcleo, como diagramas de comunicação e diagramas de estados. Além disso uma arquitetura do núcleo em alto nível é desenvolvida. Na segunda iteração da Elaboração os desenvolvedores tratam da evolução da linha de produtos, considerando UCs e características opcionais e alternativas e os pontos de variação são examinados minuciosamente. É feita a modelagem de análise dos UCs opcionais e alternativos. Na modelagem de projeto é expandida a arquitetura da LPS para incluir os componentes opcionais e variantes. A Tabela 2.1 mostra as subatividades e os artefatos produzidos para as disciplinas (ou atividades) descritas em detalhes por Gomaa (2004) (requisitos, análise e projeto). As outras duas atividades não são tratadas em detalhes pelo autor, por não diferirem muito do desenvolvimento tradicional.

Na primeira iteração de Construção, o projeto detalhado, a codificação e os testes unitários dos componentes do núcleo são executados. Também são feitos os testes de integração desses componentes. Na primeira iteração de Transição são realizados os testes sistêmicos do núcleo e produzido o núcleo. Outras iterações podem ser feitas para desenvolver os componentes opcionais e variantes. O gerente do projeto deve decidir se componentes adicionais devem ser implementados ou se devem fazer parte do processo de engenharia da aplicação.

O PU também pode ser aplicado para a engenharia de aplicação. Para cada aplicação da LPS, o processo do PU pode ser aplicado por inteiro, desde a concepção até a transição.

Tabela 2.1: Subatividades e artefatos produzidos para as atividades iniciais do processo ESPLEP

Atividade	Subatividades	Artefatos
Requisitos	Definição do escopo da LPS Modelagem de casos de uso Modelagem de características	Diagrama de UCs Especificação de UCs Diagrama de características Mapeamento UCs/características
Análise	Modelagem estática Estruturação de objetos Modelagem dinâmica Modelagem de Máq. de Estados Finitos Análise de dependência característica/-classe	Modelo estático conceitual Diagrama de classes Diagrama de comunicação Diagrama de sequência Tabela de dependência característica/-classe Diagrama de estados
Projeto	Projeto de arquitetura de LPS	Arquitetura baseada em componentes Especificação das interfaces dos componentes

Método FAST

Weiss e Lai (1999) desenvolveram um método de engenharia de LPS denominado FAST (do inglês *Family-Oriented Abstraction, Specification and Translation*) dividido em três processos. Um deles corresponde à qualificação do domínio, apresentando o objetivo de determinar a viabilidade econômica, ou não, da adoção da linha de produtos no domínio de aplicação escolhido. Se for vantajoso, são executados os processos de engenharia de domínio e de engenharia de aplicação. A engenharia de domínio consiste de um processo iterativo e contínuo de análise e de implementação do domínio, enquanto a engenharia de aplicação consiste da geração de membros da família a partir dos requisitos de determinado cliente. Os autores argumentam que a adoção do FAST pode reduzir o tempo e o custo de desenvolvimento de um membro de uma família de 60% a 80% em comparação com o seu desenvolvimento tradicional.

O objetivo da engenharia de domínio no método FAST é tornar possível a geração dos membros de uma família na engenharia de aplicação e para isso os engenheiros de domínio devem: (i) definir uma família; (ii) desenvolver uma linguagem para especificar os membros (linguagem de modelagem da aplicação); (iii) desenvolver um ambiente para, posteriormente, gerar membros da família a partir de suas especificações e (iv) definir um processo para produzir os membros usando esse ambiente.

A engenharia de aplicação no FAST tem como propósito explorar rapidamente os requisitos de uma aplicação e gerar a aplicação. Durante a análise dos requisitos do cliente, o engenheiro pode achar necessário criar modelos do membro da família, para que possa entender melhor e validar os requisitos antes de decidir pela geração do código e da documentação para a família. O processo pode ser visto na Figura 2.3, as elipses correspondem a artefatos usados ou produzidos na engenharia de aplicação e os retângulos correspondem às atividades executadas nela. Percebe-se assim que o método usa especificamente geradores de aplicação durante esse processo, que estão descritos na Seção 2.6.

Weiss e Lai (1999) apresentam também o modelo PASTA (do inglês *Process and Artifact State Transition Abstraction*) do processo FAST em que é feita a descrição dos artefatos produzidos e usados pelos engenheiros de domínio e de aplicação, das seqüências de atividades que eles usam para produzir esses artefatos e dos papéis das pessoas que produzem e usam os artefatos.

Outras Abordagens

Existem ainda outras abordagens de processos de desenvolvimento de LPS, como o PuLSE (do inglês *Product Line Software Engineering*) (Bayer *et al.*, 1999) e uma abordagem do SEI (Clements e Northrop, 2001).

O PuLSE é centrado em três principais elementos: as fases de desenvolvimento, que são fases lógicas do ciclo de vida de uma LPS; os componentes técnicos, que provêem o conhecimento técnico para operacionalizar o desenvolvimento da LPS; e os componentes de apoio, que fornecem

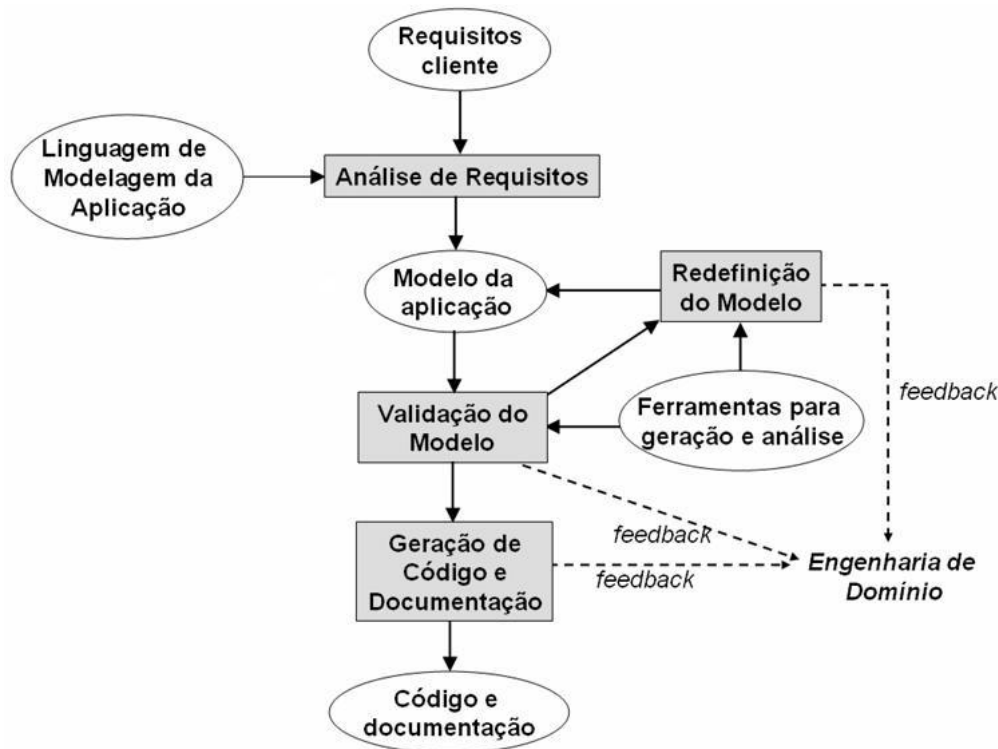


Figura 2.3: Processo da Engenharia de Aplicação no método FAST (Weiss e Lai, 1999)

diretrizes para apoiar os outros componentes (Bayer *et al.*, 1999). O processo de engenharia de aplicação do PuLSE é chamado de PuLSE-I (Bayer *et al.*, 2000). Esse processo é dependente das saídas das atividades do PuLSE, usando a infra-estrutura da linha para criar e manter um membro da LPS.

O modelo proposto pelo SEI envolve três processos: o desenvolvimento de ativos centrais, que corresponde às atividades de desenvolvimento da LPS; o desenvolvimento de produtos, que representa a atividade de engenharia de produtos usando os ativos centrais; e a gestão, que é formada pelas atividades referentes à gestão técnica e organizacional para dar apoio à linha. Eles consideram que a ordem entre o desenvolvimento de ativos centrais e de produtos não importa, pois os produtos podem ser desenvolvidos a partir de ativos centrais (engenharia avante), assim como, os ativos centrais podem ser extraídos de produtos existentes (engenharia reversa). Para realizar esses processos, foram definidas áreas de práticas relevantes para cada processo. Essas áreas foram divididas nas áreas de engenharia de software, de gestão técnica e de gestão organizacional. Além disso, Clements e Northrop (2001) definem padrões para auxiliar na aplicação das áreas de práticas de acordo com o contexto da organização.

2.4 Componentes e Frameworks de Componentes

Uma das mais importantes mudanças no processo de desenvolvimento para obter melhorias significativas na produtividade de software é deixar de fazer aplicações a partir do zero cada vez que um novo projeto for iniciado. Ao invés disso, deve-se construir o software reusando componentes já existentes na própria organização ou adquirindo componentes de outras organizações, pois muitos sistemas podem usar componentes similares ou até mesmo idênticos. Os blocos de construção a serem usados no desenvolvimento de software não devem ser limitados apenas àqueles oferecidos pela linguagem de programação, mas também incluir unidades encapsuladas de maior granularidade (D'Souza e Wills, 1998).

Há diversas definições de componentes de software na literatura, não havendo uma definição comum e precisa para o termo. Vários pesquisadores e autores apresentam definições de componente e o caracterizam da maneira mais adequada para o seu contexto e objetivos. D'Souza e Wills (1998) definem um componente de software como sendo um pacote coerente de artefatos de software que pode ser desenvolvido independentemente e distribuído como uma unidade, e que pode ser composto com outros componentes, sem alterações, para construir algo maior. Segundo Szyperski (2002), um componente de software é uma unidade de composição claramente identificável, com interfaces contratualmente especificadas e com dependências de contexto explícitas; é uma unidade de instalação independente que está sujeita a composição com outros componentes e não possui estado observável externamente. Griss (2001a), por outro lado, usa o termo componente para qualquer elemento reusável de um modelo de desenvolvimento que é fracamente acoplado a outros elementos e é projetado para o reúso.

Griss (2001a) e Atkinson *et al.* (2000) ainda relacionam componentes com linha de produtos, pois, segundo estes, os componentes fornecem a base perfeita para a aplicação prática do desenvolvimento de linhas de produto. Adicionalmente, a engenharia de software baseada em componentes pode ganhar significativamente com as idéias de linhas de produto. Espera-se que os sistemas baseados em componentes, dentro de um determinado domínio ou criados por uma certa organização, irão compartilhar diversas similaridades e, em particular, usarão vários componentes iguais. Já as variabilidades entre sistemas em uma família provavelmente estão restritas a um número relativamente pequeno de componentes.

2.4.1 Modelos de Componentes de Software

Algumas tecnologias de componentes têm surgido como apoio ao desenvolvimento de software baseado em componentes, resultando em modelos de componentes que oferecem mecanismos com os quais os engenheiros de software podem desenvolver aplicações pela composição de componentes, por meio da definição de formas de padrões e interfaces padronizadas entre componentes (Crnkovic e Larsson, 2001). Essas tecnologias surgiram por ser difícil alcançar objetivos primários como a instalação independente e a montagem dos componentes se os componentes de um sistema

forem desenvolvidos isoladamente uns dos outros, em virtude de conflitos de interfaces, formas de comunicação, etc. Para evitar esses problemas, as tecnologias aderem a certos padrões, como o modo de construção das interfaces requeridas ou fornecidas pelos componentes.

Diferentes tecnologias industriais apresentam modelos de componentes. Como exemplos desses modelos tem-se o JavaBeans da Sun, o COM+ e o .NET da Microsoft, o CCM (*Corba Component Model*) da OMG (*Object Management Group*) e o OSGi (*Open Service Gateway Initiative*) da OSGi Alliance. Os modelos evidenciam problemas práticos e são descritos em termos técnicos. Segundo Estublier e Favre (2002), existem vários detalhes de implementação nesses modelos, mas os usuários têm dificuldades para entender os seus conceitos e princípios. Além disso, os modelos focam nas últimas fases do ciclo de vida do desenvolvimento de software baseado em componentes, ou seja, na implementação, montagem e execução.

Modelo de Componente JavaBeans

O modelo de componente JavaBeans foi proposto pela Sun em 1997 como a primeira integração da noção de componentes na linguagem Java. Um *bean* corresponde a um componente. A principal qualidade desse modelo é a sua simplicidade, porém, o seu escopo é limitado, não sendo usável para o desenvolvimento baseado em componentes de grande escala (Estublier e Favre, 2002). O modelo JavaBeans interage em dois contextos diferentes: em tempo de composição, dentro da ferramenta de construção, e em tempo de execução, no ambiente de execução. Posteriormente a Sun lançou um segundo modelo de componentes distinto do JavaBeans, chamado de *Enterprise JavaBeans* (EJB). Esses dois modelos são bem distintos e não devem ser confundidos. O EJB é um modelo de componente servidor que simplifica o processo de mover a lógica de negócio para o servidor por meio da implementação de um conjunto de serviços automáticos para gerenciar os componentes (D'Souza e Wills, 1998).

O modelo de componente JavaBeans define quatro tipos de portas possíveis: propriedades, métodos, fontes e receptores de eventos. As propriedades são atributos de objeto que podem ser lidos e escritos por métodos de acesso. Os métodos são serviços com resultados especificados que o cliente pode requerer. Tanto as propriedades quanto os métodos representam serviços fornecidos pelo componente. Os eventos representam notificações de um componente, no caso das fontes de eventos são geradas notificações de determinado tipo, enquanto nos receptores de eventos são recebidas notificações.

A maioria dos componentes (*beans*) é implementada usando um objeto Java simples, e este é encapsulado no componente. Em relação à montagem dos componentes, o JavaBeans não fornece uma solução específica para a montagem, apoiando diferentes maneiras de conectar componentes (Estublier e Favre, 2002). Entretanto, ele define um modelo para o empacotamento de componentes em arquivos. Além disso, para tratar as questões de empacotamento, o JavaBeans inclui uma definição de relacionamentos de dependências entre itens de pacotes.

Modelo de Componente Corba

O CCM foi projetado com base na experiência acumulada usando o serviço Corba (do inglês *Common Object Request Broker Architecture*), o JavaBeans e o EJB. O foco do CCM é um modelo de componente para construir e implantar aplicações Corba. Uma vantagem desse modelo é seu esforço para integrar várias características envolvendo a engenharia de software, porém, como consequência tem-se uma especificação complexa que pode levar a diferentes formalismos para descrever uma aplicação de software. Mesmo sendo extensa, a especificação do CCM não descreve as fases de empacotamento e provisionamento (Estublier e Favre, 2002).

A interface do componente Corba é formada de portas dos seguintes tipos:

- *facetas (facets)*: interfaces fornecidas pelo componente para interação com o cliente;
- *receptáculos*: pontos de conexão que descrevem a habilidade de usar uma referência fornecida por algum agente externo;
- *fontes de eventos*: pontos de conexão que emitem eventos de determinado tipo;
- *receptores de eventos (event sinks)*: pontos de conexão que podem receber eventos;
- *atributos*: valores nomeados expostos por operações.

O CCM define o conceito de conexão como uma referência de objeto. Os componentes são conectados pela ligação entre facetas e receptáculos e entre fontes e receptores de eventos. A implementação de um componente é um conjunto de segmentos de códigos executáveis escritos em qualquer linguagem, que implementem ao menos uma porta. Os componentes Corba usam um container para implementar o acesso de componentes aos serviços do sistema.

Modelo de Componente .NET

A Microsoft apresenta diversas tecnologias de implementação de componentes, como o COM, que é específico para a linguagem C/C++ e conta com convenções de interoperabilidade binária e com interfaces; o DCOM, que estende o COM usando distribuição, e o MTS, que estende o DCOM com serviços de persistência e de transação (Estublier e Favre, 2002). Juntos eles constituem o COM+.

Além desses, a Microsoft elaborou o .NET, que é seu modelo de componente mais recente. Diferentemente dos outros modelos da Microsoft, ele não é baseado no COM, pois a interoperabilidade binária existente no COM é muito limitada. O .NET usa interoperabilidade de linguagens e introspecção. Para isso, o modelo define uma linguagem interna, chamada MSIL (do inglês *MicroSoft Intermediate Language*) que é parecida com o *Java Byte Code*.

A programação de componentes no .NET é representada pela abordagem de linguagem de programação. Isso significa que o programa contém as informações associadas com os relacionamentos com outros “componentes” e o compilador é responsável por gerar a informação necessária

em tempo de execução (Meyer, 2001). O modelo conta com um conector dinâmico específico que realiza as conexões entre recursos fornecidos e requeridos. Além disso, o componente no .NET consiste de módulos, que são arquivos executáveis tradicionais ou DLLs (*Dynamic Link Libraries*). A lista de módulos que compõem uma montagem é fornecida na linha de comando ao compilar o módulo principal (Estublier e Favre, 2002).

2.4.2 Frameworks de Componentes de Software

Segundo Van Ommering e Bosch (2002), um framework de componentes é uma aplicação ou parte de uma aplicação em que componentes podem ser conectados para especializar um comportamento. Dessa forma, a idéia é, ao invés de reusar componentes individualmente, construir um framework de componentes, pois a maioria dos projetos baseados em componentes de sucesso são aqueles que utilizam desenvolvimento de frameworks de componentes. Pree (1997) define framework como uma coleção de vários componentes individuais com cooperações pré-definidas entre eles, com o propósito de realizar uma determinada tarefa. Alguns desses componentes individuais são projetados para serem substituíveis, correspondendo tipicamente a classes abstratas na hierarquia de classes do framework. Esses pontos de refinamento pré-definidos são chamados de *hot spots* (Pree, 1995).

Aplicações construídas a partir de frameworks não reusam apenas código fonte, mas também o projeto, que é considerada a característica mais importante dos frameworks (Pree, 1997). O framework de componentes é bastante independente da forma como os componentes são implementados. Ele apenas requer que os componentes possam ser substituídos por componentes mais específicos que sejam compatíveis com as conexões originais.

Os frameworks de componentes se assemelham a frameworks orientados a objetos², porém, a diferença está nas dependências entre código especializado e código genérico. Enquanto os frameworks orientados a objetos usam conceitos de implementação orientada a objetos, como herança, os frameworks de componentes especificam as interfaces entre componentes conectáveis e o framework subjacente, reduzindo bastante as dependências.

Além disso, frameworks podem ser usados como componentes. Isso ocorre se um framework apenas cobrir uma parte de um domínio de aplicação (subdomínio) e se frameworks puderem ser arbitrariamente combinados, tornando-os reais componentes e sendo possível criar uma variedade de produtos pela seleção e combinação desses componentes (Van Ommering e Bosch, 2002).

²Frameworks orientados a objetos são coleções de classes organizadas em uma arquitetura abstrata com o objetivo de implementar uma família de problemas relacionados, usando herança (Johnson, 1997) (Van Ommering e Bosch, 2002).

2.5 Desenvolvimento Baseado em Componentes

O Desenvolvimento de Software Baseado em Componentes (DSBC) surgiu como uma perspectiva de desenvolvimento de software caracterizada pela composição de partes já existentes. Ele adere ao princípio de dividir para conquistar, gerenciando a complexidade, isto é, dividindo grandes problemas em partes menores e resolvendo-os. Dessa forma, são construídas soluções mais elaboradas a partir de partes mais simples (Cheesman e Daniels, 2001). Conforme Szyperski (2002), construir novas soluções pela combinação de componentes desenvolvidos e/ou comprados aumenta a qualidade e dá suporte ao rápido desenvolvimento, levando à diminuição do tempo de entrega do produto ao mercado.

D'Souza e Wills (1998) definem DSBC como uma abordagem de desenvolvimento de software em que todos os artefatos (desde códigos executáveis até especificações de interfaces, arquiteturas e modelos de negócio) podem ser construídos pela combinação, adaptação e união de todos os componentes em uma variedade de configurações. Segundo Cheesman e Daniels (2001), o DSBC é diferente das abordagens anteriores pela separação entre a especificação dos componentes e a implementação, assim como pela divisão da especificação dos componentes em interfaces.

Pressman (2002) propõe uma divisão do DSBC em duas fases. Uma fase corresponde à Engenharia de Domínio para a construção de componentes reusáveis e a outra fase corresponde ao DSBC em si, ou seja, é feita uma análise e projeto arquitetural e, em seguida, componentes são qualificados, adaptados e compostos de acordo com a necessidade da aplicação. Villela (2000), de modo semelhante, une a engenharia de domínio ao DSBC, definindo o Odyssey-DE em dois processos: o processo de engenharia de domínio do Odyssey e o processo de engenharia de aplicações do Odyssey. De maneira geral, o objetivo desses autores é separar as atividades do DSBC tal que seja permitido que elas sejam realizadas por organizações totalmente diferentes.

Vários autores têm apresentado métodos para o desenvolvimento de software baseado em componentes, dentre os quais destacam-se o método Catalysis (D'Souza e Wills, 1998), o método Kobra (Atkinson *et al.*, 2001) e o método *UML Components* (Cheesman e Daniels, 2001). Os três usam notação baseada na UML (*Unified Modeling Language*) com pequenas adaptações e oferecem abordagens para a modelagem de componentes. Nas subseções seguintes é apresentada uma visão geral de cada um desses métodos.

2.5.1 Catalysis

O método Catalysis (D'Souza e Wills, 1998) é abrangente, relativamente complexo e dá suporte à modelagem de diferentes domínios de aplicação. O método enfatiza a identificação e generalização de partes ou módulos de um software que está sendo desenvolvido para que se torne um componente e possa ser reusado em outros desenvolvimentos. Os componentes podem ser modelados como um conjunto de classes dentro de um pacote, com uma interface pública comum –

tanto a interface oferecida como a requerida. O método é baseado na composição de componentes existentes, abrangendo requisitos, código, padrões de projeto e arquiteturas.

De modo sistemático, Catalysis fornece um processo com o propósito de construir modelos precisos desde os requisitos, manter esses modelos, refatorá-los, extrair padrões e realizar engenharia reversa a partir de descrições detalhadas para modelos abstratos. D'Souza e Wills (1998) definem o processo subjacente ao Catalysis como um conjunto de padrões de processos. Isso é justificado pelo argumento de que não há um único processo que atenda a todas as necessidades, pois os pontos de partida, os objetivos e/ou as restrições podem ser diferentes.

O método Catalysis considera que o DSBC deve ser iterativo, com entregas incrementais, porém deve haver separação dos interesses diferentes quando apropriado, como requisitos de usuários, arquitetura e código. Ele se baseia em uma definição abstrata de conceitos e, no decorrer do desenvolvimento do sistema, mecanismos de refinamentos são aplicados. Dessa maneira, o desenvolvimento é visto como uma série de refinamentos dos conceitos, princípios e abstrações para níveis mais baixos de implementação, até chegar ao código. Além disso, o método sugere que as atividades de especificação, projeto, implementação e teste sejam realizadas recursivamente para modelagem de negócio, especificação de componentes e implementação de componentes, e permite que vários caminhos sejam utilizados, cada um definindo uma sequência de tarefas e um conjunto de artefatos que devem ser gerados.

Segundo Bunse e Atkinson (2001), embora a estratégia para chegar ao código a partir dos modelos seja boa, o método não soluciona problemas de mapeamento dos elementos da análise para o projeto e do projeto para a codificação. Outro problema levantado por eles é que o Catalysis não define em qual nível de refinamento se deve descrever todas as principais decisões, nem o impacto que os requisitos funcionais podem ter. De acordo com eles, isso resulta em modelos abstratos que ainda não são implementáveis ou estão muito longe dos detalhes necessários para a codificação.

2.5.2 KobrA

O método KobrA (Atkinson *et al.*, 2001) foi desenvolvido no Instituto Fraunhofer para representar uma síntese de diversas tecnologias de engenharia de software, incluindo desenvolvimento de LPS, DSBC, frameworks, modelagem de qualidade e de processo e inspeções centradas na arquitetura. Essas tecnologias foram integradas no KobrA com o objetivo de fornecer uma abordagem sistemática para o desenvolvimento de aplicações baseadas em componentes. Portanto, o método dá apoio ao desenvolvimento dirigido por modelos (*model-driven development*) (Atkinson e Muthig, 2002); a uma abordagem de LPS (Atkinson *et al.*, 2000); e à modelagem de componentes com UML. Segundo Atkinson *et al.* (2001), isso permite que os benefícios do desenvolvimento baseado em componentes possam ser obtidos em todo o ciclo de vida do software e possa aumentar significativamente a reusabilidade dos componentes.

KobrA propõe um apoio concreto para o desenvolvimento de frameworks de domínio específico baseado em componentes. Para desenvolver uma aplicação, o framework genérico, que é o artefato principal produzido pelo método, é instanciado pela tomada de decisões sobre quais funcionalidades farão parte da aplicação a ser gerada. Kobra apóia os diversos conceitos de LPS para o desenvolvimento e evolução de aplicações, pois apresenta como base a abordagem PuLSE, tendo sido feito um mapeamento das atividades do PuLSE no Kobra por Atkinson, Bayer e Muthig (2000).

Cada componente KobrA (*Komponent*) do framework é descrito por um diagrama UML adequado, como se fosse um sistema independente, conhecido como princípio de localidade. A modelagem é usada para descrever a estrutura e o comportamento essencial de cada componente, independentemente da tecnologia de implementação, pela especificação e realização dos componentes. A especificação do componente descreve as suas propriedades externas e visíveis e especifica seus requisitos, consistindo de três diagramas (funcional, estrutural e comportamental) que descrevem as diferentes variações do componente. A realização define como o componente satisfaz os requisitos pela descrição da arquitetura e do projeto do componente, que são modelados pelos diagramas de estrutura, atividade e interação.

2.5.3 UML Components

Cheesman e Daniels (2001) definiram o método de desenvolvimento baseado em componentes denominado *UML Components*. O seu processo se baseia no RUP (do inglês *Rational Unified Process*), seguindo um ciclo de vida iterativo e incremental e usando o conceito de fluxos de trabalho (*workflow*). Cada fluxo de trabalho define uma seqüência de atividades que produz um resultado de valor observável (Kruchten, 2000). Além disso, o método se baseia na notação UML (Fowler, 2004), utilizando-a como linguagem de especificação desde as fases iniciais do desenvolvimento, com casos de uso, até as fases finais de projeto, com diagramas de classes e de seqüência.

O método tem o objetivo de produzir uma arquitetura de componentes e as especificações dos componentes, de acordo com os requisitos de negócio do sistema a ser implementado com componentes, conforme é mostrado na Figura 2.4. No processo são mostrados os fluxos de trabalho (*workflows*) e os artefatos produzidos e consumidos pelos fluxos de trabalho, representados respectivamente por caixas e setas. *UML Components* é composto pelos fluxos de trabalho de requisitos, especificação, provisão, montagem, teste e implantação. Os fluxos de requisitos, teste e implantação correspondem aos especificados no RUP, enquanto os fluxos de especificação, provisão e montagem correspondem a modificações nos fluxos de trabalho de análise, projeto e implementação do RUP.

O processo de desenvolvimento tem como entrada os requisitos de negócio de um sistema. Esses requisitos produzem o modelo conceitual de negócios e os modelos de casos de uso no fluxo de Requisitos. A Especificação recebe os artefatos gerados nos Requisitos, os recursos existentes no sistema e as restrições técnicas (como modelo arquitetural e/ou ferramentas já definidas), e gera

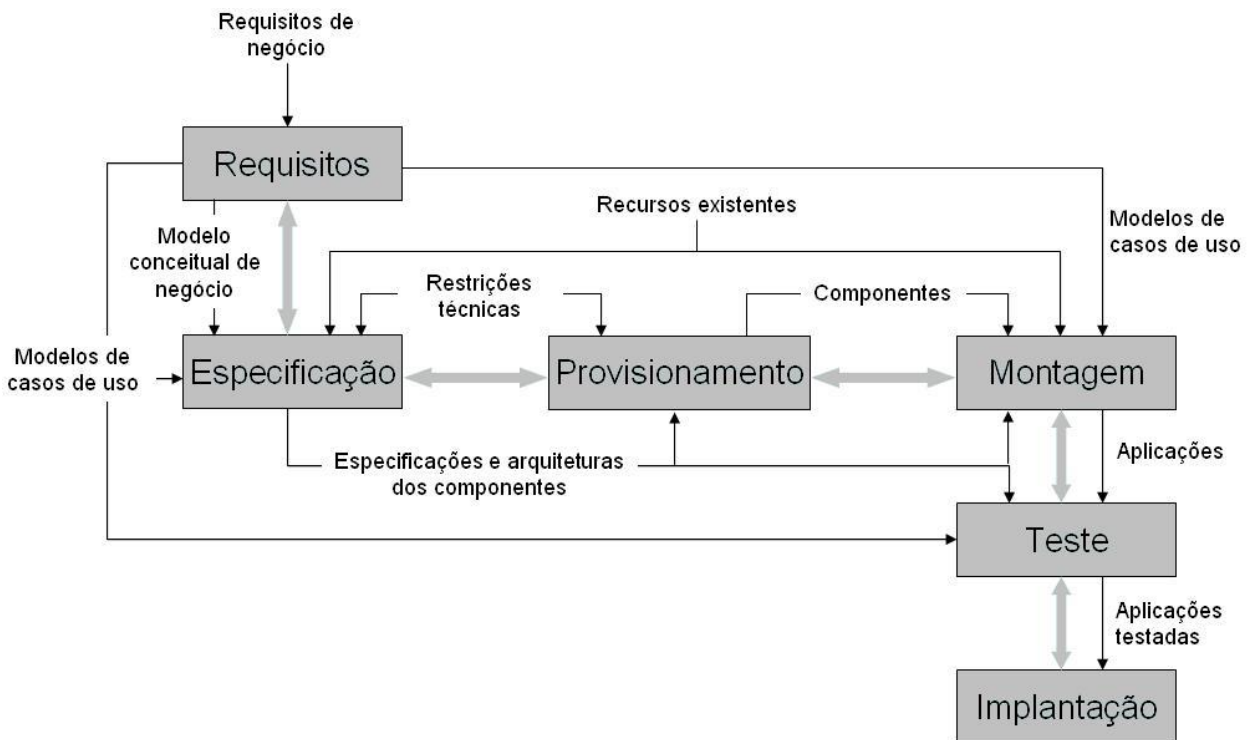


Figura 2.4: Processo de Desenvolvimento do método *UML Components* (Cheesman e Daniels, 2001)

um conjunto de especificações de componentes e uma arquitetura de componentes para o sistema. Uma especificação de componente define as especificações das interfaces requeridas e fornecidas do componente e a arquitetura de componentes define como os componentes interagem entre si. Para obter esse resultado, a Especificação é dividida em três etapas específicas: identificação, interação e especificação de componentes.

Com a definição das especificações dos componentes e da arquitetura dos componentes, o Provisionamento determina quais componentes devem ser implementados (ou obtidos de terceiros) e quais devem ser reutilizados. Em seguida, a Montagem é responsável pela combinação dos componentes, usando os recursos existentes, os modelos de casos de uso e os componentes necessários para desenvolver a aplicação desejada. Após a Montagem, a aplicação é testada e em seguida disponibilizada para operação pelos usuários na fase de Implantação.

2.6 Geradores de Aplicação

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. O programa é então compilado para criar uma aplicação executável (Cleaveland, 1988).

Eles auxiliam uma organização a construir múltiplos produtos de uma família com mais facilidade do que pela maneira de implementação tradicional. Dessa forma, os geradores de aplicação facilitam o reúso e a adaptação do projeto do sistema. Adicionalmente, a sua automação pode aumentar significativamente a produtividade durante o desenvolvimento e a manutenção (Cleaveland, 1988). Além de código, os geradores podem produzir outros artefatos, tais como documentação do usuário e do software, casos de teste, diagramas e figuras.

De forma geral, um gerador de aplicação executa os seguintes passos (Czarnecki e Eisenecker, 1999):

- verifica se o sistema especificado pode ser construído, validando a especificação de entrada e relatando possíveis erros ou avisos de inconsistências;
- completa a especificação, adicionando as configurações-padrão existentes;
- gera a implementação, unindo componentes de implementação.

Em um desenvolvimento típico, o analista e o projetista de sistema constroem aplicações específicas. No desenvolvimento de geradores, o analista e o projetista de domínio constroem os geradores de aplicação que são usados pelos projetistas de sistema. O analista de domínio especifica os requisitos de um gerador de aplicação para uma série de problemas e o projetista de domínio usa essas especificações e implementa-as em um gerador. Conforme Czarnecki e Eisenecker (1999), o gerador de aplicação funciona se (1) projetar os componentes de implementação para servirem para uma arquitetura de linha de produtos em comum; (2) modelar o conhecimento de configuração, tal que descreva como traduzir os requisitos abstratos em conjuntos específicos de componentes; e (3) implementar o conhecimento de configuração usando geradores. Wu *et al.* (2002) sugerem um método para construir uma aplicação em três estágios: modelagem de requisitos, construção da especificação e construção da aplicação. Cleaveland (1988) fornece uma abordagem mais detalhada para construir geradores em sete passos:

1. Reconhecimento de domínio: reconhecer onde um gerador deve ser usado é o passo mais difícil, uma forma é pelo reconhecimento de padrões que ocorram no código ou em artefatos de mais alto nível de abstração.
2. Definição de fronteiras de domínios: determinar o alcance do gerador, sabendo quais características devem ser incluídas ou excluídas.
3. Definição de modelo: determinar um modelo matemático para que o gerador seja mais compreensível, consistente e completo, pois cada característica poderá ser explicada em termos do modelo.
4. Definição de partes fixas e variáveis: definir as partes que estarão sob o controle do projetista do sistema (variáveis) e as partes que não poderão ser alteradas (fixas). As partes variáveis

correspondem geralmente à especificação do sistema, enquanto as partes fixas são assertivas fixas do domínio ou da implementação.

5. Definição da entrada da especificação: as entradas das especificações podem ser feitas de forma interativa, em que o usuário seleciona as características desejadas por meio de escolhas em uma seqüência de formulários ou menus, ou podem ser especificadas de forma gráfica ou textual.
6. Definição dos produtos: definir o tipo de produto do gerador, podendo ser de diversos tipos, como um programa, documentação ou dados de teste.
7. Implementação do gerador: consiste na escrita do programa que traduz a linguagem da especificação no produto desejado, podendo ser usada uma ferramenta de desenvolvimento.

Weiss e Lai (1999) descrevem ainda duas formas possíveis para construir um gerador de aplicação: construir um compilador ou um compositor. A construção de um **compilador** consiste na criação de analisadores léxico, sintático e semântico para uma linguagem. No caso de um gerador de aplicação, consistiria da construção de um compilador para a Linguagem de Modelagem da Aplicação (LMA), cujo papel é o de modelar o comportamento das aplicações de uma família a serem geradas a partir da linguagem. Diferentemente, a construção de um **compositor** consiste na criação de um projeto de software, na derivação de um conjunto de gabaritos (do inglês *templates*) a partir do projeto, na criação de um mapeamento entre a especificação e esses gabaritos, para em seguida uma ferramenta utilizar ambos para gerar artefatos. Em outras palavras, a geração dos produtos da família será feita pela composição de gabaritos – implementações de módulos do projeto da família. Dessa forma, de acordo com o domínio da aplicação, pode-se escolher por compilar ou combinar os modelos da aplicação para gerar o código e a documentação de uma aplicação.

Os autores ilustram o uso de um compositor para uma família de sistemas para gerenciar bóias náuticas FWS (do inglês *Floating Weather Station*), como pode ser visto na Figura 2.8. O exemplo do desenvolvimento dessa família de bóias náuticas usando um gerador é explicado em detalhes a seguir.

As bóias náuticas FWS são utilizadas para realizar a medição da velocidade do vento no mar. Cada bóia é equipada com um ou mais sensores de vento, um transmissor de rádio e um computador de bordo. Vários sensores de diferentes resoluções são espalhados pela superfície da bóia para captar a velocidade do vento. O computador de bordo mantém o histórico das leituras da velocidade obtida por cada sensor e, em intervalos regulares, calcula a média ponderada das medições realizadas pelos sensores e emite essa informação por meio do transmissor de rádio para uma torre de controle. A torre monitora então as velocidades do vento captadas por diversas bóias espalhadas no mar.

No domínio das bóias náuticas existem algumas características similares, como a transmissão de mensagens, a média ponderada, os sensores, o transmissor e o *driver*. Os aspectos variáveis são apresentados na Tabela 2.2.

Tabela 2.2: Variabilidades das Bóias Náuticas

Variabilidade	Descrição	Identificador
Resolução	A resolução de cada sensor pode variar	ResSensor
Número de sensores	O número de sensores da velocidade do vento pode variar	NumSensores
Período de transmissão	Período de transmissão de cada sensor	PeriodoTrans
Período do sensor	O período de captura do sensor pode variar	PeriodoSensor
Tipo de mensagem transmitida	Os tipos de mensagem que a bóia transmite podem variar em conteúdo e formato	TipoMsg
Histórico	O tamanho do histórico de leituras dos sensores pode variar	Historico
Peso de sensor de alta resolução	O peso do sensor de alta resolução para calcular a média ponderada pode variar	PesoAltaRes
Peso de sensor de baixa resolução	O peso do sensor de baixa resolução para calcular a média ponderada pode variar	PesoBaixaRes

A LMA da família de bóias náuticas é uma linguagem de configuração. Essa linguagem deve apoiar as definições das variabilidades das bóias, como número de sensores diferentes, resoluções diferentes para cada sensor, períodos de transmissão de mensagens e de leitura dos sensores diferentes. Cada especificação define a configuração para uma bóia náutica e deve incluir os parâmetros de variação. O tradutor deve verificar que a configuração é válida e deve gerar o código para a configuração. Um exemplo de especificação de uma bóia náutica é mostrada na Figura 2.5.

```

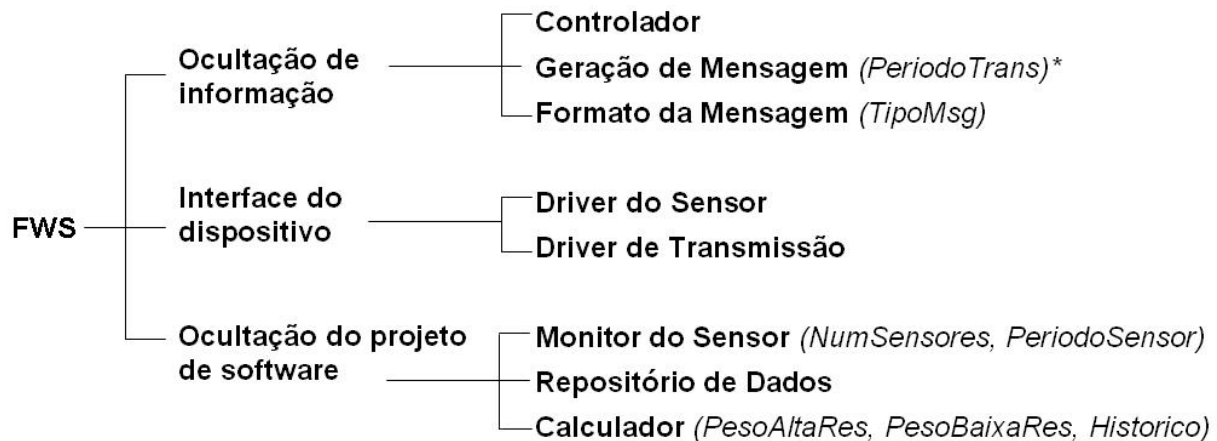
NumSensores(2)
PeriodoSensor(5)
Historico(3)
TipoMsg(SHORTMSG)
SensorBaixaRes(1)
PesoBaixaRes(50)
SensorAltaRes(1)
PesoAltaRes(100)

```

Figura 2.5: Exemplo de uma especificação de uma bóia náutica

Em seguida são implementados os módulos da família, consistindo de uma biblioteca de garbados usada na composição dos produtos da família; o programa de composição para gerar os produtos; uma ferramenta para criar a especificação de cada bóia e um simulador para auxiliar na validação do produto gerado. Cada variabilidade e parâmetro de variação que especifica uma bóia náutica é associada a um módulo, como visto na Figura 2.6. Em seguida deve ser feita uma

correspondência entre as partes variáveis e os gabaritos e entre os módulos que apresentam dependências.



* As anotações de parênteses se referem a parâmetros de variação

Figura 2.6: Hierarquia de módulos das bóias náuticas (Weiss e Lai, 1999)

Na Figura 2.7 é ilustrado um exemplo de um gabarito, referente ao módulo Monitor do Sensor. Ele apresenta duas partes de variabilidade nas bóias náuticas, relacionadas a dois parâmetros de variabilidade, *PeriodoSensor* e *NumSensores*, como mostrado na Figura 2.6. Uma parte corresponde ao intervalo de tempo entre cada captura do sensor (linhas 4-5), que depende do *PeriodoSensor*. Outra parte variável corresponde à leitura de velocidade do vento do sensor e de escrita dessa leitura no repositório de dados (linhas 10-11), pois deve haver repetição dessas operações para cada sensor existente, que é definido pelo *NumSensores*.

```

1  class MonitorSensor extends Thread {
2      public void run() {
3          while (true) {
4              #Dormir por um PeriodoSensor, em que PeriodoSensor é uma constante
5              #cujo valor é estabelecido pelo gerador
6
7              try {Thread.sleep(FWS.PeriodoSensor);}
8              catch (InterruptedException e) {}
9
10             #Para cada sensor incluído para a bóia náutica, uma leitura do sensor e
11             #uma escrita no repositório de dados é adicionada aqui.
12
13             ...
14         }
15     }
16 }

```

Figura 2.7: Gabarito do módulo de Monitor do Sensor

Como foi decidido construir o gerador de aplicação usando um compositor, o gerador precisa mapear os programas em gabaritos de uma biblioteca que será usada para compor os produtos da

família, chamado de mapeamento de composição do sistema. Ele descreve, para cada parâmetro de variação, os gabaritos correspondentes que serão necessários para implementar a funcionalidade representada pelo parâmetro. Além disso, ele incorpora as regras para transformar cada gabarito em código. O gerador implementa então o mapeamento analisando cada valor do programa na especificação, obtendo os seus respectivos gabaritos e seguindo regras para criar o código, como visto na Figura 2.8. Finaliza-se assim a engenharia de domínio da família de bóias náuticas.

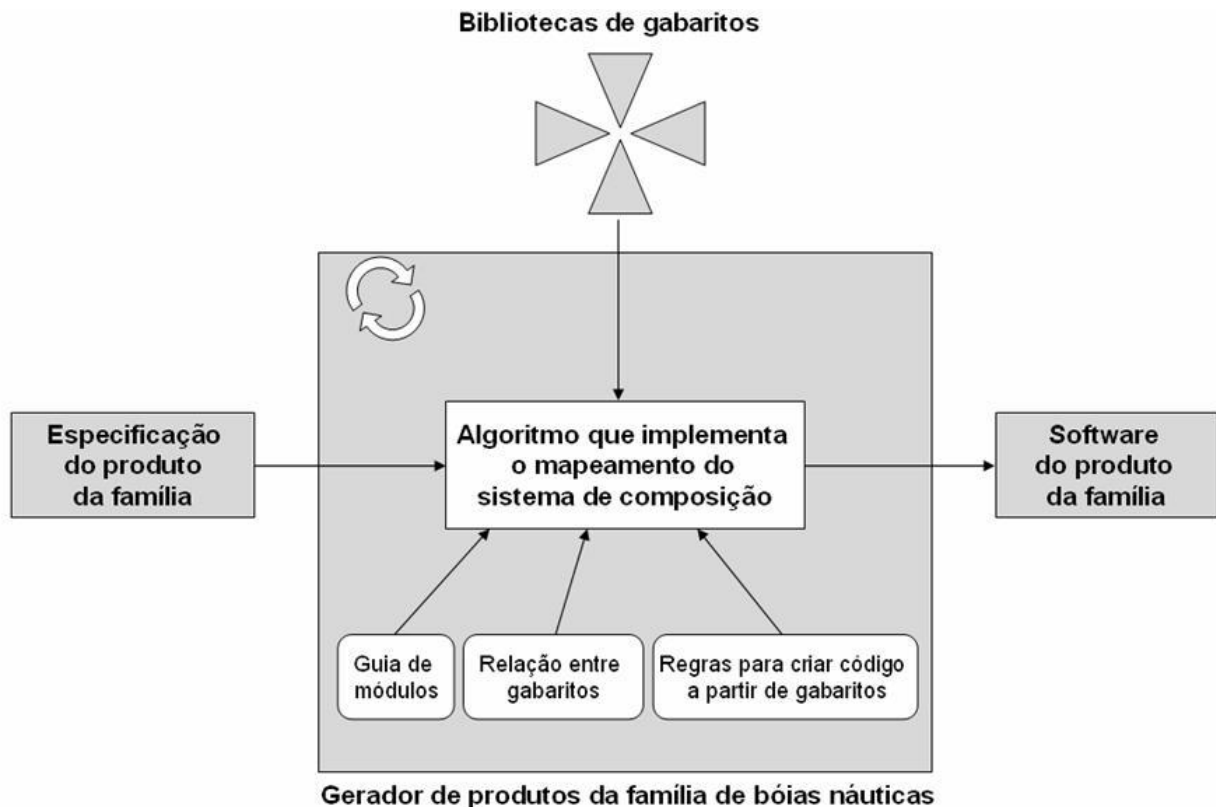


Figura 2.8: Estratégia de geração de bóias náuticas por composição (Weiss e Lai, 1999)

Para realizar a engenharia de aplicação, os requisitos da bóia náutica são analisados e constrói-se uma descrição em LMA, como a apresentada na Figura 2.5. Além disso, o compositor é executado para criar o código do produto e, após testar a nova bóia náutica usando um simulador, o produto é entregue.

2.6.1 Gerador de Aplicação Configurável

Em geral, os geradores de aplicação são ferramentas que produzem artefatos para um domínio específico e eles podem ser custosos e complexos de desenvolver, o que pode não justificar sua construção para determinados domínios em que o custo de desenvolver um gerador (engenharia de domínio), somado com o custo de desenvolver aplicações com esse gerador (engenharia de aplicação), seja maior do que o custo de desenvolver as mesmas aplicações sem o uso do gerador. Por outro lado, os geradores de aplicação configuráveis são geradores que podem ser adaptados para

dar apoio a domínios diferentes e não apenas a um domínio específico. A principal vantagem dessa abordagem é que as atividades de configuração de um gerador de aplicação configurável podem ser menos custosas do que a construção completa de um gerador de aplicação específico, diminuindo o custo de uso do gerador e permitindo a utilização das técnicas de geração para domínios que normalmente não apresentam viabilidade econômica para investir nessas ferramentas (Shimabukuro, 2006).

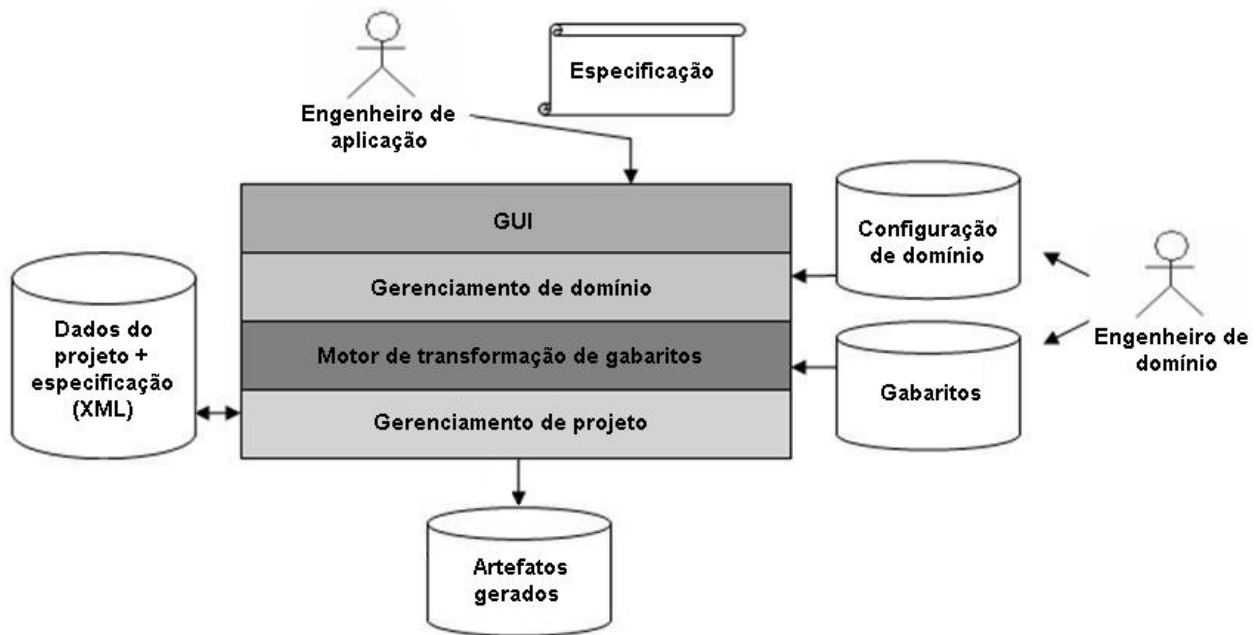
Um gerador de aplicação configurável é uma ferramenta que deve ser configurada para apresentar as mesmas características de um gerador de aplicação específico, ou seja, receber uma especificação, armazená-la em meio persistente, permitir a edição e re-edição dessa especificação, validá-la e transformá-la em artefatos de software. O gerador de aplicação configurável pode, após o seu desenvolvimento, ser configurado para produzir artefatos em diversos domínios, e cada instância configurada desses geradores pode produzir artefatos para diversas aplicações.

Captor

O gerador de aplicação configurável Captor (do inglês *Configurable Application Generator*) foi inicialmente projetado para ser um gerador configurável por linguagens de padrões e, na primeira fase do desenvolvimento, seus requisitos foram definidos com base no gerador de aplicação específico GREN-Wizard (Braga e Masiero, 2003), que é um gerador baseado em linguagens de padrões. Ao finalizar o desenvolvimento desse gerador configurável, os autores decidiram que a ferramenta deveria ser configurada não apenas por linguagens de padrões, mas também por outras linguagens de modelagem de aplicações que podem ser definidas para domínios específicos. Esse processo levou a uma segunda fase de desenvolvimento, que culminou na criação do gerador de aplicação configurável Captor (Shimabukuro, 2006).

Dessa forma, o Captor tem por objetivo facilitar a geração de artefatos de software em variados domínios. Vários tipos de artefatos podem ser gerados pelo Captor, como código, documentação e testes (Shimabukuro *et al.*, 2006). Sua arquitetura, mostrada na Figura 2.9, é composta por quatro módulos: gerenciamento de interface (GUI), gerenciamento de domínio, motor de transformação de gabaritos e gerenciamento de projeto. O Captor foi desenvolvido em Java e se baseia em arquivos XML e gabaritos XSL para realizar a configuração para domínios específicos.

As informações sobre o domínio são incluídas por meio de uma linguagem de alto nível capaz de representar as variabilidades do domínio. Uma instância da linguagem é fornecida por formulários que o engenheiro de aplicação preenche na interface gráfica do Captor. Adicionalmente, foi delineado um processo de desenvolvimento baseado em geradores de aplicação configuráveis composto por três etapas: desenvolvimento do gerador configurável, engenharia de domínio e engenharia de aplicação. Na etapa de desenvolvimento, o gerador configurável é analisado, projetado, implementado e testado. Na etapa de engenharia de domínio, o gerador configurável é instanciado para um determinado domínio e, finalmente, na engenharia de aplicação o gerador é utilizado para gerar artefatos específicos do domínio para o qual foi configurado.

**Figura 2.9:** Arquitetura do Captor

Shimabukuro (2006) realizou três estudos de casos com o Captor para configurá-lo em domínios distintos: persistência, gestão de recursos de negócios e bóias náuticas. Eles permitiram que o Captor fosse configurado para a geração de artefatos como código-fonte, documentos UML e casos de testes.

No Captor são utilizadas LMAs declarativas que são especificadas em um conjunto de formulários organizados hierarquicamente em forma de árvore. Cada formulário contém um conjunto de campos, representados por elementos gráficos. Os formulários podem apresentar pequenas variações, dependendo das escolhas do usuário e das necessidades dos clientes. Após a definição da estrutura hierárquica, o engenheiro de domínio deve definir a forma como o formulário deve se apresentar ao engenheiro da aplicação, ou seja, deve definir quais campos cada formulário contém e quais são os valores válidos dos dados inseridos nesses campos.

O fluxo de execução durante o uso do Captor é dividido em duas etapas: criação/edição da especificação e transformação da especificação em artefatos. Na primeira etapa o engenheiro de aplicação deve inserir a especificação de um produto de uma família na GUI do Captor e, em seguida, o Captor deve validar a especificação e persisti-la no formato XML. Na segunda etapa o Captor utiliza a especificação persistida no formato XML e um conjunto de gabaritos para gerar artefatos de software, como é ilustrado na Figura 2.10.

O Captor fornece apoio para a seleção de gabaritos por meio de uma linguagem, denominada MTL (do inglês *Mapping Transformation Language*). Para cada domínio em que o Captor é configurado, o engenheiro de domínio deve fornecer um arquivo de mapeamento de transformação de gabaritos. Esse arquivo tem diversos objetivos, como realizar asserções no modelo da aplicação e selecionar os gabaritos que devem ser utilizados para gerar os artefatos.

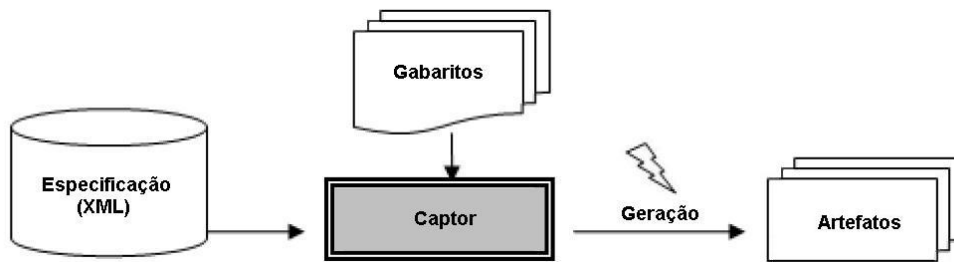


Figura 2.10: Etapa do fluxo de execução do Captor para geração de artefatos (Shimabukuro, 2006)

Na Figura 2.11 são ilustrados os arquivos necessários para realizar a configuração do Captor. O arquivo de configuração de interface gráfica define como a interface com o usuário vai ser adaptada para receber uma especificação e quais são os critérios de validação desses dados. Os gabaritos são arquivos de texto que contém a parte fixa dos artefatos que devem ser gerados e instruções sobre como processar, com base nos dados da especificação, as partes variáveis desses artefatos. O arquivo de MTL é utilizado pelo Captor para selecionar, dependendo dos dados da especificação, quais gabaritos devem ser utilizados na geração. Finalmente, os arquivos de pré e pós-processamento realizam processamentos específicos nos artefatos do domínio.

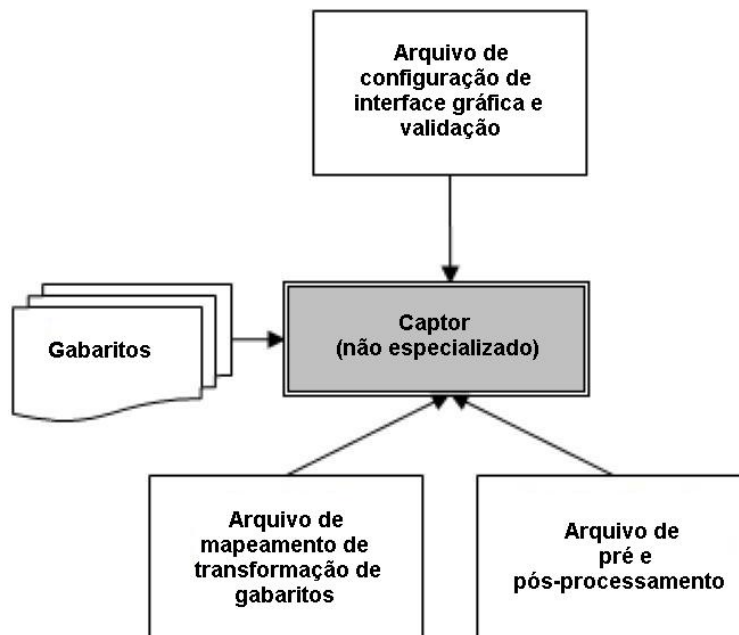


Figura 2.11: Configuração do Captor para um determinado domínio (Shimabukuro, 2006)

2.7 Considerações Finais

Neste capítulo foram apresentadas algumas abordagens para o reúso de software que podem trazer diversos benefícios operacionais e estratégicos para projetos de desenvolvimento com o reúso de artefatos.

No caso de existirem diversas aplicações de uma mesma família que apresentem mais características em comum do que características que os distinguem, é vantajoso desenvolver uma linha de produtos de software, pois o grau de reúso deve ser grande. Os métodos de desenvolvimento de LPS dividem o processo de desenvolvimento quase da mesma forma, mas cada método foca mais em uma parte: ou na engenharia de domínio ou na engenharia de aplicação. Portanto, para ter um processo de desenvolvimento mais completo é interessante convergir essas linhas de pesquisa para ter eficiência tanto na engenharia de domínio quanto na engenharia de aplicação.

É interessante que esse desenvolvimento da LPS seja ágil e para isso ter um método de desenvolvimento que seja ágil também. Entre os métodos de desenvolvimento de LPS, o PLUS parece ser o mais próximo disso, pois considera a possibilidade de desenvolver a linha integrando com o processo unificado ou com o modelo em espiral. Apesar de ser bem didático para a engenharia de domínio, o PLUS é superficial em termos de engenharia da aplicação e pode ser interessante usar um método como o FAST em conjunto para melhorar e agilizar a engenharia de aplicações de uma LPS.

O FAST sugere o uso de geradores de aplicações. Para não ser necessário desenvolver totalmente um gerador para a LPS pode ser usado um gerador de aplicação configurável, podendo diminuir tempo e esforço. Dessa maneira, é preciso apenas configurar o gerador para o domínio da linha, especificar as partes variáveis da aplicação e gerar a implementação. No FAST, a configuração de um gerador para um domínio é feita de modo cascata, mas para agilizar a configuração do gerador é melhor optar por definir a linguagem de modelagem da aplicação de forma incremental.

A maioria das linhas de produtos desenvolvidas possui arquitetura baseada em componentes para facilitar o reúso entre aplicações da linha, mas não detalha como os componentes devem ser identificados e implementados. Seria interessante ter disponível uma LPS completa com um projeto detalhado de componentes e com uma implementação real. Para auxiliar na identificação dos componentes pode ser usado um método de desenvolvimento baseado em componentes. Uma boa opção é o *UML Components* que, apesar de não ser específico para LPS, é bem didático e percebe-se a possibilidade de uso no desenvolvimento da LPS. De qualquer maneira, o método não possui detalhes de implementação.

Nas pesquisas realizadas, percebeu-se a dificuldade de encontrar uma LPS desenvolvida que não seja trivial e que esteja disponível para pesquisas, portanto, é interessante desenvolver uma linha como prova de conceito para analisar questões relacionadas ao seu desenvolvimento e para servir de apoio a vários estudos.

Arquitetura de Software, Componentes e Aspectos

3.1 Considerações Iniciais

A base para o desenvolvimento de uma família de produtos é a arquitetura do software e os componentes a serem implementados. Podem existir interesses transversais no software e isso refletirá na arquitetura e nos componentes, entrecortando as unidades arquiteturais e dos componentes. Portanto, a programação orientada a aspectos apresenta um importante relacionamento com esses assuntos, fornecendo conceitos que permitem expressar os interesses transversais separadamente em uma estrutura modular aspectual.

Neste capítulo são apresentadas essas questões importantes para o desenvolvimento da proposta. Na Seção 3.2 é apresentada uma visão geral da programação orientada a aspectos, assim como a linguagem orientada a aspectos AspectJ e a linguagem FuseJ, que é uma linguagem para o DSBC que utiliza as idéias da orientação a aspectos. Na Seção 3.3 é apresentada uma introdução à arquitetura de software e às linguagens de descrição de arquiteturas de software e como ilustração discute-se brevemente a Acme, pois incorpora os principais conceitos dessas linguagens. Na Seção 3.4 é apresentado o desenvolvimento de software baseado em componentes e em aspectos, além de discutir algumas abordagens existentes relacionadas a componentes aspectuais. Por fim, na Seção 3.5 são apresentadas as considerações finais do capítulo.

3.2 Programação Orientada a Aspectos

Dijkstra (1976) definiu o termo “interesse” (*concern*) como sendo qualquer parte de um sistema em que se foca a atenção em um momento da atividade de projeto, deixando de lado, conscientemente, as demais partes do problema (separação de interesses). Ossher e Tarr (2001) consideram interesse como sendo o principal critério para decompor software em partes menores e mais gerenciáveis e compreensíveis que tenham significado para um engenheiro de software.

As linguagens de programação apresentam limitações que permitem encapsular adequadamente apenas interesses para os quais foram criadas, enquanto que os interesses de outros tipos, como os transversais (*crosscutting concerns*), por exemplo, são implementados de maneira a ficarem espalhados pelas divisões ou módulos de um sistema, causando problemas como o espalhamento e o entrelaçamento de código, repetindo lógica semelhante em diferentes módulos e prejudicando o entendimento, projeto, desenvolvimento, reúso e manutenção dos sistemas. Exemplos típicos de interesses transversais são interesses como persistência, registros (*logging*), concorrência, técnicas de cache (*caching*), distribuição, controle de acesso, mecanismos de tolerância a falhas, sincronização e regras de negócio (Cibran *et al.*, 2003) que descrevem lógica específica de negócios.

Essas limitações das linguagens na decomposição do sistema em apenas uma dimensão recebem o nome de tirania da decomposição dominante (Tarr *et al.*, 1999). Diante desse problema, um grupo de pesquisadores do centro de pesquisa da Xerox, em Palo Alto, propôs uma abordagem chamada de Programação Orientada a Aspectos (POA) que procura fornecer conceitos e mecanismos de programação para separar e encapsular esses interesses transversais para eliminar o espalhamento e entrelaçamento de código, resultando em sistemas mais legíveis, fáceis de entender, implementar, integrar, reusar, personalizar, evoluir e manter (Kiczales *et al.*, 1997). De maneira geral, a POA pode ser entendida como o desejo de fazer declarações quantificadas em relação ao comportamento de programas, e essas quantificações terem preferência sobre programas escritos por programadores inconscientes (Filman e Friedman, 2000).

Binkley *et al.* (2006) definem o Desenvolvimento de Software Orientado a Aspectos (DSOA) como uma abordagem de programação em que interesses transversais são isolados e extraídos para módulos separados, chamados de aspectos. O aspecto adiciona uma funcionalidade ao código-base pela interceptação do fluxo de execução, sem a necessidade do código-base mencionar o código do aspecto explicitamente (Binkley *et al.*, 2006). Em outras palavras, o código-base permanece alheio à funcionalidade adicionada por um aspecto e é o aspecto que especifica (quantifica) os locais no código-base afetados pela nova funcionalidade (Filman e Friedman, 2000). Um aspecto é então um interesse transversal bem modularizado (Kiczales *et al.*, 2001).

A implementação de um aspecto é feita por meio de interceptações em pontos bem definidos no fluxo de execução do código-base, como em chamadas a métodos e construtores e na modificação de valores de atributos. Esses pontos de interceptação são chamados de pontos de junção e são

definidos pelo aspecto de acordo com regras da linguagem de POA utilizada. Ao interceptar esses pontos, o aspecto pode executar um trecho de código antes, depois, ou no lugar do ponto interceptado. Como os aspectos são desenvolvidos em módulos separados dos módulos do código-base, há a necessidade de um processo de combinação (*weaving*) dos aspectos com o programa para poder gerar o sistema final executável com todos os interesses implementados. Esse processo pode ser realizado tanto em tempo de compilação (combinação estática) quanto em tempo de execução (combinação dinâmica).

Existem dois tipos diferenciados de linguagens de POA: linguagens de domínio específico e linguagens de propósito geral. As linguagens de domínio específico abordam somente alguns tipos de interesses transversais de um determinado domínio, como a linguagem COOL (Lopes, 1997), que trata do interesse de *locking*/exclusão mútua, e a linguagem RIDL (Lopes, 1997), que trata o interesse de invocação remota. As linguagens de propósito geral oferecem mecanismos para a definição de pontos do sistema nos quais aspectos possam alterar a semântica de sistemas de domínios gerais, como a linguagem AspectJ (Kiczales *et al.*, 2001).

Além disso, pesquisas têm sido realizadas com o intuito de identificar os aspectos de um sistema desde as fases iniciais do desenvolvimento de software e não apenas durante a implementação, pois os interesses transversais são parte inerente dos sistemas de software. Clarke e Baniassad (2005) propõem o desenvolvimento de software orientado a aspectos usando temas, incluindo as fases de requisitos, análise e projeto considerando aspectos, antes mesmo da implementação propriamente dita.

Diversos pesquisadores têm proposto usar programação orientada a aspectos para auxiliar na implementação de uma LPS, como Apel *et al.* (2006), Kastner *et al.* (2007), Heo e Choi (2006), Lee *et al.* (2006), Anastasopoulos e Muthig (2004) e Figueiredo *et al.* (2008). A maioria fez pesquisas qualitativas referentes ao uso de aspectos em uma LPS. Figueiredo *et al.* (2008) fizeram estudos quantitativos comparando o uso da programação orientada a objetos e da programação orientada a aspectos em uma LPS. Outros autores têm pesquisado o uso de aspectos para implementação específica de características de uma LPS (Mezini e Ostermann, 2004; Apel e Batory, 2006; Apel *et al.*, 2006).

3.2.1 A Linguagem AspectJ

AspectJ (Kiczales *et al.*, 2001) é uma extensão orientada a aspectos para a linguagem orientada a objetos Java. Ela é uma linguagem de propósito geral que oferece mecanismos para permitir implementar interesses transversais em módulos separados. A linguagem AspectJ foi desenvolvida inicialmente na Xerox, mas tornou-se posteriormente um projeto da comunidade *Eclipse* de software livre (AspectJ Team, 2008). Para permitir a implementação de aspectos, a linguagem AspectJ introduz algumas construções, tais como: aspectos (*aspects*); conjuntos de pontos de junção (*pointcuts*), que são pontos bem determinados na execução de um código-base; e adendos (*advices*), que implementam o comportamento adicional para cada ponto de junção. Além disso, podem

ser definidos atributos e métodos que alteram a estrutura estática das classes afetadas pelo aspecto, que são chamadas de declarações inter-tipos (*intertype declarations*).

Os **aspectos** são unidades da implementação transversal que são similares a classes de várias maneiras, mas a diferença mais importante é que aspectos podem entrecortar tipos, ou seja, afetar e modificar outras classes (Kiselev, 2002). Outras diferenças são que as classes podem ser diretamente instanciadas enquanto os aspectos não o podem e que as classes não podem estender aspectos enquanto os aspectos podem estender outros aspectos.

Os **conjuntos de ponto de junção** identificam um conjunto de pontos bem definidos no fluxo de execução de um programa, nos quais serão executados comportamentos adicionais inseridos por aspectos. Esses conjuntos podem identificar um único ponto de junção ou a composição de vários deles, usando operadores lógicos. Os pontos de junção podem ser chamadas e execução de construtores de classes e de métodos, iniciação de classe, acesso a atributos, execução de tratador de exceção e de adendos e iniciação de objetos (Laddad, 2003).

Os **adendos** executam quando o aspecto interceptar a classe nos pontos de junção determinado pelo designador dos conjuntos de ponto de junção. Os adendos podem ser de três tipos básicos: o *before*, que é executado antes do ponto de junção ser executado; o *after*, que é executado depois do ponto de junção ser executado; e o *around*, que é executado no lugar do ponto de junção, sendo que após ser executado pode, ou não, chamar a execução do ponto de junção. Os adendos são construções semelhantes aos métodos, mas não podem ser chamados diretamente pelo programa e nem pelo próprio aspecto, pois sua execução é feita automaticamente após a interceptação no ponto de junção. Os adendos não possuem nome, não têm especificadores de acesso (por exemplo *public* e *private*) e têm acesso a variáveis especiais das execuções dos pontos de junção, como a assinatura dos métodos interceptados.

Um exemplo da codificação de um aspecto é mostrado na Figura 3.1. O programa tem o interesse primário de imprimir o texto “Olá Mundo” no console. Esse interesse é implementado na classe *Teste*, que tem dois métodos. O primeiro método, chamado de *olaMundo()*, implementa o único interesse do sistema e quando chamado ele imprime o texto “Olá Mundo” no console. Ele é chamado pelo método *main()* da mesma classe (linhas 6-9). É adicionado um novo interesse no sistema para registrar (*log*) a execução de todas as funções de saída. Esse interesse é transversal, pois o registro deve ser feito em cada uma das funções existentes. O ponto de junção é feito então na chamada ao método *olaMundo()* (linha 13), que é o local em que o interesse transversal entrecorta o código, pois o método é o responsável pela impressão de um texto. O adendo deve ter o seu código executado logo antes do ponto de junção, pois ele é do tipo *before* (linha 14). Dessa forma, é impresso o texto “Antes da chamada” sempre antes de imprimir o texto “Olá Mundo”.

3.2.2 A Linguagem FuseJ

FuseJ é uma linguagem de programação que recupera idéias da programação orientada a aspectos e permite implementar todos os interesses como *Java Beans*, o modelo de componentes

```
1 public class Teste {  
2     public void olaMundo() {  
3         System.out.println("Olá Mundo");  
4     }  
5  
6     public static void main(String args[]) {  
7         Teste teste = new Teste();  
8         teste.olaMundo();  
9     }  
10 }  
11  
12 public aspect TesteAspecto {  
13     pointcut registro() : call(public void olaMundo());  
14     before(): registro() {  
15         System.out.println("Antes da chamada");  
16     }  
17 }
```

Figura 3.1: Exemplo de aspecto em AspectJ

baseado em Java (Suvée *et al.*, 2006). Dessa forma, a linguagem ilustra uma arquitetura orientada a componentes unificada no nível de implementação e introduz conceitos mapeados do mundo real, tendo como objetivo manter o acoplamento entre componentes o mais baixo possível e, assim, alcançar maior reusabilidade. Tal objetivo do FuseJ pode ser alcançado também utilizando o framework do Spring (Spring, 2008).

Existem diversas tecnologias disponíveis que almejam integrar as idéias e os conceitos de DSOA e DSBC. Algumas delas introduzem uma abordagem assimétrica parecida ao AspectJ, em que interesses transversais são implementados pelo uso de uma linguagem de POA dedicada, como é o caso da linguagem JAsCo (Suvée *et al.*, 2003). Outras tecnologias são baseadas em frameworks e implementam aspectos usando uma linguagem de programação base (Suvée *et al.*, 2006).

O FuseJ surgiu a partir de pesquisas relacionadas à introdução de uma abordagem simétrica e unificada para combinar as idéias e os conceitos de desenvolvimento orientado a aspectos e DSBC. Suvée *et al.* (2005) não consideram necessário um módulo especializado para o aspecto, ao invés disso, propõem mecanismos de composição orientados a aspectos. Com isso, eles introduzem aspectos como entidades especializadas no FuseJ, aplicando esses mecanismos de composição orientados a aspectos em construções de componentes existentes. Dessa maneira, ao se representar um elemento do software como um método, deve ser possível compor esse método com outros como se fosse um adendo. Isso permite prorrogar a escolha de um mecanismo específico de composição e permite empregar esse mecanismo em módulos de software já existentes.

A linguagem FuseJ propõe o conceito de especificação de serviço que define o conjunto de operações que o componente deve prover para o ambiente e o conjunto de operações que deve esperar do ambiente. Mesmo que propriedades não-funcionais existam no sistema, FuseJ fornece e modela essas propriedades como componentes normais. Essas propriedades serão compostas posteriormente, de forma aspectual, levando em consideração os interesses específicos da aplicação (Suvée *et al.*, 2006).

Para descrever o processo de composição dos componentes, a linguagem de configuração do FuseJ usa um construtor de configuração explícito (*configuration*) que atua como mediador, descrevendo como dois ou mais componentes devem interagir pela ligação de operações requeridas/fornecidas. Além disso, uma configuração pode descrever interações orientadas a aspectos pela declaração do papel da fonte (*source role*) como sendo aspectual. As interações transversais podem ser de três tipos: *before*, *after* e *around*. Essas interações são definidas da mesma forma do AspectJ. Entretanto, o FuseJ¹ ainda não apresenta algumas técnicas de composição e de encapsulamento orientadas a aspectos, como a precedência e a combinação de aspectos (Suvée *et al.*, 2006).

A estrutura da entidade de configuração, mostrada na Figura 3.2, é formada por ao menos um *linklet* (linhas 2-8). Cada *linklet* liga as operações definidas em um ou mais componentes e é geralmente formado de quatro partes individuais: (i) papel do alvo, que enumera o conjunto de operações a serem executadas (linha 3); (ii) papel da fonte, que enumera o conjunto de operações que agem como gatilho (linhas 4-5); (iii) mapeamento de propriedades, parte opcional que enumera o conjunto de mapeamentos de propriedades, descrito em termos de fonte, alvo ou operações externas (linha 6); e (iv) especificação de condição, parte opcional que enumera o conjunto de pré-condições.

```

1 configuration <name> configures (<comp>|<serv>)+ as <serv> {
2   (linklet <linkname> {
3     execute|expose : (<op_comp>|<servop>)           //papel do alvo
4     for|before|after|around|as :                   //tipo de interação (normal ou aspectual)
5       (<op_comp>|<op_serv>)+                         //papel da fonte
6     (where : (<mapeamento_parametro>)+)?           //mapeamento de propriedade (opcional)
7     (when : (<op_comp>|<op_serv>)+)?                 //especificação de condição (opcional)
8   })+
9 }
```

Figura 3.2: Estrutura geral da entidade de configuração do FuseJ (Suvée *et al.*, 2006)

A Figura 3.3 ilustra a configuração que especifica uma interação aspectual entre componentes de um subsistema de controle de transferência de fragmentos de arquivos compartilhados (Suvée *et al.*, 2006). Ela configura os componentes TransferNetC e Logger, que estão de acordo com a especificação de serviço TransferNetS, como o novo componente LoggedTransferNetC. A configuração é responsável por garantir que cada execução de uma operação que seja parte do componente TransferNetC seja registrada para consultas futuras. O alvo é então a operação log do componente Logger (linhas 5-6). O registro (*log*) entrecorta todas as operações do componente TransferNetC e a interação é do tipo *before* (linhas 7-8). Finalmente, a cláusula *where* inicia o parâmetro *st* com a assinatura do método da operação que o desencadeou (linhas 9-10). Na Figura 3.4 é mostrada essa configuração da interação entre os componentes. O componente

¹O grupo de pesquisa do FuseJ parece ter descontinuado a pesquisa usando o FuseJ. Um foco deles atualmente tem sido uma especialização da linguagem AspectJ, a StrongAspectJ (De Fraine *et al.*, 2008)

TransferNetC apresenta duas operações entrecortadas (getFileFrag e findFileFrag) pela operação log do componente Logger.

```

1 configuration LoggedTransferNetC configures
2   TransferNetC, Logger as TransferNetS {
3
4   linklet log {
5     execute:
6       Logger.log(String st);
7     before:
8       TransferNetC.*(..);
9     where:
10      st = Source.getMethodSignature();
11   }
12 }
13 }

```

Figura 3.3: Exemplo de codificação de interação orientada a aspecto entre os componentes TransferNetC e Logger (Suvée *et al.*, 2006)

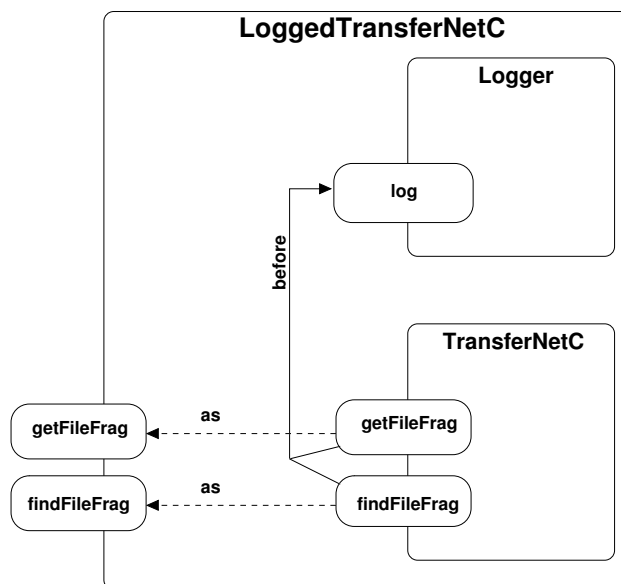


Figura 3.4: Interação entre componentes TransferNetC e Logger (Suvée *et al.*, 2006)

3.3 Arquitetura de Software

O engenheiro de software freqüentemente descreve a arquitetura de seu sistema. A descrição trata de propriedades de alto nível do sistema, como a organização geral, a decomposição em componentes, as associações das funcionalidades dos componentes e a forma como os componentes interagem. Shaw *et al.* (1995) afirmam que a arquitetura de um sistema de software define o sistema em termos de componentes e de interações entre esses componentes. Além de especificar a

estrutura e a topologia do sistema, a arquitetura mostra a correspondência almejada entre os requisitos do sistema e os elementos do sistema implementado. Os modelos arquiteturais podem então esclarecer diferenças estruturais e semânticas entre componentes e interações. Os elementos são definidos independentemente para que possam ser reusados em diferentes contextos.

Logo, é importante encontrar notações apropriadas para descrever os sistemas baseados em componentes e, por esse motivo, diversos pesquisadores propuseram notações formais para representar e analisar projetos arquiteturais, genericamente referenciadas como linguagens de descrição de arquitetura (ADL - *Architecture Description Language*). O objetivo dessas linguagens é auxiliar projetistas a primeiro definirem a arquitetura de software com abstrações que achem úteis e então fazer uma transição suave para o código (Shaw *et al.*, 1995). Assim, as ADLs oferecem tanto um framework conceitual quanto uma sintaxe concreta para caracterizar arquiteturas de software (Garlan *et al.*, 1997). De forma superficial, uma ADL para aplicações de software foca na estrutura de alto nível da aplicação, ao invés de focar nos detalhes de implementação de módulos específicos (Vestal, 1993). Medvidovic e Taylor (2000) tentam definir ADLs de forma mais concisa e precisa da seguinte forma: uma ADL deve modelar componentes, conectores e suas configurações, explicitamente, e, além disso, para ser verdadeiramente útil e usável, ela deve fornecer suporte por meio de uma ferramenta para o desenvolvimento baseado em arquitetura e a sua evolução.

Diversas ADLs têm sido propostas, cada uma fornecendo características complementares para o desenvolvimento e a análise da arquitetura de aplicações. Além disso, algumas são de domínio específico e outras de propósito geral (Medvidovic e Taylor, 2000). Segundo Garlan *et al.* (2000), apesar da diversidade, todas as ADLs compartilham uma base conceitual similar, ou ontologia, que determina um núcleo comum de conceitos e interesses para a descrição arquitetural. Os principais elementos dessa ontologia são:

- componentes: representam os elementos computacionais primários de um sistema;
- conectores: representam interações entre componentes, mediando as atividades de comunicação e coordenação entre componentes;
- sistemas: representam configurações de componentes e conectores;
- propriedades: representam informações semânticas de um sistema e dos seus componentes;
- restrições: representam afirmações de um projeto arquitetural que devem permanecer verdadeiras ao longo de sua evolução;
- estilos: representam famílias ou sistemas relacionados e tipicamente definem um vocabulário de tipos e de regras de elementos de projeto para compô-los.

Medvidovic e Taylor (2000) separam os conceitos de descrição de arquitetura de maneira diferente, dividindo-os em apenas três blocos de construção: componentes, conectores e configurações arquiteturais. Os dois primeiros apresentam a mesma representação de Garlan *et al.* (2000)

e as configurações arquiteturais definem as estruturas arquiteturais e como os componentes e conectores são ligados, correspondendo assim ao elemento de sistemas da ontologia. Além disso, Medvidovic e Taylor (2000) afirmam que os componentes e os conectores podem ter interfaces, tipos, semânticas e restrições associadas, porém, apenas interfaces explícitas de componentes são características requeridas em ADLs.

3.3.1 A Linguagem Acme

Cada ADL e suas ferramentas de suporte operam isoladamente, dificultando a integração das ferramentas e o compartilhamento de descrições arquiteturais. A linguagem Acme (Garlan *et al.*, 2000) foi desenvolvida como um esforço conjunto da comunidade de pesquisa de arquitetura de software como um formato comum de troca entre ferramentas de projeto arquitetural. Acme fornece um framework estrutural para caracterizar arquiteturas, além de ter facilidades de anotação para informações adicionais específicas de ADLs. Ela é baseada na premissa de que existem similaridades suficientes nos requisitos e nas características das ADLs tal que informações significativas independente de ADLs possam ser compartilhadas (Garlan *et al.*, 1997).

A linguagem Acme incorpora a ontologia de arquitetura descrita anteriormente e fornece uma linguagem extensível semanticamente e um ferramental rico para análise e integração de ferramentas arquiteturais desenvolvidas independentemente (Garlan *et al.*, 2000), por meio do mapeamento de especificações de arquiteturas de uma ADL para outra. De acordo com Medvidovic e Taylor (2000), a linguagem Acme em si não é uma ADL, ela contém diversas características semelhantes a uma ADL e corresponde a uma linguagem de intercâmbio entre descrições de arquiteturas (*Architecture Description Interchange Language*). Diferentemente, Garlan *et al.* (2000) consideram a linguagem Acme como sendo uma ADL de segunda geração, pois é construída em cima da experiência de outras ADLs e fornece os elementos essenciais do projeto da arquitetura.

A estrutura arquitetural básica na Acme é descrita com componentes, conectores, sistemas, anexos, portas, papéis e representações. Os componentes são encapsulamentos que apóiam múltiplas interfaces conhecidas como portas. As portas são unidas a portas de outros componentes usando os conectores, que têm papéis como intermediários. Esses papéis dos conectores são ligados diretamente a portas. Os anexos definem então um conjunto de associações entre portas e papéis. Os conjuntos de componentes, de conectores e de anexos que descrevem a topologia do sistema ficam contidos no sistema.

Como um exemplo tem-se a Figura 3.5 com um diagrama de arquitetura trivial, contendo dois componentes – um cliente e um servidor – associados por um conector RPC. A Figura 3.6 contém a descrição Acme dessa arquitetura. Os componentes cliente e servidor são declarados como tendo cada um uma única porta, respectivamente, enviar-requisicao e receber-requisicao (linhas 2-3). O conector rpc possui dois papéis, chamados de chamador e chamado (linha 4). Enfim, a topologia do sistema é declarada por meio de uma listagem de anexos (linhas 5-7).

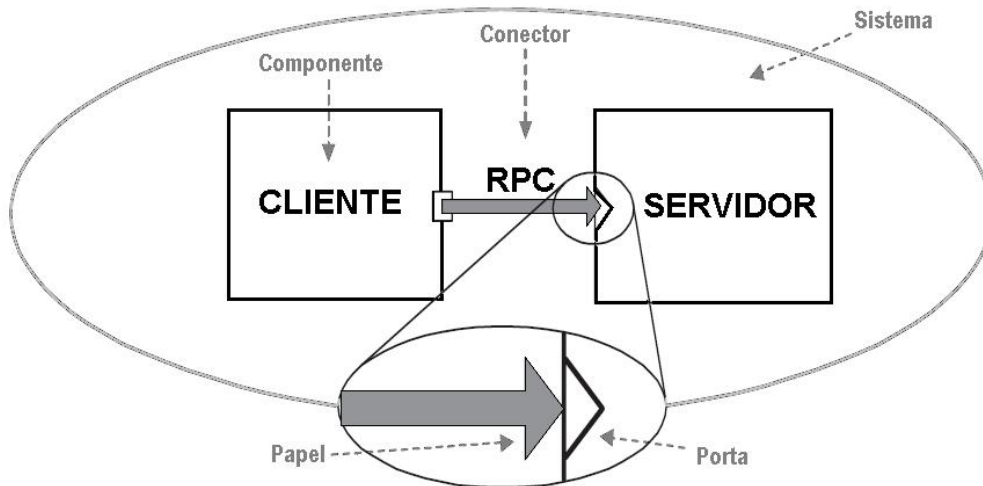


Figura 3.5: Diagrama de um sistema simples cliente-servidor na Acme (Garlan *et al.*, 2000)

```

1 System cs_simples {
2   Component cliente = { Port enviar-requisicao }
3   Component servidor = { Port receber-requisicao }
4   Connector rpc = { Roles { chamador, chamado } }
5   Attachments : {
6     cliente.enviar-requisicao to rpc.chamador;
7     servidor.receber-requisicao to rpc.chamado;
8   }
9 }

```

Figura 3.6: Sistema simples de cliente-servidor na Acme (Garlan *et al.*, 2000)

Para descrever um dado elemento (componente, conector, porta ou papel) em maiores detalhes são feitas decomposições alternativas do elemento por meio das representações. Portanto, uma representação é uma descrição mais refinada de um elemento. Existem ainda outros elementos Acme que apóiam características arquiteturais mais sofisticadas, como é o caso das propriedades e dos estilos. As propriedades são um mecanismo para anotar elementos de projeto com informações detalhadas e, geralmente, não são estruturadas. Elas podem ser anexadas, na forma de anotações, a qualquer um dos elementos básicos Acme. Os estilos definem conjuntos de tipos de componentes, conectores, propriedades e conjuntos de regras que especificam como os elementos devem ser compostos de forma válida em um domínio reusável.

3.3.2 O Desenvolvimento Orientado a Aspectos e as Linguagens de Descrição de Arquitetura

Alguns interesses arquiteturais não podem ser capturados de forma modular usando as abstrações existentes nas ADLs e na Acme (Batista *et al.*, 2006b), pois alguns interesses são transversais

até mesmo no nível de projeto, podendo ser especificados nas unidades arquiteturais de forma localizada. De forma semelhante à noção de aspecto no nível de implementação (Kiczales *et al.*, 1997), esses interesses entrecortam as unidades arquiteturais e denotam aspectos arquiteturais (*architectural aspects*) (Cuesta *et al.*, 2005; Baniassad *et al.*, 2006).

Como modo de fornecer apoio à modularização de interesses transversais, foram propostas algumas linguagens de descrição de arquiteturas orientadas a aspectos (Pérez *et al.*, 2003; Pinto *et al.*, 2005), ora como extensões de ADLs existentes, ora desenvolvidas do zero usando abstrações da POA, como aspectos, pontos de junção e adendos. Entretanto, de acordo com Batista *et al.* (2006a), ainda existe pouco consenso em como integrar as ADLs com o desenvolvimento de software orientado a aspecto. Por causa dos resultados de um estudo detalhado das questões consideradas relevantes na integração de ADLs e do desenvolvimento orientado a aspectos, eles propõem a modelagem de interesses transversais usando as mesmas abstrações das convenções das ADLs e fazendo apenas pequenas adaptações, ao invés de introduzir novas abstrações que elevem os conceitos de linguagens de programação para o nível da arquitetura de software. Com essa proposta, os autores realizam uma extensão da linguagem Acme, a AspectualACME (Batista *et al.*, 2006b), com o objetivo de integrar aspectos e ADLs. Eles estendem a Acme pela introdução de conectores aspectuais (uma extensão da noção tradicional de conectores) e de suporte de quantificação no nível configuracional.

3.4 Desenvolvimento de Software Baseado em Componentes e em Aspectos

A engenharia de software baseada em componentes tem o objetivo de estruturar um sistema pela separação de interesses em entidades claramente definidas, os componentes. Esses componentes devem ser elementos reusáveis, oferecendo mecanismos de fácil evolução e adaptação. Como eles podem ser reusados em vários contextos e em diferentes domínios de aplicação, é difícil prever todos os requisitos não-funcionais a que eles devem satisfazer. Portanto, para ter componentes mais genéricos, as propriedades não-funcionais e os interesses dependentes de contexto devem ser implementados separadamente dos componentes e sua integração no sistema deve fazer parte do processo de composição (Cottenier e Elrad, 2004).

Assim como no desenvolvimento com programação orientada a objetos, o desenvolvimento baseado em componentes também sofre do problema da tirania da decomposição dominante (Ossher e Tarr, 1999), pois as questões de entrelaçamento e espalhamento aparecem também no DSBC quando existem interesses transversais (Lieberherr *et al.*, 1999) (Duclos *et al.*, 2002). Esses interesses podem entrecortar tanto a estrutura interna do componente (intra-componente) quanto as suas interfaces (inter-componente) (Cottenier e Elrad, 2004). Dessa forma, é difícil modularizar esses interesses em componentes normais e tem-se sugerido o apoio da POA no DSBC. Com a

POA, os interesses podem ser modularizados e os componentes podem ser refinados de forma não-invasiva.

O comportamento fornecido pelos aspectos não é tão diferente daquele dos componentes normais, ambos implementam alguma funcionalidade requerida pela aplicação e eles diferem apenas pela forma como interagem com o resto do sistema (Suvée *et al.*, 2006). Apesar disso, a integração entre aspectos e componentes não é uma tarefa simples, principalmente, ao tratar interesses transversais do tipo intra-componente, pois os componentes possuem certos princípios e objetivos que devem ser respeitados e são desafios na integração, como (i) o encapsulamento, (ii) a certificação de qualidade e (iii) a previsibilidade de composição dos componentes (Cottenier e Elrad, 2004).

O primeiro desafio é que os componentes em geral são utilizados como caixa preta e só se tem acesso às suas interfaces. Um aspecto pode facilmente acessar a estrutura interna do componente, quebrando assim a sua política de encapsulamento. Por esse mesmo motivo, chega-se ao segundo desafio, pois se um aspecto é utilizado e ele acessa o interior de um componente, a garantia de qualidade pode ser totalmente perdida, pois não se pode garantir que o componente atende às especificações originais e aquele que reusar o componente não poderá exigir a garantia dos atributos de qualidade. Em relação ao terceiro desafio, com a composição entre componentes sem aspectos, sabe-se, em termos gerais, qual o resultado da composição. Com o seu poder de expressividade, um aspecto pode mudar o fluxo de execução de um programa, mudar a hierarquia de classes, introduzir métodos no componente afetado, etc. Portanto, a composição não tem, a princípio, resultados previsíveis.

Uma solução para esses desafios é comprometer a expressividade dos aspectos, permitindo que os aspectos apenas atuem nas operações expostas nas interfaces dos componentes e não permitindo a extensão de operações por meio de declarações inter-tipos. Essa solução é usada em diversas abordagens que integram DSBC e POA, como as propostas dos autores Pessemier *et al.* (2006), Clemente *et al.* (2002b) e Eler (2006). Outras abordagens optam por tentar manter a expressividade e utilizam algum meio dos componentes não saberem da existência de aspectos. Como exemplos dessas abordagens tem-se o *OpenModules* de Aldrich (2004) e o AOCE (do inglês *Aspect Oriented Component Engineering*) de Grundy e Patel (2001).

Além desses desafios na união dos conceitos de aspectos no DSBC, também existem dificuldades na integração dos conceitos de componentes na POA. Isso ocorre pois os componentes são independentes de contexto, dinâmicos e interagem entre si. Ademais, em geral, a instalação dos aspectos é geralmente estática, eles são dependentes de contexto e não há modelos adequados para prover interações específicas entre aspectos (Suvée *et al.*, 2003).

3.4.1 Abordagens de Componentes Aspectuais

Os componentes são úteis não apenas para a modelagem de composição comportamental, mas também são bons construtores para expressar aspectos. Esse novo tipo de componente é chamado de componente aspectual e expressa cada aspecto separadamente em uma estrutura modular

(Lieberherr *et al.*, 1999). O componente aspectual representa um interesse transversal e é um componente normal que fornece como serviço o código de um adendo que representa o comportamento a ser combinado com um conjunto de componentes normais (Pessemier *et al.*, 2006).

Pessemier *et al.* (2006) propõem uma abordagem simétrica, considerando aspectos como componentes. Eles afirmam melhorar a abordagem de componentes, fornecendo suporte à POA, e melhorar a abordagem de aspectos, aplicando conceitos de DSBC à POA. O modelo é chamado de FAC (do inglês *Fractal Aspect Component*) e é uma extensão de um modelo de componente chamado Fractal (Bruneton *et al.*, 2004). A proposta conta com três noções principais: componente aspectual, domínio de aspecto (*aspectual domain*) e ligação de aspecto (*aspectual binding*). O componente aspectual é o encapsulamento do código de um adendo, representado por um componente normal Fractal. O domínio de um aspecto é a retificação dos componentes selecionados pelo componente aspectual. A ligação de aspecto é o relacionamento implícito entre um componente aspectual e o componente em que ele é aplicado. Esses conceitos são mapeados no modelo de componentes Fractal. Ainda estão sendo feitos estudos para considerar o nível arquitetural dos aspectos no DSBC.

Clemente e Hernández (2003) apresentam um processo chamado de ACBSE (do inglês *Aspect Component Based Software Engineering*) em que a programação orientada a aspectos é utilizada para descrever e implementar as dependências entre componentes, durante as fases de Projeto e Especificação, Implementação, Empacotamento, Montagem e Implantação dos componentes. Para chegar a esse processo, eles focaram no estudo do modelo CCM (*Corba Component Model*) e na sua extensão orientada a aspecto, o AspectCCM (Clemente *et al.*, 2002b). Para desenvolver o AspectCCM, utilizaram UML para a modelagem e separaram a definição de dependências entre componentes (intrínseca e não-intrínseca) durante a fase de Projeto e Especificação usando técnicas de POA (Clemente *et al.*, 2002a). Além disso, são usados descritores XML para descrever as propriedades dos componentes do sistema na fase de empacotamento e o código de interconexão é gerado pela tradução do XML.

Eler (2006) propõe um método para o Desenvolvimento de Software Baseado em Componentes e Aspectos (DSBC/A). O método é uma adaptação do método *UML Components* (Cheesman e Daniels, 2001) com modificações e inclusões de algumas atividades para considerar aspectos no DSBC. No método de DSBC/A, os componentes são considerados como caixas-pretas e a interação entre os componentes e os aspectos é projetada com o objetivo de preservar o encapsulamento dos componentes, permitindo que os aspectos apenas operem nas interfaces dos componentes. O método proposto tem o objetivo de, a partir do documento de requisitos de um sistema, produzir uma arquitetura de componentes que contenha componentes-base (normais) e componentes transversais (aspectuais), bem como suas especificações. Os componentes transversais possuem o comportamento de um aspecto, tendo a capacidade de entrecortar outros componentes nas operações de suas interfaces e adicionar ou substituir algum comportamento.

3.5 Considerações Finais

É importante utilizar os conceitos e os mecanismos da programação orientada a aspectos para separar e encapsular possíveis interesses transversais, pois assim podem ser obtidos sistemas mais fáceis de entender, implementar, integrar, reusar e manter. Como forma de modularizar os interesses transversais existentes em arquiteturas de software e nos componentes do sistema, vários autores propuseram linguagens de descrição de arquiteturas orientadas a aspectos e métodos de desenvolvimento baseado em componentes e em aspectos. Logo, é importante em uma arquitetura baseada em componentes também considerar aspectos. Havendo uma linha de produtos de software, é interessante usar aspectos não só para tratar requisitos não-funcionais, mas também para representar as variabilidades da linha.

Ao desenvolver uma linha de produtos de software com arquitetura baseada em componentes, as características comuns e as variabilidades são projetadas por meio de componentes. Eles são adicionados ou substituídos para obter aplicações-referência. Os aspectos devem ser usados para representar requisitos não-funcionais da linha, separando interesses transversais. Além disso, é interessante usar aspectos para auxiliar a representar as variabilidades de uma linha sem quebrar o encapsulamento fornecido pela arquitetura baseada em componentes, ou seja, entrecortando apenas as operações das interfaces dos componentes. Essas variabilidades podem representar tanto requisitos funcionais quanto não-funcionais.

As arquiteturas são importantes para LPSs e uma forma de definir a linguagem de descrição de domínio que possua uma arquitetura baseada em componentes seria usando os conceitos de ADLs. Tratando-se de uma LPS, geralmente a definição de domínio é feita em termos de características, portanto, a linguagem de descrição de domínio pode usar os conceitos de ADLs para mostrar como devem ser associadas as características da LPS para obter a arquitetura baseada em componentes.

Desenvolvimento de Linhas de Produtos de Software

4.1 Considerações Iniciais

Os processos de Engenharia de Domínio e de Aplicação devem ser conduzidos para desenvolver uma LPS e produzir suas aplicações. Este capítulo foca o processo de Engenharia de Domínio para desenvolver uma LPS. Alguns princípios foram adotados para o processo de desenvolvimento da LPS proposta e tomou-se como base o método PLUS (Gomaa, 2004), que foi descrito na Seção 2.3.1 do Capítulo 2, para desenvolver a LPS. Conforme o que foi discutido nessa seção, Gomaa (2004) define um processo de desenvolvimento específico para o método PLUS, chamado ESPLEP. No contexto desta dissertação decidiu-se por adaptar esse processo de acordo com necessidades encontradas. As adaptações realizadas são apresentadas e o processo resultante é descrito.

O capítulo inicia com a definição de alguns princípios que foram adotados para o desenvolvimento da LPS na Seção 4.2. Em seguida, na Seção 4.3, são apresentadas as adaptações propostas ao processo ESPLEP. Essas adaptações foram implementadas na LPS como é detalhado na Seção 4.4. Essa seção é dividida na engenharia de domínio e de aplicação da Linha de Produtos de Software de Bilhetes Eletrônicos e Transporte municipal (LPS-BET). Por sua vez, a engenharia de domínio é dividida nos ciclos incrementais de desenvolvimento da LPS-BET, mostrando alguns dos seus artefatos. Posteriormente, na Seção 4.5 são fornecidas informações específicas referentes à construção da LPS-BET. Finalmente, na Seção 4.6 são apresentadas as considerações finais do processo de desenvolvimento da LPS-BET.

4.2 Princípios Adotados para o Desenvolvimento da LPS

Considerando o contexto do projeto de pesquisa desta dissertação, definiu-se o princípio de aplicar um processo de desenvolvimento de LPS baseado em engenharia avante, pois não havia sistemas disponíveis nesse mesmo domínio que pudessem ser usados para obter artefatos já prontos. Ao invés disso, foi necessário captar os requisitos a partir de três aplicações que foram escolhidas como referências. Outra premissa foi o uso de uma evolução pró-ativa da LPS. Inicialmente, decidiu-se por considerar três aplicações-referência para a LPS, considerando que elas cobririam a maior parte do escopo de produtos da LPS que se desejava, e por planejar o desenvolvimento da LPS com base nessas aplicações pré-definidas. Logo, não seria o caso de usar a abordagem reativa ou a extrativa. Adicionalmente, os incrementos no desenvolvimento da LPS são guiados por essas aplicações-referência. Dessa forma, a meta é desenvolver a cada ciclo de desenvolvimento um grupo de características tal que possam vir a produzir uma aplicação operacional que havia sido planejada previamente. No desenvolvimento de uma LPS, o núcleo a ser desenvolvido inicialmente não precisa necessariamente ser operacional (executável e utilizável por usuários). Neste caso optou-se por produzir um núcleo que pudesse ser executado e assim já ser possível ter desde cedo agilidade no desenvolvimento e poder verificar e validar as características básicas da LPS.

Esta decisão foi tomada porque parte do interesse de pesquisa deste trabalho é investigar a agilidade no processo de desenvolvimento de LPS. Além disso, procurou-se investigar o projeto da LPS considerando componentes caixa-preta para verificar como essa decisão influencia em outros fatores relacionados à arquitetura de componentes de LPS e no reúso com esses tipos de componentes. Para isso, optou-se por usar uma arquitetura baseada em componentes do tipo caixa-preta. Também foi considerado o uso de aspectos com componentes para avaliar a interação deles com os componentes e o uso de aspectos para representar variabilidades e interesses transversais, com o objetivo de avaliar em que situações pode ser melhor usar aspectos.

Esses princípios guiaram a pesquisa realizada no desenvolvimento da LPS-BET. Como explicado, alguns apenas por serem decisões relacionadas ao contexto em que o projeto de mestrado se encontra e outros pelo interesse de investigação de alguns fatores relacionados ao processo de desenvolver uma LPS e ao uso de certas soluções tecnológicas, como componentes caixa-preta e aspectos.

4.3 Adaptações Propostas ao Processo ESPLEP

A análise de domínios considera as funcionalidades que são comuns a todas as aplicações do domínio (núcleo da LPS), aqueles que são opcionais (existentes apenas para alguns produtos da LPS) e os alternativos (escolhidos a partir de um conjunto de possibilidades). Um modelo conceitual geral é criado representando as partes comuns e variáveis. Em seguida, um diagrama de características (*features*) pode ser desenvolvido para sintetizar essas partes de uma LPS. Uma

possibilidade é então elaborar o projeto inteiro do domínio modelado. A implementação pode ser feita posteriormente, em uma versão ou em vários incrementos. Essa solução parece ser não-econômica e complexa (Gomaa, 2004; Atkinson e Muthig, 2002).

Outra possibilidade é usar um processo mais ágil que seja iterativo e incremental, em que a LPS primeiro tem sua arquitetura de componentes projetada e é implementada em uma versão que contém apenas características básicas (do núcleo, comuns a todas as aplicações). Então essa versão é incrementada por um subgrupo de variabilidades opcionais e alternativas, como proposto por Gomaa (2004), incrementando assim a arquitetura de componentes e a implementação do núcleo. Para isso, pode-se optar por usar ou adaptar um processo de desenvolvimento iterativo e incremental baseado no PU (Process Unificado), como o ESPLEP, permitindo que o progresso seja mais visível para os usuários e haja maior retroalimentação no desenvolvimento.

Para integrar o PU ao método PLUS, Gomaa (2004) sugere que se faça a fase de elaboração em duas iterações. A primeira iteração compreende as atividades relacionadas à elaboração do núcleo. Uma segunda iteração é necessária para a evolução da linha de produtos, ou seja, para a elaboração das características e dos casos de uso opcionais e alternativos. Dessa forma, a elaboração é feita primeiro para toda a LPS antes de seguir para a construção. A construção e a transição também podem ser feitas em iterações. Entretanto, assim a LPS só fica disponível ao final de todo o desenvolvimento, não havendo entregas progressivas de possíveis aplicações da LPS.

Considera-se importante que versões da LPS com conjuntos de características sejam desenvolvidas em ciclos de desenvolvimento para ter diversas versões prontas e testadas ao longo do projeto, que possam fornecer apoio a outras versões da LPS, tornando o desenvolvimento mais ágil. Para isso, deve-se percorrer rapidamente as fases do PU, diferente do proposto na integração com o método PLUS. A fim de produzir uma nova versão, inicia-se um novo ciclo, um ciclo de evolução, com um novo conjunto de características. Cada versão corresponde a um novo incremento. Esse incremento pode ser horizontal ou vertical, cujas diferenças serão explicadas na seção a seguir. Essa adaptação não exclui a possibilidade de cada fase de um incremento possuir iterações, como em um desenvolvimento tradicional.

De forma geral, as subatividades das disciplinas/atividades são executadas como o método PLUS orienta e é usada a mesma notação do PLUS. Houve apenas a inserção de uma nova subatividade para as atividades de análise e projeto na elaboração. Nessa subatividade é feita uma antecipação da análise e do projeto de algumas características de outros incrementos. Essa subatividade é descrita na próxima seção.

Na engenharia de aplicação do processo ESPLEP, a arquitetura baseada em componentes da LPS é adaptada e projetada para que uma aplicação que seja membro da LPS possa ser derivada. Em termos da engenharia de aplicação, prefere-se usar o método FAST, exposto na Seção 2.3.1 do Capítulo 2. Weiss e Lai (1999) sugerem que já na engenharia de domínio deve-se: fazer uma linguagem (linguagem de modelagem de aplicação) para especificar membros da LPS; desenvolver um ambiente para, posteriormente, gerar esses membros; e definir um processo para produzir os membros. Essas atividades são realizadas na fase de transição dos incrementos que produzem

aplicações-referência, ao invés de apenas fazer a transição sugerida pelo processo ESPLEP. A cada incremento, uma parte da linguagem vai sendo definida, de acordo com as características e os casos de uso que fazem parte do incremento. A linguagem a ser definida consiste de uma receita (do inglês *cookbook* (Johnson, 1992) ou guia de montagem) para obter a aplicação-referência com base nos ativos centrais (*core assets*) disponíveis. Essa receita pode ser manual ou automatizada. A diferença entre as duas é detalhada na próxima seção.

4.3.1 Engenharia de Domínio

A escolha dos incrementos a serem produzidos em cada ciclo de desenvolvimento da engenharia de domínio pode ser feita de forma horizontal ou vertical, influenciando bastante o projeto da arquitetura baseada em componentes que implementa as variabilidades da LPS. Incrementos horizontais são planejados pela inclusão de um subgrupo de características que atendem a uma aplicação-referência específica, mas que não contém necessariamente todas as possíveis variabilidades de cada característica incluída no incremento. No caso dos incrementos verticais, todas as variabilidades de um subgrupo de características escolhidas são implementadas de forma geral e completa, mas não produzem necessariamente uma aplicação-referência. Essas possibilidades são mostradas esquematicamente na Figura 4.1. Para a LPS-BET, por exemplo, um incremento horizontal poderia ser um que gerasse o sistema BET para a cidade de São Carlos, enquanto um incremento vertical seria um que possuísse todas as possíveis formas de viagens de integração especificadas durante a análise de domínio. No caso, correspondem a integração por tempo, número de viagens que podem ser integradas, a existência de linhas que permitem a integração e o uso de terminais para passageiros realizarem integração sem uso de cartão.

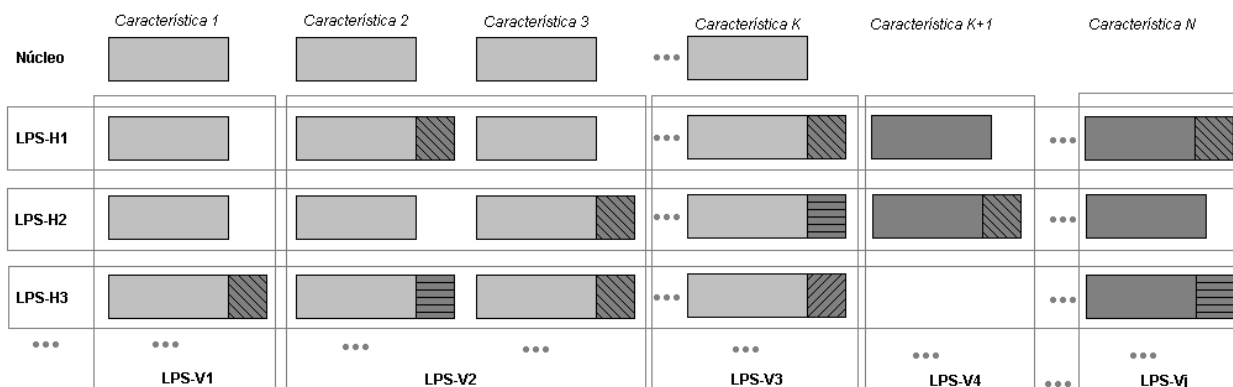


Figura 4.1: Incrementos verticais e horizontais

O comportamento das variabilidades em incrementos horizontais é mostrado na Figura 4.1 por diferentes sombreamentos de variabilidades estendendo uma característica básica contida no núcleo. A figura ilustra, por exemplo, características que não aparecem no núcleo e apenas aparecem em incrementos posteriores; características que aparecem no núcleo e que são estendidas de uma

forma para uma aplicação-referência e de outra forma para outra aplicação, etc. Com a adoção de um processo evolutivo horizontal, cada variabilidade necessita de um projeto cuidadoso, pois pode requerer refatoração em incrementos posteriores. Em outras palavras, o projeto que é adequado para um incremento pode não o ser para um incremento posterior.

Os incrementos horizontais são mais realistas economicamente, pois a LPS evolui à medida que novas aplicações-referência precisam ser incorporadas na linha de produtos, mesmo que esses incrementos possam requerer um maior retrabalho quando a linha evolui. Por outro lado, incrementos verticais – mesmo quando não produzem uma aplicação-referência após os primeiros incrementos – possuem a vantagem de permitir que cada característica escolhida seja analisada e projetada globalmente, incluindo as suas possíveis variabilidades dentro do domínio. Ao optar por incrementos horizontais, pode-se mitigar o retrabalho realizando uma análise e projeto parcial de algumas variabilidades de outras aplicações-referência. Ela deve ser feita após a análise e projeto da aplicação-referência em questão e antes de começar a implementar essa aplicação, para já refinar o projeto dessa aplicação e diminuir possível retrabalho em incrementos posteriores.

Considerando ainda o desenvolvimento usando incrementos horizontais, cada incremento produz uma aplicação-referência e, portanto, corresponde a um ciclo de desenvolvimento do PU. O ciclo inicial de desenvolvimento produz o núcleo e os outros ciclos produzem as aplicações-referência, ou seja, os ciclos/incrementos são guiados por aplicações-referência. No início da engenharia de domínio devem ser planejados os incrementos da LPS e a ordem na qual devem ser desenvolvidos. Na Figura 4.2 pode-se ver que o primeiro ciclo corresponde ao desenvolvimento dos casos de uso do núcleo da LPS. Dessa forma é produzido o núcleo operacional. Os casos de uso (UCs) que vão sendo implementados passam a fazer parte dos ativos centrais (*core assets*) da LPS. O primeiro incremento do núcleo corresponde ao desenvolvimento de casos de uso que produzam a aplicação-referência 1 e que não possam ser reusados do núcleo. Cada incremento pode reusar os casos de uso que necessitar dos incrementos anteriores, pois são todos ativos centrais da LPS. Conseqüentemente, o incremento i pode utilizar os casos de uso desenvolvidos em todos os $i - 1$ incrementos anteriores e, obrigatoriamente, reusa todos os casos de uso do núcleo. Além disso, alguns casos de uso são específicos da aplicação-referência i e, portanto, precisam ser desenvolvidos nesse incremento e passam a fazer parte dos ativos centrais a partir do término do incremento i .

Caso ainda não se saiba no início as aplicações que devem ser desenvolvidas, pode-se construir cenários de aplicações-referência da LPS. Dessa forma, os cenários se relacionam especificamente a características que eles podem ter.

A ordem dos incrementos deve ser definida de acordo com as características ou os casos de uso de cada aplicação-referência. Com base neles pode ser feita uma estimativa de esforço para o desenvolvimento de cada uma. Sugere-se que os incrementos sejam organizados com o objetivo de que o desenvolvimento seja mais ágil, produzindo resultados que irão requerer menor esforço em termos do que já se tem, reusando funcionalidades do incremento anterior. Se for considerado que cada característica requer o mesmo esforço de desenvolvimento, a prioridade seria dada para

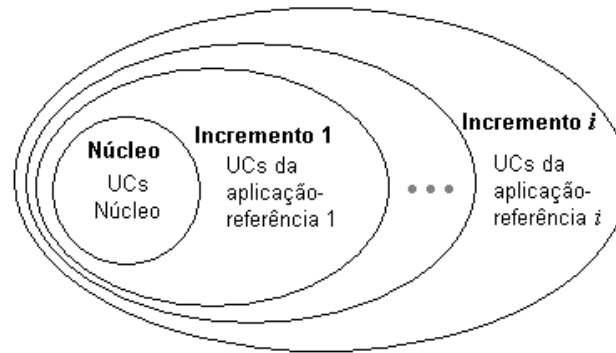


Figura 4.2: Incrementos de desenvolvimento de uma LPS

a aplicação-referência que tivesse a menor quantidade adicional de características ou casos de uso. De acordo com o projeto ou a organização, pode-se optar por outros critérios para priorização das aplicações-referência. Por exemplo, a organização pode já precisar desenvolver as aplicações-referência em uma determinada ordem ou alguma aplicação pode possuir maior urgência do que outras.

Cada ciclo iterativo é dividido em quatro fases: Concepção, Elaboração, Construção e Transição, e produz uma versão ou incremento da LPS. Isso pode ser visto na Figura 4.3. Cada fase pode envolver múltiplas iterações antes que ela seja completada. Durante uma fase, atividades são realizadas em várias disciplinas, como requisitos, análise e projeto e implementação, mas o esforço em cada uma muda ao longo das fases, assim como o esforço das fases nos diferentes incrementos.

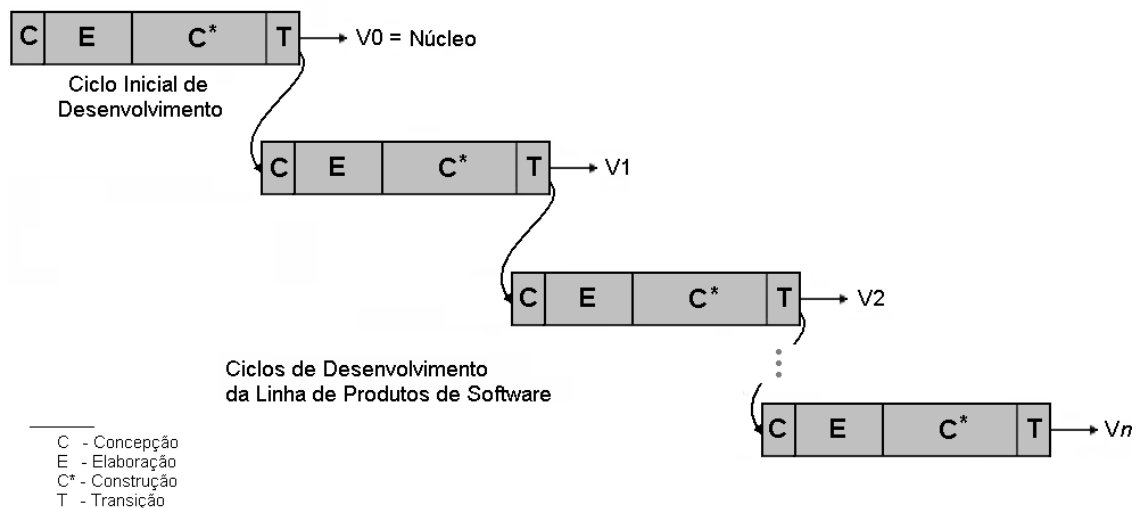


Figura 4.3: Ciclos de desenvolvimento da LPS e as suas fases

Na concepção determina-se o escopo da LPS e a partir dele faz-se uma elicitação inicial dos requisitos do ciclo de desenvolvimento, ou incremento. Assim, pode-se desenvolver uma estimativa e cronograma inicial para o desenvolvimento da LPS. A fase de elaboração foi dividida em três subatividades: (i) a especificação detalhada dos casos de uso da aplicação-referência; (ii) uma

subatividade de projeto completo da aplicação-referência; e (iii) uma subatividade de projeto parcial ágil de casos de uso de aplicações de outros incrementos, tendo uma visão do projeto adiante para algumas variabilidades que são relevantes que podem implicar em uma melhoria ou mudança no projeto da aplicação-referência sendo desenvolvida, refinando o projeto da subatividade (ii). O projeto da subatividade (ii) é então usado para a implementação da aplicação-referência na fase de construção e o projeto parcial da subatividade (iii) serve de entrada para a subatividade (ii) do incremento seguinte.

O projeto da arquitetura pode ser feito considerando componentes, como sugerido por Gomaa (2004). Caso a escolha seja a de usar componentes, ainda há a escolha de usar só componentes do tipo caixa-preta, só caixa-branca ou uma combinação dos dois tipos. Além disso, pode-se usar o desenvolvimento orientado a aspectos para auxiliar na separação de interesses transversais ou para a implementação das variabilidades (Mezini e Ostermann, 2004; Apel e Batory, 2006; Heo e Choi, 2006; Lee *et al.*, 2006; Anastasopoulos e Muthig, 2004). Pode-se ainda escolher por combinar aspectos, componentes e frameworks para desenvolver uma LPS (Griss, 2000).

Com a arquitetura de componentes definida pode-se iniciar a fase de construção, que consiste da implementação e teste dos casos de uso (UCs) analisados e projetados para a aplicação. Finalmente, a transição assegura que a aplicação-referência possa ficar disponível para os seus usuários finais no momento que for feita a engenharia de aplicação. Dessa forma, na transição elabora-se uma receita para guiar a posterior engenharia da aplicação-referência, pode-se liberar o uso da aplicação-referência para executar testes e prepara-se o ambiente para o próximo ciclo a ser iniciado. A forma como é feita a receita depende de como se planeja fazer a engenharia de aplicação¹.

No ciclo de desenvolvimento do primeiro incremento faz-se a análise e o projeto da LPS de forma geral e um projeto detalhado do núcleo da LPS. Em seguida o núcleo é desenvolvido. Especificamente, na concepção são verificadas as similaridades e as variabilidades da LPS, para que o escopo dos incrementos possa ser definido. Para isso são elaborados documentos de requisitos dos sistemas da LPS e modelado o diagrama de características para auxiliar a delimitação dos incrementos e a definição do cronograma, escopo e estimativa do esforço de desenvolvimento da LPS. Esse planejamento ainda não é feito de forma detalhada para toda a LPS, pois ainda não existem informações suficientes para tal. O detalhamento é feito na concepção dos ciclos seguintes.

Na elaboração desse incremento é realizada a engenharia de requisitos, a análise e o projeto da linha em geral e é detalhado o núcleo da LPS. Com o amadurecimento em relação aos requisitos da LPS, pode-se modelar o diagrama de casos de uso da LPS e especificar os casos de uso básicos para a LPS (similares para todas as suas aplicações-referência). Pode-se então realizar o projeto geral da LPS, definindo artefatos, como o modelo conceitual e de classes e a arquitetura geral do sistema. Especificamente para o núcleo, outros artefatos são elaborados, como diagramas de estados e de comunicação; e a arquitetura baseada em componentes é detalhada, especificando as interfaces e os contratos das operações. Como dito anteriormente, para manter um projeto consistente com a

¹O processo de engenharia de aplicação é abordado na próxima seção e os detalhes da engenharia de aplicação para a LPS-BET são mostrados no Capítulo 6.

LPS e que diminua retrabalho nos outros incrementos, é feita a análise e o projeto parcial de alguns casos de uso que aparecerão em incrementos posteriores. Essa atividade pode levar a atualizações no projeto já feito para o núcleo, que correspondem a refinamentos da arquitetura modelada. Os casos de uso são selecionados de acordo com a importância no desenvolvimento e projeto da LPS, como fatores de risco, complexidade e acoplamento com outras funcionalidades.

Após elaborar esses artefatos, a construção é iniciada focando-se apenas no núcleo. Os requisitos do núcleo que foram especificados nos casos de uso e projetados na arquitetura são implementados e testados. Essa fase leva mais tempo do que as outras fases do incremento. Como deve haver mais similaridades do que variabilidades na LPS, então o núcleo deve ser de tamanho considerável para compensar o desenvolvimento de uma LPS. Portanto, a fase de construção desse incremento provavelmente também será a de maior esforço em relação a cada fase de construção dos incrementos seguintes. O núcleo desenvolvido é operacional, ou seja, ele funciona como uma aplicação-referência. Pode coincidir do núcleo corresponder a uma aplicação da LPS, que seria a mais simples (básica), mas isso nem sempre ocorre. Mesmo assim, considera-se importante desenvolver as características básicas para que se tenha um incremento operacional. Dessa forma, verificação e validação das características básicas já podem ser feitas, antecipando possíveis erros que só seriam percebidos na integração com operações que o tornassem funcionais, sem as quais poderiam não ser perceptíveis.

Os ciclos iterativos seguintes produzem aplicações-referência. As principais diferenças do primeiro ciclo, que produz o núcleo, em comparação com os outros ciclos são em relação à fase de concepção, que é mais curta para estes, pois consiste basicamente da definição do escopo e do cronograma detalhado do ciclo; e à fase de transição, que é mais longa², pois nestes ciclos deve-se preparar o ambiente e uma receita para fazer a engenharia de aplicação posteriormente. Isso consome mais tempo do que na transição do núcleo, pois ele não possui variabilidades para serem representadas na receita para a engenharia de aplicação. Na elaboração, tem-se como entrada da análise e do projeto os requisitos elicitados anteriormente para a aplicação-referência e os artefatos produzidos nos incrementos anteriores, como a arquitetura e o projeto parcial. Com base neles é feito o projeto para os casos de uso adicionais para desenvolver a aplicação-referência e um projeto parcial de casos de uso por vir, como foi explicado anteriormente. Assim são implementados apenas os casos de uso opcionais e variantes específicos da aplicação-referência na fase de construção.

4.3.2 Engenharia de Aplicação

A engenharia de aplicação tem como propósito explorar rapidamente os requisitos de uma aplicação e produzir a aplicação. A forma como a aplicação é desenvolvida depende de como a LPS foi projetada e implementada e pode ser feita, por exemplo, usando geradores de aplicação,

²Caso o núcleo corresponda a uma aplicação-referência, esta afirmação não se aplica necessariamente, pois pode-se achar necessário realizar uma receita simples para a posterior engenharia de aplicação.

composição de componentes, instanciação de frameworks e compilação condicional (Figueiredo *et al.*, 2008). Independente da forma como será produzida a aplicação, inicialmente é feita uma análise da aplicação a ser produzida. A partir dessa análise são definidas as características da aplicação. Caso as características já tenham sido previstas na engenharia de domínio, pode-se partir para a montagem da aplicação em si. Caso alguma característica não tenha sido planejada para a LPS, a equipe deve verificar se é válido adicionar a característica na LPS ou apenas desenvolver a característica específica para a aplicação.

A montagem da aplicação pode ser feita de duas formas: manual ou automatizada. Ao fazê-la manualmente, deve-se seguir a receita desenvolvida ao longo da engenharia de domínio (especificamente nas fases de transição) para saber como montar a aplicação. A receita contém diretrizes em relação a quais artefatos fazem parte da aplicação e como e onde dispô-los. Por exemplo, assumindo que a arquitetura seja baseada em componentes, a receita informa quais componentes devem ser reusados para produzir a aplicação e de que maneira eles devem ser acoplados e eventuais necessidades de código novo de ligação (*glue code*). Um engenheiro de aplicação então seguiria essa receita passo a passo para produzir a aplicação-referência desejada.

Para fazer a montagem de forma automatizada, a receita pode ser processada automaticamente por um gerador de aplicação que reconheça o domínio da LPS. Pode-se desenvolver um gerador de aplicação próprio para o domínio ou pode-se optar por usar um gerador de aplicações configurável. Caso se decida por seguir essa opção, deve-se configurar o gerador de aplicação para o domínio específico, fornecendo as funcionalidades e as variabilidades da LPS e formas como podem ser combinadas para produzir aplicações. Essas funcionalidades podem ser representadas por características ou casos de uso, por exemplo. No âmbito de geradores de aplicação, a receita corresponde a uma linguagem de modelagem de aplicação (LMA). Ela deve ser especificada durante as fases de transição da engenharia do domínio a fim de ser processada pelo gerador no processo de engenharia de aplicação. O engenheiro de aplicação tem a atribuição de informar ao gerador as funcionalidades específicas da aplicação a ser gerada. O gerador de aplicação recebe essas informações como entrada, processa-as de acordo com as diretrizes da LMA e, como resultado, produz a aplicação-referência desejada pelo engenheiro de aplicação. Esse processo de desenvolvimento da LMA e da engenharia de aplicações é detalhado para a LPS-BET no Capítulo 6.

4.4 Processo ESPLEP Adaptado e Instanciado para a LPS-BET

A LPS desenvolvida para servir de estudo de caso consiste da gestão de bilhetes eletrônicos de transporte, chamada LPS-BET. Esses sistemas facilitam o uso de ônibus de transporte municipal, oferecendo várias funcionalidades para os passageiros e as empresas viárias, como uso de um cartão plástico para pagar o transporte, abertura automática de catracas, pagamento unificado de viagens, integração de viagens e fornecimento de informações *on-line* para os passageiros.

O software integra e automatiza a rede de transporte com um sistema centralizado que mantém os dados dos passageiros, cartões, linhas, ônibus e viagens. Ônibus são equipados com um validador que lê um cartão e se comunica com o sistema central (por exemplo por RFID - *Radio Frequency Identification*) para debitar uma viagem no cartão do passageiro. Pode também haver um sistema de integração de ônibus que permite ao usuário pagar uma única passagem para realizar várias viagens. Além disso, passageiros podem usar a *web* para consultar suas viagens e saldo.

O domínio do sistema foi analisado e a LPS-BET foi projetada com o objetivo de ser capaz de derivar ao menos três produtos (que nesse contexto são aplicações de software) baseados nos sistemas BET existentes de três cidades brasileiras: São Carlos (São Paulo), Fortaleza (Ceará) e Campo Grande (Mato Grosso do Sul). A especificação desses três sistemas foi feita para gerar inicialmente essas três aplicações – aplicações-referência. Os requisitos especificados de cada um dos sistemas foram analisados e comparados em detalhes, realizando um mapeamento entre os requisitos dos três sistemas. O mapeamento foi feito para que (a) as similaridades da LPS pudessem ser extraídas e identificadas a partir dos requisitos comuns, (b) os requisitos não existentes em todas as especificações fossem identificados e especificados como variabilidades opcionais e (c) os requisitos que possuísem variações nas especificações pudessem ter pontos de variação definidos. Dessa forma, a comparação permitiu a extração das características e a modelagem da LPS em termos de similaridades e variabilidades.

Foi considerado importante ter uma aplicação completa logo no começo do processo de desenvolvimento, optando assim por usar os ciclos de incrementos horizontais, que permitem a geração de uma dessas aplicações a cada incremento. Por esse motivo, foram planejados quatro incrementos, um produzindo o núcleo da LPS-BET e os outros uma aplicação da linha:

- *Incremento 1:* Desenvolvimento de casos de uso do núcleo da LPS.
- *Incremento 2:* Reúso do núcleo (incremento 1) e desenvolvimento de casos de uso específicos da aplicação-referência de Fortaleza.
- *Incremento 3:* Reúso do núcleo (incremento 1) e de alguns casos de uso desenvolvidos (incremento 2) e desenvolvimento de casos de uso específicos da aplicação-referência de Campo Grande.
- *Incremento 4:* Reúso do núcleo (incremento 1) e de alguns casos de uso desenvolvidos (incrementos 2 e 3) e desenvolvimento de casos de uso específicos da aplicação-referência de São Carlos.

Desses incrementos inicialmente planejados, os três primeiros foram de fato produzidos como parte deste trabalho de mestrado. Para completar o ciclo de desenvolvimento do último incremento (São Carlos) falta apenas a implementação de uma característica, que ficou como trabalho futuro. Decidiu-se por analisar o uso de aspectos para a implementação de variabilidades durante os ciclos

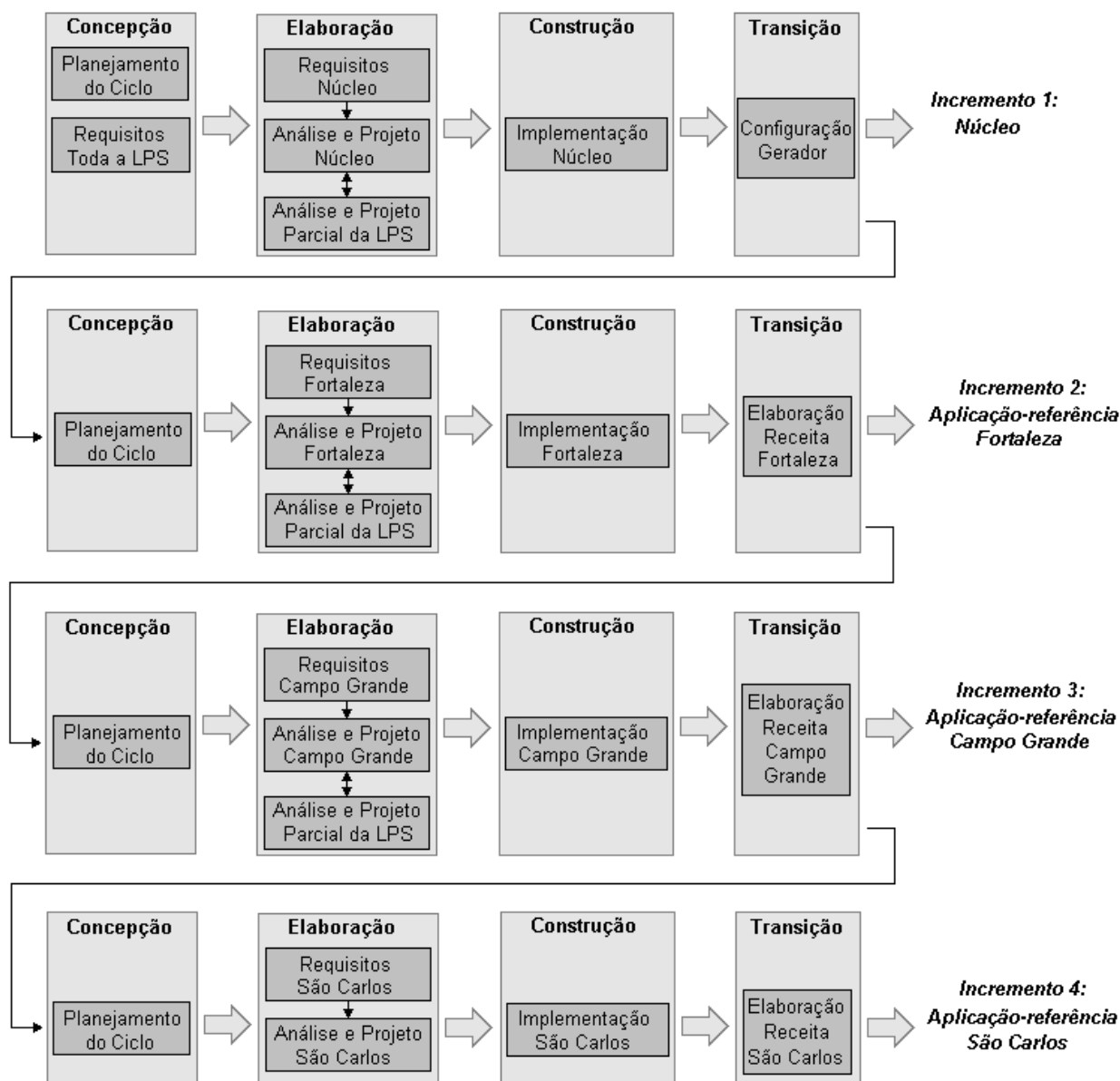


Figura 4.4: Visão geral do processo de desenvolvimento da LPS-BET

de desenvolvimento dos incrementos, por ser considerado no contexto desta dissertação como sendo tema de pesquisa a ser investigado.

A Figura 4.4 ilustra o processo de desenvolvimento da LPS-BET, evidenciando as disciplinas de requisitos, análise, projeto e implementação, assim como a elaboração da receita de composição que deve ser usada posteriormente na engenharia de aplicação. Na concepção do primeiro ciclo foi planejado de forma geral o desenvolvimento da LPS e em detalhes o desenvolvimento do núcleo e foram elicitados os requisitos das três aplicações-referência. Com base nos requisitos pôde-se modelar o diagrama de casos de uso da LPS; verificar quais casos de uso deveriam existir em cada aplicação para que seus requisitos fossem alcançados e então iniciar a elaboração pela especificação detalhada dos casos de uso do núcleo. A partir da especificação dos casos de uso,

a análise e o projeto do núcleo foram feitos. No projeto parcial do primeiro incremento foram analisados alguns casos de uso que existem para Fortaleza, Campo Grande e São Carlos. Esse projeto parcial ofereceu retroalimentação para o projeto do núcleo, para que pudesse ser refinado. Com base nos requisitos e no projeto foi implementado o núcleo da LPS, o qual foi reusado para as implementações das aplicações-referência nos outros ciclos. Na transição do núcleo foi feita a configuração inicial do gerador a ser usado na engenharia de aplicações da LPS-BET. Na Figura 4.4 não está sendo mostrada a associação entre os requisitos e a análise e o projeto de ciclos diferentes que deve ficar subentendido.

A partir do fim do primeiro ciclo de desenvolvimento pôde-se planejar melhor o desenvolvimento da LPS. Dessa forma, foi possível realizar um planejamento detalhado do desenvolvimento da primeira aplicação-referência (Fortaleza) na concepção do seu ciclo de desenvolvimento. Em seguida os requisitos e o projeto do núcleo serviram de base para poder completar a fase de elaboração de Fortaleza. O projeto parcial no ciclo de desenvolvimento de Fortaleza foi feito analisando casos de uso existentes para Campo Grande e São Carlos. Finalizando os requisitos e o projeto de Fortaleza, iniciou-se a implementação de casos de uso adicionais e, junto com os já implementados no incremento anterior, desenvolveu-se a aplicação de Fortaleza. Com base na implementação, elaborou-se a receita para Fortaleza, refinando a configuração do gerador. No incremento de Campo Grande, optou-se por não fazer o projeto parcial, pois considerou-se que não havia mais casos de uso que necessitassem de análise, embora isso pudesse ter sido feito (por esse motivo deixou-se a subatividade representada na figura). Na construção, o BET de Campo Grande foi implementado reusando alguns casos de uso e desenvolvendo novos casos de uso. Em seguida, na transição foi elaborada a receita para gerar Campo Grande. No ciclo de desenvolvimento do último incremento não foi mais necessário realizar um projeto parcial pois não há mais incrementos posteriores. Foram implementados alguns casos de uso específicos de São Carlos e reusados aqueles já desenvolvidos em ciclos anteriores, faltando apenas a implementação de uma variabilidade para obter a aplicação-referência de São Carlos.

4.4.1 Ciclo de Desenvolvimento do Núcleo

Primeiro desenvolveu-se um documento de requisitos para cada um dos sistemas. Para o levantamento de tais requisitos tomou-se como base a experiência de algumas pessoas no uso do sistema de transporte urbano das cidades sob o papel de passageiros e os sites³ disponibilizados pelas empresas de viação. Os documentos de requisitos para as três aplicações-referência podem ser vistos nos Apêndices A, B e C. A partir dos documentos de requisitos dos três sistemas, o diagrama de características foi modelado. Na Figura 4.5 pode-se ver o diagrama de características para o núcleo da LPS-BET (características comuns) usando a notação do Gomaa (2004). As características comuns da LPS-BET são descritas a seguir:

³Fortaleza: <http://www.vtefortaleza.com.br>, <http://www.sindionibus.com.br>

Campo Grande: <http://www.assetur.com.br/pagintegracao.html>

São Carlos: <http://www.saocarlos.sp.gov.br>

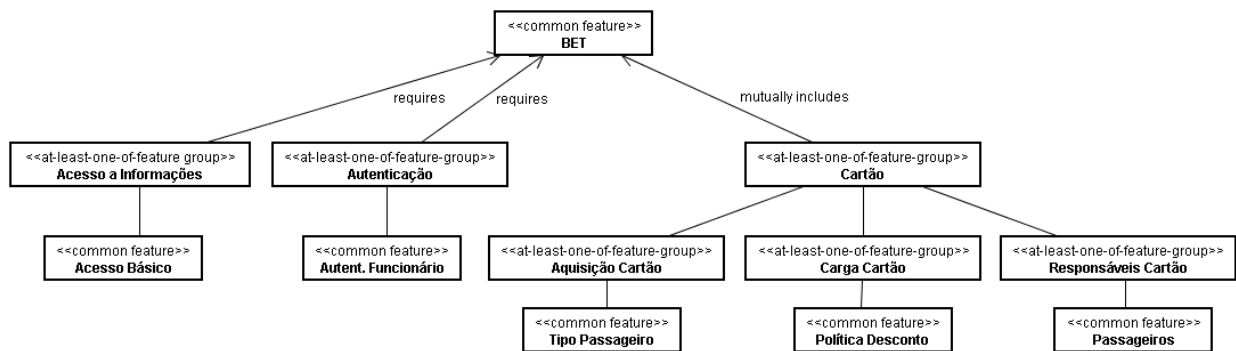


Figura 4.5: Diagrama de Características do núcleo da LPS-BET

- *Acesso Básico*: corresponde à característica de acesso de um usuário, nesse caso o passageiro, a informações básicas da parte web do sistema BET referentes à consulta de dados do passageiro e dos seus cartões, como a data de validade e o saldo do cartão.
- *Autenticação de Funcionário*: corresponde à autenticação que é necessária para os funcionários da empresa viária (atendente e manutenedor) para utilizarem a parte web do sistema BET. De acordo com o cargo do funcionário há autorização para utilizar algumas funcionalidades do sistema. O atendente pode usar apenas as funcionalidades referentes ao cartão, ao passageiro e à carga e aquisição de um cartão.
- *Tipo de Passageiro*: corresponde à existência de categorias de passageiros que possuem atributos específicos. Para adquirir um cartão, ele deve ser associado a um tipo específico de passageiro da empresa viária.
- *Política de Desconto*: corresponde a uma característica relacionada à carga dos cartões em que é fornecida uma percentagem de desconto para a compra de passagens usando um cartão. Isso ocorre de acordo com a política de desconto que está relacionada ao tipo de passageiro do cartão.
- *Passageiros*: corresponde à característica relacionada ao papel responsável pela carga de um cartão. O passageiro deve requerer a carga de um determinado valor no cartão para um atendente da empresa viária.

A Tabela 4.1 mostra as características (opcionais e alternativas) que devem ser incorporadas ao núcleo da linha para obter cada uma das três aplicações-referência.

Para a disciplina de requisitos, além dos documentos de requisitos, são feitos o diagrama de casos de uso e as especificações dos casos de uso. Um mapeamento é feito entre requisitos, características e casos de uso para poder rastreá-los posteriormente. Os casos de uso básicos são especificados no incremento 1 e os casos de uso opcionais são especificados no incremento em que forem usados primeiro. Um fragmento do diagrama de casos de uso é mostrado na Figura 4.6,

Tabela 4.1: Características opcionais ou alternativas para aplicações da LPS-BET

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

resumindo os casos de uso relacionados ao uso do ônibus por um passageiro para realizar uma viagem e pelo cobrador para registrar uma corrida, possuindo estereótipos que identificam se são básicos («kernel») ou opcionais («optional»). Essas subatividades descritas até agora correspondem a subatividades de requisitos da fase de concepção do incremento do núcleo, com exceção da especificação dos casos de uso que correspondem a uma subatividade de requisitos já da fase de elaboração.

Os casos de uso necessários para o desenvolvimento de cada aplicação-referência da LPS-BET já podem ser identificados e pode-se ter uma idéia da quantidade de casos de uso que são reusados nos incrementos e a quantidade que deve ser desenvolvida. A Tabela 4.2 mostra os casos de uso a serem desenvolvidos em cada ciclo de desenvolvimento de incrementos da LPS. O reúso só pode ser feito de casos de uso já implementados em incrementos anteriores, por esse motivo a relação de reúso das aplicações-referência mostrada na tabela se assemelha a uma matriz triangular. A última coluna totaliza os casos de uso presentes em cada aplicação-referência, considerando aqueles reusados e aqueles especificamente desenvolvidos.

Tabela 4.2: Casos de uso necessário para os incrementos da LPS-BET

Aplicação-referência	Quantidade de Casos de Uso					Total
	Desenvolvimento	Reúso do Núcleo	Reúso de Fortaleza	Reúso de Campo Grande	Reúso de São Carlos	
Núcleo	21	-	-	-	-	21
Fortaleza	3	21	-	-	-	24
Campo Grande	5	21	2	-	-	28
São Carlos	2	21	0	4	-	27

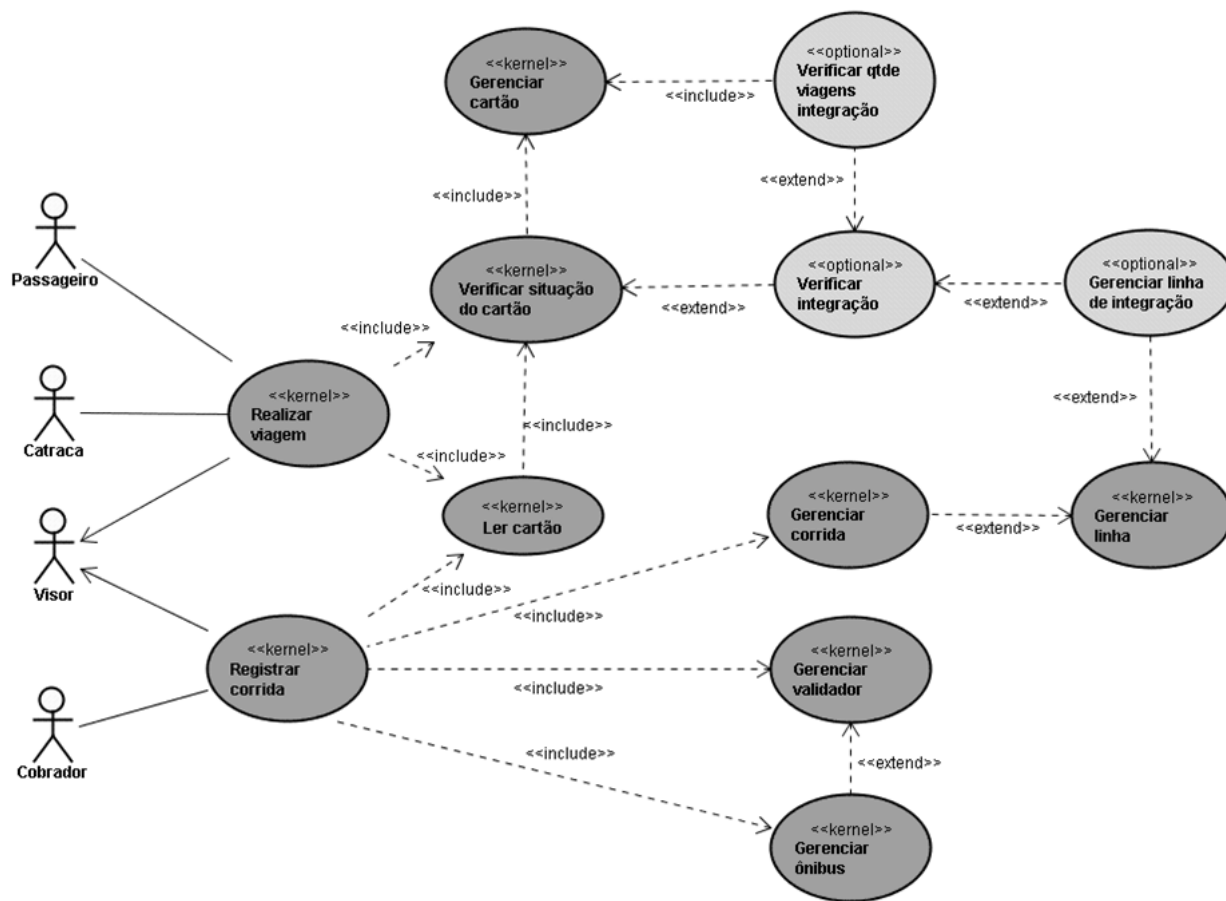


Figura 4.6: Parte do diagrama de casos de uso da LPS-BET

A Figura 4.7 mostra a arquitetura da LPS-BET, projetada na fase de Elaboração. Ela combina duas instâncias do padrão arquitetural cliente/servidor em multicamadas (Gomaa, 2004): uma arquitetura em três camadas para o cliente do ônibus e seu servidor; e uma arquitetura em quatro camadas para o sistema web de informação, ambos compartilhando a mesma camada de domínio de negócio.

Em um nível mais baixo, cada camada possui uma arquitetura interna de componentes, como é mostrado para o núcleo da LPS-BET na Figura 4.8. Há uma instância de cada módulo servidor (Servidor WEB, Servidor Ônibus, Domínio BET e Persistência) e várias ocorrências de dois módulos clientes (Cliente Ônibus e Interface de Usuário). Externos a esses módulos há um ambiente com usuários e dispositivos externos que interagem com a LPS. Internos a esses módulos estão os componentes projetados para tratar as características comuns da LPS-BET. Componentes de sistema e de negócio (representados respectivamente com os estereótipos «system component» e «business component» na figura) foram derivados usando diretrizes sugeridas por Cheesman e Daniels (2001), enquanto componentes controladores foram derivados usando sugestões de Gomaa (2004).

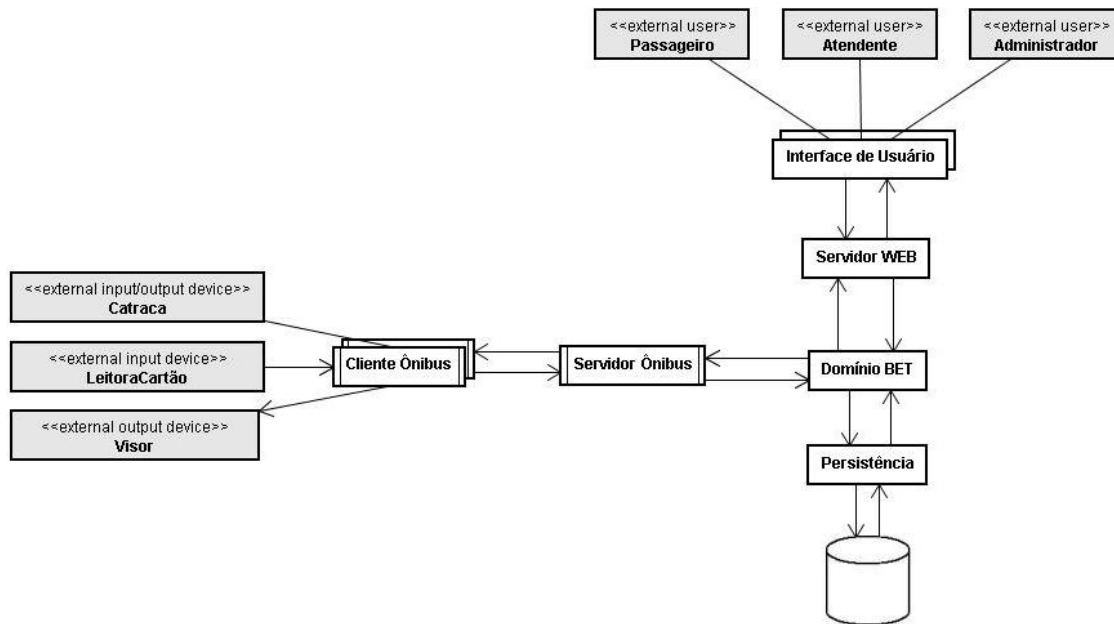


Figura 4.7: Arquitetura da LPS-BET

Componentes de sistema são identificados a partir das operações que emergem do diagrama de casos de uso e componentes de negócio são obtidos do refinamento do modelo conceitual. As operações dos componentes de sistema usam operações das interfaces de negócio (Cheesman e Daniels, 2001). Um componente controlador executa conceitualmente um *statechart* e controla uma parte do sistema e um componente coordenador gerencia diversos componentes controladores (Gomaa, 2004). Há também um aspecto de autenticação presente na arquitetura de componentes que será explicado melhor no próximo capítulo.

Parte da arquitetura de componentes do núcleo é mostrada em mais detalhes na Figura 4.9, que inclui interfaces requeridas e fornecidas, assim como componentes que as requerem e as fornecem. Os dispositivos externos e o subsistema do Cliente Ônibus presentes em um ônibus são mostrados do lado esquerdo da figura e o subsistema do Servidor Ônibus e seus componentes relacionados (partes do subsistema Domínio BET) para processar uma viagem de um passageiro ou uma corrida de um ônibus são mostrados do lado direito. Nessa figura, os dispositivos externos LeitoraCartão, Catraca e Visor são colocados como sendo parte da LPS, pois é necessário que eles sejam projetados e implementados para o funcionamento da LPS, passando a serem representados também por componentes da LPS-BET. Os usuários cobrador e passageiro (representados na Figura 4.8) interagem com a LPS por meio das interfaces fornecidas ILeitora e IGirarCatraca pelos componentes LeitoraCartão e Catraca, respectivamente.

No projeto parcial foram analisadas as características relacionadas às *Formas de Integração* e a característica de *Pagamento de Cartão*, utilizando assim os casos de uso listados na Tabela 4.3. Essas características foram escolhidas por tratarem dois tipos diferentes de projeto de variabilidades: projeto com novas classes, que possuem menor acoplamento e projeto com novas subclasses, que

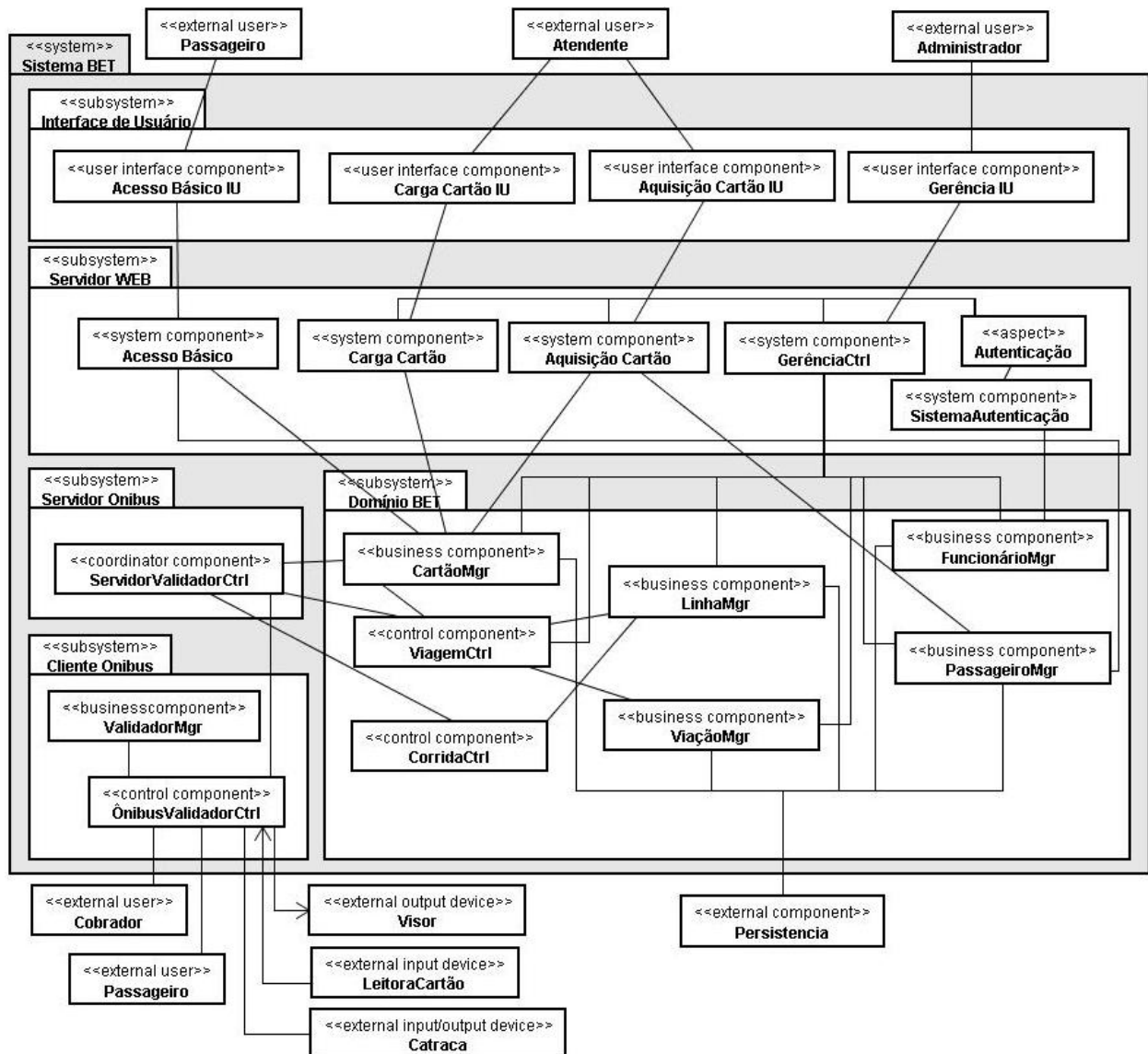


Figura 4.8: Arquitetura de componentes do núcleo da LPS-BET

possuem maior acoplamento. Isso será tratado em detalhes no próximo capítulo. Com esse projeto foi possível refinar a arquitetura de componentes do núcleo para que menos retrabalho fosse necessário posteriormente nos outros incrementos ao inserir variabilidades do grupo de características de *Forma de Integração* e da característica *Pagamento de Cartão*. A partir da arquitetura de componentes do núcleo, pôde-se fazer a implementação dos seus componentes durante a fase de construção.

Do ponto de vista técnico, foi usado o *framework* Spring (Spring, 2008) para construir a arquitetura de componentes. O Spring oferece a possibilidade de declarar quais componentes serão usados e como eles se ligam uns aos outros, por meio de interfaces. Há várias maneiras de definir essa configuração dos componentes usando Spring, mas a mais comum é por meio de arquivos XML, conhecidos como “contextos de aplicação”. Na linguagem definida por esse padrão XML, os com-

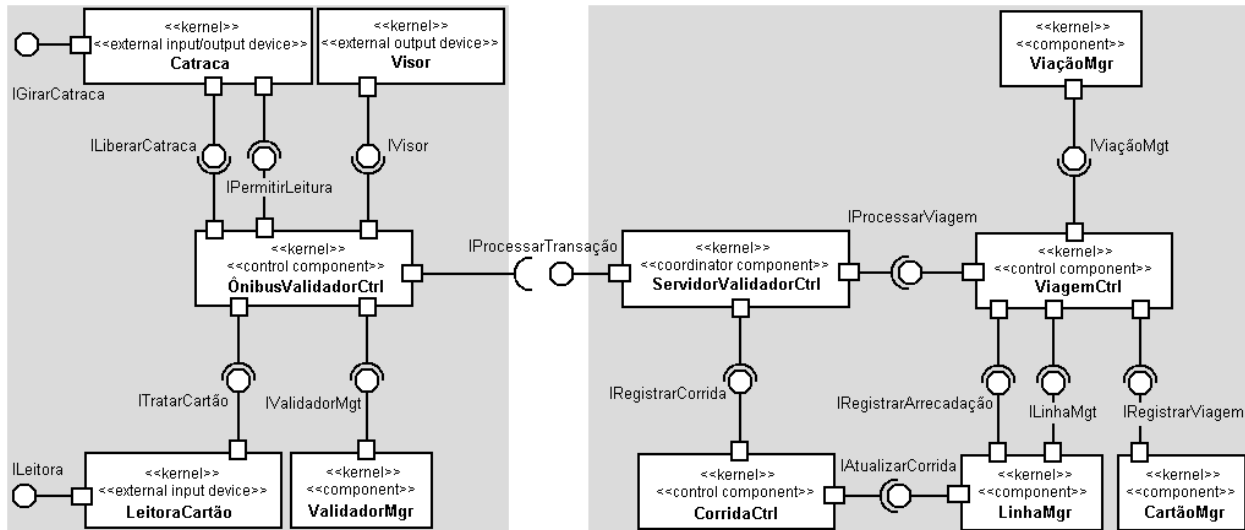


Figura 4.9: Parte dos componentes e interfaces do núcleo da LPS-BET

Tabela 4.3: Casos de uso da análise e do projeto parcial no ciclo de desenvolvimento do núcleo

Caso de Uso	Fortaleza	Campo Grande	São Carlos
Verificar pagamento do cartão	X		
Gerenciar terminal	X	X	
Gerenciar linha de integração		X	X
Verificar integração		X	X

ponentes são denominados beans. Cada bean pode ter um conjunto de propriedades (property, no XML), por meio das quais esse bean pode referenciar outros beans. Inicialmente a idéia era usar a linguagem FuseJ (apresentada na Seção 3.2.2), mas houve dificuldades para configurar o ambiente, pois o desenvolvimento da linguagem parece ter sido descontinuado. De qualquer maneira, a forma de definir a configuração de componentes usando o Spring corresponde a uma idéia semelhante à da linguagem FuseJ. A linguagem de configuração de beans do Spring corresponde a uma ADL não-hierárquica com conectores implícitos (McVeigh, 2008). É uma forma de definir a linguagem de descrição de domínio pelo modo como os componentes da LPS são interligados para associar características da LPS e obter uma arquitetura baseada em componentes.

Na Figura 4.10 pode ser vista a configuração de um dos componentes mostrados na arquitetura de componentes do núcleo. Neste exemplo o componente ViagemCtrl requer interfaces de outros quatro componentes: interfaceRegistrarViagem, interfaceRegistrarArrecadação, interfaceLinhaMgt, e interfaceViacaoMgt. Na nomenclatura do Spring, essa requisição de interfaces é conhecida pelo nome de “dependência”. O uso de interfaces é extremamente importante, já que assim o desenvolvedor pode indicar declarativamente quais implementações dessas interfaces serão usadas, sem interferir na relação de dependência entre os componentes. Essa flexibilidade é explorada na LPS-BET para trocar as implementações de modo a gerar novos produtos da linha, como será visto nos próximos capítulos.

```
1 <bean id="ViagemCtrl" class="lps.bet.basico.viagemCtrl.ViagemCtrl">
2   <property name="interfaceRegistrarViagem">
3     <ref bean="CartaoMgr"/>
4   </property>
5   <property name="interfaceRegistrarArrecadacao">
6     <ref bean="LinhaMgr"/>
7   </property>
8   <property name="interfaceLinhaMgt">
9     <ref bean="LinhaMgr"/>
10  </property>
11  <property name="interfaceViacaoMgt">
12    <ref bean="ViacaoMgr"/>
13  </property>
14 </bean>
```

Figura 4.10: Bean para injeção de dependências do componente ViagemCtrl

Ao finalizar a fase de Construção do núcleo, iniciou-se a fase de Transição e pôde-se partir para a configuração inicial do gerador Captor para o novo domínio da LPS-BET. Para isso foi necessário considerar todas as características da linha e a combinação válida dessas características. Portanto, essa configuração inicial foi feita sem possuir ainda as variabilidades implementadas e baseando-se no diagrama de características da LPS-BET. Nesse momento ainda não era possível elaborar a receita para a geração das aplicações-referência, pois para isso seria necessário ter as variabilidades desenvolvidas. Mesmo assim, já foi feito um esboço inicial dos passos considerados necessários em uma receita manual para analisar algumas atividades importantes para que a implementação das variabilidades não dificultassem a posterior elaboração das receitas da LPS. Os detalhes da configuração do Captor para a LPS-BET são apresentados no Capítulo 6.

4.4.2 Ciclo de Desenvolvimento da Aplicação-Referência de Fortaleza

Como visto na Tabela 4.1, a aplicação-referência de Fortaleza possui três características além daquelas já implementadas para o incremento do núcleo: *Terminal*, *Pagamento de Cartão* e *Empresas Usuárias*. A Tabela 4.4 mostra os casos de uso que especificam os requisitos dessas características e que precisaram ser adicionados aos casos de uso do núcleo para poder gerar a aplicação BET de Fortaleza. Na elaboração, a especificação desses casos de uso foi detalhada e assim pôde-se realizar a análise e o projeto desses casos de uso para a LPS-BET. O caso de uso de *Gerenciar Terminal* já havia sido parcialmente projetado no ciclo anterior, sendo possível reusar esse projeto parcial e refiná-lo.

Após a análise e o projeto dos três casos de uso, foi realizada a subatividade de análise e projeto parcial de casos de uso de incrementos posteriores. Os casos de uso analisados parcialmente foram *Gerenciar Linha de Integração* e *Verificar Integração*, ambos existentes tanto para Campo Grande quanto para São Carlos. Esse projeto permitiu fazer refinamentos na arquitetura de componentes de Fortaleza e, dessa forma, chegou-se ao final da fase de elaboração com a arquitetura de compo-

Tabela 4.4: Características e casos de uso opcionais da aplicação-referência de Fortaleza

Nome da Característica	Categoria da Característica	Nome do Caso de Uso	Categoria do Caso de Uso
Terminal	«optional feature»	Gerenciar terminal	«optional»
Pagamento Cartão	«optional feature»	Verificar pagamento do cartão	«optional»
Empresas Usuárias	«optional feature»	Gerenciar empresa usuária	«optional»

nentes da aplicação-referência de Fortaleza, mostrada na Figura 4.11. Os componentes em cinza são componentes que não existiam para o núcleo e que são necessários para a aplicação-referência de Fortaleza.

No domínio BET para Fortaleza são necessários mais três componentes de negócio para implementar as novas variabilidades: *EmpresaUsuaríaMgr*, *PagamentoCartãoMgr* e *TerminalMgr*. Esses componentes, assim como todos os outros componentes de negócio, são persistidos, ficando ligados ao componente de Persistência. O componente de negócio *CartãoMgr* precisa de um controlador adicional por causa do *Pagamento de Cartão*, representado no componente *CartãoPgtoCartãoCtrl*. Além disso, alguns componentes de sistema precisam ser alterados e, conseqüentemente, suas interfaces de usuário para contemplar as novas variabilidades. Os seus nomes representam as variabilidades adicionais que eles implementam. O componente de sistema de gerência é responsável por permitir a manipulação (criação, alteração, remoção e busca) das diversas entidades do domínio pelo administrador. Por esse motivo, cada nova funcionalidade inserida na LPS requer alteração nesse componente e na sua respectiva interface para o administrador. Assim, o componente de gerência para Fortaleza é chamado de *GerênciaPgtoCartao-EmprUsuáriaTerminalCtrl* e requer também ligação com os novos componentes de negócio (*PagamentoCartãoMgr*, *EmpresaUsuaríaMgr* e *TerminalMgr*). Também pode ser percebida na figura a necessidade de um usuário adicional no sistema, a *Empresa Usuária*, que pode fazer acesso à interface web do sistema BET.

No próximo capítulo é apresentada uma explicação detalhada sobre como foram projetados os componentes para implementação das variabilidades. Com base nessa arquitetura de componentes e na especificação dos componentes e das suas interfaces foi iniciada a fase de construção dos componentes de variabilidades (em cinza na figura), reusando os componentes básicos implementados no núcleo.

Após finalizar a construção da aplicação-referência de Fortaleza, iniciou-se a fase de transição. Nessa fase foi elaborada a receita para poder gerar a aplicação-referência de Fortaleza na engenharia de aplicação.

Inicialmente, decidiu-se por fazer uma receita a ser usada manualmente. Ela possuía diretrizes para juntar cada uma das variabilidades de Fortaleza (Terminal, Pagamento de Cartão e Empresas Usuárias) ao núcleo com o objetivo de obter uma aplicação executável. Essa receita manual não foi elaborada em detalhes, pois o objetivo era especificar os requisitos para elaborar o gabarito que seria usado pelo gerador e para verificar se seria possível obter as aplicações-referência das duas

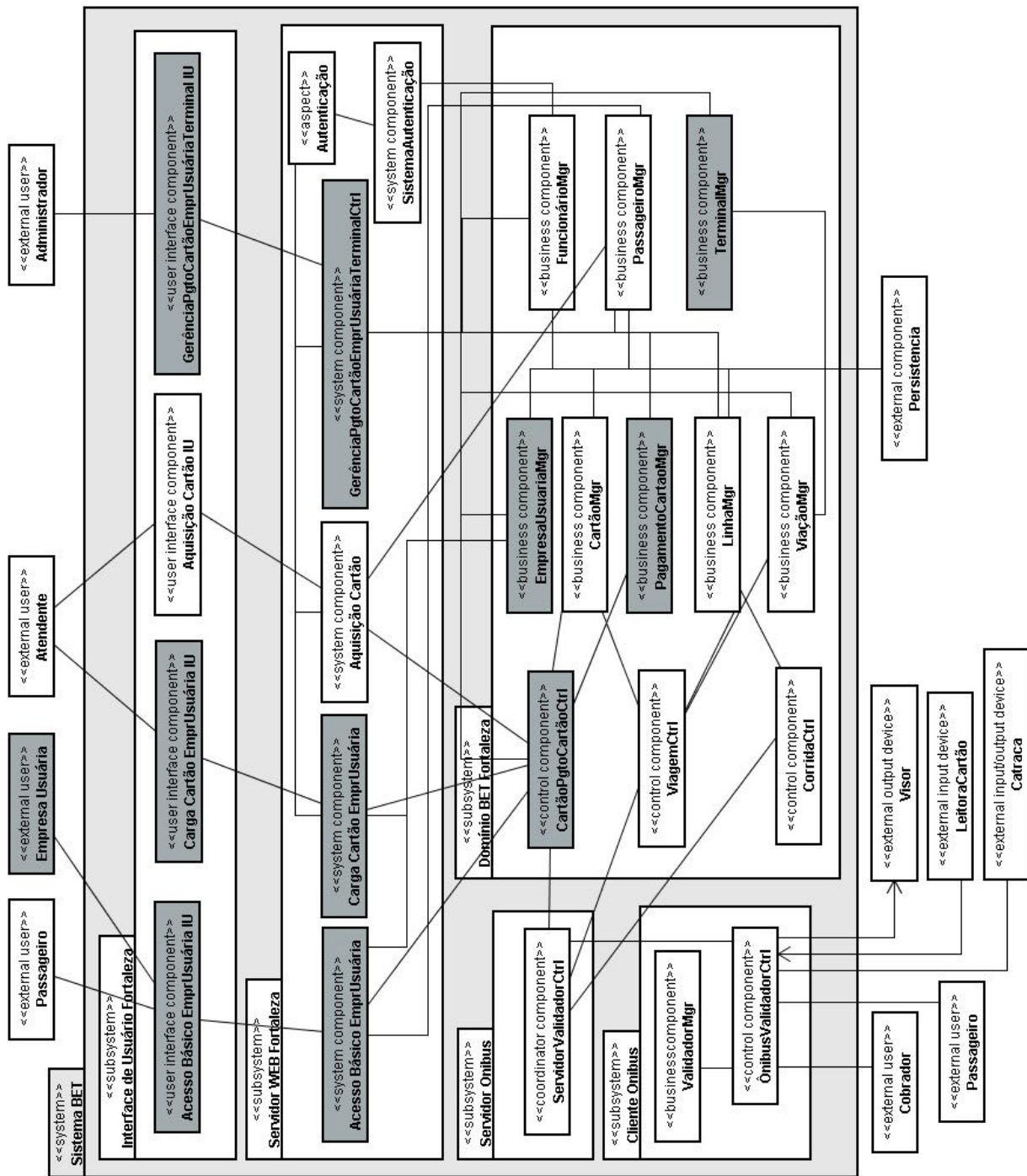


Figura 4.11: Arquitetura de Componentes para a aplicação-referência de Fortaleza

maneiras. A receita manual foi aplicada para obter a aplicação-referência de Fortaleza e ser usada como base para a solução usando o gerador. Isso facilitou o posterior desenvolvimento do gabarito para obter a aplicação-referência de Fortaleza de forma automatizada.

A receita automatizada foi implementada e pôde ser usada no Captor para gerar aplicações executáveis usando combinações válidas das variabilidades implementadas para Fortaleza. No

Capítulo 6 são explicadas em detalhes as atividades relacionadas à elaboração das receitas para geração das aplicações.

4.4.3 Ciclo de Desenvolvimento da Aplicação-Referência de Campo Grande

Além das características que fazem parte do núcleo da LPS-BET, a aplicação-referência de Campo Grande possui sete características, como é mostrado na Tabela 4.5. Dessas características, duas existem também para a aplicação-referência de Fortaleza (*Terminal e Empresas Usuárias*). Como o incremento de Fortaleza já foi gerado, elas puderam ser reusadas. O reúso foi feito tanto das especificações dos casos de uso relacionados a essas características, quanto dos componentes que estão relacionados às características. Portanto, apenas cinco características necessitaram ser desenvolvidas durante o ciclo de desenvolvimento da aplicação-referência de Campo Grande e, conseqüentemente, seis casos de uso. Para isso, o planejamento na fase de concepção do ciclo envolveu apenas esses seis casos de uso.

Tabela 4.5: Características e casos de uso opcionais da aplicação-referência de Campo Grande

Nome da Característica	Categoria da Característica	Nome do Caso de Uso	Categoria do Caso de Uso
Acesso Adicional	«optional feature»	Consultar viagens, Imprimir extrato	«optional»
Tempo	«optional feature»	Verificar integração	«optional»
Linha de Integração	«optional feature»	Gerenciar linha de integração	«optional»
Número Viagens Integração	«optional feature»	Verificar quantidade de viagens de integração	«optional»
Terminal	«optional feature»	Gerenciar terminal	«optional»
Número Cartões	«default feature»	Adquirir cartão	«kernel»
Empresas Usuárias	«optional feature»	Gerenciar empresa usuária	«optional»

No projeto parcial do ciclo de desenvolvimento de Fortaleza haviam sido analisados os casos de uso *Gerenciar Linha de Integração* e *Verificar Integração*, assim pôde-se partir dos artefatos resultantes dessa subatividade para realizar a análise e o projeto total desses casos de uso. Os outros casos de uso passaram por seu processo normal de especificação de casos de uso, seguida da análise e do projeto da arquitetura de componentes da aplicação-referência de Campo Grande, vista na Figura 4.12. Ao chegar à subatividade de projeto parcial de características dos incrementos posteriores, foram enumeradas as características de São Carlos e percebeu-se, com base na experiência de desenvolvimento dos ciclos anteriores, que essas características não possuíam funcionalidades muito diferentes das já feitas e, portanto, optou-se por não fazer um projeto parcial nesse ciclo.

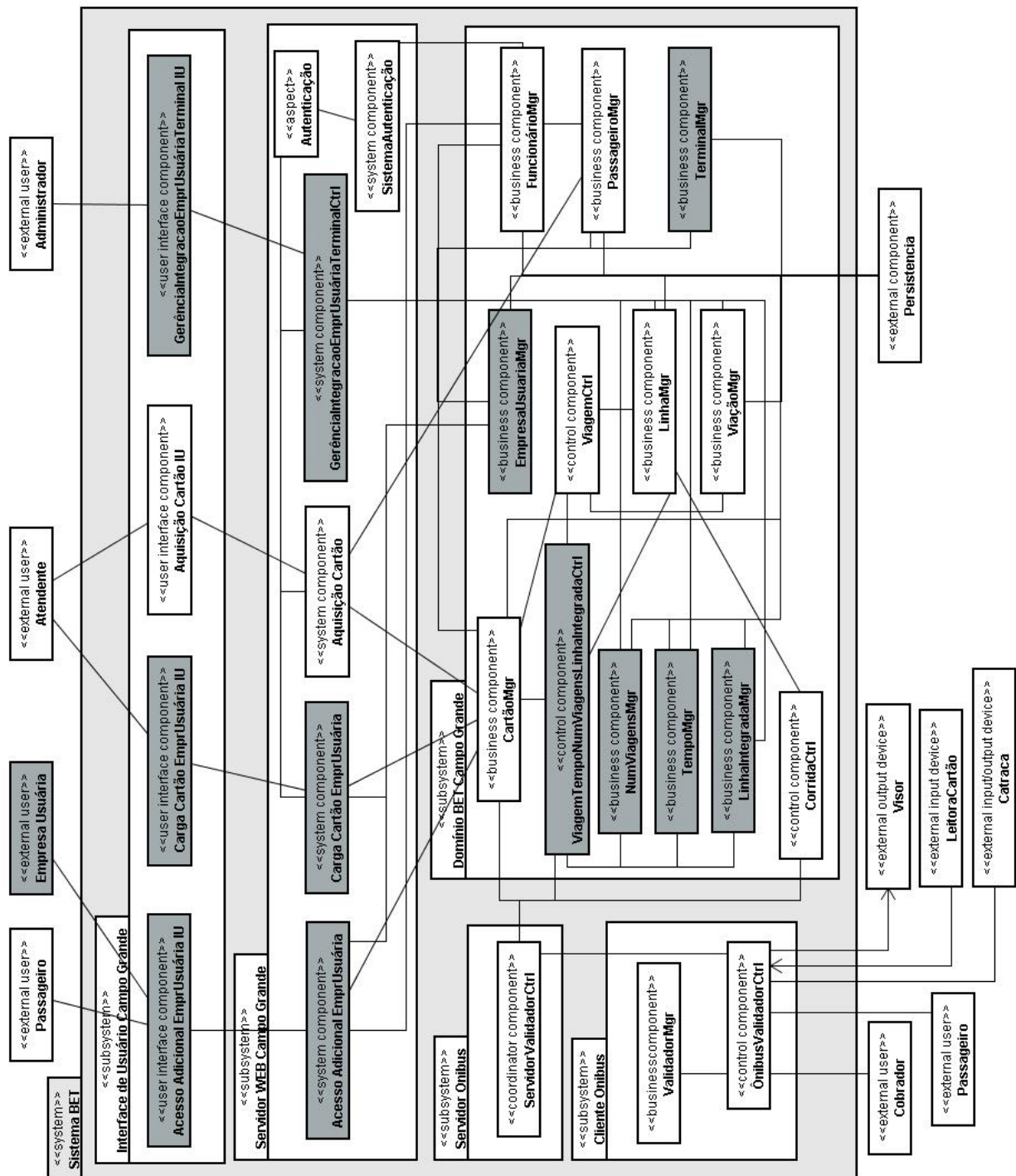


Figura 4.12: Arquitetura de Componentes para a aplicação-referência de Campo Grande

Para implementar o subsistema de domínio do BET para Campo Grande puderam ser reusados os componentes de negócio *EmpresaUsuaiaMgr* e *TerminalMgr* do incremento de Fortaleza e foram necessários mais três componentes de negócio relacionados a características de *Integração*: *NumViagensMgr*, *TempoMgr* e *LinhaIntegradaMgr*. O controle da viagem passa a necessitar de funcionalidades para realizar a integração por tempo, com um número máximo de

viagens de integração e considerando um conjunto possível de linhas integradas. Para realizar esse controle é necessário um novo componente controlador, chamado `ViagemTempoNumViagens-LinhaIntegradaCtrl`. Ele passa a ser chamado pelo `ServidorValidadorCtrl`.

Em termos do servidor web para a aplicação-referência de Campo Grande, foram necessárias duas mudanças em comparação com Fortaleza: uma referente ao componente de acesso ao sistema, que passa a ter um acesso adicional, sendo chamado então de `AcessoAdicional EmprUsuaría`; e outra associada à gerência, que deve tratar as novas entidades das características de integração para Campo Grande (tempo, número de viagens e linha integrada), representado pelo componente `GerênciaIntegraçãoEmprUsuaríaTerminalCtrl`.

A fase de construção da aplicação-referência de Campo Grande tomou como base essa arquitetura de componentes, sendo implementados os componentes necessários e reusando aqueles que fazem parte de Campo Grande e já foram desenvolvidos nos incrementos anteriores.

Na fase de transição foi elaborada a receita para poder gerar a aplicação-referência de Campo Grande com as suas variabilidades, refinando a linguagem de modelagem de aplicação da LPS-BET no Captor que foi desenvolvida na fase de transição do ciclo de desenvolvimento de Fortaleza. Iniciou-se tomando como base a receita manual feita anteriormente, porém, percebeu-se que, como algumas características de Fortaleza não existem em Campo Grande, era necessário não só fornecer diretrizes para adicionar as novas variabilidades, mas também diretrizes para remover as variabilidades de Fortaleza que não estariam em Campo Grande. Outra solução seria sempre partir do núcleo e as diretrizes seriam sempre para adicionar novas variabilidades. Nesse caso, a desvantagem ocorre quando as diferenças entre as aplicações-referência é pequena, pois a cada aplicação inicia-se do núcleo. Conseqüentemente, se houver maior diferença entre as aplicações-referência do que entre elas e o núcleo é melhor realizar receitas manuais partindo de aplicações anteriores e inserindo e removendo variabilidades, senão é melhor partir do núcleo para cada aplicação-referência.

Como a receita manual é apenas um esboço com a finalidade de elaborar a receita automatizada, mas que requer certo esforço de qualquer maneira, optou-se nessa fase por não continuar fazendo a receita manual para todas as variabilidades e partir logo para incrementar o gabarito a ser usado pelo Captor, incorporando as variabilidades de Campo Grande. As variabilidades de integração, de número de cartões e de acesso adicional passaram a fazer parte também da LMA do domínio BET no Captor.

4.4.4 Ciclo de Desenvolvimento da Aplicação-Referência de São Carlos

A aplicação-referência de São Carlos possui seis características além das características que fazem parte do núcleo da LPS-BET. Uma é referente a um requisito não-funcional (*Autenticação de Passageiro*) e as outras a requisitos funcionais. A Tabela 4.6 mostra essas características e os casos

de uso que as implementam. Alguns casos de uso necessitaram ter sua implementação alterada para tratar variabilidades, os quais estão representados pelo estereótipo «kernel» na tabela.

Tabela 4.6: Características e casos de uso opcionais da aplicação-referência de Campo Grande

Nome da Característica	Categoria da Característica	Nome do Caso de Uso	Categoria do Caso de Uso
Acesso Adicional	«optional feature»	Consultar viagens, Imprimir extrato	«optional»
Autenticação Passageiro	«optional feature»	Autenticar usuário	«kernel»
Tempo	«optional feature»	Verificar integração	«optional»
Linha de Integração	«optional feature»	Gerenciar linha de integração	«optional»
Combinação Cartões	«alternative feature»	Adquirir cartão, Gerenciar Tipo de Passageiro	«kernel»
Limite de Passagens	«optional feature»	Gerenciar limite de passagens, Verificar limite de passagens	«optional»

Três variabilidades já existiam para Campo Grande, portanto, houve reúso dessas variabilidades no ciclo de desenvolvimento. As outras três variabilidades necessitaram ser analisadas e projetadas. A arquitetura resultante dessas atividades pode ser vista na Figura 4.13.

Para implementar o subsistema de domínio do BET para São Carlos puderam ser reusados os componentes de negócio *TempoMgr* e *LinhaIntegradaMgr* do incremento de Campo Grande e foram planejados mais dois componentes de negócio: *LimPassagensMgr* e *CombinacaoMgr*. O controle da viagem passa a necessitar de um subconjunto de funcionalidades de integração implementadas para Campo Grande, a integração por tempo e com um conjunto possível de linhas integradas. Para realizar esse controle é preciso um novo componente controlador, chamado *ViagemTempoLinhaIntegradaCtrl*. Ele passa a ser chamado pelo *ServidorValidadorCtrl*. Para implementar esse controlador foi necessário pouco esforço, já que em Campo Grande a integração é mais abrangente, podendo haver reúso interno de código. Porém, não foi possível reusar todo o componente controlador. Considerando o servidor web para a aplicação-referência de São Carlos, foram necessárias diversas mudanças para incorporar as novas variabilidades. Apenas o componente *Carga Cartão* pôde ser reusado completamente.

A fase de construção desse ciclo tomou como base essa arquitetura de componentes. Das três variabilidades que necessitavam ser desenvolvidas, duas foram implementadas (*Autenticação de Passageiro* e *Acesso Adicional*) e uma será implementada futuramente pelo grupo de pesquisa (*Combinação de Cartões*). Portanto, faltou a implementação dos componentes *CombinacaoMgr*, *GerenciaIntegracaoCombinacaoLimPassagensCtrl* e *AquisicaoCartãoCombinação*.

Mesmo não tendo sido realizada completamente a fase de construção, a linguagem de modelagem de aplicação foi refinada para poder gerar a aplicação-referência de São Carlos com as

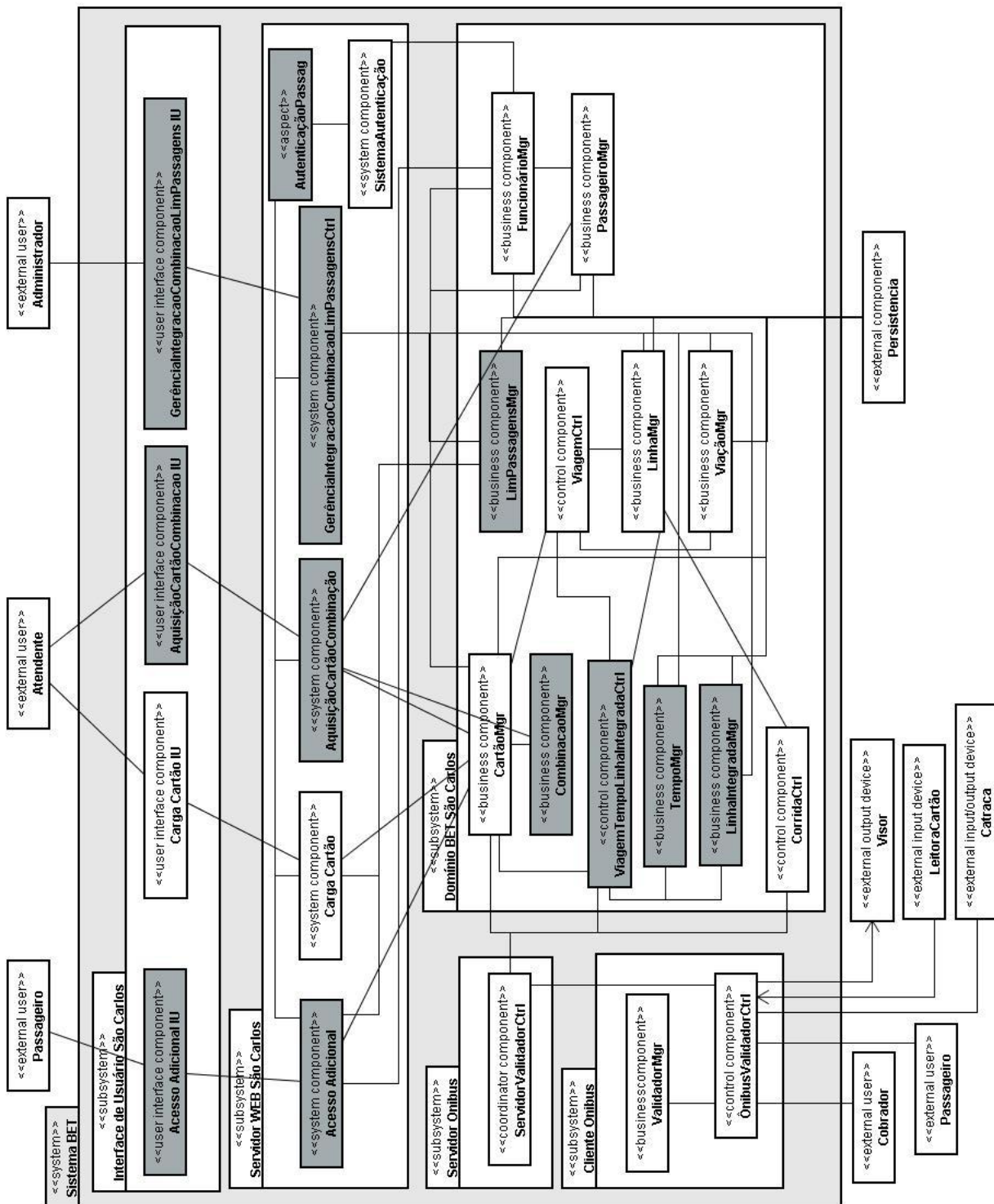


Figura 4.13: Arquitetura de Componentes para a aplicação-referência de São Carlos

variabilidades desenvolvidas. Nessa fase, a receita foi elaborada apenas para ser usada de forma automatizada pelo Captor. Portanto, o gabarito foi incrementado e passou a considerar as variabilidades implementadas em São Carlos, além de algumas combinações válidas que elas podem ter com outras variabilidades.

4.5 Informações sobre a Construção da LPS-BET

A LPS-BET foi modelada seguindo a notação UML na ferramenta Jude (Jude, 2008). Os artefatos desenvolvidos ao longo do processo de desenvolvimento foram mantidos sob o controle de versão do TortoiseSVN (Tigris, 2008) e permaneceram hospedados em um repositório do Google⁴ (Google, 2008).

A implementação da LPS foi feita usando a linguagem de programação orientada a objetos Java (seguindo as diretrizes do modelo JavaBeans) e a linguagem de programação orientada a aspectos AspectJ (AspectJ Team, 2008) no ambiente Eclipse (Eclipse, 2008). O banco de dados adotado foi o PostgreSQL (PostgreSQL, 2008), porém, a aplicação pode executar sobre qualquer outra base de dados relacional. Para realizar o mapeamento objeto-relacional e poder persistir os dados na base foi usado o framework Hibernate (Hibernate, 2008). Além disso, o Spring (Spring, 2008) foi utilizado para fazer inversão de controle (especificamente a injeção de dependência) (Fowler, 2004), usar o MVC (*Model-View-Controller*) para a parte web e facilitar o uso do Hibernate utilizando-se de sua integração com o framework. O Velocity (Apache, 2008b) foi aplicado como mecanismo para geração de páginas, pois permite um maior desacoplamento entre a apresentação e a lógica de negócios, podendo reforçar mais a separação entre ambos. Como ferramenta de *build* foi utilizado o Maven 2 (Apache, 2008a), por ser declarativo e permitir uma maior abstração. O Maven já possui um conjunto de convenções pré-estabelecidas que facilitam as atividades de *build*.

Durante a construção e a transição dos ciclos de desenvolvimento da LPS foram realizados testes do domínio BET usando a estratégia de amostras de aplicações (*Sample Application Strategy*) (Pohl *et al.*, 2005). Essa estratégia consiste da validação na engenharia de domínio de algumas aplicações que sejam sistemas representativos da LPS-BET para validar os ativos centrais do domínio. Com isso, algumas variabilidades da LPS já são testadas, porém, não elimina a necessidade de realizar testes na engenharia de aplicação, pois nem todas as combinações de aplicações possíveis são testadas na engenharia de domínio.

Os testes consistiram basicamente de testes funcionais manuais, sendo feita a verificação e a validação das funcionalidades do núcleo operacional e das aplicações-referência, assim como de algumas outras aplicações representativas usando combinações de variabilidades. Apenas foram elaborados testes unitários para as classes com maior dificuldade de entendimento no nível de implementação, nesse caso usando o JUnit (JUnit, 2008). Os problemas encontrados foram sendo registrados no site do repositório para serem corrigidos.

Com essa infra-estrutura fez-se a engenharia de domínio da LPS-BET. A partir disso, pôde-se fazer a engenharia das aplicações. Optou-se por usar um gerador de aplicação configurável (GAC), pois assim, o gerador pôde ser configurado para o domínio BET. O gerador usado foi desenvolvido pelo próprio grupo de pesquisa – o Captor (Shimabukuro, 2006) – que usa a linguagem de trans-

⁴<http://code.google.com/p/bet>

formação XSLT (do inglês *eXtensible Stylesheet Language Transformations*) (W3C, 2008) para a implementação da linguagem de modelagem de aplicação.

Em termos de tempo, a engenharia de domínio levou aproximadamente um ano e dois meses. As fases de concepção e de elaboração do primeiro ciclo duraram em torno de quatro meses e a fase de implementação do núcleo cinco meses. Os outros ciclos de desenvolvimento duraram no total cinco meses, incluindo todas as suas fases. A curva de aprendizado das tecnologias e a grande quantidade de casos de uso no núcleo (31 casos de uso para a LPS, sendo 21 do núcleo) explica o motivo da implementação do núcleo superar o tempo para o desenvolvimento de cada incremento da LPS-BET. A organização da implementação da LPS-BET pode ser vista na Figura 4.14, distinguindo a parte básica (à esquerda na Figura) e as variabilidades (à direita na Figura).

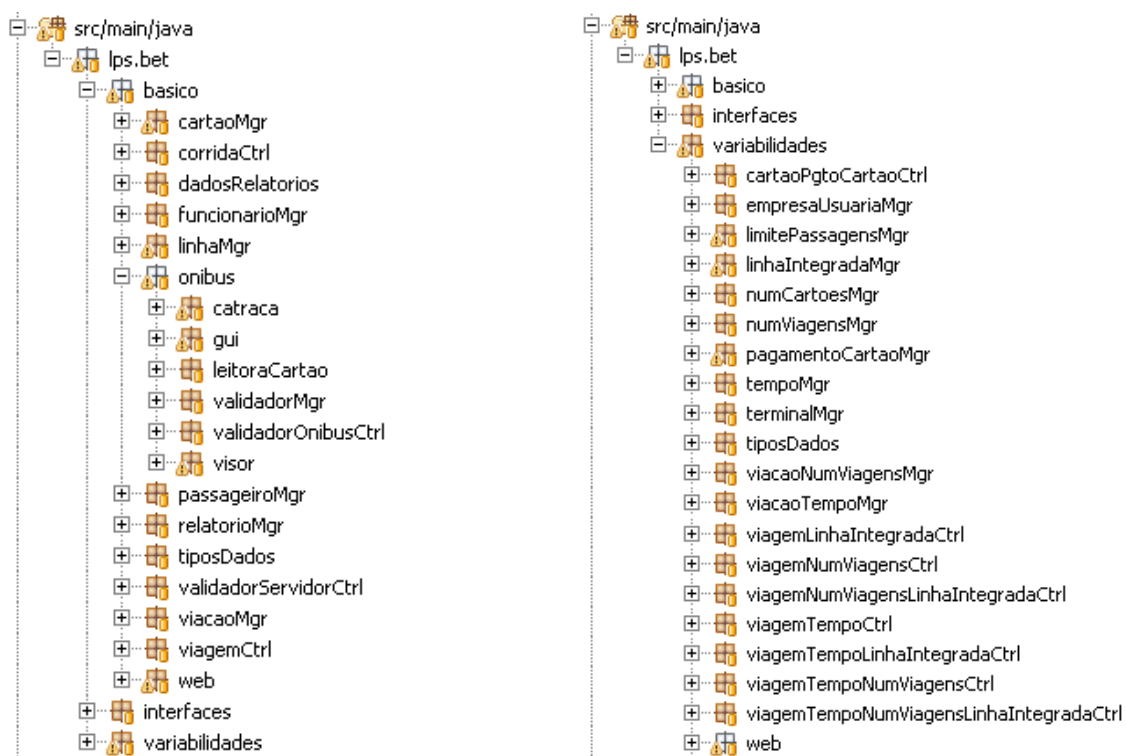


Figura 4.14: Organização da implementação da LPS-BET

A Tabela 4.7 mostra algumas métricas gerais da LPS-BET, usando o plugin Metrics (Sauer, 2008) no Eclipse, como a quantidade de atributos, métodos, classes, aspectos e componentes que foram desenvolvidos. A métrica de número de Linhas de Código (*Lines of Code* - LOC) não considera comentários e linhas em branco e também foram excluídas da contagem as linhas referentes a classes de teste. Outra métrica relacionada a contagem de linhas de código é a de Linhas de Código dos Métodos (*Method Lines of Code* - MLOC), que considera apenas as linhas de código dentro do corpo dos métodos. Os aspectos quantificados consistem de dois aspectos usados para o núcleo da LPS e os outros cinco aspectos correspondem a alternativas de projeto e de implementação de variabilidades de cinco componentes, podendo ser usados como substitutos desses componentes.

Tabela 4.7: Métricas gerais da LPS-BET

Métrica	Total
Número de Classes	126
Número de Atributos	356
Número de Métodos	1319
Número de Interfaces	36
Número de Componentes	61
Número de Aspectos	7
Número de Linhas de Código (LOC)	9.441
Linhas de Código dos Métodos (MLOC)	4.724

Na Tabela 4.8 foi feita uma comparação de algumas métricas para o núcleo da LPS-BET e as variabilidades implementadas para as aplicações-referência em termos do valor das métricas e dos percentuais em relação ao total da LPS. As entidades representam os conceitos implementados e que são persistidos pelo mapeamento objeto-relacional na base de dados.

Tabela 4.8: Métricas relacionadas ao núcleo e às variabilidades da LPS-BET

Métrica	Núcleo	Variabilidades	TOTAL
Quantidade de Componentes	27 (44,26%)	34 (55,74%)	61
Número de Linhas de Código (LOC)	4.856 (51,44%)	4.585 (48,56%)	9.441
Entidades	15 (60%)	10 (40%)	25

Calcularam-se as médias de classes, atributos, métodos e linhas de código (LOC) por componente na Tabela 4.9. Foi feita uma diferenciação na média para os componentes do núcleo e para os componentes de variabilidades da LPS, além de mostrar a média considerando toda a LPS. Percebe-se que os componentes do núcleo têm em média um tamanho maior do que os componentes das variabilidades.

Tabela 4.9: Médias de classes, atributos, métodos e LOC por componente para o núcleo, as variabilidades e a LPS-BET

Média por componente	Núcleo	Variabilidades	LPS
Classes por componente	2,63	1,62	2,07
Atributos por componente	7,07	4,85	5,84
Métodos por componente	26,70	17,59	21,62
LOC por componente	179,85	134,85	154,77

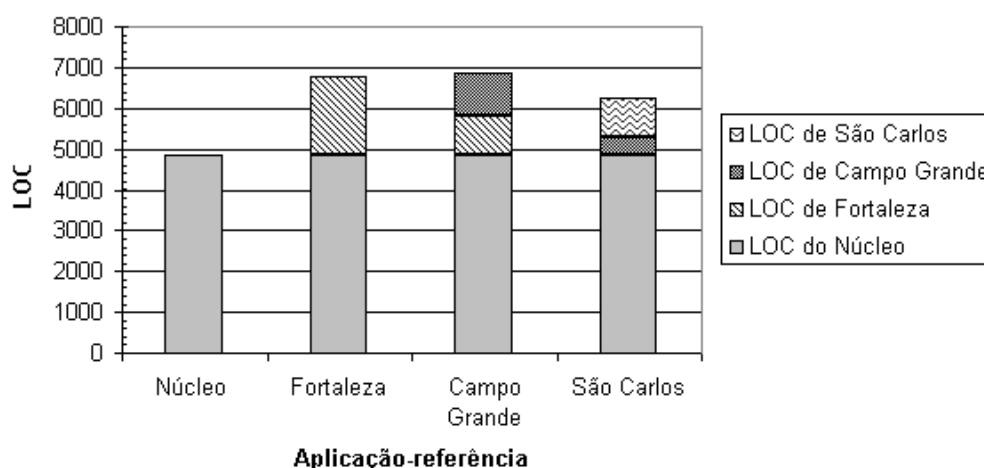
Para se ter uma idéia de percentagem de reúso do núcleo nas aplicações-referência da LPS-BET, foi feita a Tabela 4.10. Essa tabela considera as linhas de código (LOC) que são reusadas do núcleo a cada incremento e a quantidade de linhas de código (LOC) para obter as características adicionais (variabilidades da LPS). Essas duas partes correspondem à implementação de cada aplicação-referência. Na tabela, o maior grau de reúso é obtido em São Carlos. Nesse cálculo não é considerado o reúso de variabilidades nas aplicações-referência, que é mostrado a seguir na seção.

Tabela 4.10: Percentagem de Reúso do Núcleo da LPS-BET

Aplicação	Núcleo	Variabilidades	% de Reúso do Núcleo	% de Variabilidades
LPS	4856	4585	-	-
Fortaleza	“	1936	71,50	28,50
Campo Grande	“	2023	70,59	29,41
São Carlos	“	1406	77,55	22,45

Vale ressaltar que essa percentagem não inclui a implementação de uma variabilidade de São Carlos (*Combinação de Cartões*), que ao ser implementada diminuirá o valor da percentagem de reúso e aumentará o percentual de desenvolvimento. Outra observação é que existiu reúso de código de Fortaleza em Campo Grande e de Campo Grande em São Carlos, porém, esse valor foi deixado de fora nessa figura para mostrar a percentagem de reúso sem importar a ordem de implementação das aplicações-referência.

Na Figura 4.15 é considerado o reúso das variabilidades entre as aplicações-referência, além do reúso do núcleo. Primeiro foi desenvolvido o núcleo com 4.856 LOC, que foi reusado em todas as aplicações-referência. Para a aplicação-referência de Fortaleza, foram necessárias implementar 1.936 LOC de variabilidades. Em Campo Grande, aproximadamente 50% dessas linhas desenvolvidas em Fortaleza puderam ser reusadas. Adicionalmente, foram implementadas 1.054 LOC no ciclo de Campo Grande. No ciclo de desenvolvimento de São Carlos, foram reusadas 423 LOC das desenvolvidas em Campo Grande, consistindo um reúso de aproximadamente 40% de variabilidades implementadas em Campo Grande. Não houve reúso em São Carlos de variabilidades de Fortaleza. Para esse caso é importante a ordem dos incrementos produzidos, pois influencia no que é reusado em cada aplicação-referência.

**Figura 4.15:** Linhas de Código (LOC) resultantes dos ciclos de desenvolvimento da LPS-BET

Para se ter uma idéia da percentagem de reúso em cada aplicação-referência é mostrada a Figura 4.16. A figura mostra o reúso e o desenvolvimento em termos de percentagem de código para a aplicação-referência em si. Por Fortaleza ter sido desenvolvida primeiro, a aplicação-referência

é formada por 71,50% de reúso do núcleo e o resto foi implementado no seu ciclo de desenvolvimento. A aplicação-referência de Campo Grande é formada por 70,57% do núcleo, 14,09% de reúso de Fortaleza e 15,32% por código desenvolvido em seu ciclo. Portanto, Campo Grande teve 84,66% de reúso no total. No momento, a aplicação-referência de São Carlos é formada por 77,55% de reúso do núcleo, 6,76% de reúso de Campo Grande (totalizando 84,31% de reúso) e 15,70% de código implementado especificamente para São Carlos.

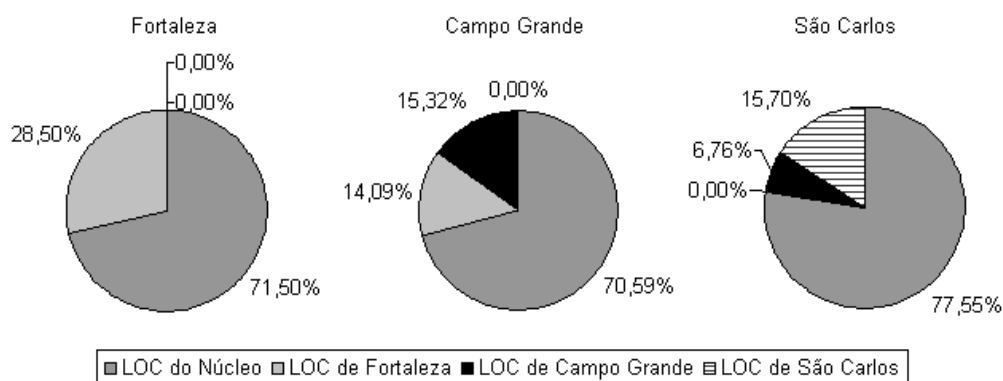


Figura 4.16: Percentagem de reúso e desenvolvimento das aplicações-referência da LPS-BET

4.6 Considerações Finais

Neste capítulo foi apresentado o processo de desenvolvimento da LPS-BET. O processo ES-PLP foi adaptado de acordo com princípios adotados para investigações a serem realizadas no decorrer do desenvolvimento da LPS-BET. Optou-se por desenvolver em ciclos horizontais que fornecessem aplicações-referência da linha, assim o desenvolvimento foi feito em 4 ciclos incrementais, produzindo as aplicações-referência do núcleo, de Fortaleza, de Campo Grande e São Carlos. Outra adaptação no processo foi a inserção de uma subatividade para as atividades de análise e projeto que permitiu antecipar mudanças relacionadas a outras variabilidades para minimizar retrabalho e durante o desenvolvimento percebeu-se que isso auxiliou no processo. Adicionalmente, as atividades relacionadas à engenharia de aplicação do processo ESPLP foram substituídas pelo FAST. Dessa forma, ainda na fase de transição de cada ciclo é feito um incremento da linguagem de modelagem de aplicação. Com isso, foi definido um processo de desenvolvimento de LPS baseado em aplicações-referência e com uma arquitetura baseada em componentes e utilizando geradores de aplicação para obter os produtos da linha.

A linha de produtos desenvolvida corresponde a uma junção de sistema de tempo real e um sistema de informação e teve todas as suas atividades documentadas para que pudesse ser usada para apoiar estudos posteriores. A LPS-BET já foi utilizada como exemplo em disciplina da pós-graduação para ilustrar a aplicação de conceitos relacionados à engenharia de software, assim

como foi usada em um curso de difusão sobre LPS. Um exemplo de LPS para uso em estudos é o Arcade Game Maker (SEI, 2008), mas essa LPS não possui a implementação disponível.

Os requisitos da LPS-BET foram elicitados considerando sistemas de bilhetes de transporte municipal que possuem apenas um meio de transporte. Portanto, sistemas que possuem integração de metrô e ônibus, por exemplo, não estão contemplados no escopo da linha. Para incluir essas funcionalidades seria necessário analisar o domínio para verificar se valeria a pena adicionar as funcionalidades na mesma linha.

Em termos de implementação, inicialmente houveram dificuldades relacionadas ao conhecimento nas tecnologias a serem usadas para implementar a linha, entretanto, as escolhas, como o uso do Spring, acabaram por facilitar o desenvolvimento e a adição de variabilidades na LPS. Foi possível alcançar um alto grau de reúso de características comuns (em torno de 70%) e houve também alto grau de reúso entre variabilidades das aplicações-referência (entre 6,76 e 14,09%), havendo um desenvolvimento específico médio de 15,5% para as aplicações-referência que puderam ter reúso de variabilidades (Campo Grande e São Carlos) e essa percentagem pôde auxiliar em reúsos de ciclos seguintes.

A granularidade de componentes depende do método de DSBC utilizado. Como foi usado o *UML Components*, os componentes ficaram de acordo com a granularidade sugerida pelo método. Usando essa granularidade pequena com componentes do tipo caixa-preta percebeu-se que é preciso fazer bastante requisições de operações de interfaces de outros componentes. Se a granularidade fosse maior a maioria das requisições ficariam mais internas e entre os componentes diminuiria, diminuindo o acoplamento. Em compensação, percebeu-se que a granularidade pequena facilita a adição e alteração de variabilidades.

Decisões de Projeto da LPS-BET

5.1 Considerações Iniciais

No capítulo anterior foi apresentado o processo adotado para o desenvolvimento da LPS-BET. Neste capítulo serão mostrados e discutidos detalhes do projeto da LPS para variabilidades da LPS, ou seja, características existentes para alguma aplicação-referência e que não estavam presentes no núcleo. Usando uma solução com componentes, deve-se verificar como devem ser inseridas essas novas características: ou pela composição de componentes, ou pelo uso de aspectos para auxiliar a interligar os componentes. É importante analisar as diferentes formas de projeto possíveis e avaliar cada solução para minimizar futuro retrabalho e facilitar manutenção. Além disso, são discutidas as diferentes soluções usando componentes caixa-preta e/ou caixa branca e as suas vantagens e desvantagens.

O capítulo é organizado da seguinte forma: na Seção 5.2 são apresentadas decisões de projeto de uma arquitetura baseada em componentes, comparando soluções com componentes caixa-preta e caixa-branca. Em seguida na Seção 5.3 são apresentadas as decisões relacionadas ao projeto da LPS-BET considerando o uso de aspectos para implementação tanto de requisitos não-funcionais, quanto de variabilidades. Na Seção 5.4 são expostas as considerações finais em relação às diferentes decisões de projeto tomadas no projeto da LPS-BET.

5.2 Projeto Baseado em Componentes

Algumas características variantes da LPS-BET mostradas na Tabela 4.1 são discutidas nesta seção para ilustrar como decisões de projeto são influenciadas pelas decisões tomadas em relação

ao processo adotado para o desenvolvimento da LPS e ao tipo de componente (caixa-preta ou caixa-branca). Duas características (*Terminal* e *Linha Integrada*) são modeladas com novas classes e duas outras características (*Tempo* e *Número de Viagens de Integração*) são modeladas com subclasses (com novos atributos e métodos). Por razões de simplicidade, os modelos de classes estão mostrando apenas atributos.

5.2.1 Modelagem com novas classes

Inicialmente é considerada a característica opcional *Terminal*, existente apenas para as cidades de Fortaleza e Campo Grande, considerada assim na fase de elaboração do incremento 2 da LPS-BET. A Figura 5.1 mostra parte do diagrama de características relacionadas a essa característica. O sistema BET de Fortaleza possui apenas terminais como forma de integração de viagens de passageiros. Terminais funcionam como pontos especiais onde passageiros podem trocar de ônibus sem pagar uma nova passagem. Outras formas mais sofisticadas de integração ocorrem nos sistemas BET das outras duas cidades, correspondendo a outras variabilidades do grupo de característica *Forma de Integração*.

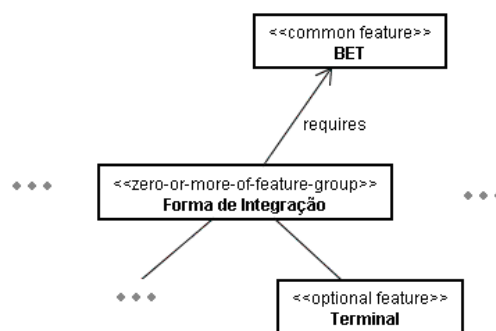


Figura 5.1: Parte do diagrama de características relacionadas a *Terminal*

A Figura 5.2 mostra o modelo de classes usado para implementar as operações relacionadas às linhas da empresa viária. As classes *Linha*, *Corrida*, *MeioDePassagem* e *Ônibus* (representados com o estereótipo «kernel») são encapsuladas em um componente básico chamado *LinhaMgr*, representado anteriormente na Figura 4.8. O projeto da característica *Terminal* requer a inclusão da classe *Terminal* no modelo (representado pelo estereótipo «optional»). De forma geral, a adição de novas características ao projeto da LPS implica em adicionar e/ou modificar classes, operações e atributos. Similarmente, componentes podem necessitar de adições tal que variabilidades sejam refletidas na arquitetura dos componentes. Há muitas maneiras de tratar essas mudanças, cada uma com vantagens e desvantagens que refletem nas decisões tomadas para o projeto da LPS.

Uma forma de tratar a necessidade de inclusão de operações e atributos dentro de classes existentes e a inclusão de novas classes é adicionar as classes diretamente dentro dos seus componentes e adicionar as operações de acordo com as novas necessidades. Para o exemplo dado, deveria haver dois componentes, o componente básico *LinhaMgr* e o componente alternativo, que seria

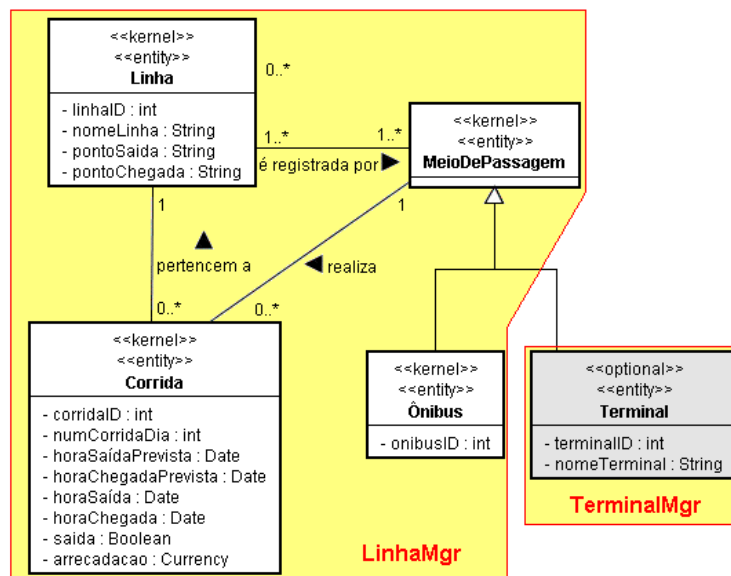


Figura 5.2: Parte do modelo de classes relacionado à característica *Terminal*

chamado *LinhaTerminalMgr*. Eles seriam usados respectivamente para a aplicação básica e para as aplicações-referência de Fortaleza e Campo Grande.

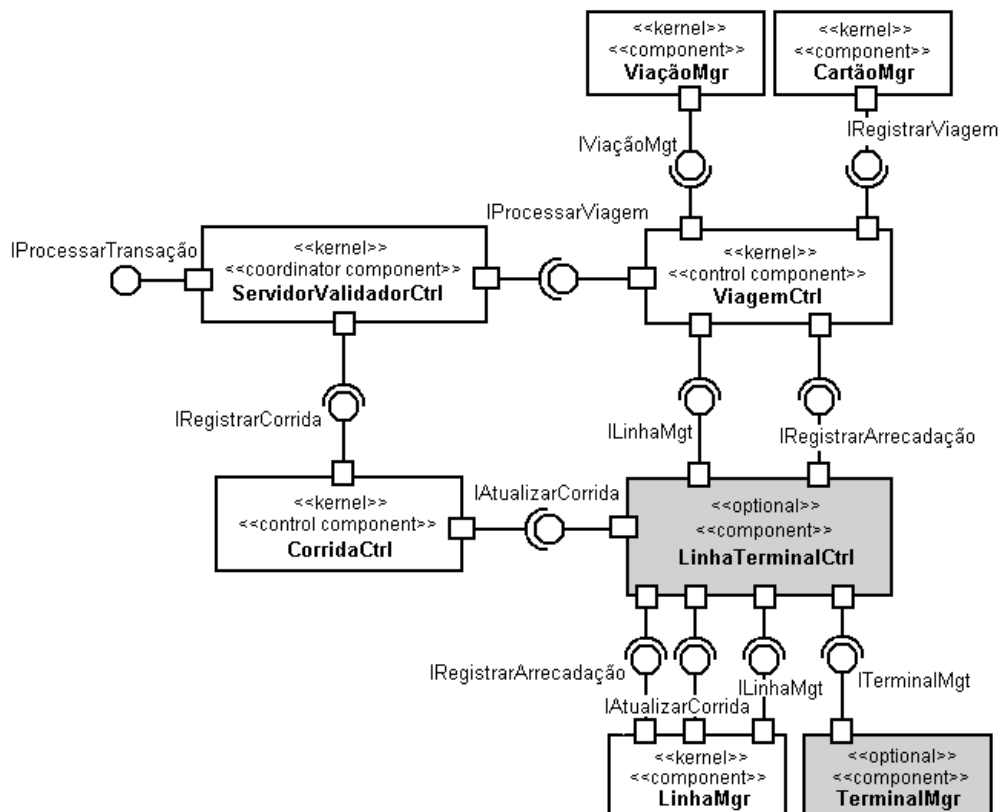
A vantagem dessa solução é a sua facilidade de implementação e composição. Existem, entretanto, várias desvantagens. Pode haver futuros problemas com manutenibilidade, porque a solução tende a duplicar código e, assim, futuras modificações podem requerer atualizações nos dois componentes alternativos. Adicionalmente, o componente original básico *LinhaMgr* deve ser uma caixa-branca, porque sua adaptação requer conhecimento e acesso a sua estrutura interna.

Para incluir classes, operações e atributos sem ter acesso interno à implementação de componentes desenvolvidos previamente – correspondendo aos ativos centrais – é necessário projetar caixas pretas. Preferiu-se usar esse tipo de componente na LPS-BET para que operações e atributos fiquem separados em novas classes, criando novos componentes específicos de uma determinada variabilidade, ao invés de deixar as operações e os atributos serem adicionados a classes existentes. Esses componentes podem ser ligados com componentes controladores (representados pelo sufixo “*Ctrl*” no nome do componente) projetados para satisfazer os novos requisitos e controlar o uso de variabilidades com características básicas. Esse tipo de componente é mais específico do que aqueles componentes de controle definidos por Goma (2004) e mostrados na arquitetura de componentes da Figura 4.9. O componente controlador requer as interfaces do novo componente e do componente básico, além de possuir lógica adicional para que a nova variabilidade seja adicionada reusando completamente o componente básico (sem mudar o componente). A desvantagem dessa solução é a necessidade de uma maior interação entre componentes e maior quantidade de acessos à base de dados, o que diminui a eficiência. A Tabela 5.1 resume essas vantagens e desvantagens do uso de cada tipo de componente.

Tabela 5.1: Vantagens e desvantagens no uso de componentes do tipo caixa-branca ou caixa-preta

Tipo de Componente	Vantagens	Desvantagens
Caixa-branca	Facilidade de implementação Facilidade de composição	Duplicação de código Dificuldade de manutenção
Caixa-preta	Maior separação de interesses Facilidade de manutenção	Maior interação entre componentes Pior desempenho

Conseqüentemente, como mostrado na Figura 5.2, ao invés de incluir a classe *Terminal* dentro do componente *LinhaMgr*, um novo componente de negócio é criado para a classe com seus atributos e operações, chamado *TerminalMgr*, e o componente *LinhaMgr* é reusado sem qualquer alteração. O uso desses componentes é gerenciado por outro componente, um componente controlador (*LinhaTerminalCtrl*). As interfaces do componente *LinhaMgr* não são alteradas (*IRegistrarArrecadacao*, *IAtualizarCorrida* e *ILinhaMgt*) e os componentes que as requerem também não necessitam de modificação. Esses componentes são o *ViagemCtrl* que requer as interfaces *ILinhaMgt* e *IRegistrarArrecadacao* e o *CorridaCtrl* que requer a interface *IAtualizarCorrida*.

**Figura 5.3:** Arquitetura de componentes parcial incluindo a característica *Terminal*

Parte da arquitetura de componentes incluindo a característica *Terminal* pode ser vista na Figura 5.3. Os componentes *ViagemCtrl* e *CorridaCtrl* que estavam anteriormente associados diretamente ao componente *LinhaMgr* (Figura 4.9) continuam a requerer as mesmas interfaces,

agora fornecidas pelo `LinhaTerminalCtrl`. O controlador pode então requerer operações da interface `ITerminalMgt` ou das interfaces do componente `LinhaMgr`. A interface `ILinhaMgr` é requerida pelos componentes `ViagemCtrl` e `GerênciaCtrl` (mostrados anteriormente na Figura 4.8) e as interfaces `IAtualizarCorrida` e `IRegistrarArrecadacao` são requeridas respectivamente pelos componentes `CorridaCtrl` e `ViagemCtrl`.

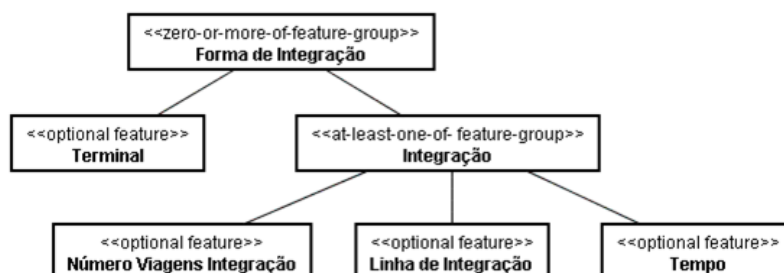


Figura 5.4: Parte do diagrama de características relacionadas à característica *Formas de Integração*

Além da característica *Terminal*, o grupo de características *Formas de Integração* também contém o grupo de características *Integração* (Tabela 4.1). *Integração* consiste no uso de linhas integradas pré-definidas e/ou no uso de ônibus dentro de um intervalo de tempo. Pode também haver um número máximo de viagens de integração permitidas dentro de um intervalo de tempo. Essas características (*Linha Integrada*, *Tempo* e *Número de Viagens de Integração*) podem ser vistas na Figura 5.4.

Agora será abordada a característica *Linha Integrada* das aplicações-referência de São Carlos e Campo Grande. Essa característica foi analisada e projetada parcialmente na fase de Elaboração do ciclo de desenvolvimento de Fortaleza e depois a característica teve seu projeto detalhado na fase de Elaboração do ciclo de Campo Grande. Por fim, no ciclo de São Carlos, o projeto e a sua implementação puderam ser reusados.

No modelo de classes a característica *Linha Integrada* é representada por uma classe opcional chamada `LinhaIntegrada` e é associada com a classe `Linha`, que podem ser vistas na Figura 5.5. A classe opcional `LinhaIntegrada` é encapsulada em um novo componente chamado `LinhaIntegradaMgr`. Como no exemplo anterior, um componente controlador é criado. Ele é mostrado na Figura 5.6.

Esse componente controlador tem uma interface nova `ILinhaIntegradaMgt` além daquelas de componentes correspondentes de versões anteriores. Essa interface é requerida por um controlador de viagem (representado pelo componente `ViagemCtrl`) para validar a integração de uma viagem de acordo com as características selecionadas do grupo de características de *Integração*. No caso da característica *Linha Integrada*, o controlador de viagem precisa validar se a linha corresponde a uma linha integrada. Conseqüentemente, o componente `ViagemCtrl` também precisa ser alterado para validar a informação fornecida de acordo com as regras de negócio da característica *Linha*

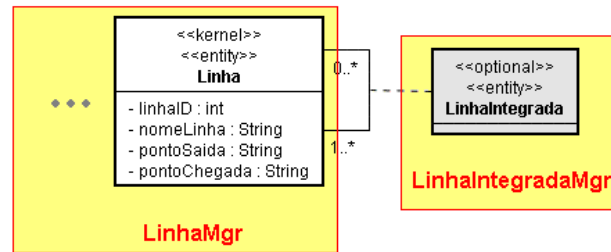


Figura 5.5: A característica *LinhaIntegrada* no modelo de classes

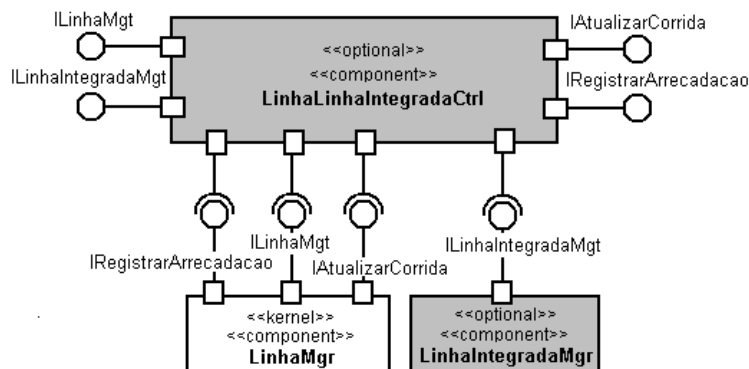


Figura 5.6: Componentes para implementar a característica *Linha Integrada*

Integrada. Possíveis soluções são: substituir o componente *ViagemCtrl* por outro; usar composição, projetando um novo componente; ou separar o interesse adicional em um aspecto (Kiczales *et al.*, 1997; Suvée *et al.*, 2006). Essas soluções serão analisadas em detalhes no decorrer do capítulo.

5.2.2 Modelagem com novas subclasses

As outras duas características do grupo de características de *Integração* são *Tempo* (existente em Campo Grande e São Carlos) e *Número de Viagens de Integração* (existente apenas em Campo Grande). A Figura 5.7 apresenta as classes *EmpresaViaria*, *SistemaViarioUrbano* e *Tarifa* que fazem parte da LPS-BET e são encapsuladas no componente *ViacaoMgr*. As características *Tempo* e *Número de Viagens de Integração* implicam em pontos de variação na classe *SistemaViarioUrbano*, alterando atributos e operações dessa classe que podem ser colocados em subclasses (representadas pelo estereótipo «variant» na Figura 5.7), contrastando com os exemplos anteriores, em que era necessário inserir uma nova classe no modelo.

Uma opção neste caso é usar classes parametrizadas (Gomaa, 2004) como mostrado na Figura 5.8, mas essa opção não foi adotada para manter os interesses separados e evitar colocar características diferentes na mesma classe e consequentemente no mesmo componente. Outra opção seria a de usar subclasses variantes como extensões da classe *SistemaViarioUrbano* (uma classe básica com pontos de variação abstratos). Isso implicaria em uma solução usando com-

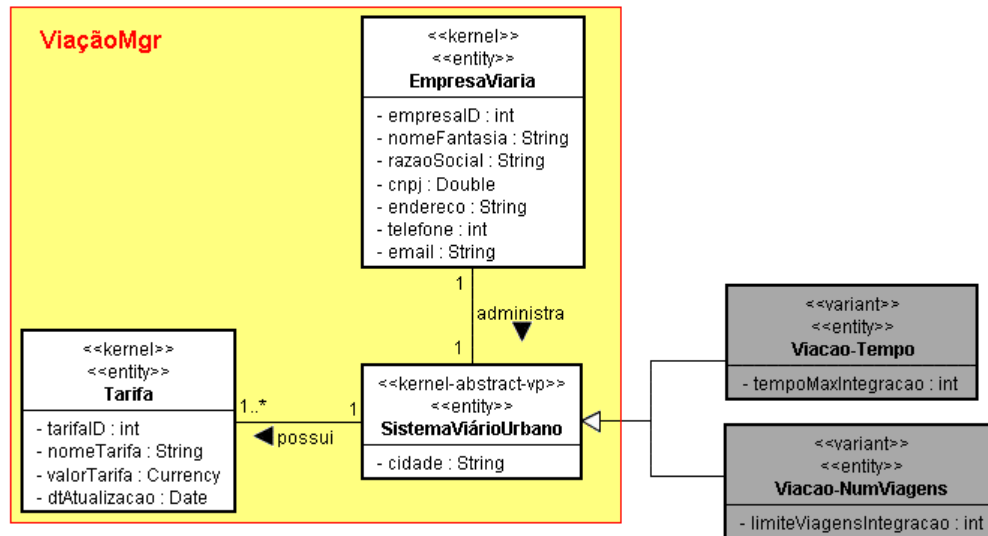


Figura 5.7: As características *Tempo* e *Número de Viagens de Integração* no modelo de classes

ponente caixa-branca, como é mostrado na Figura 5.9. Essa opção não foi adotada para poder manter os componentes como caixas-pretas. Portanto, escolheu-se uma terceira opção: usar classes independentes da classe básica SistemaViarioUrbano e separar essas características em novos componentes chamados TempoMgr e NumViagensMgr, com as interfaces ITempoMgt e INumViagensMgt respectivamente como interfaces fornecidas. Esses componentes caixa-preta são mostrados na Figura 5.10. As operações da interface ITempoMgt são mostradas na Figura 5.11. Essas três opções são mostradas na Tabela 5.2, listando suas características, vantagens e desvantagens.

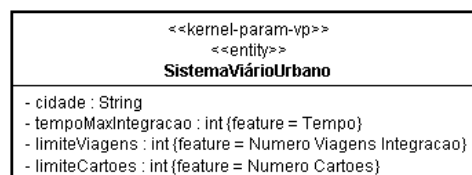


Figura 5.8: SistemaViarioUrbano como uma classe parametrizada com pontos internos de variação

Os componentes TempoMgr e NumViagensMgr são projetados similarmente, pois ambos são variantes do mesmo componente e possuem o mesmo tipo de associação com o componente básico e operações requeridas. Portanto, daqui em diante dá-se enfoque ao projeto do componente TempoMgr, e pode-se assumir um projeto análogo para o componente NumViagensMgr.

É preciso um componente controlador para obter e tratar informações dos componentes TempoMgr e ViagemMgr. Uma solução como aquela mostrada para LinhaIntegradaMgr na Figura 5.6 é possível. Outra semelhança ao exemplo da *Linha Integrada* é a necessidade de adicionar uma interface no componente controlador, nesse caso porque o controlador de viagem (ViagemCtrl)

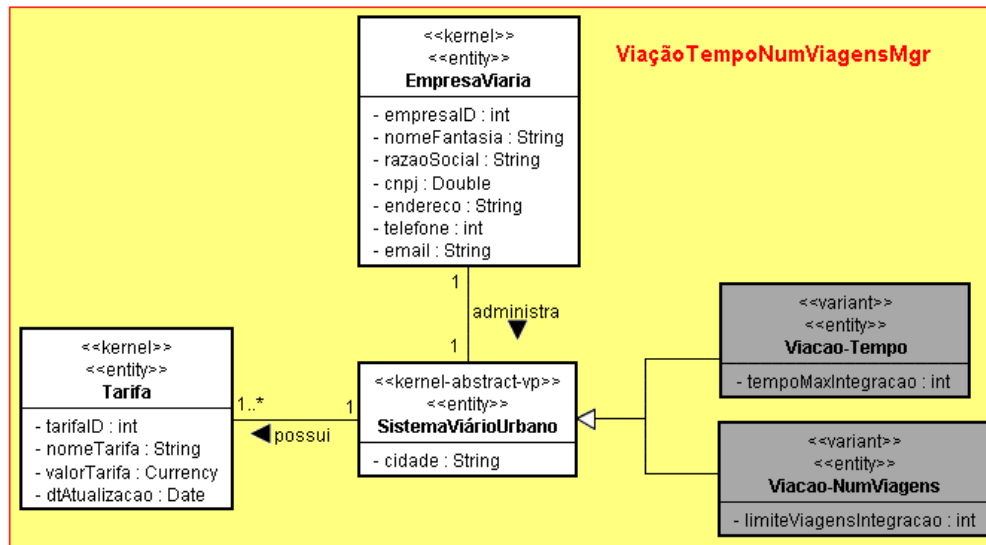


Figura 5.9: Classes em um componente caixa-branca, incluindo as características *Tempo* e *Número de Viagens de Integração*



Figura 5.10: Componentes caixa-preta de negócio para o grupo de características de *Integração*

Tabela 5.2: Diferentes opções de projeto de variabilidades que requerem novos atributos e operações em uma LPS

Opção	Característica	Vantagens	Desvantagens
Classe parametrizada	- Alterações em nível de atributos e operações - Componente caixa-branca	- Menor quantidade de classes - Melhor desempenho	- Não há separação de interesses
Subclasses variantes	- Alterações em nível de classes - Componente caixa-branca	- Separação de interesses (por classes)	- Dificuldade de manutenção
Classes independentes	- Alterações em nível de componentes independentes - Componente caixa-preta	- Facilidade de manutenção - Separação de interesses (por componentes)	- Possível explosão de classes e de componentes - Pior desempenho

precisa verificar uma possível integração dependendo do intervalo de tempo desde a última viagem. Portanto, o componente ViagemCtrl também precisa ser alterado para implementar essa

```

1 public interface ITempoMgt {
2
3     public int buscarTempo();
4     public void criarTempo(int tempoMaxIntegracao);
5     public void atualizarTempo(int tempoMaxIntegracao);
6
7 }

```

Figura 5.11: Operações da Interface ITempoMgt

nova variabilidade e requerer a nova interface do componente TempoMgr fornecida pelo seu controlador.

Para componentes caixa-preta, uma opção seria desenvolver dois novos componentes controladores (solução *a*). Entretanto, pode-se optar por um componente controlador que execute as funcionalidades de ambos os controladores, pois sempre serão usados juntos e possuem os mesmos interesses. Essa segunda opção (solução *b*) evita o desenvolvimento de um dos controladores e substitui o componente básico ViagemCtrl pelo seu componente variante, ViagemTempoCtrl. A solução para incluir a característica *Tempo* na arquitetura usando esse componente é mostrada na Figura 5.12. Para essa solução o componente ViagemCtrl não é reusado.

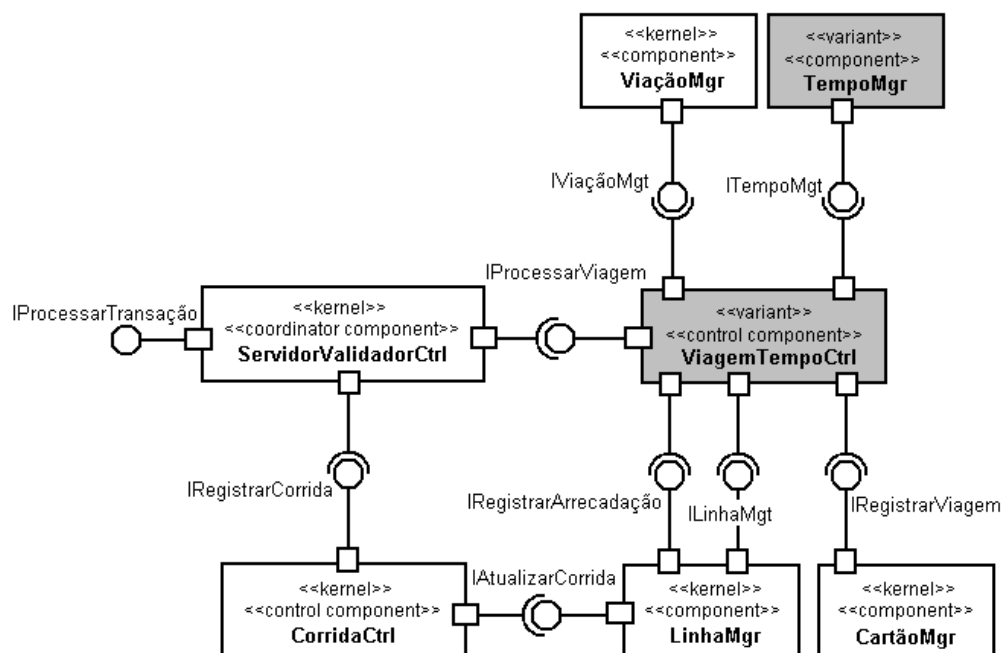


Figura 5.12: Solução *b* para integrar a característica *Tempo* na arquitetura da LPS-BET

Uma terceira e melhor solução usando componentes caixa-preta seria a de já projetar inicialmente o componente ViagemCtrl de tal modo que ele pudesse ser reusado por todas as aplicações e um componente controlador separado de ViagemCtrl pudesse ser adicionado para uma variabilidade da característica de *Integração*, que no nosso exemplo corresponde a *Tempo*. Essa

solução (solução *c*) é mostrada na Figura 5.13. O componente *TempoViagemCtrl* é criado e não demanda mudanças nos componentes básicos. Esse componente fornece a mesma interface *IProcessarViagem* para o componente *ServidorValidadorCtrl*, que assim não requer alteração. O novo componente requer a interface *ITempoMgt* para verificar uma possível integração por tempo e a interface *IProcessarViagem*, implementada pelo componente *ViagemCtrl*, para poder encaminhar o controle de uma viagem que não corresponda a uma integração.

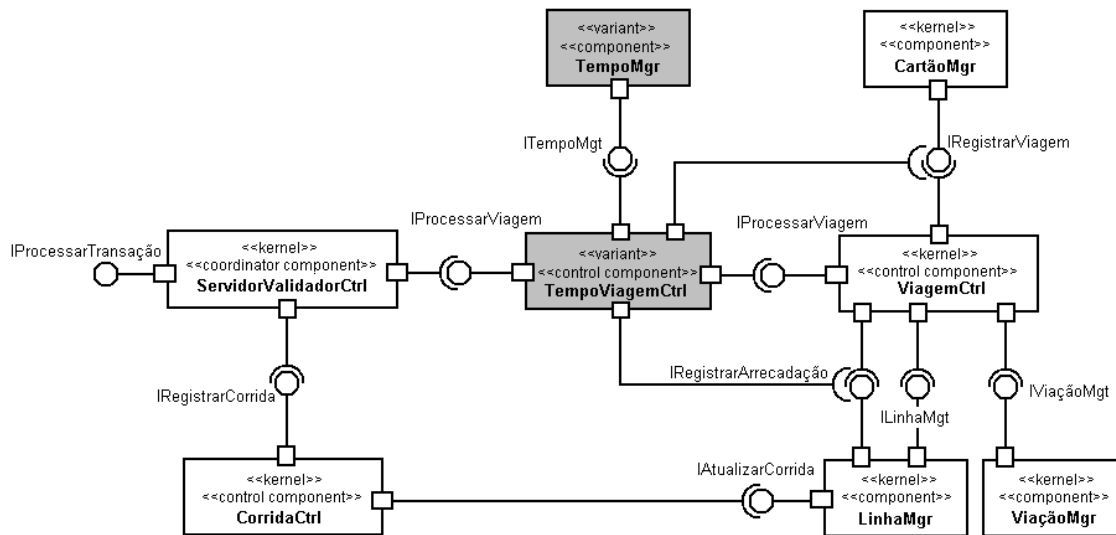


Figura 5.13: Solução *c* para integrar a característica *Tempo* na arquitetura da LPS-BET

O componente *TempoViagemCtrl* encapsula uma classe que implementa o método *processarViagem* da interface *IProcessarViagem* (interface mostrada na Figura 5.14), que será requerida pelo componente *ServidorValidadorCtrl*. A implementação do componente *TempoViagemCtrl* é apresentada na Figura 5.15. A classe *TempoViagemCtrl* requer quatro interfaces (linhas 03-06): *ITempoMgt* (fornecida pelo componente variante *TempoMgr*), *IRegistrarViagem* (fornecida por *CartãoMgr*), *IRegistrarArrecadação* (fornecido por *LinhaMgr*) e *IProcessarViagem* (fornecido pelo *ViagemCtrl*). O método *processarViagem* (linhas 08-15) é responsável por chamar o método *processarViagem* do componente *ViagemCtrl* se não estiver dentro do intervalo de tempo de uma viagem de integração. Para verificar isso, o método *verificarIntegracao* (linhas 17-29) é chamado com os parâmetros *cartaoID* e *onibusID* (linha 10). Esse método compara o tempo máximo de integração com o tempo que passou desde a última viagem e tem como resposta se a viagem corresponde ou não a uma integração. Caso haja integração por tempo, o método *processarIntegracao* (linhas 31-36) é chamado para atualizar os dados da viagem do passageiro e o número de passageiros na corrida do ônibus, sem fazer a arrecadação do valor da passagem.

Em termos do arquivo XML de contexto da aplicação, a configuração desses componentes e a ligação dos componentes relacionados ao *TempoViagemCtrl* é mostrada na Figura 5.16. O componente *ServidorValidadorCtrl* (linhas 01-11) requer três interfaces (linhas 02, 05 e 08),

```

1 public interface IProcessarViagem {
2
3     public String processarViagem(int cartaoID, int onibusID);
4
5 }

```

Figura 5.14: Interface IProcessarViagem

```

1 public class TempoViagemCtrl implements IProcessarViagem{
2
3     ITempoMgt interfaceTempoMgt;
4     IRegistrarViagem interfaceRegistrarViagem;
5     IRegistrarArrecadacao interfaceRegistrarArrecadacao;
6     IProcessarViagem interfaceProcessarViagem;
7
8     public String processarViagem(int cartaoID, int onibusID) {
9         String estado="INT-NOK";
10        estado = verificarIntegracao(cartaoID, onibusID);
11        if (!estado.equals("INT-OK"))
12            return interfaceProcessarViagem.processarViagem(cartaoID, onibusID);
13        else
14            return estado;
15    }
16
17    private String verificarIntegracao(int cartaoID, int onibusID) {
18        String estado = "INT-NOK";
19        long tempoDecorrido = Long.MAX_VALUE;
20        int tempoMaxIntegracao = interfaceTempoMgt.buscarTempo();
21        Viagem viagem = interfaceRegistrarViagem.buscarUltimaViagem(cartaoID);
22        if (viagem != null) {
23            Calendar horaUltimaViagem = viagem.getHora();
24            tempoDecorrido = Calendar.getInstance().getTimeInMillis() - horaUltimaViagem.getTimeInMillis();
25            if (tempoDecorrido <= tempoMaxIntegracao)
26                estado = processarIntegracao(onibusID, viagem);
27        }
28        return estado;
29    }
30
31    private String processarIntegracao(int onibusID, Viagem viagem) {
32        interfaceRegistrarArrecadacao.registrarArrecadacao(onibusID, 0);
33        viagem.setNumViagens(viagem.getNumViagens()+1);
34        interfaceRegistrarViagem.alterarViagem(viagem);
35        return "INT-OK";
36    }
37 }

```

Figura 5.15: Classe TempoViagemCtrl do componente de mesmo nome com implementação da interface IProcessarViagem

fornecidas pelos componentes CorridaCtrl, TempoViagemCtrl e CartãoMgr. Diferentemente da configuração do componente no núcleo, que requeria a interfaceProcessarViagem implementada pelo componente ViagemCtrl no lugar do TempoViagemCtrl (linha 06).

Percebeu-se pela implementação da classe do componente TempoViagemCtrl que ele requer quatro interfaces. Isso também pode ser visto no contexto da aplicação da Figura 5.16. Uma das interfaces requeridas é a interfaceTempoMgt (linha 14) fornecida pelo componente TempoMgr

(linha 15), que por sua vez é definido nas linhas 28-32 e não requer uma interface, pois é um componente de negócio e apenas possui dependência da propriedade `sessionFactory` (linhas 29-31) que é relacionada à persistência da entidade *Viação-Tempo*.

```

1  <bean id="ServidorValidadorCtrl" class="lps.bet.basico.servidorValidadorCtrl.ServidorValidadorCtrl">
2      <property name="interfaceRegistrarCorrida">
3          <ref bean="CorridaCtrl"/>
4      </property>
5      <property name="interfaceProcessarViagem">
6          <ref bean="TempoViagemCtrl"/>
7      </property>
8      <property name="interfaceCartaoMgt">
9          <ref bean="CartaoMgr"/>
10     </property>
11 </bean>
12
13 <bean id="TempoViagemCtrl" class="lps.bet.variabilidades.tempoViagemCtrl.TempoViagemCtrl">
14     <property name="interfaceTempoMgt">
15         <ref bean="TempoMgr"/>
16     </property>
17     <property name="interfaceRegistrarViagem">
18         <ref bean="CartaoMgr"/>
19     </property>
20     <property name="interfaceRegistrarArrecadacao">
21         <ref bean="LinhaMgr"/>
22     </property>
23     <property name="interfaceProcessarViagem">
24         <ref bean="ViagemCtrl"/>
25     </property>
26 </bean>
27
28 <bean id="TempoMgr" class="lps.bet.variabilidades.tempoMgr.TempoDAO">
29     <property name="sessionFactory">
30         <ref bean="sessionFactory"/>
31     </property>
32 </bean>

```

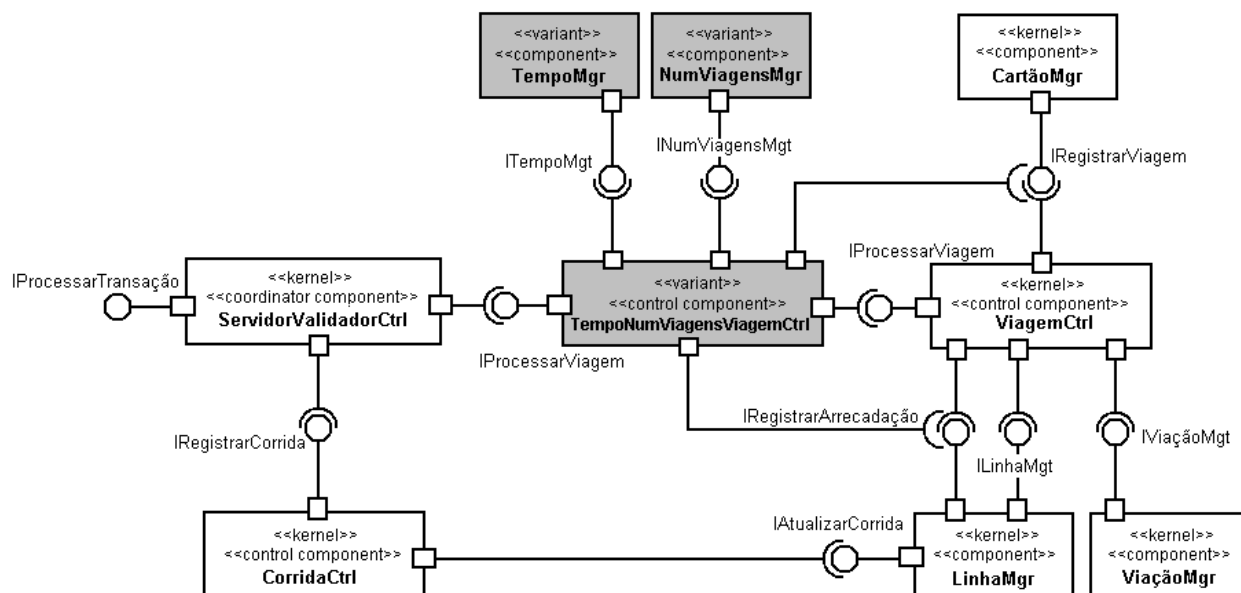
Figura 5.16: Beans relacionados ao componente *TempoViagemCtrl*

Como explicado anteriormente, a característica de *Número de Viagens de Integração* pode ser projetada da mesma maneira como foi ilustrado para a característica de *Tempo*. O mesmo é verdade para a característica de *Linha Integrada*, que também requer uma alteração no componente *ViagemCtrl* e apenas o tipo de validação é diferente. Portanto, a Figura 5.13 poderia ser redesenhada com os componentes *NumViagensViagemCtrl* e *NumViagensMgr* ao invés de *TempoViagemCtrl* e *TempoMgr*, ou até mesmo com os componentes *LinhaIntegradaViagemCtrl* e *LinhaIntegradaMgr*. Na Tabela 5.3 estão sintetizadas as três soluções explicadas para inserir variabilidade na arquitetura de componentes da LPS-BET, mostrando a quantidade de componentes controladores adicionais necessários para cada solução, uma descrição das soluções e níveis (alto, médio e baixo) comparativos de reúso e de manutenibilidade dos componentes controladores.

Entretanto, essas variabilidades de integração normalmente não aparecem sozinhas. A combinação dessas características pode ser requerida em uma aplicação. Na Figura 5.17 é mostrado um exemplo da combinação das características de *Tempo* e *Número de Viagens de Integração*.

Tabela 5.3: Soluções de junção de variabilidades por meio de controladores na arquitetura baseada em componentes caixa-preta

Solução	Descrição	Qtd. Controladores	Reúso	Manutenibilidade	Desempenho
<i>a</i>	Um controlador com a nova variabilidade substitui um básico e um controlador adicional específico para unir informações dos componentes de negócio (básico e opcional)	2	Baixo	Média	Médio
<i>b</i>	Um novo controlador com a nova variabilidade substitui um básico e ainda une as informações dos componentes de negócio (básico e opcional)	1	Baixo	Baixa	Médio-Alto
<i>c</i>	O controlador básico é reusado e apenas especificidades da variabilidade são do novo controlador, ligando-se com o componente de negócio opcional	1	Alto	Alta	Médio

**Figura 5.17:** Uma solução para integrar as características *Tempo* e *Número de Viagens* na arquitetura da LPS-BET

Finalmente, deve-se perceber que caso componentes caixa-branca tivessem sido usados no lugar de componentes caixa-preta, a classe do componente `ViagemCtrl` poderia ter sido reusada de modo que uma subclasse especializasse o seu comportamento e adicionasse o método de verificação de integração considerando uma característica de integração como *Tempo*, por exemplo. Tanto a classe quanto a subclasse estariam em um único componente que poderia se chamar `ViagemTempoCtrl`. Os componentes `TempoMgr` e `ViaçãoMgr` mostrados anteriormente na seção seriam substituídos por um único componente com as classes de ambos. A classe

Viação-Tempo estaria dentro do componente ViaçãoMgr e se chamaria ViaçãoTempoMgr por exemplo, da mesma forma como ilustrado na Figura 5.9. Com essa solução, o componente ViagemTempoCtrl estaria apenas conectado ao componente ViaçãoTempoMgr. Outra solução usando componentes caixa-branca seria a de manter dentro de um único componente as classes dos componentes ViaçãoTempoMgr e ViagemTempoCtrl. Entretanto, haveria vários interesses em um mesmo componente.

Outra consideração em relação ao projeto de componentes é o retrabalho que pode ser necessário ao passar de um ciclo de desenvolvimento para outro. As atividades de análise e projeto parcial da fase de elaboração tentam minimizar o retrabalho, embora este possa não ser totalmente eliminado. Na LPS-BET, o componente ViagemCtrl do núcleo passou por alterações no ciclo de desenvolvimento de Campo Grande, pois percebeu-se que, ao mudar a sua forma de implementação, poderia ser usada uma solução melhor na arquitetura de componentes (a solução *c* exposta anteriormente nesta seção). Com ela seria possível o reúso do componente para as diversas aplicações-referência. No desenvolvimento do núcleo achava-se que a solução seria eficiente para tal, mas só percebeu-se posteriormente que não seria possível.

5.3 Uso de Aspectos no Projeto da LPS-BET

Como visto na Seção 3.2 do Capítulo 3, aspectos são geralmente usados para implementar requisitos não-funcionais. Além disso, mostrou-se que eles também têm sido investigados para representar variabilidades de LPS. A arquitetura da LPS-BET é baseada em componentes caixa-preta e decidiu-se analisar o uso de aspectos para representar variabilidades nessa arquitetura, assim como para representar requisitos não-funcionais. Por esse motivo, foram feitos estudos para avaliar os prós e contras do uso de aspectos em uma arquitetura baseada em componentes caixa-preta.

5.3.1 Aspectos para Implementar Requisitos Não-Funcionais

A arquitetura de componentes da LPS-BET mostrada na Figura 4.8 apresentou o aspecto de autenticação, que corresponde a um requisito não-funcional do sistema e existe para as três aplicações-referência, sendo, portanto, parte do núcleo. A autenticação é projetada e implementada como um aspecto, porque senão o seu código estaria espalhado e entrelaçado em vários componentes (Carga Cartão, Aquisição Cartão, GerênciaCtrl). Na solução sem aspectos, cada um desses componentes teria que ter acesso a métodos responsáveis por autenticar o usuário e acessar a autorização do usuário para os diferentes módulos do sistema. O código estaria repetido em diferentes componentes e qualquer alteração nas regras de negócio de autenticação levaria a mudanças em todos os componentes. Com o uso de aspectos isso não ocorre.

O aspecto de autenticação é na verdade dividido em dois aspectos, um especificamente para a autenticação do usuário (Autenticacao) e outro para autorizar as páginas da web em que o usuário

pode navegar (Autorizacao), de acordo com um determinado nível de acesso. Ambos os aspectos possuem o mesmo ponto de junção e o aspecto de autenticação precede o aspecto de autorização.

A especificação desses aspectos é mostrada na Figura 5.18. Para autenticar o usuário, o aspecto de 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 ser feita antes de entrar em funções do sistema web. Diferentemente, o aspecto de autorização não requer operações de uma interface e esse aspecto tem um adendo do tipo around, obtendo o nível de acesso do usuário (método adicionarMenu) e depois adicionando o menu de acordo com o nível permitido para o usuário (método adicionarMenu).

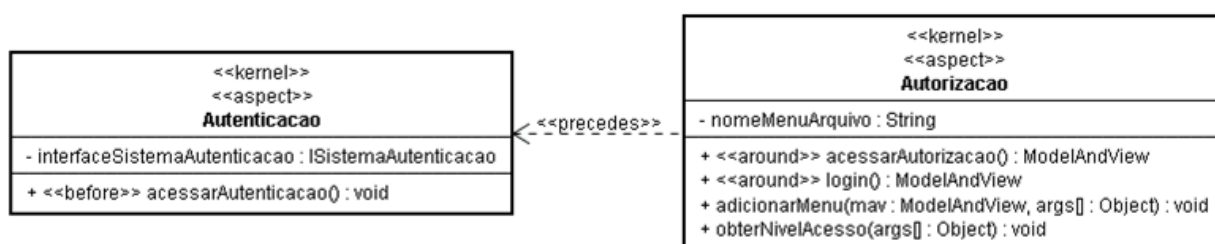


Figura 5.18: Adendos, operações e atributos dos aspectos Autenticação e Autorização

A Figura 5.19 fornece a arquitetura dos aspectos Autenticação e Autorização. Os aspectos entrecortam as interfaces dos componentes GerênciaCtrl, Aquisição Cartão e Carga Cartão. Esses componentes requerem diversas interfaces de outros componentes do domínio BET, porém, estes outros não estão sendo representadas na figura para focar apenas naquelas relacionadas aos aspectos. Com o uso desses aspectos para autenticação e autorização não há retrabalho no decorrer dos ciclos de desenvolvimento com a alteração dos componentes do subsistema Servidor WEB, pois o aspecto já teve seu ponto de junção configurado para entrecortar todos os métodos handleRequestInternal (método usado especificamente por controladores do MVC (Spring, 2008)) dos componentes que fazem parte do subsistema Servidor WEB, com exceção do Acesso Básico.

A implementação do aspecto de autenticação pode ser vista na Figura 5.20. O ponto de junção (linhas 05-07) entrecorta o método handleRequestInternal, contanto que não seja do próprio gerenciamento de login e não seja do acesso básico, pois no núcleo não deve haver autenticação de passageiro. O adendo do tipo before se encontra na linha 09. O sistema direciona para a página de login caso ainda não tenha havido um login (linha 16-17) ou caso o usuário não esteja autenticado ou se a sua sessão expirou (linha 20-21). Do contrário, a sessão do usuário é atualizada (linha 23).

Para São Carlos, a autenticação precisa ser feita também para o acesso web dos passageiros. Esse caso é tratado como a implementação de uma variabilidade normal, sendo implementada uma nova versão para substituir o aspecto Autenticação, chamada AutenticaçãoPassag, que apenas possui um ponto de entrecorte diferente, pois também entrecorta o método handleRequestInternal da interface do componente AcessoAdicional.

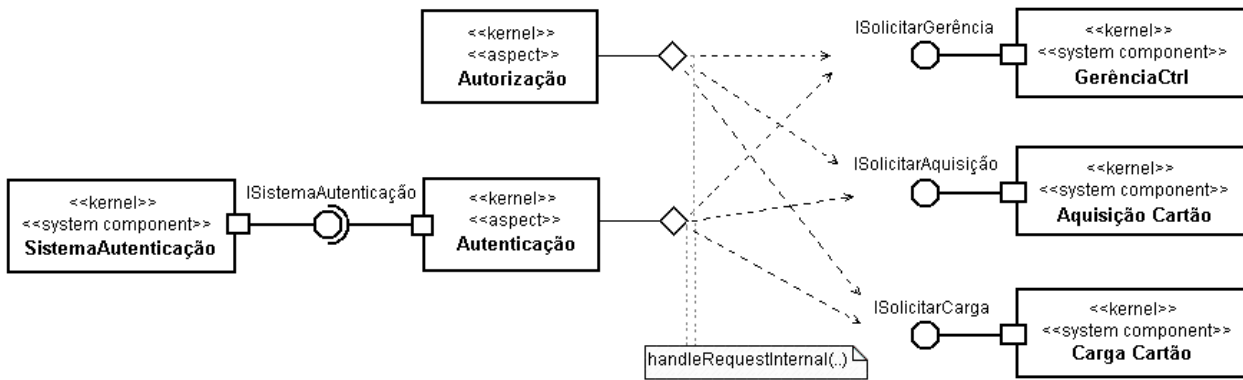


Figura 5.19: Arquitetura dos aspectos Autenticação e Autorização e dos componentes entrecortados

```

1 public aspect Autenticacao {
2
3     ISistemaAutenticacao interfaceAutenticacao;
4
5     pointcut acessarAutenticacao() : execution(* lps.bet...handleRequestInternal(..) &&
6         !execution(* lps.bet.basico.web.autenticacao.GerenciaLogin.handleRequestInternal(..) &&
7         !execution(* lps.bet.basico.web.acessoBasico.AcessoBasico.handleRequestInternal(..));
8
9     before() : acessarAutenticacao () {
10         Object[] args = thisJoinPoint.getArgs();
11         if (args.length > 0) {
12             HttpServletRequest request = (HttpServletRequest) args[0];
13             HttpServletResponse response = (HttpServletResponse) args[1];
14             HttpSession sessao = request.getSession(false);
15
16             if (sessao == null)
17                 response.sendRedirect("login.html");
18             else {
19                 String login = (String) sessao.getAttribute("login");
20                 if (!interfaceAutenticacao.estaAutenticado(login) || interfaceAutenticacao.estaExpirada(request))
21                     response.sendRedirect("login.html");
22                 else
23                     interfaceAutenticacao.atualizarSessao(request);
24             }
25         }
26     }
27 }
  
```

Figura 5.20: Implementação do aspecto Autenticação

Persistência também é normalmente um caso em que aspectos podem ser usados para representar os interesses transversais. Entretanto, persistência é uma situação especial, pois não é completamente ortogonal ao código-base (Rashid e Chitchyan, 2003). O armazenamento e a atualização possuem pontos de junção explícitos no código-fonte que podem ser associados a essas operações de persistência, isto é, podem ser adicionados em uma aplicação sem que ela seja desenvolvida com consciência da persistência. Em contrapartida, a remoção de objetos de um banco de dados não pode ser associada com algum ponto de junção. O mesmo problema ocorre para as consultas – deve haver chamadas explícitas a operações de banco de dados quando é necessário realizar al-

um tipo de consulta e isso depende do algoritmo da aplicação. Conseqüentemente, a persistência não pode ser completamente negligenciada no código-base de uma aplicação, havendo no mínimo uma implementação com consciência parcial, pois o engenheiro de software precisa ao menos estar ciente dos locais do código em que a aplicação precisa remover e consultar dados (Camargo, 2006).

A implementação da persistência utilizando o desenvolvimento orientado a aspectos (Rashid e Chitchyan, 2003; Soares, 2004; Couto *et al.*, 2005) já foi estudada por vários autores e tem uma solução feita pelo próprio grupo de pesquisa (Camargo, 2006). Além disso, várias implementações de persistência orientadas a aspectos usam declarações intertipos para inserir código no código-base, o que no caso deste trabalho violaria o encapsulamento dos componentes. Portanto, decidiu-se por não implementar esse interesse com aspectos para a LPS-BET e por usar o Hibernate (Hibernate, 2008).

5.3.2 Aspectos para Implementar Variabilidades da LPS-BET

Inicialmente a solução de projeto de variabilidades da LPS-BET foi feita e implementada considerando uma arquitetura puramente baseada em componentes, havendo o uso de aspectos apenas para representar requisitos não-funcionais. A partir dessa solução baseada em componentes foi projetada outra solução baseada em componentes e em aspectos. Nessa solução os aspectos passam a ser utilizados para representar variabilidades. Como são usados componentes do tipo caixa-preta, os aspectos são usados de modo a interceptar apenas operações das interfaces. Um estudo de caso foi realizado com o objetivo de comparar ambas as soluções de implementação de variabilidades: a solução de projeto e de implementação de variabilidades apenas com componentes e a solução usando componentes e aspectos.

O uso de aspectos será ilustrado considerando o mesmo exemplo da Seção 5.2: as características do grupo de características *Integração (Tempo, Linha Integrada e Número de Viagens de Integração)*. Como visto no projeto desses componentes, os controladores dos três variantes possuem um comportamento bastante semelhante, pois todos requerem alterações no mesmo componente (*ViagemCtrl*) e eles devem ser capazes de processar uma integração pela requisição de duas interfaces: *IRegistrarViagem* e *IRegistrarArrecadação*. Eles diferem em termos do componente de negócio que irão precisar (*TempoMgr*, *NumViagensMgr* e *LinhaIntegradaMgr*) e, portanto, diferem em relação à interface que irão requerer. Isso significa que eles serão distintos em relação às regras de negócio para verificação da existência de uma integração.

Portanto, com a escolha de usar aspectos para implementar essas variabilidades, um aspecto abstrato foi usado para generalizar as similaridades desses três tipos de integração. Cada variabilidade corresponde a um aspecto e implementa um aspecto abstrato (*IntegraçãoCtrl*), herdando o ponto de junção, o adendo e um método (*processarIntegração*), além de duas interfaces comuns que são requeridas. O aspecto *IntegraçãoCtrl* define um método abstrato chamado

verificarIntegração que deve ser implementado por cada aspecto especializado. Esses aspectos são mostrados na Figura 5.21.

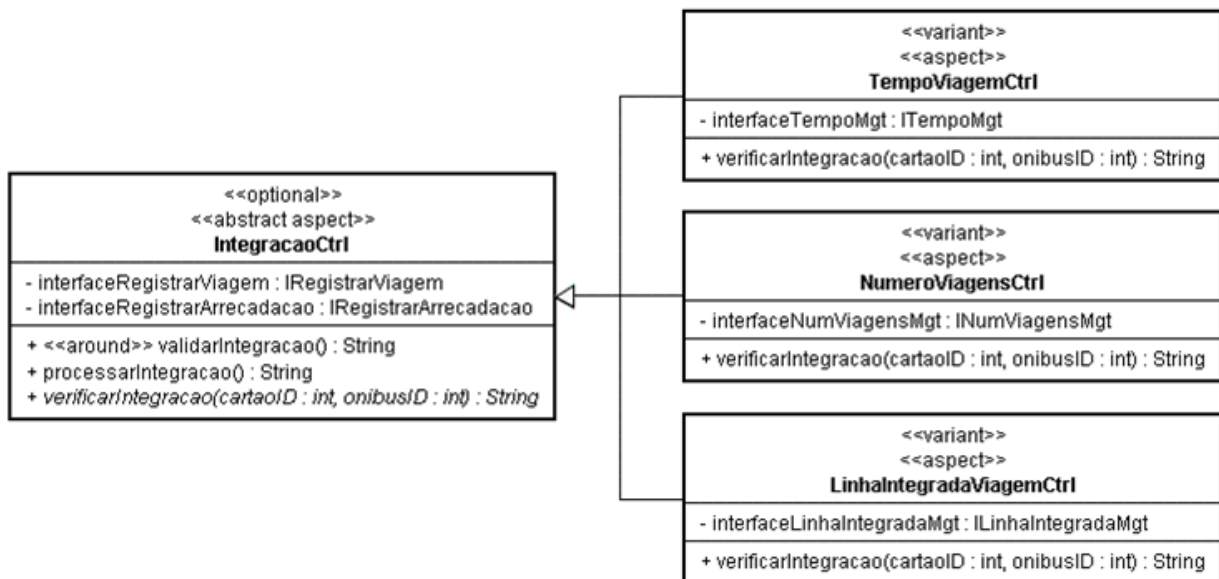


Figura 5.21: Aspecto abstrato e aspectos contratos para representar a característica *Integração*

A implementação do aspecto abstrato *IntegraçãoCtrl* usando AspectJ (Kiczales *et al.*, 2001) é apresentada na Figura 5.22. 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).

A Figura 5.23 mostra o projeto da característica *Tempo* usando aspectos. Esse projeto é correspondente à solução mostrada na Figura 5.13 que usa apenas componentes. A principal diferença é que o componente *ServidorValidadorCtrl* não precisa requerer a interface *IProcessarViagem* de um novo componente, mas continua requerendo a interface fornecida pelo componente básico *ViagemCtrl*. O aspecto *IntegracaoCtrl* entrecorta a interface e é estendido pelo aspecto *TempoViagemCtrl* que requer as operações da interface *ITempoMgt* do componente de negócio *TempoMgr*.

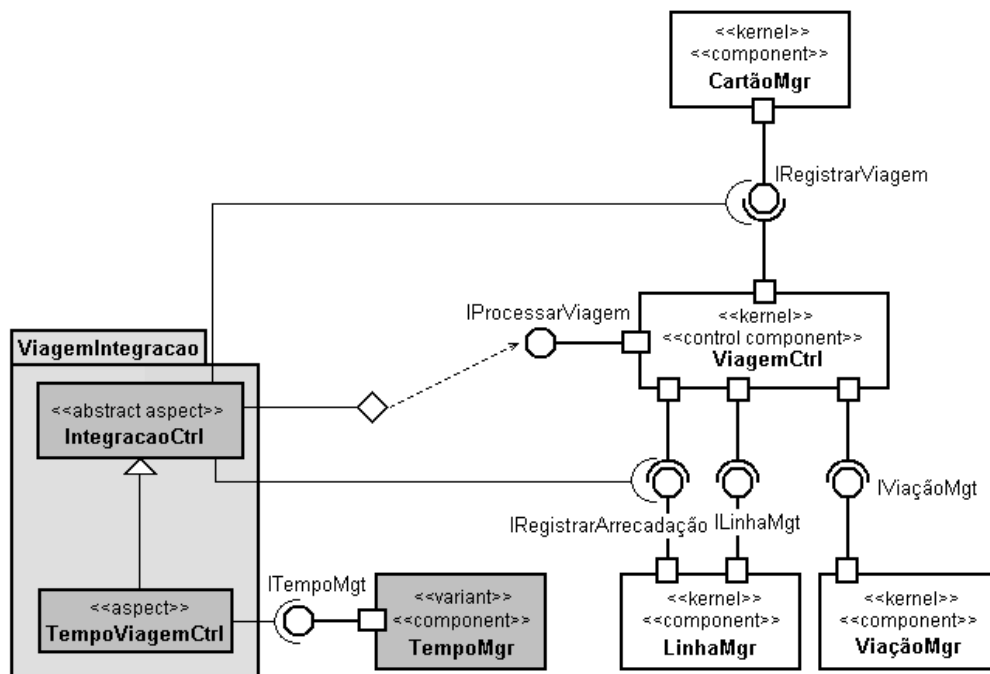
A implementação do aspecto *TempoViagemCtrl* é mostrada na Figura 5.24. Ele estende o aspecto abstrato *IntegraçãoCtrl* (linha 01) e, portanto, pode utilizar o método *processarIntegração* (linha 16). O aspecto verifica se a integração se aplica comparando o tempo decorrido desde a última viagem (linha 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 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 5.22: Implementação do aspecto abstrato IntegraçãoCtrl

Figura 5.23: Uma solução usando aspectos para adicionar a característica *Tempo* na arquitetura

Para a solução com componentes pode não ser necessário alterar componentes do núcleo para a adição de variabilidades, mas ao menos é preciso alterar a configuração no contexto de apli-

```

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 = interfaceTempoMgt.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 5.24: Implementação do aspecto ViagemTempoCtrl

cação do componente que implementa a interface requerida pelo componente básico, como visto anteriormente na Figura 5.16. Portanto, os componentes básicos em si podem não possuir conhecimento da inserção de variabilidades, pois são usadas interfaces com nomes equivalentes. Contudo, como a interface é fornecida por um componente diferente é necessário alterar a configuração dos componentes básicos requerendo essa interfaces pela referência (ref) ao componente que implementa a interface. Ao usar aspectos essa alteração no contexto da aplicação não é necessária, pois o aspecto entrecorta a interface sem que os componentes básicos tenham conhecimento do mesmo. No exemplo da característica de *Tempo*, para a solução com aspectos o componente básico *ServidorValidadorCtrl* não seria modificado nem na implementação nem na sua configuração. A Figura 5.25 mostra como fica a configuração para esse exemplo usando aspecto.

A solução aqui apresentada para o uso de aspectos não implementa a variabilidade como um todo com o uso de aspectos, pois deseja-se continuar com as vantagens do uso de componentes caixa-preta. Os componentes de negócio são extraídos do refinamento do modelo conceitual da mesma forma como são extraídos no projeto sem o uso de aspectos. Os aspectos substituem componentes controladores e de sistema. Eles entrecortam componentes básicos, adicionam lógicas de sistema ou de controle e obtêm informações necessárias de componentes de negócio. Dessa forma, os aspectos atuam como uma espécie de código de ligação (*glue code*) entre o núcleo e as variabilidades da LPS.

Em soluções para implementar uma variabilidade tendo um componente básico A que pode ser reusado sem problemas e é preciso adicionar um componente B, a solução usando aspectos é similar àquela mostrada nesta seção. O aspecto entrecorta a interface do componente A, realiza um processamento e, de acordo com o resultado, retorna o fluxo para o componente A. O processamento feito pelo aspecto pode consistir em uma chamada a operações do componente B, caso


```

1 <bean id="ServidorValidadorCtrl" class="lps.bet.basico.servidorValidadorCtrl.ServidorValidadorCtrl">
2   <property name="interfaceRegistrarCorrida">
3     <ref bean="CorridaCtrl"/>
4   </property>
5   <property name="interfaceProcessarViagem">
6     <ref bean="ViagemCtrl"/>
7   </property>
8   <property name="interfaceCartaoMgt">
9     <ref bean="CartaoMgr"/>
10  </property>
11 </bean>
12
13 <bean id="ViagemTempoCtrlAspecto" class="lps.bet.variabilidades.ViagemTempoCtrl" factory-method="aspectOf">
14   <property name="interfaceTempoMgt">
15     <ref bean="TempoMgr"/>
16   </property>
17   <property name="interfaceCartaoMgt">
18     <ref bean="CartaoMgr"/>
19   </property>
20   <property name="interfaceRegistrarArrecadacao">
21     <ref bean="LinhaMgr"/>
22   </property>
23 </bean>
24
25 <bean id="TempoMgr" class="lps.bet.variabilidades.tempoMgr.TempoDAO">
26   <property name="sessionFactory">
27     <ref bean="sessionFactory"/>
28   </property>
29 </bean>

```

Figura 5.25: Beans relacionados ao aspecto TempoViagemCtrl

o componente seja de negócio, ou em uma implementação das operações que estariam no componente B, caso o componente seja de controle ou de sistema. Como explicado anteriormente, a vantagem dessa solução é que evita a necessidade de alterar a configuração de componentes que requerem o componente A do núcleo.

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.

5.4 Considerações Finais

Neste capítulo foram apresentadas e discutidas diferentes soluções baseadas em componentes e em aspectos para implementar variabilidades. Podem ser usados componentes caixa-branca pela facilidade de implementação e de composição ou podem ser usados componentes caixa-preta para se ter uma maior separação de interesses e facilidade de manutenção. Essas soluções foram analisadas tanto para casos de modelagem de novas classes como para modelagem de novas subclasses com novos atributos e métodos para classes já existentes.

No contexto dessas categorias de modelagem foram fornecidas diversas soluções usando componentes, discutindo vantagens e desvantagens de cada uma, usando a linha de produtos desenvolvida. Como resultado, tem-se uma linha de produtos de software com um projeto detalhado da arquitetura baseada em componentes.

Além de considerar componentes para projetar a arquitetura da LPS, também foi analisado o uso de aspectos juntamente com os componentes como forma de representar as variabilidades. Adicionalmente, foi apresentada uma comparação do projeto apenas com componentes e com aspectos e componentes para algumas variabilidades, mostrando vantagens e desvantagens para cada solução. Os aspectos também foram utilizados para implementar requisitos não-funcionais que estão presentes nas aplicações-referência da LPS: autenticação e autorização.

Portanto, a LPS-BET possui o projeto de algumas variabilidades usando componentes e aspectos, mas a quantidade de aspectos desenvolvidos não fornece uma análise mais aprofundada em relação ao uso de aspectos em uma LPS com arquitetura baseada em componentes caixa-preta, sendo interessante investigar esses estudos a fundo.

Uso do Gerador Captor para a Engenharia de Aplicações da LPS-BET

6.1 Considerações Iniciais

Para automatizar a estratégia de engenharia de aplicação delineada no Capítulo 4 é usado o gerador de aplicações configurável Captor (Schimabukuro *et al.*, 2006). O gerador é configurado para a LPS-BET durante a engenharia de domínio, na qual são desenvolvidos os ativos centrais da linha em ciclos incrementais e há a atividade de elaboração incremental de uma receita (linguagem de modelagem da aplicação) para gerar aplicações a partir de determinados requisitos.

Este capítulo está organizado da seguinte forma: na Seção 6.2 é apresentada a configuração do gerador Captor para a LPS-BET; em seguida, na Seção 6.3, o Captor configurado é utilizado pelo engenheiro de aplicação para selecionar as variabilidades a serem considerações na geração de uma aplicação da linha pelo Captor, como o núcleo e a aplicação-referência de Campo Grande e, finalmente, na Seção 6.4 são apresentadas as considerações finais deste capítulo.

6.2 Configuração do Captor para o Domínio BET

A fase de Transição é realizada após a implementação e os testes executados na construção de cada ciclo de desenvolvimento. Nessa fase deve ser elaborada uma receita para guiar a posterior engenharia da aplicação-referência produzida no ciclo. Portanto, para cada fase de transição da

engenharia de domínio da LPS-BET, uma parte da LMA foi definida para a LPS e o domínio BET foi sendo configurado no Captor de forma incremental.

A configuração foi iniciada na transição do ciclo de desenvolvimento do núcleo. Nesse momento as características existentes na LPS-BET foram definidas no gerador e o Captor foi configurado para produzir um núcleo operacional, ou seja, com as características comuns da linha. Na fase de transição do ciclo de desenvolvimento da aplicação-referência de Fortaleza, a configuração do Captor foi incrementada para que a aplicação pudesse ser gerada pelo Captor, incluindo as características comuns e as variabilidades de Fortaleza. O mesmo processo foi feito para a fase de transição do ciclo de desenvolvimento de Campo Grande. Assim, no final dessa fase de transição, o Captor se encontrava configurado para as características comuns da LPS e para as características opcionais e variantes de Fortaleza e de Campo Grande. Na transição do ciclo de desenvolvimento da aplicação-referência de São Carlos, o gerador foi configurado para as variabilidades implementadas para São Carlos. Como esse último ciclo não foi totalmente implementado, a receita configurada no Captor para essa aplicação-referência não considera a existência da característica *Combinação de Cartões*.

A escolha de configurar o Captor incrementalmente e não apenas no final da engenharia de domínio da LPS-BET levou a um pequeno retrabalho na LMA. Contudo, considerou-se importante realizar a configuração em incrementos, pois a geração facilitava a validação das aplicações-referência, os testes realizados para verificar o funcionamento da LPS-BET com as variabilidades que iam sendo implementadas e a combinação das variabilidades existentes.

No Captor são utilizadas LMAs declarativas que são especificadas em um conjunto de formulários organizados hierarquicamente em forma de árvore. O diagrama de características da LPS-BET foi tomado como base para definir esses formulários durante a fase de transição do ciclo de desenvolvimento do núcleo, ou seja, os formulários puderam ser definidos sem ter as variabilidades implementadas. As características comuns não foram representadas, pois sempre fazem parte da LPS-BET, inclusive os aspectos que implementam requisitos não-funcionais (autenticação e autorização). Portanto, a árvore de formulários do Captor foi construída considerando grupos de características opcionais e variantes da LPS-BET. A Figura 6.1 mostra a árvore de formulários usada no Captor para o domínio BET. Essa estrutura hierárquica é definida no Captor, conforme é ilustrado na Figura 6.2.

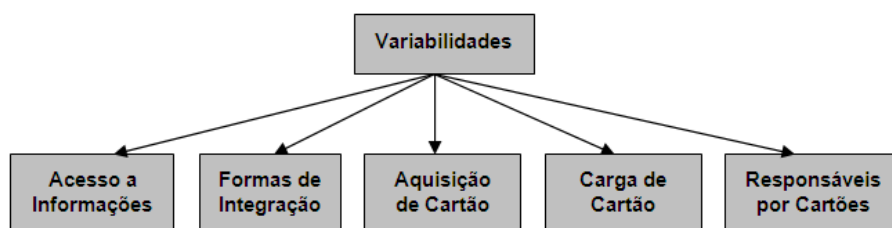


Figura 6.1: Árvore de formulários definidos no Captor para a LPS-BET

Form: New Form - Variant: Default

Id: 1

Form name: Variabilidades

Form name: "Acesso a Informacoes" - Min childs: 1 - Id: 2Max childs: 1

Form name: "Formas de Integracao" - Min childs: 1 - Id: 3Max childs: 1

Form name: "Aquisicao de Cartao" - Min childs: 1 - Id: 4Max childs: 1

Form name: "Carga de Cartao" - Min childs: 1 - Id: 5Max childs: 1

Next forms: Form name: "Responsaveis por Cartoes" - Min childs: 1 - Id: 6Max childs: 1

Add

Edit

Remove

Up

Down

Figura 6.2: Estrutura hierárquica dos formulários para a LPS-BET

Cada formulário da LMA no Captor contém elementos que definem variantes. Esses variantes foram definidos como sendo as características do grupo de características do formulário. A Tabela 6.1 mostra a listagem de elementos variantes definidos por formulário no Captor e os seus possíveis valores.

Tabela 6.1: Disposição dos elementos variantes nos formulários

Formulário	Elementos Variantes	Possíveis Valores
Variabilidades	Nome da Aplicação	<nome da aplicação>
Acesso a Informações	Acesso Adicional	inexistente, existente
Formas de Integração	Terminal	inexistente, existente
	Integração	inexistente, tempo, número de viagens e linha integrada (combinação das variáveis)
	Solução de Integração	com componentes, com aspectos
Aquisição de Cartão	Pagamento de Cartão	inexistente, existente
	Restrição de Cartões	inexistente, número de cartões, combinação de cartões
Carga de Cartão	Limite de Passagens	inexistente, existente
Responsáveis por Cartões	Empresas Usuárias	inexistente, existente

Com exceção dos elementos variantes Nome da Aplicação e Solução de Integração, os elementos variantes correspondem a características da LPS-BET. Como padrão é assumido o valor inexistente para os elementos correspondentes a características. O elemento Nome da Aplicação deve ser preenchido durante a engenharia de aplicação com o nome da aplicação a ser desenvolvida (por exemplo, Fortaleza, Campo Grande e São Carlos). O variante Solução de Integração foi criado para optar pela solução de implementação das variabilidades existentes de integração na aplicação utilizando componentes ou aspectos. A opção não se aplica para todas as variabilidades da LPS-BET, pois até o momento apenas as características de integração foram implementadas usando aspectos. A Figura 6.3 mostra os dados definidos para o elemento Integração e ilustra os parâmetros preenchidos para os diversos variantes.

Após finalizar a configuração do Captor para a LPS-BET, os gabaritos foram criados. O Captor armazena as informações inseridas na sua interface gráfica em uma representação textual no

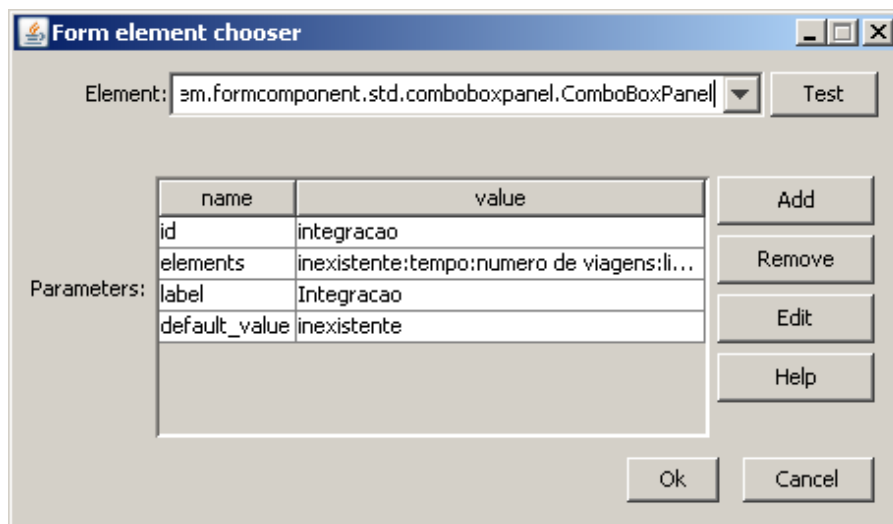


Figura 6.3: Definição do elemento variante Integração

formato XML. A estrutura em XML que o Captor utiliza para persistir a especificação possui uma parte fixa (independente de domínio) e uma parte variável (dependente de domínio) (Shimabukuro, 2006).

A parte variável representa os dados contidos nos elementos de cada formulário, como é ilustrado na Figura 6.4. A marcação `project` (linha 03-05) representa o nome do projeto e a marcação `forms` contém os dados dos formulários da especificação. As marcações `data` representam os elementos variantes dos formulários (`form`) e o conteúdo dessas marcações é definido na engenharia da aplicação. A figura mostra nas linhas 13 a 19 que a aplicação usada como exemplo possui a característica `Terminal` no grupo `Formas de Integração` e possui a característica de `Tempo de integração`. Além disso, a solução a ser usada na geração dessas características corresponde àquela usando componentes para representar variabilidades.

Geralmente há um gabarito para cada artefato gerado e com base no diagrama de mapeamento de variabilidades é possível determinar quais definições do projeto vão afetar qual gabarito (Shimabukuro, 2006). Esses gabaritos devem ser implementados usando a linguagem de transformação chamada XSL (W3C, 2008), que corresponde à linguagem de transformação utilizada pelo Captor. Para o caso da LPS-BET, como é usado o arquivo XML de “contexto de aplicação” (artefato chamado `bet-servlet.xml.xsl`) para configurar os componentes a serem usados na aplicação e como os componentes são acoplados, não é necessário fazer um gabarito para cada artefato gerado e para cada variabilidade. Dessa forma só é preciso elaborar um gabarito a partir desse artefato para mapear as variabilidades que a aplicação requer dada a especificação definida na interface gráfica do Captor. A parte fixa do gabarito é definida no ciclo de desenvolvimento do núcleo, enquanto as partes variáveis são definidas incrementalmente nos outros ciclos. São realizados testes de regressão em cada um desses incrementos para validar os gabaritos ao passo em que são refinados e para se certificar que os gabaritos não criaram problemas na aplicação-referência já implementada.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <formsData>
3   <project>
4     <name>Nome do Projeto</name>
5   </project>
6
7   <forms>
8     <form id="1.1" variant="Variabilidades">
9       <data>
10        <textatt name="1">Nome da Aplicação</textatt>
11      </data>
12      ...
13     <form id="3.1" variant="Formas de Integracao">
14       <data>
15        <combo name="terminal">existente</combo>
16        <combo name="integracao">tempo</combo>
17        <combo name="solucaoIntegracao">com componentes</combo>
18      </data>
19    </form>
20    ...
21  </forms>
22 </formsData>
23
```

Figura 6.4: Exemplo de Estrutura XML da Especificação do Captor

Na Figura 6.5 é mostrado um fragmento do gabarito do arquivo de configuração. Os comentários sinalizam as configurações que foram removidas. A configuração do componente de negócio LinhaMgr é mostrada para exemplificar uma parte fixa no gabarito (linhas 09-14). Essa parte foi definida na fase de transição do desenvolvimento do núcleo. Para ilustrar uma parte variável no gabarito, utiliza-se o componente de negócio TerminalMgr (linhas 21-23), que deve ser incorporado na transformação caso o elemento variante Terminal seja definido como existente (linhas 19-20). Essa parte variável foi incorporada ao gabarito na fase de transição do ciclo de desenvolvimento da aplicação-referência de Fortaleza.

Além disso, foi definido mais um gabarito para personalizar a página principal do site (artefato chamado menu.xml.xsl) da aplicação BET, colocando o nome definido no Captor, além de fornecer links adicionais no menu para algumas variabilidades. Na Figura 6.6 é mostrada uma parte do gabarito menu.xml.xsl em que são considerados os elementos variantes Nome da Aplicação e Terminal. O gabarito mostra uma parte variável em que o valor do atributo do formulário Variabilidades é colocado como cabeçalho (linha 12) e outra parte em que se houver Terminal (valor existente) como forma de integração, é inserido um link para a gerência de terminais (linhas 22-25). Outros elementos variantes que são definidos nesse gabarito são a Integração utilizando Linha Integrada e as Empresas Usuárias, porém, não estão mostrados na Figura 6.6.

Para indicar ao Captor quais gabaritos devem ser utilizados no processo de geração de artefatos deve ser definido um arquivo de mapeamento de transformação de gabaritos usando a linguagem MTL (do inglês *Mapping Transformation Language*) criada especificamente para o Captor. Na Figura 6.7 é apresentado o arquivo de mapeamento da LPS-BET. A marcação *task* define a transformação de um gabarito (com extensão .xsl) em um arquivo de saída (linhas 8 e 14). Portanto,

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
3  <xsl:output method="xml" doctype-system="http://www.springframework.org/dtd/spring-beans.dtd"/>
4  <xsl:template match="/">
5    <beans>
6      <!--Beans relacionados ao sistema web-->
7      <!--Beans de Componentes de Negócio do Núcleo-->
8      ...
9      <bean id="LinhaMgr" class="lps.bet.basico.linhaMgr.LinhaMgr">
10         <property name="linhaDAO" ref="LinhaDAO"/>
11         <property name="corridaDAO" ref="CorridaDAO"/>
12         <property name="validadorDAO" ref="ValidadorDAO"/>
13         <property name="onibusDAO" ref="OnibusDAO"/>
14       </bean>
15       ...
16     <!-- Beans relacionados ao Hibernate-->
17     <!-- Partes Variáveis do gabarito-->
18     ...
19     <xsl:if
20       test="formsData/forms/form/form[@variant='Formas de Integraçã']/data/combo[@name='terminal']='existente'">
21       <bean id="TerminalMgr" class="lps.bet.variabilidades.terminalMgr.TerminalMgr">
22         <property name="terminalDAO" ref="TerminalDAO"/>
23       </bean>
24       <!--Configuração dos Beans GerenciaTerminal e TerminalDAO-->
25     </xsl:if>
26     ...
27   </beans>
28 </xsl:template>
29 </xsl:stylesheet>

```

Figura 6.5: Parte do gabarito bet-servlet.xml.xsl

a transformação de gabaritos na LPS-BET deve gerar os arquivos bet-servlet.xml e menu.xml. Esses arquivos configurados pelo Captor juntamente com os outros artefatos da LPS-BET formam a aplicação definida no Captor. O sistema gerado utiliza apenas as variabilidades escolhidas pelo engenheiro da aplicação.

6.3 Engenharia de Aplicações da LPS-BET

Na engenharia de aplicações da LPS-BET é gerada uma aplicação com o uso do Captor configurado para o domínio específico do BET de acordo com os requisitos da aplicação. O primeiro passo para a engenharia de aplicações é elicitar os requisitos da aplicação a ser obtida.

Inicialmente, deve-se verificar se a aplicação realmente se encontra dentro do domínio BET. Se a aplicação não estiver no escopo do domínio BET, não é o caso de gerar a aplicação pelo Captor a partir da LPS-BET, pois existem mais variabilidades do que similaridades entre essa aplicação e o domínio. Senão, em seguida, deve-se verificar se as características da aplicação já foram previstas na LPS-BET. Caso todas estejam previstas na LPS-BET, não haverá problemas na geração da aplicação pelo Captor, sendo apenas necessário identificar as características da LPS-BET que a aplicação possuirá e fornecer como entrada ao gerador. Do contrário, existe ao menos uma característica que não foi planejada para a LPS-BET. Deve ser analisado então se é válido imple-


```

1 <?xml version="1.0" encoding="ISO-8859-1"?>
2
3 <xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4
5 <xsl:output method="xml" encoding="ISO-8859-1" indent="yes"/>
6
7 <xsl:template match="/">
8   <div class="header">
9     ...
10    <div class="logo">
11      ...
12      <h2><xsl:value-of select="formsData/forms/form[@variant='Variabilidades']/data/textatt[@name='1']"/></h2>
13    </div>
14  </div>
15  <div class="bar menu">
16    <ul>
17      ...
18      <li nivel="3"><a href="#">Linha</a>
19      <ul>
20        <li nivel="5"><a href="gerenciaLinha.html">Linha</a></li>
21        ...
22      <xsl:if
23        test="formsData/forms/form/form[@variant='Formas de Integracao']/data/combo[@name='terminal']='existente' ">
24        <li nivel="5"><a href="gerenciaTerminal.html">Terminal</a></li>
25      </xsl:if>
26    </ul>
27  </li>
28  ...
29 </ul>
30 </div>
31
32 </xsl:template>
33 </xsl:stylesheet>

```

Figura 6.6: Parte do gabarito menu.xml.xsl

```

1 <composer name="FIT">
2   <main>
3     <callTask id="betServlet"/>
4     <callTask id="menu"/>
5   </main>
6
7   <tasks>
8     <task id="betServlet">
9       <compose>
10        <template>bet-servlet.xml.xsl</template>
11        <newFilename>bet-servlet.xml</newFilename>
12      </compose>
13    </task>
14    <task id="menu">
15      <compose>
16        <template>menu.xml.xsl</template>
17        <newFilename>menu.xml</newFilename>
18      </compose>
19    </task>
20  </tasks>
21 </composer>

```

Figura 6.7: Arquivo de Mapeamento de Gabaritos para a LPS-BET

mentar a característica e adicioná-la aos ativos centrais da LPS-BET ou se é preferível desenvolver a característica especificamente para a aplicação desejada. Caso a característica seja inserida na LPS-BET, deve-se também realizar alterações na configuração do Captor para o domínio BET e nos seus gabaritos, para que a nova característica esteja disponível para as próximas aplicações que surgirem e até possa ser incorporada em aplicações que já estejam em produção, pela geração de novas versões.

As características da aplicação devem ser especificadas no domínio BET do Captor. A partir dessas informações das variabilidades da aplicação, o Captor utiliza a LMA da LPS-BET e pode gerar os artefatos específicos da aplicação, definindo os ativos centrais da linha que devem ser usados e como devem estar conectados para gerar a aplicação desejada.

As aplicações geradas devem ser testadas para complementar as atividades de testes realizadas durante a engenharia do domínio. São executados testes sistêmicos da aplicação para validá-la de acordo com os requisitos e as características levantados no início da engenharia da aplicação. O teste deve garantir que a aplicação realiza os requisitos da aplicação e que nenhum requisito tenha sido omitido (Pohl *et al.*, 2005). Além disso, nos testes da aplicação é verificado se os pontos variantes da aplicação estão acoplados corretamente às características básicas da LPS-BET e às outras variabilidades da aplicação. Isso deve ser verificado tanto no arquivo de configuração dos componentes quanto no teste funcional da aplicação. É importante verificar se existem características configuradas que não devem fazer parte da aplicação e se características necessárias para a aplicação foram desconsideradas (Pohl *et al.*, 2005).

Adicionalmente, como está sendo usado o gerador para obter as aplicações, também devem ser feitos testes para verificar a integração entre o gerador e a aplicação e a corretude da configuração realizada no gerador para a LPS-BET. Algumas combinações de variabilidades já foram testadas durante os testes de domínio, mas mesmo assim devem ser testadas novamente após geradas pelo Captor.

No caso de se ter levantado no início da engenharia de aplicação que alguns requisitos não existiam na LPS-BET e que eles seriam implementados especificamente para a aplicação, esses requisitos são implementados e acoplados a uma versão da aplicação gerada com as características que existem na LPS-BET. Esses novos requisitos são específicos da aplicação e devem ser testados detalhadamente, pois não passaram por quaisquer testes.

6.3.1 Engenharia da Aplicação BET de Campo Grande

A seguir é detalhado o processo de engenharia da aplicação BET de Campo Grande. Como essa aplicação foi previamente analisada na análise da LPS-BET como uma aplicação-referência, já se tem idéia dos requisitos e das características que a aplicação deve possuir. As características existentes para a aplicação de Campo Grande são mostradas na Figura 6.8. Como visto na Figura, as variabilidades para Campo Grande são *Acesso Adicional*, *Terminal*, *Tempo* e *Linha de Integração*, *Número de Viagens de Integração*, *Número de Cartões* e *Empresas Usuárias*.

A partir das características identificadas para a aplicação é possível criar o modelo da aplicação de Campo Grande. Com esse intuito deve-se executar o Captor e criar um projeto no domínio BET. A interface gráfica do Captor relacionada a essa função é mostrada na Figura 6.9. Na figura estão visíveis os diversos domínios configurados no Captor que podem ser usados para a geração de aplicações. A LPS-BET encontra-se evidenciada na figura e é chamada de *DominioBET*.

No Captor, o domínio BET requer que um nome seja dado à aplicação a ser gerada no formulário Variabilidades, como pode ser visto na Figura 6.10. Do lado esquerdo da figura pode ser vista a hierarquia de formulários para construção da aplicação na LPS-BET. As características identificadas e especificadas pelo engenheiro de aplicação para a aplicação de Campo Grande devem ser informadas nesses formulários.

A seleção dos valores dos elementos dos diversos formulários para a aplicação de Campo Grande é feita de acordo com as características. Caso a característica não exista para a aplicação, é selecionada a opção inexistente para o elemento variante. A Figura 6.11 mostra como são preenchidos esses formulários para que a aplicação de Campo Grande possa ser gerada com as características que a definem. Por exemplo, a característica de *Pagamento de Cartão* não existe para Campo Grande, sendo marcada como inexistente. Para a engenharia da aplicação de Fortaleza esse elemento estaria marcado como existente. Como o Captor não possui implementada a relação entre elementos variantes, como dois elementos que sejam mutuamente exclusivos, caso das características *Número de Cartões* e *Combinação de Cartões* do grupo *Restrição de Cartões*, o formulário foi definido de forma que só fosse possível selecionar uma opção sem selecionar a outra.

Para que o Captor possa gerar os gabaritos da aplicação de Campo Grande, o engenheiro deve salvar o projeto da aplicação. Com isso, o Captor gera o arquivo no qual estão os valores dos elementos variantes informados pelo engenheiro de aplicação. O Captor apresenta um console que registra a geração do código XML com base nos formulários preenchidos, como pode ser visto na Figura 6.12.

O código XML gerado pelo Captor é mostrado na Figura 6.13. O nome da aplicação é Campo Grande (linha 9) e as características existentes para Campo Grande podem ser vistas nas linhas 13, 18, 19, 26 e 36. Optou-se pela solução de integração com componentes (linha 20). Caso fosse escolhido o valor para o elemento Solução de Integração “com aspectos”, apareceria no lugar da linha 20 a seguinte linha de código `<combo name=”solucaoIntegracao”>com aspectos</combo>`.

A aplicação pode ser gerada após essa atividade. O Captor gera dois artefatos a partir dos gabaritos definidos na engenharia de domínio e do código XML definindo os elementos variantes: o `bet-servlet.xml` e o `menu.xml`. Essa atividade pode ser verificada no console do Captor visto na Figura 6.14.

Internamente ao Captor, o gabarito indica como cada característica é obtida pela composição de componentes. Na Figura 6.15 é mostrado um extrato do gabarito referente a algumas características de Campo Grande. Nas linhas 05-09 e 28-29 há código que sempre faz parte do arquivo

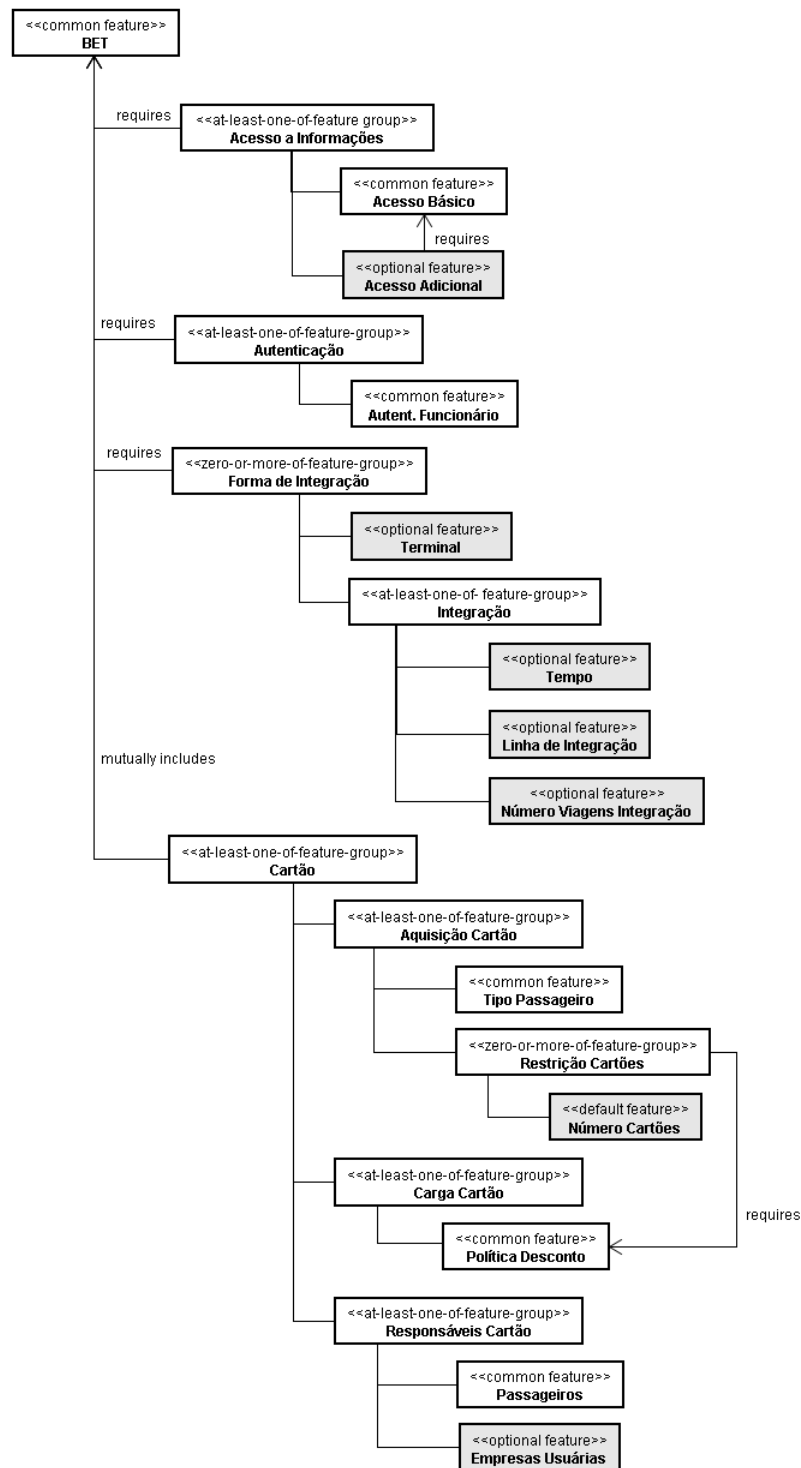


Figura 6.8: Diagrama de Características para a aplicação de Campo Grande

gerado. Na linha 09 é definida uma interface requerida pelo componente ValidadorServidorCtrl, porém, o componente que fornece essa interface varia de acordo com o processamento realizado entre as linhas 10-27. A primeira verificação é a solução para a característica de integração. Caso seja com aspectos, o componente básico ViagemCtrl implementa a interface (linhas 24-

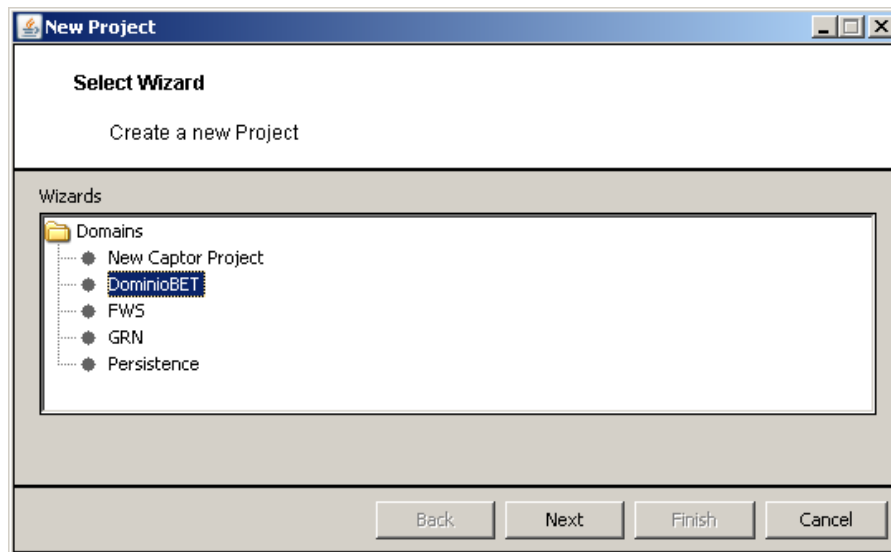


Figura 6.9: Criação de um novo projeto para geração de uma aplicação pelo Captor

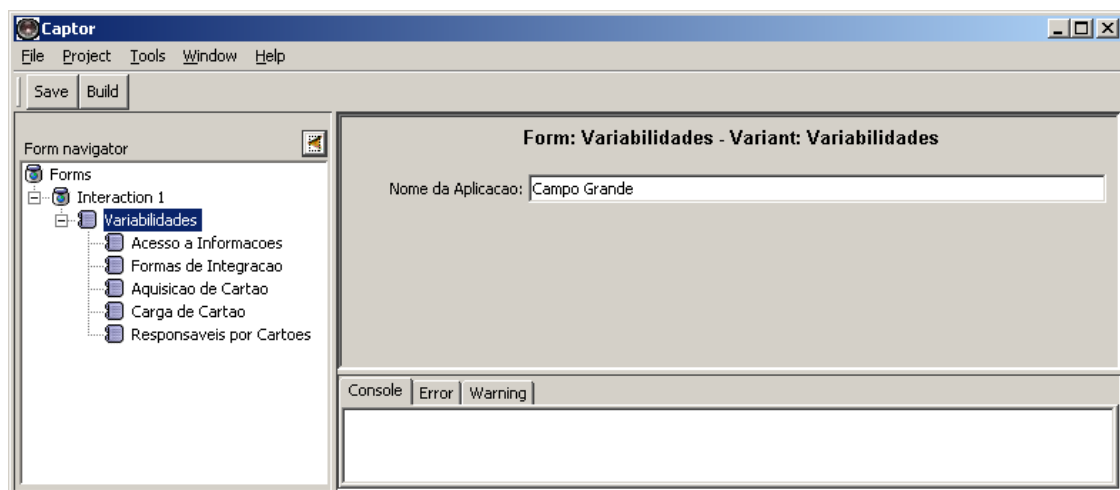


Figura 6.10: Definição do elemento Nome da Aplicação para a aplicação de Campo Grande

26). Caso contrário, componentes devem ser usados para implementar a variabilidade (linhas 11-23). Para Campo Grande as formas de integração são por tempo, número de viagens e linha integrada, portanto, a interface deve ser implementada pelo componente *ViagemTempo-NumViagensLinhaIntegradaCtrl* (linhas 15-18). A figura também ilustra a definição ou não da característica *Terminal* (linhas 31-37). Caso exista a característica, serão definidos os componentes relacionados a essa característica, como *TerminalMgr* (linhas 33-35).

O arquivo de configuração (*bet-servlet.xml*) que é gerado define então os componentes que existem para a aplicação de Campo Grande e a forma como estão conectados os componentes. Esses componentes que são configurados fazem parte dos ativos centrais da LPS-BET. Um fragmento do arquivo de configuração gerado é mostrado na Figura 6.16. É mostrada a definição de um bean da parte básica da LPS (linhas 04-09) e a configuração de uma interface requerida por

The image displays five stacked forms, each with a title and a variant name, and a dropdown menu for data selection:

- Form: Acesso a Informacoes - Variant: Acesso a Informacoes**
Acesso Adicional:
- Form: Formas de Integracao - Variant: Formas de Integracao**
Terminal:
Integracao:
Solucao de Integracao:
- Form: Aquisicao de Cartao - Variant: Aquisicao de Cartao**
Pagamento de Cartao:
Restrição de Cartoes:
- Form: Carga de Cartao - Variant: Carga de Cartao**
Limite de Passagens:
- Form: Responsaveis por Cartoes - Variant: Responsaveis por Cartoes**
Empresas Usurias:

Figura 6.11: Definição dos valores dos elementos para a aplicação de Campo Grande

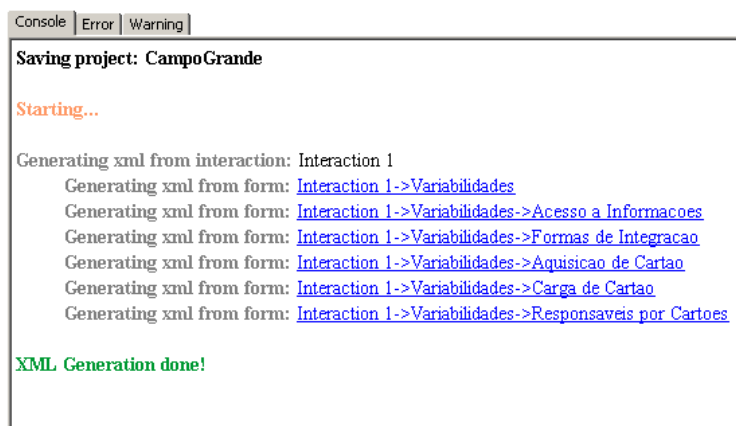
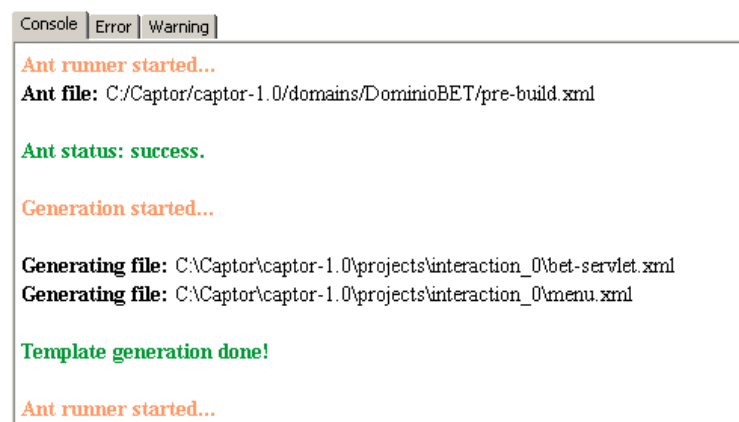


Figura 6.12: Captor gera arquivo XML com valores dos elementos variantes dos formulários

ele, a interfaceProcessarViagem (linha 06). Para a aplicação de Campo Grande, a interface é implementada pelo componente ViagemTempoNumViagensLinhaIntegradaCtrl, pois existem as variabilidades *Tempo*, *Número de Viagens* e *Linhas Integradas*. Portanto, também devem ser definidos os componentes de negócio dessas variabilidades (linhas 17-25). Além disso, Campo Grande requer mais dois componentes de negócio que não existem para o núcleo: TerminalMgr e EmpresaUsuarialMgr. Esses dois componentes também existiriam na geração do BET de Fortaleza.

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <formsData>
3   <project>
4     <name>CampoGrande</name>
5   </project>
6   <forms>
7     <form id="1.1" variant="Variabilidades">
8       <data>
9         <textatt name="1">Campo Grande</textatt>
10      </data>
11     <form id="2.1" variant="Acesso a Informacoes">
12       <data>
13         <combo name="acessoAdicional">existente</combo>
14       </data>
15     </form>
16     <form id="3.1" variant="Formas de Integracao">
17       <data>
18         <combo name="terminal">existente</combo>
19         <combo name="integracao">tempo, numero de viagens, linha integrada</combo>
20         <combo name="solucaoIntegracao">com componentes</combo>
21       </data>
22     </form>
23     <form id="4.1" variant="Aquisicao de Cartao">
24       <data>
25         <combo name="pgtoCartao">inexistente</combo>
26         <combo name="restricaoCartoes">numero de cartoes</combo>
27       </data>
28     </form>
29     <form id="5.1" variant="Carga de Cartao">
30       <data>
31         <combo name="limitePassagens">inexistente</combo>
32       </data>
33     </form>
34     <form id="6.1" variant="Responsaveis por Cartoes">
35       <data>
36         <combo name="empresasUsuaris">existente</combo>
37       </data>
38     </form>
39   </forms>
40 </formsData>
```

Figura 6.13: Estrutura XML da Especificação de Campo Grande no Captor



The screenshot shows a console window with tabs for 'Console', 'Error', and 'Warning'. The output text is as follows:

```
Ant runner started...
Ant file: C:/Captor/captor-1.0/domains/DominioBET/pre-build.xml

Ant status: success.

Generation started...

Generating file: C:\Captor\captor-1.0\projects\interaction_0\bet-servlet.xml
Generating file: C:\Captor\captor-1.0\projects\interaction_0\menu.xml

Template generation done!

Ant runner started...
```

Figura 6.14: Captor gera arquivo XML com valores dos elementos variantes dos formulários

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
3  <xsl:output method="xml" doctype-system="http://www.springframework.org/dtd/spring-beans.dtd"/>
4  <xsl:template match="/">
5    <beans>
6      ...
7      <bean id="ValidadorServidorCtrl" class="lps.bet.basico.validadorServidorCtrl.ValidadorServidorCtrl">
8        ...
9        <property name="interfaceProcessarViagem">
10         <xsl:choose>
11           <xsl:when test="formsData/forms/form/form[@variant='Formas de Integracao']/data/combo
12             [@name='solucaoIntegracao']='com componentes' ">
13             <xsl:choose>
14               ...
15               <xsl:when test="formsData/forms/form/form[@variant='Formas de Integracao']/data/combo
16                 [@name='integracao']='tempo, numero de viagens, linha integrada'">
17                 <ref bean="ViagemTempoNumViagensLinhaIntegradaCtrl"/>
18               </xsl:when>
19               <xsl:otherwise>
20                 <ref bean="ViagemCtrl"/>
21               </xsl:otherwise>
22             </xsl:choose>
23           </xsl:when>
24           <xsl:otherwise>
25             <ref bean="ViagemCtrl"/>
26           </xsl:otherwise>
27         </xsl:choose>
28       </property>
29     </bean>
30     ...
31     <xsl:if
32       test="formsData/forms/form/form[@variant='Formas de Integracao']/data/combo[@name='terminal']='existente'">
33       <bean id="TerminalMgr" class="lps.bet.variabilidades.terminalMgr.TerminalMgr">
34         <property name="terminalDAO" ref="TerminalDAO"/>
35       </bean>
36       ...
37     </xsl:if>
38     ...
39   </beans>
40 </xsl:template>
41 </xsl:stylesheet>

```

Figura 6.15: Parte do gabarito bet-servlet.xml.xsl que identifica características de Campo Grande

Caso a aplicação gerada fosse o núcleo, então o componente relacionado à viagem que forneceria a interfaceProcessarViagem seria ViagemCtrl e, caso fosse a aplicação de São Carlos, seria ViagemTempoLinhaIntegradaCtrl. Para esse segundo caso, a parte do arquivo de configuração referente às características da viagem possuiria apenas as características de *Tempo* e de *Linha Integrada*. Assim, não haveria o componente NumViagensMgr definido nas linhas 20-22 e deveria haver o componente ViagemTempoLinhaCtrl no lugar do componente controlador configurado nas linhas 26 a 35.

A partir desses artefatos gerados pelo Captor é possível construir (*build*) o sistema BET para Campo Grande. A aplicação de Campo Grande faz uso apenas dos componentes que estão configurados no bet-servlet.xml e a parte web da aplicação possui uma interface gráfica que identifica a aplicação que foi gerada usando o elemento Nome da Aplicação. Na Figura 6.17 pode ser


```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans>
3   ...
4   <bean class="basico.validadorServidorCtrl.ValidadorServidorCtrl" id="ValidadorServidorCtrl">
5     ...
6     <property name="interfaceProcessarViagem">
7       <ref bean="ViagemTempoNumViagensLinhaIntegradaCtrl"/>
8     </property>
9   </bean>
10  ...
11  <bean id="TerminalMgr" class="variabilidades.terminalMgr.TerminalMgr">
12    <property name="terminalDAO" ref="TerminalDAO"/>
13  </bean>
14  <bean id="EmpresaUsuarialMgr" class="variabilidades.empresaUsuarialMgr.EmpresaUsuarialMgr">
15    <property name="empresaUsuarialDAO" ref="EmpresaUsuarialDAO"/>
16  </bean>
17  <bean id="TempoMgr" class="variabilidades.tempoMgr.TempoDAO">
18    <property name="sessionFactory" ref="sessionFactory"/>
19  </bean>
20  <bean id="NumViagensMgr" class="variabilidades.numViagensMgr.NumViagensDAO">
21    <property name="sessionFactory" ref="sessionFactory"/>
22  </bean>
23  <bean id="LinhaIntegradaMgr" class="variabilidades.linhaIntegradaMgr.LinhaIntegradaDAO">
24    <property name="sessionFactory" ref="sessionFactory"/>
25  </bean>
26  <bean id="ViagemTempoNumViagensLinhaIntegradaCtrl"
27    class="variabilidades.viagemTempoNumViagensLinhaIntegradaCtrl.ViagemTempoNumViagensLinhaIntegradaCtrl">
28    <property name="interfaceTempoMgt" ref="TempoMgr"/>
29    <property name="interfaceNumViagensMgt" ref="NumViagensMgr"/>
30    <property name="interfaceLinhaIntegradaMgt" ref="LinhaIntegradaMgr"/>
31    <property name="interfaceLinhaMgt" ref="LinhaMgr"/>
32    <property name="interfaceCartaoMgt" ref="CartaoMgr"/>
33    <property name="interfaceRegistrarArrecadacao" ref="LinhaMgr"/>
34    <property name="interfaceProcessarViagem" ref="ViagemCtrl"/>
35  </bean>
36  ...
37 </beans>

```

Figura 6.16: Fragmento do arquivo de configuração gerado pelo Captor para Campo Grande

visualizado o site com nível de permissão de um administrador, pois apenas ele pode gerenciar as entidades visualizadas no submenu de Linha. Além disso, pode-se perceber que a aplicação gerada possui algumas variabilidades, como *Linha Integrada* e *Terminal*, que estão listadas no submenu. Também há a característica de *Empresas Usuárias*, pois a consulta pode ser feita considerando o CNPJ de uma empresa. Adicionalmente, ainda há a característica *Acesso Adicional*. Essa funcionalidade é exibida ao se selecionar o link Consultar Viagens, que se encontra relacionado ao cartão de um passageiro.

A aplicação gerada possui também o subsistema cliente que existe nos ônibus. Esse subsistema não é alterado para as diferentes aplicações da LPS-BET. O subsistema implementado é composto de um simulador do validador, como visto na Figura 6.18. Por meio dele podem ser registradas as corridas e realizadas as viagens dos passageiros.

A aplicação passa então por testes sistêmicos para validar a aplicação gerada. Esses testes verificam se todos os requisitos da aplicação de Campo Grande estão presentes na aplicação gerada e se os componentes referentes a características adicionais para Campo Grande estão acoplados

BET Gestão
Campo Grande

Consulta Viação Cartão Funcionário Passageiro Linha Relatórios Sair

Acesso Básico Sistema BET

CPF do Passageiro:
 CNPJ da Empresa:

Dados do Passageiro

Identificador do Passageiro: 6
 Nome: Otavio
 CPF: 998987343

Lista de Cartões

ID	Nome do Passageiro	Tipo	Data de Aquisição	Data de Validade	Saldo	
9	Otavio	estudante	30/12/2007	30/12/2007	0.0	Consultar Viagens
11	Otavio	idoso	30/12/2007	30/12/2007	123.0	Consultar Viagens
3	Otavio	normal	01/01/2005	01/01/2010	0.0	Consultar Viagens

Menu Lateral:

- Linha
- Linha Integrada
- Corrida
- Validador
- Ônibus
- Terminal

Figura 6.17: Aplicação web do sistema BET de Campo Grande

Simulador do Validador do Ônibus

Leitora Cartão:

Viagem permitida.

Figura 6.18: Simulador do validador da LPS-BET

corretamente aos componentes do núcleo da LPS-BET. Além disso, é verificada a integração entre o gerador e a aplicação de Campo Grande, para validar o gabarito utilizado para gerar Campo Grande.

6.3.2 Engenharia de Outras Aplicações

Seguindo o mesmo processo descrito anteriormente para a engenharia da aplicação de Campo Grande foram geradas as outras aplicações-referência da LPS-BET (Fortaleza e São Carlos). A

LPS-BET ainda pode produzir diversas outras aplicações de forma automática usando o Captor, contanto que as suas características façam parte da LPS-BET desenvolvida e a combinação de características seja válida. A seguir é descrita a engenharia de aplicação para o núcleo operacional e para uma aplicação que seja formada por uma combinação de variabilidades na LPS-BET.

Núcleo Operacional

A engenharia de aplicação também pode ser usada para gerar o núcleo, pois ele é operacional. O mesmo processo explicado para a engenharia da aplicação de Campo Grande deve ser seguido, com a diferença de que as características dos formulários devem ser todas selecionadas como “inexistentes”, pois apenas os elementos variantes são mostrados na interface do Captor. Com isso, a aplicação gerada possui apenas as características básicas da LPS-BET. Isso deve ser verificado tanto no arquivo de configuração gerado, quanto na aplicação funcional.

Na Figura 6.19 é mostrada a parte web do núcleo. As características opcionais ou variantes não pertencem ao núcleo, portanto, as variabilidades de Campo Grande que foram mostradas anteriormente não são encontradas no núcleo. Como exemplo dessas características verifica-se no submenu de Linha que não há Linha Integrada e Terminal. Também se percebe pela figura que não há a característica de *Empresas Usuárias*, pois não há a opção da consulta ser feita para as empresas usuárias. Além disso, a característica *Acesso Adicional* é inexistente, pois não há possibilidade de consultar as viagens realizadas pelo passageiro.

ID	Nome do Passageiro	Tipo	Data de Aquisição	Data de Validade	Saldo
9	Otavio	estudante	30/12/2007	30/12/2007	0.0
11	Otavio	idoso	30/12/2007	30/12/2007	123.0
3	Otavio	normal	01/01/2005	01/01/2010	0.0

© Copyright: 2008 Donegan Paula

Figura 6.19: Aplicação web do núcleo da LPS-BET

O subsistema cliente do Ônibus não possui variabilidade e, portanto, para qualquer aplicação gerada que seja da LPS-BET utiliza o mesmo subsistema, cuja interface gráfica foi mostrada na Figura 6.18. Isso ocorre porque o controle de possíveis viagens de integração não é realizada nesse subsistema. Esse controle é feito por um componente controlador que faz parte do servidor do sistema BET.

Combinação Válida Aleatória

Caso fosse necessário produzir uma outra aplicação do domínio BET diferente das aplicações-referência, com base no levantamento de requisitos e nas características da aplicação, seria preciso verificar se a aplicação possui os requisitos existentes na LPS-BET. Para possuir uma combinação válida da linha é preciso inicialmente analisar o diagrama de características da LPS-BET. Em relação ao fluxo de negócio em que essas características são implementadas na LPS-BET, pode-se analisar outros artefatos produzidos no processo de desenvolvimento da LPS-BET, como o diagrama de casos de uso e os diagramas de colaboração. Outra opção é consultar um especialista na LPS-BET.

Como exemplo é mostrada a engenharia de uma aplicação considerando algumas características aleatórias existentes para a linha e que sejam válidas para a LPS-BET. As características para essa aplicação são *Pagamento de Cartão*, *Empresas Usuárias* e *Integração por Tempo* e com *Linha Integrada*. Essas características são inseridas no Captor por meio dos formulários da LPS-BET. Ao salvar essas informações no Captor é gerado o arquivo XML visto na Figura 6.20.

As variabilidades *Pagamento de Cartão* e *Empresas Usuárias* são definidas nas linhas 25 e 36, respectivamente. Há integração apenas por tempo e considerando linhas integradas. Portanto, o elemento variante Terminal não existe (linha 18) e na linha 19 o elemento variante integração recebe o valor “tempo, linha integrada”.

Esse arquivo XML gerado pelo Captor serve de entrada para o processamento dos gabaritos para geração das aplicações da LPS-BET. Um fragmento do artefato bet-servlet.xml gerado é mostrado na Figura 6.21. A característica *Pagamento de Cartão* requer a configuração do componente de negócio *PagamentoCartãoMgr* (linhas 11-13). Além disso, mais três componentes de negócio para variabilidades são configurados: *EmpresaUsuarialMgr*, *TempoMgr* e *LinhaIntegradaMgr* (linhas 14-22). Para realizar a integração considerando tempo e linhas integradas, o componente *ValidadorServidorCtrl* requer a interface *interfaceProcessarViagem* implementada pelo componente *ViagemTempoLinhaIntegradaCtrl* (linha 07). Esse componente é configurado nas linhas 23-31.

A partir da combinação de variabilidades e dos artefatos gerados pelo Captor, pode ser obtida uma nova aplicação da LPS-BET. A parte web desse exemplo é mostrada na Figura 6.22. Percebe-se assim facilmente algumas das suas diferenças e semelhanças em relação ao núcleo e à aplicação de Campo Grande. Essa aplicação possui a variabilidade *Linha Integrada*, visualizada no submenu de *Linha*. Com exceção dessa característica, as características do submenu

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <formsData>
3   <project>
4     <name>CombinacaoAleatoria</name>
5   </project>
6   <forms>
7     <form id="1.1" variant="Variabilidades">
8       <data>
9         <textatt name="1">Combinação Aleatória</textatt>
10      </data>
11     <form id="2.1" variant="Acesso a Informacoes">
12       <data>
13         <combo name="acessoAdicional">inexistente</combo>
14       </data>
15     </form>
16     <form id="3.1" variant="Formas de Integracao">
17       <data>
18         <combo name="terminal">inexistente</combo>
19         <combo name="integracao">tempo, linha integrada</combo>
20         <combo name="solucaoIntegracao">com componentes</combo>
21       </data>
22     </form>
23     <form id="4.1" variant="Aquisicao de Cartao">
24       <data>
25         <combo name="pgtoCartao">existente</combo>
26         <combo name="restricaoCartoes">inexistente</combo>
27       </data>
28     </form>
29     <form id="5.1" variant="Carga de Cartao">
30       <data>
31         <combo name="limitePassagens">inexistente</combo>
32       </data>
33     </form>
34     <form id="6.1" variant="Responsaveis por Cartoes">
35       <data>
36         <combo name="empresasUsuaris">existente</combo>
37       </data>
38     </form>
39   </forms>
40 </formsData>
```

Figura 6.20: Exemplo de Estrutura XML da Especificação de uma combinação aleatória de características no Captor

Linhas são semelhantes ao núcleo, não possuindo Terminal. Além disso, na página web mostrada pode-se perceber a característica Empresas Usuárias, que não existe para o núcleo, mas existe para Campo Grande. A característica de Pagamento de Cartão não é visualizada nessa página, porém, existe para essa aplicação e não é usada para a aplicação de Campo Grande.

6.4 Considerações Finais

Neste capítulo foi apresentada a configuração do Captor para o domínio BET. Essa configuração foi iniciada na fase de transição do ciclo de desenvolvimento do núcleo e foi sendo refinada nas outras fases de transição dos ciclos de desenvolvimento. Para configurar o Captor foi necessário definir os formulários e seus elementos variantes, implementar os gabaritos para gerar as diferen-

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans>
3      ...
4      <bean class="basico.validadorServidorCtrl.ValidadorServidorCtrl" id="ValidadorServidorCtrl">
5          ...
6          <property name="interfaceProcessarViagem">
7              <ref bean="ViagemTempoLinhaIntegradaCtrl"/>
8          </property>
9      </bean>
10     ...
11     <bean id="PagamentoCartaoMgr" class="variabilidades.pagamentoCartaoMgr.PagamentoCartaoMgr">
12         <property name="pagamentoCartaoDAO" ref="PagamentoCartaoDAO"/>
13     </bean>
14     <bean id="EmpresaUsuarialMgr" class="variabilidades.empresausuarialMgr.EmpresaUsuarialMgr">
15         <property name="empresausuarialDAO" ref="EmpresausuarialDAO"/>
16     </bean>
17     <bean id="TempoMgr" class="variabilidades.tempoMgr.TempoDAO">
18         <property name="sessionFactory" ref="sessionFactory"/>
19     </bean>
20     <bean id="LinhaIntegradaMgr" class="variabilidades.linhaIntegradaMgr.LinhaIntegradaDAO">
21         <property name="sessionFactory" ref="sessionFactory"/>
22     </bean>
23     <bean id="ViagemTempoLinhaIntegradaCtrl"
24         class="variabilidades.viagemTempoLinhaIntegradaCtrl.ViagemTempoLinhaIntegradaCtrl">
25         <property name="interfaceTempoMgt" ref="TempoMgr"/>
26         <property name="interfaceLinhaIntegradaMgt" ref="LinhaIntegradaMgr"/>
27         <property name="interfaceLinhaMgt" ref="LinhaMgr"/>
28         <property name="interfaceCartaoMgt" ref="CartaoMgr"/>
29         <property name="interfaceRegistrarArrecadacao" ref="LinhaMgr"/>
30         <property name="interfaceProcessarViagem" ref="ViagemCtrl"/>
31     </bean>
32     ...
33 </beans>

```

Figura 6.21: Fragmento do arquivo de configuração gerado pelo Captor para aplicação com combinação de características da LPS-BET

tes características da LPS-BET escolhendo os componentes que devem fazer parte da aplicação e como eles devem ser acoplados.

A configuração implementada em incrementos permitiu validar o uso do gerador ainda no começo e auxiliar na verificação dos requisitos implementados, antecipando a correção de alguns defeitos que outrora poderiam ser verificados apenas na engenharia de aplicações.

Também foi mostrado como é feita a engenharia de aplicações da LPS-BET, usando como exemplos o núcleo, a aplicação-referência de Campo Grande e uma aplicação diferente com uma combinação válida de variabilidades da LPS-BET. Dessa forma foi finalizado o processo de desenvolvimento da LPS-BET utilizando o processo ESPLEP adaptado, que facilitou a engenharia de aplicações.

Uma das dificuldades encontradas durante a configuração do Captor foi em relação à linguagem XSL utilizada. Na época o Captor foi desenvolvido usando uma versão inicial da linguagem e influenciando na estrutura da LMA desenvolvida. Por esse motivo, a LMA não é muito flexível e o código fica menos legível, dificultando manutenções. Além disso, as funcionalidades do Captor podem dificultar a configuração de alguns relacionamentos entre variabilidades, pois não há grande

The screenshot shows the BETGestão web application interface. At the top, there is a header with the logo 'BETGestão' and the subtitle 'Combinação Aleatória'. Below the header is a navigation bar with tabs: Consulta, Viação, Cartão, Funcionário, Passageiro, Linha, Relatórios, and Sair. The 'Linha' tab is selected, showing a dropdown menu with options: Linha, Linha Integrada, Corrida, Validador, and Ônibus. Below the navigation bar, there is a section titled 'Acesso Básico Sistema BET' with input fields for 'CPF do Passageiro:' and 'CNPJ da Empresa:', and a 'Buscar' button. Below this, there is a section titled 'Dados do Passageiro' displaying the following information: Identificador do Passageiro: 6, Nome: Otavio, and CPF: 998987343. At the bottom, there is a section titled 'Lista de Cartões' displaying a table with the following data:

ID	Nome do Passageiro	Tipo	Data de Aquisição	Data de Validade	Saldo
9	Otavio	estudante	30/12/2007	30/12/2007	0.0
11	Otavio	idoso	30/12/2007	30/12/2007	123.0
3	Otavio	normal	01/01/2005	01/01/2010	0.0

Figura 6.22: Aplicação web para uma combinação aleatória de características da LPS-BET

variedade de elementos de formulários. Apesar de que Shimabukuro (2006) sugere que outros elementos podem ser facilmente implementados para usar no Captor.

Conclusão

7.1 Considerações Iniciais

O trabalho realizado divide-se em duas fases: o desenvolvimento dos ativos centrais da LPS-BET e o uso do Captor para a geração de aplicações da LPS-BET. Na primeira fase foi seguido um processo de desenvolvimento iterativo e incremental. Os ativos centrais foram obtidos em ciclos de desenvolvimento incrementais que desenvolveram o núcleo e três aplicações-referência da LPS-BET. Ao longo desse desenvolvimento foram investigadas questões relacionadas ao processo de desenvolvimento em si, ao projeto da LPS usando componentes caixa-preta versus caixa-branca e ao uso de aspectos para representar variabilidades além de requisitos tipicamente transversais. Para realizar a engenharia de aplicações de forma automatizada foi usado o gerador de aplicações configurável Captor. Ele foi configurado para as variabilidades da LPS-BET em incrementos a cada ciclo de desenvolvimento, desenvolvendo uma linguagem de modelagem de aplicações (LMA) da LPS-BET. Na engenharia de aplicações puderam ser geradas diversas aplicações pela seleção de variabilidades com combinações válidas para a LPS-BET. Essas variabilidades são especificadas pela LMA e a partir dela o gerador de aplicações gera artefatos.

7.2 Contribuições

A principal contribuição deste trabalho é a proposta da adaptação do processo ESPLEP (Gomaa, 2004) com o objetivo de ter um processo mais ágil, de projetar e desenvolver características com re-trabalho mínimo e de facilitar a engenharia de aplicações com as seguintes propriedades:

engenharia de domínio iterativa e incremental, desenvolvimento da LPS usando engenharia avante, incrementos baseados em aplicações-referência, arquitetura da LPS baseada em componentes e aplicações geradas por um gerador de aplicações configurável a partir de uma LMA baseada no modelo de características.

Adicionalmente, foi realizado um estudo detalhado de alternativas para projeto de componentes caixa-preta com o objetivo de facilitar a composição de componentes e reusar os componentes sem modificá-los. Com isso, apresentou-se uma proposta de como fazer e compôr componentes caixa-preta com base no modelo de características e no diagrama de classes. Também, investigou-se como aspectos podem ser usados com componentes para implementar requisitos não-funcionais que são transversais e variabilidades funcionais e não-funcionais. Apresentou-se uma comparação entre duas forma de implementação: usando apenas programação orientada a objetos e outra usando também programação orientada a aspectos. Além disso, percebeu-se a facilidade de gerar uma solução ou outra, dependendo das necessidades e interesses do engenheiro de aplicações.

Finalmente, há a contribuição de produção de uma LPS praticamente completa e não trivial, que será usada pelo grupo de pesquisa para apoiar outras pesquisas.

7.3 Trabalhos Futuros

A partir do desenvolvimento da LPS foi possível disponibilizar uma linha de produtos de software para uso como apoio a outros estudos. Atualmente, a linha de produtos de software pode ser usada para ensino em cursos de graduação e pós-graduação referentes ao desenvolvimento de linhas de produtos de software e a arquiteturas baseadas em componentes e em aspectos. Os estudantes podem realizar trabalhos tais como instanciar aplicações da LPS-BET e implementar novas variabilidades da LPS-BET.

Além disso, a LPS desenvolvida pode ser utilizada para outras pesquisas. Inicialmente será completado o desenvolvimento da variabilidade da linha que ainda falta. Para isso será feita a implementação dos componentes da variabilidade e a configuração do gerador para obter aplicações que tenham essa variabilidade.

Como trabalho futuro, deseja-se desenvolver uma versão da LPS-BET com todas as variabilidades implementadas por aspectos. A partir dessa nova versão será feita uma análise mais aprofundada da representação de variabilidades usando aspectos. Serão coletadas métricas e executados experimentos para comparar as duas soluções (Figueiredo *et al.*, 2008).

Adicionalmente, pretende-se pesquisar sobre testes em linhas de produtos de Software, analisando como os testes de regressão devem ser feitos e qual deve ser o escopo dos testes ao adicionar uma nova variabilidade. Além disso, almeja-se investigar o uso de “*build-in testing*” para testes dos componentes de uma LPS (Gross, 2005; Wang *et al.*, 1999).

Referências Bibliográficas

- ALDRICH, J. Open Modules: A Proposal for Modular Reasoning in Aspect-Oriented Programming. In: *FOAL 2004 Proceedings - Foundations of Aspect-Oriented Languages, Workshop at AOSD*, Lancaster, UK, 2004, p. 7–18.
- ANASTASOPOULOS, M.; GACEK, C. Implementing Product Line Variabilities. *ACM Sigsoft Software Engineering Notes*, v. 23, n. 3, p. 109–117, 2001.
- ANASTASOPOULOS, M.; MUTHIG, D. An Evaluation of Aspect-Oriented Programming as a Product Line Implementation Technology. In: *Proceedings of the 8th International Conference Software Reuse - ICSR 2004*, Springer, Madrid, Spain, 2004, p. 141–156.
- APACHE The Apache Maven Project. Acessado em junho, 2008a.
Disponível em: <http://maven.apache.org/>
- APACHE The Apache Velocity Project. Acessado em junho, 2008b.
Disponível em: <http://velocity.apache.org/>
- APEL, S.; BATORY, D. When to Use Features and Aspects? A Case Study. In: *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, ACM Press New York, NY, USA, 2006, p. 59–68.
- APEL, S.; LEICH, T.; SAAKE, G. Aspectual Mixin Layers: Aspects and Features in Concert. In: *Proceedings of the 28th International Conference on Software Engineering - ICSE*, ACM New York, Shanghai, China, 2006, p. 122–131.
- ASPECTJ TEAM The AspectJ Programming Guide. Acessado em junho, 2008.
Disponível em: <http://www.eclipse.org/aspectj/doc/released/progguide/>
- ATKINSON, C.; BAYER, J.; BUNSE, C.; LAITENBERGER, O.; LAQUA, R.; KAMSTIES, E.; MUTHIG, D.; PAECH, B.; WÜST, J.; ZETTEL, J. *Component-based Product Line Engineering with UML*. Component Series, Addison-Wesley, 464 p., 2001.

- ATKINSON, C.; BAYER, J.; MUTHIG, D. Component-Based Product Line Development: The KobrA Approach. In: *Proceedings of the 1st Software Product Line Conference*, Denver, USA, 2000, p. 28–31.
- ATKINSON, C.; MUTHIG, D. Enhancing Component Reusability through Product Line Technology. In: *Proceedings of the 7th International Conference in Software Reuse (ICSR 02): Methods, Techniques and Tools*, Springer, Austin, USA, 2002, p. 93–108.
- BACHMANN, F.; GOEDICKE, M.; LEITE, J.; NORD, R.; POHL, K.; RAMESH, B.; VILBIG, A. 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, 2003, p. 66–80.
- BANIASSAD, E.; CLEMENTS, P.; ARAUJO, J.; MOREIRA, A.; RASHID, A.; TEKINERDOGAN, B. Discovering Early Aspects. *Software, IEEE*, v. 23, n. 1, p. 61–70, 2006.
- BATISTA, T.; CHAVEZ, C.; GARCIA, A.; RASHID, A.; SANT’ANNA, C.; KULESZA, U.; CASTOR FILHO, F. Reflections on Architectural Connection: Seven Issues on Aspects and ADLs. In: *Proceedings of the 2006 International Workshop on Early Aspects at ICSE*, ACM Press New York, Shangai, China, 2006a, p. 3–10.
- BATISTA, T. V.; CHAVEZ, C.; GARCIA, A.; SANT’ANNA, C.; KULESZA, U.; LUCENA, C. Aspectual Connectors: Supporting the Seamless Integration of Aspects and ADLs. *Simpósio Brasileiro de Engenharia de Software - SBES 2006*, p. 17–32, Florianópolis, Brasil, 2006b.
- BAYER, J.; FLEGE, O.; KNAUBER, P.; LAQUA, R.; MUTHIG, D.; SCHMID, K.; WIDEN, T.; DEBAUD, J. PuLSE: a Methodology to Develop Software Product Lines. In: *Proceedings of the 1999 ACM SIGSOFT Symposium on Software Reusability*, ACM Press, Los Angeles, USA, 1999, p. 122–131.
- BAYER, J.; GACEK, C.; MUTHIG, D.; WIDEN, T. PuLSE-I: Deriving Instances from a Product Line Infrastructure. In: *Proceedings of 7th IEEE International Conference and Workshop on the Engineering of Computer Based Systems - ECBS*, 2000, p. 237–245.
- BECKER, M.; KAISERSLAUTERN, G. 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, 2003, p. 9.
- BIGGERSTAFF, T.; PERLIS, A. *Software Reusability: Vol. 1, Concepts and Models*. ACM Press New York, NY, USA, 425 p., 1989.
- BINKLEY, D.; CECCATO, M.; HARMAN, M.; RICCA, F.; TONELLA, P. Tool-Supported Refactoring of Existing Object-Oriented Code into Aspects. *IEEE Transactions on Software Engineering*, v. 32, n. 9, p. 698–717, 2006.

- BOEHM, B. A Spiral Model of Software Development and Enhancement. *ACM SIGSOFT Software Engineering Notes*, v. 11, n. 4, p. 22–42, 1986.
- BOSCH, J.; FLORIJN, G.; GREEFHORST, D.; KUUSELA, J.; OBBINK, J.; POHL, K. Variability Issues in Software Product Lines. *Software Product-Family Engineering: 4th International Workshop - PFE 2001*, p. 11–19, Bilbao, Spain, 2002.
- BRAGA, R.; MASIERO, P. Building a wizard for framework instantiation based on a pattern language. *Lecture Notes on Computer Science*, v. 2817, p. 95–106, 2003.
- BRUNETON, E.; COUPAYE, T.; LECLERCQ, M.; QUEMA, V.; STEFANI, J. An Open Component Model and its Support in Java. *Lecture Notes of the 2004 International Symposium on Component-Based Software Engineering*, v. 3054, p. 7–22, 2004.
- CAMARGO, V. V. D. *Frameworks Transversais: Definições, Classificações, Arquitetura e Utilização em um Processo de Desenvolvimento de Software*. Tese de Doutorado, Instituto de Ciências Matemáticas e de Computação - ICMC, Universidade de São Paulo (USP), 2006.
- CHEESMAN, J.; DANIELS, J. *UML Components: A Simple Process for Specifying Component-Based Software*. Addison-Wesley Boston, 208 p., 2001.
- CIBRAN, M.; D'HONDT, M.; JONCKERS, V. Aspect-Oriented Programming for Connecting Business Rules. In: *Proceedings of the 6th International Conference on Business Information Systems - BIS'03*, Colorado, USA, 2003, p. 24.
- CLARKE, S.; BANIASSAD, E. *Aspect-Oriented Analysis and Design: The Theme Approach*. Addison-Wesley Professional, 400 p., 2005.
- CLEAVELAND, J. C. Building application generators. *Software, IEEE*, v. 5, n. 4, p. 25–33, 1988.
- CLEMENTE, P.; HERNÁNDEZ, J. Aspect Component Based Software Engineering. In: *Proceedings of the 2nd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software - ACP4IS*, Boston, USA, 2003, p. 39–42.
- CLEMENTE, P.; SÁNCHEZ, F.; PÉREZ, M. Modeling with UML Component-Based and Aspect Oriented Programming Systems. In: *Proceedings of the 7th International Workshop on Component-Oriented Programming - WCOP'02 at the European Conference on Object Oriented Programming (ECOOP'02)*, Malaga, Spain, 2002a, p. 7.
- CLEMENTE, P. J.; HERNANDEZ, J.; MURILLO, J. M.; PEREZ, M. A.; SANCHEZ, F. AspectCCM: an aspect-oriented extension of the Corba Component Model. In: *Proceedings of the 28th Euromicro Conference*, Dortmund, Germany, 2002b, p. 10–16.
- CLEMENTS, P.; NORTHROP, L. *Software Product Lines: Practices and Patterns*. Addison-Wesley Boston, 576 p., 2001.

- COTTENIER, T.; ELRAD, T. Validation of Context-Dependent Aspect-Oriented Adaptations to Components. In: *Proceedings of the Workshop on Component-Oriented Programming - WCOP 2004 at ECOOP 2003*, Darmstadt, Germany, 2004, p. 7.
- COUTO, C. F. M.; VALENTE, M. T. O.; BIGONHA, R. S. Um Arcabouço Orientado por Aspectos para Implementação Automatizada de Persistência. In: *Anais do 2o. Workshop Brasileiro de Desenvolvimento Orientado a Aspectos - WASP'05 no XIX Simpósio Brasileiro de Engenharia de Software - SBES*, Uberlândia, Brasil, 2005, p. 8.
- CRNKOVIC, I.; LARSSON, M. Component-based Software Engineering - New Paradigm of Software Development. In: *Invited talk and report, MIPRO 2001 proceedings*, Opatija, Croatia, 2001, p. 523–524.
- Disponível em: <http://www.mrtc.mdh.se/index.php?choice=publications&id=0293>
- CUESTA, C.; PILAR ROMAY, M.; FUENTE, P.; BARRIO-SOLORZANO, M. Architectural Aspects of Architectural Aspects. *2nd European Workshop on Software Architecture (EWSA), Lecture Notes in Computer Science*, v. 3527, p. 247–262, 2005.
- CZARNECKI, K.; EISENECKER, U. Components and Generative Programming. *ACM SIGSOFT*, v. 24, n. 6, p. 2–19, 1999.
- DE FRAINE, B.; SÜDHOLT, M.; JONCKERS, V. StrongAspectJ: flexible and safe pointcut/advice bindings. In: *Proceedings of the 7th International Conference on Aspect-Oriented Software Development - AOSD*, ACM New York, NY, USA, 2008, p. 60–71.
- DIJKSTRA, E. *A Discipline of Programming*. Prentice Hall Series in Automatic Computation, 217 p., Upper Saddle River, USA, 1976.
- D'SOUZA, D.; WILLS, A. *Objects, Components, and Frameworks with UML: the Catalysis approach*. Addison-Wesley Longman Publishing Co., Inc., 785 p., Boston, USA, 1998.
- DUCLOS, F.; ESTUBLIER, J.; MORAT, P. Describing and using non functional aspects in component based applications. In: *Proceedings of the 1st International Conference on Aspect-Oriented Software Development - AOSD*, ACM Press New York, Enschede, The Netherlands, 2002, p. 65–75.
- ECLIPSE The Eclipse Foundation. Acessado em junho, 2008.
- Disponível em: <http://www.eclipse.org/>
- ELER, M. M. *Um Método para o Desenvolvimento de Software baseado em Componentes e Aspectos*. Dissertação de Mestrado, Instituto de Ciências Matemáticas e de Computação - ICMC, Universidade de São Paulo (USP), 2006.

- ESTUBLIER, J.; FAVRE, J. Component Models and Technology. *Building Reliable Component-Based Software Systems*, p. 57–86, 2002.
- FAVARO, J.; FAVARO, K.; FAVARO, P. Value Based Software Reuse Investment. *Annals of Software Engineering*, v. 5, p. 5–52, 1998.
- FIGUEIREDO, E.; CACHO, N.; SANTANNA, C.; MONTEIRO, M.; KULESZA, U.; GARCIA, A.; SOARES, S.; FERRARI, F.; KHAN, S.; FILHO, F.; DANTAS, F. Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. In: *Proceedings of the 30th International Conference on Software Engineering - ICSE*, ACM New York, Leipzig, Germany, 2008, p. 261–270.
- FILMAN, R.; FRIEDMAN, D. Aspect-Oriented Programming is Quantification and Obliviousness. *Workshop on Advanced Separation of Concerns at OOPSLA 2000*, p. 7, Minneapolis, USA, 2000.
- FINKELSTEIN, A.; KRAMER, J. Software Engineering: A Roadmap. In: *Proceedings of the Conference on the Future of Software Engineering - FoSE*, ACM Press New York, NY, USA, 2000, p. 3–22.
- FOWLER, M. *UML Distilled : a Brief Guide to the Standard Object Modeling Language*. 3a ed. Addison-Wesley Reading, 185 p., 2004.
- FRAKES, W.; ISODA, S. Success Factors of Systematic Reuse. *Software, IEEE*, v. 11, n. 5, p. 14–19, 1994.
- FRAKES, W.; KANG, K. Software Reuse Research: Status and Future. *IEEE Transactions on Software Engineering*, v. 31, n. 7, p. 529–536, 2005.
- FREEMAN, P. Reusable Software Engineering: Concepts and Research Directions. *IEEE Tutorial Software Reusability*, p. 10–25, 1987.
- GARLAN, D.; MONROE, R.; WILE, D. Acme: An Architecture Description Interchange Language. In: *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research - Cascon*, IBM Press, Ontario, Canada, 1997, p. 169–183.
- GARLAN, D.; MONROE, R.; WILE, D. Acme: Architectural Description of Component-Based Systems. *Foundations of Component-Based Systems*, p. 47–68, 2000.
- GOMAA, H. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison-Wesley Boston, 736 p., 2004.
- GOOGLE Google Code. Acessado em junho, 2008.
Disponível em: <http://code.google.com/>

- GRISS, M. Implementing Product-Line Features by Composing Component Aspects. *First International Software Product-Line Conference*, p. 271–288, Denver, USA, 2000.
- GRISS, M. CBSE Success Factors: Integrating Architecture, Process, and Organization. *Component-Based Software Engineering: Putting the Pieces Together*, p. 143–160, 2001a.
- GRISS, M. Product-Line Architectures. *Component-Based Software Engineering: Putting the Pieces Together*, p. 405–420, 2001b.
- GRISS, M.; FAVARO, J.; D’ALESSANDRO, M. Integrating Feature Modeling with the RSEB. In: *Proceedings of the Fifth International Conference on Software Reuse*, Victoria, Canada, 1998, p. 76–85.
- GROSS, H.-G. *Component-Based Software Testing with UML*. Springer, 340 p., 2005.
- GRUNDY, J.; PATEL, R. Developing Software Components with the UML, Enterprise Java Beans and Aspects. In: *Proceedings of the 2001 Australian Software Engineering Conference*, Canberra, Australia, 2001, p. 26–29.
- HEO, S.; CHOI, E. Representation of Variability in Software Product Line Using Aspect-Oriented Programming. In: *Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications - SERA 2006*, IEEE Computer Society, Washington, USA, 2006, p. 66–73.
- HIBERNATE Hibernate - Relational Persistence for Java and .NET. Acessado em junho, 2008.
Disponível em: <http://www.hibernate.org/>
- JACOBSON, I.; BOOCH, G.; RUMBAUGH, J. *The Unified Software Development Process*. Addison-Wesley Longman Publishing Co., Inc., 512 p., Boston, USA, 1999.
- JOHNSON, R. Documenting frameworks using patterns. In: *Proceedings of the Conference on Object-oriented programming systems, languages, and applications - OOPSLA*, ACM New York, Vancouver, Canada, 1992, p. 63–76.
- JOHNSON, R. Components, Frameworks, Patterns. In: *Proceedings of the 1997 Symposium on Software Reusability, ACM SIGSOFT Software Engineering Notes*, ACM Press New York, Boston, USA, 1997, p. 10–17.
- JUDE Jude UML Modeling Tool. Acessado em junho, 2008.
Disponível em: <http://jude.change-vision.com/jude-web/index.html>
- JUNIOR, E. A. O.; GIMENES, I. M. S.; HUZITA, E. H. M.; MALDONADO, J. C. A. Variability Management Process for Software Product Lines. In: *Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research - Cascon*, 2005, p. 225–241.

JUNIT JUnit.org - Resources for Test Driven Development. Acessado em junho, 2008.

Disponível em: <http://www.junit.org/>

KANG, K.; KIM, S.; LEE, J.; KIM, K.; SHIN, E.; HUH, M. FORM: A Feature-Oriented Reuse Method with Domain-specific Reference Architectures. *Annals of Software Engineering*, v. 5, p. 143–168, 1998.

KANG, K.; *et al.* *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-021, Carnegie Mellon University, Software Engineering Institute, 161 p., 1990.

KASTNER, C.; APEL, S.; BATORY, D. A Case Study Implementing Features Using AspectJ. In: *Proceedings of the 11th International Software Product Line Conference - SPLC*, Kyoto, Japan, 2007, p. 223–232.

KICZALES, G.; HILSDALE, E.; HUGUNIN, J.; KERSTEN, M.; PALM, J.; GRISWOLD, W. An Overview of AspectJ. In: *Proceedings of the 15th European Conference on Object-Oriented Programming, Lecture Notes in Computer Science*, Springer-Verlag, London, UK, 2001, p. 327–353.

KICZALES, G.; LAMPING, J.; MENDHEKAR, A.; MAEDA, C.; LOPES, C. V.; LOINGTIER, J.-M.; IRWIN, J. Aspect-Oriented Programming. In: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science 1241*, Springer-Verlag, Finland, 1997.

KISELEV, I. *Aspect-Oriented Programming with AspectJ*. Sams, 288 p., Indianapolis, USA, 2002.

KRUCHTEN, P. *The Rational Unified Process: An Introduction*. 2a ed. Addison-Wesley, 320 p., 2000.

KRUEGER, C. Software reuse. *ACM Computing Surveys (CSUR)*, v. 24, n. 2, p. 131–183, New York, USA, 1992.

KRUEGER, C. Easing the Transition to Software Mass Customization. In: *Proceedings on the 4th International Workshop Software Product-Family Engineering: - PFE 2001: Revised Papers*, Springer, Bilbao, Spain, 2002.

LADDAD, R. *AspectJ in Action: Practical Aspect-oriented Programming*. Manning, 512 p., 2003.

LEE, K.; KANG, K.; KIM, M.; PARK, S. Combining Feature-Oriented Analysis and Aspect-Oriented Programming for Product Line Asset Development. In: *Proceedings of 10th International Software Product Line Conference - SPLC 2006*, Baltimore, USA, 2006, p. 103–112.

- LIEBERHERR, K.; LORENZ, D.; MEZINI, M. *Programming with Aspectual Components*. Relatório Técnico NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA 02115, 1999.
- Disponível em: <http://www.ccs.neu.edu/home/lorenz/papers/reports/NU-CCS-99-01.html>
- LOPES, C. *D: A Language Framework for Distributed Programming*. Tese de Doutorado, Northeastern University, 1997.
- MCVEIGH, A. The Rich Engineering Heritage Behind Dependency Injection. Acessado em junho, Architect Zone, 2008.
- Disponível em: <http://architects.dzone.com/articles/rich-engineering-heritage-behi>
- MEDVIDOVIC, N.; TAYLOR, R. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, v. 26, n. 1, p. 70–93, 2000.
- MEYER, B. .NET is Coming [Microsoft Web Services Platform]. *Computer*, v. 34, n. 8, p. 92–97, 2001.
- MEZINI, M.; OSTERMANN, K. Variability Management with Feature-Oriented Programming and Aspects. In: *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM Press New York, California, USA, 2004, p. 127–136.
- OMMERING, R. Building Product Populations with Software Components. *Proceedings of the 24rd International Conference on Software Engineering - ICSE 2002*, p. 255–265, Orlando, USA, 2002.
- OSSHERR, H.; TARR, P. *Multi-dimensional Separation of Concerns in Hyperspace*. IBM TJ Watson Research Center, RC 21452(96717)16APR99, 1999.
- OSSHERR, H.; TARR, P. Using Multidimensional Separation of Concerns to (Re)Shape Evolving Software. *Communications of the ACM*, v. 44, n. 10, p. 43–50, 2001.
- OSTERWEIL, L. J. A Future For Software Engineering? In: *Proceedings of the 2nd Conference on the Future of Software Engineering - FoSE' 07*, IEEE Computer Society, Washington, USA, 2007, p. 1–11.
- PARNAS, D. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering* 5, 2, p. 128–138, 1979.

- PÉREZ, J.; RAMOS, I.; JAÉN, J.; LETELIER, P.; NAVARRO, E. PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures. In: *Proceedings of the 3rd IEEE International Conference on Quality Software - QSIC 2003*, IEEE Computer Society, Dallas, USA, 2003, p. 59–66.
- PESSEMIER, N.; SEINTURIER, L.; COUPAYE, T.; DUCHIEN, L. A Model for Developing Component-Based and Aspect-Oriented Systems. *5th International Symposium on Software Composition*, v. 4089, p. 259–274, 2006.
- PINTO, M.; FUENTES, L.; TROYA, J. A Dynamic Component and Aspect-Oriented Platform. *The Computer Journal*, v. 48, n. 4, p. 401–420, 2005.
- POHL, K.; BÖCKLE, G.; LINDEN, F. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 468 p., 2005.
- POSTGRESQL PostgreSQL Global Development Group. Acessado em junho, 2008.
Disponível em: <http://www.postgresql.org/>
- PREE, W. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley Longman, 268 p., Reading, Mass, 1995.
- PREE, W. Component-based Software Development - A New Paradigm in Software Engineering? *Software - Concepts and Tools*, v. 18, n. 4, p. 169–174, 1997.
- PRESSMAN, R. *Software engineering: A practitioner's approach*. 5a ed. McGraw-Hill, Inc., 880 p., new York, USA, 2002.
- PRIETO-DIAZ, R.; ARANGO, G. *Domain Analysis and Software Systems Modeling*. IEEE Computer Society Press, 312 p., Los Alamitos, CA, USA, 1991.
- RASHID, A.; CHITCHYAN, R. Persistence as an Aspect. In: *Proceedings of the 2nd International Conference on Aspect-oriented Software Development*, ACM Press New York, Boston, USA, 2003, p. 120–129.
- ROBERTS, D.; JOHNSON, R.; *et al.* Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks. *Pattern Languages of Program Design*, v. 3, p. 15, 1997.
- SAUER, F. Metrics 1.3.6 - Getting Started. Acessado em junho, 2008.
Disponível em: <http://metrics.sourceforge.net/>
- SCHIMABUKURO, E. K. J.; MASIERO, P. C.; BRAGA, R. T. V. Captor: Um Gerador de Aplicações Configurável. In: *Anais do XIII Sessão de Ferramentas do Simpósio Brasileiro de Engenharia de Software*, Florianópolis, Brasil, 2006, p. 6.

- SEI Arcade Game Maker Pedagogical Product Line. Acessado em junho, Software Engineering Institute - Carnegie Mellon, 2008.
Disponível em: <http://www.sei.cmu.edu/productlines/pp1/>
- SHAW, M.; DELINE, R.; KLEIN, D.; ROSS, T.; YOUNG, D.; ZELESNIK, G. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, v. 21, n. 4, p. 314–335, 1995.
- SHIMABUKURO, E. K. J. *Um Gerador de Aplicações Configurável*. Dissertação de Mestrado, Instituto de Ciências Matemáticas e de Computação - ICMC, Universidade de São Paulo (USP), 2006.
- SOARES, S. C. B. *An Aspect-Oriented Implementation Method*. Tese de Doutorado, Centro de Informática - CIn, Universidade Federal de Pernambuco (UFPE), 2004.
- SPRING Spring Framework. Acessado em junho, 2008.
Disponível em: <http://www.springframework.org/>
- SUVÉE, D.; FRAINE, B. D.; VANDERPERREN, W. A Symmetric and Unified Approach Towards Combining Aspect-Oriented and Component-Based Software Development. In: *Component-Based Software Engineering*, Springer Berlin / Heidelberg, Lecture Notes in Computer Science, 2006, p. 114–122.
- SUVÉE, D.; VANDERPERREN, W.; JONCKERS, V. JAsCo: An Aspect-Oriented Approach Tailored for Component Based Software Development. In: *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development - AOSD*, ACM Press New York, Boston, USA, 2003, p. 21–29.
- SUVEE, D.; VANDERPERREN, W.; WAGELAAR, D.; JONCKERS, V. There Are No Aspects. In: *Proceedings of the Software Composition Workshop - SC 2004*, Elsevier, Electronic Notes in Theoretical Computer Science, 2005, p. 153–174.
- SZYPERSKI, C. *Component Software: Beyond Object-Oriented Programming*. 2a ed. Addison-Wesley, Harlow, England, 624 p., 2002.
- TARR, P.; OSSHER, H.; HARRISON, W.; SUTTON JR, S. M. N Degrees of Separation: Multidimensional Separation of Concerns. 21st Int. In: *Proceedings of the 1999 International Conference on Software Engineering - ICSE*, ACM, Los Angeles, USA, 1999, p. 107–119.
- TAYLOR, R.; HOEK, A. Software Design and Architecture: The Once and Future Focus of Software Engineering. In: *Proceedings of the International Conference on Software Engineering - ICSE*, IEEE Computer Society Washington, DC, USA, 2007, p. 226–243.
- TIGRIS Tortoise SVN. Acessado em junho, 2008.
Disponível em: <http://tortoisesvn.tigris.org/>

- VAN OMMERING, R.; BOSCH, J. Components in Product-Line Architectures. *Building Reliable Component-Based Software Systems*, p. 207–221, 2002.
- VESTAL, S. A cursory Overview and Comparison of Four Architecture Description Languages. *Technical Report, Minneapolis*, p. 10, 1993.
- VILLELA, R. M. M. B. *Busca e Recuperação de Componentes em Ambientes de Reutilização de Software*. Tese de Doutorado, COPPE, Universidade Federal do Rio de Janeiro (UFRJ), 2000.
- W3C Architecture Domain - The Extensible Stylesheet Language Family (XSL). Acessado em junho, 2008.
Disponível em: <http://www.w3.org/Style/XSL/>
- WANG, Y.; KING, G.; WICKBURG, H. A method for built-in tests in component-based software maintenance. In: *Proceedings of the 3rd European Conference on Software Maintenance and Reengineering - CSMR '99*, Washington, DC: IEEE Computer Society, 1999, p. 186.
- WEISS, D.; LAI, C. *Software Product-line Engineering: a Family-based Software Development Process*. Addison-Wesley Longman Publishing Co., Inc., 448 p., Boston, USA, 1999.
- WU, J.; HSIA, T.; CHANG, I.; TSAI, S. Application Generator: A Framework and Methodology for IS Construction. In: *Proceedings of the 36th Hawaii International Conference on System Sciences, HICSS-36*, IEEE, Hawaii, 2002, p. 10.
- ZAND, M.; SAMADZADEH, M. Software Reuse Issues and Perspectives. *Potentials, IEEE*, v. 13, n. 3, p. 15–19, 1994.

Documento de Requisitos do Sistema BET de Fortaleza

Introdução

Propósito

Este documento descreve os requisitos de software para o sistema de controle de Bilhetes Eletrônicos para Transporte (BET) no município de Fortaleza. Destina-se ao projetista, desenvolvedor e administrador do BET.

Escopo

A função do BET é dar apoio computadorizado à rede de ônibus da cidade de Fortaleza.

Visão Geral

O restante deste documento está organizado como segue: inicialmente, definem-se alguns termos importantes para entendimento do documento na Seção A. A Seção A contém uma descrição geral do BET. A Seção A identifica os requisitos funcionais específicos, as interfaces externas e os requisitos de desempenho do BET.

Definições

- **Cartão:** Cartão eletrônico (tecnologia SmartCard) que permite a recarga de valores em dinheiro para serem utilizados no transporte municipal.
- **Passageiro:** Pessoa que utiliza o sistema de transporte para realizar viagens de ônibus (na própria cidade). Existem categorias que possuem desconto no valor da passagem, como por exemplo estudantes. Outros pagam valor integral.
- **Ônibus:** Um veículo de transporte que possui uma leitora (dispositivo para leitura do cartão e comunicação com o sistema da empresa). Essa comunicação é on-line e ocorre via um sistema de radiofrequência (RFID) ou tecnologia semelhante. (Detalhes desta comunicação podem ser abstraídos para os propósitos deste projeto)
- **Linha:** Trecho percorrido pelo ônibus desde o ponto-inicial até o ponto-final. Cada linha tem diversos horários de saída do ponto-inicial e é identificada por um nome e um código.
- **Recarga:** O cartão é adquirido com número definido de passagens, convertidas em reais, e pode ser carregado mensalmente com um número máximo de passagens.
- **Validador:** É um equipamento instalado em todos os ônibus. Ela é programável e é composta de um visor de LCD para emitir pequenas mensagens ao passageiro, um atuador para liberar a catraca e um leitor de cartão.
- **Terminal:** Local de embarque e desembarque de passageiros de ônibus, tal que possa ser feita a integração entre diversas linhas sem que haja pagamento de uma nova passagem. A entrada em um terminal também pode ser feita pelo uso do validador.

Descrição Geral

Perspectiva do Produto

O sistema BET não trabalha independentemente. Ele depende das informações centralizadas na empresa viária.

Funções do Produto

O software deve permitir que a rede BET trabalhe de forma computadorizada. A empresa viária possui um computador central que mantém os dados de todos os passageiros, cartões, linhas, ônibus, etc. O validador instalado em um ônibus ou em um terminal aceita um cartão como entrada e comunica-se com o computador da empresa viária por rádio para realizar o débito da passagem no cartão do passageiro.

Existe um sistema de terminais de integração de ônibus, que permite ao usuário pagar um único valor de passagem, desde que as viagens ocorram em ônibus que participem dos terminais de integração e a mudança de ônibus seja feita dentro dos terminais, não sendo necessário passar o cartão novamente no validador.

Existem também computadores espalhadas nas agências da empresa de transportes que permitem ao passageiro fazer consulta do seu saldo no cartão.

Características do Usuário

O **passageiro** interage com o sistema BET via uma leitora instalada no ônibus. É desejável que seja bem fácil usar a leitora mas, em caso de dúvida, deve haver algum tipo de suporte ao passageiro. Além disso, o passageiro pode usar terminais disponíveis nas agências da empresa de transportes para consultar as viagens realizadas no período.

O **atendente** é a pessoa responsável pelo registro de passageiros no sistema, assim como pela criação e carga de cartões para os passageiros.

O **administrador** é a pessoa responsável por manter o sistema, por exemplo, conectar uma nova leitora em um ônibus ou reiniciar a leitora quando necessário.

Requisitos Específicos

Requisitos Funcionais

R1 - O sistema deve permitir ao passageiro utilizar seu cartão para pagar uma viagem feita em ônibus. O cartão pode ser usado nas leitoras dos validadores existentes tanto nos ônibus quanto nos terminais. O cartão armazena um identificador do cartão físico.

R2 - Sempre que um cartão é inserido, o sistema deve verificar se se trata de um cartão válido por meio de acesso ao computador central. O validador envia uma mensagem contendo os identificadores do cartão e do ônibus para o computador central, que verifica a validade do cartão. Além disso, um funcionário da empresa (cobrador) estará dentro do ônibus para verificar se a foto impressa no cartão confere com o passageiro.

R3 - Um cartão é válido se a informação nele contida puder ser lida e repassada para o computador central, se sua data de validade não tiver expirado e se o passageiro tiver saldo de viagens disponível. Dessa forma, o valor referente a uma viagem é diminuído no registro do cartão no computador central.

R4 - Se o passageiro utilizou seu cartão em uma viagem há menos de 90 minutos, o sistema deve verificar se a viagem atual está em uma linha complementar à linha da viagem anterior. Em caso positivo, não haverá débito e a catraca será liberada; caso contrário, o valor da passagem será

debitado do total disponível no cartão e o saldo restante deverá ser exibido no visor da leitora, com a subsequente liberação da roleta.

R5 - Se o cartão do passageiro não tiver saldo suficiente para cobrir o valor da passagem, o sistema deve imprimir uma mensagem no visor do validador e um sistema manual deve ser usado, ou seja, o passageiro deve pagar o valor em dinheiro para o funcionário da empresa (cobrador).

R6 - Após a passagem de um passageiro pela catraca, tenha sido ela liberada pelo cobrador ou pelo sistema, a catraca volta a ficar na posição travada. No caso dos terminais, após passar pela catraca, o passageiro pode optar por usar qualquer ônibus que tenha um ponto de parada no terminal. A entrada do ônibus em um terminal é feita por uma porta que dá acesso diretamente ao interior do ônibus, sem passar pela catraca, não havendo assim uso de cartão ou pagamento manual de passagem.

R7 - O sistema tem cinco tipos de cartões: o vale-transporte convencional, classicamente utilizado pelos usuários do benefício do vale-transporte; o vale-transporte operacional, fornecido aos funcionários e operadores das empresas viárias; o vale-transporte gratuidade, fornecido aos passageiros maiores de 65 anos; o cartão estudantil, fornecido aos estudantes do município e o vale-transporte avulso, fornecido a passageiros comuns que desejam possuir um cartão para facilitar o pagamento das passagens.

R8 - O vale-transporte convencional, confeccionado na cor vermelha, é adquirido por empresas usuárias da cidade para seus funcionários (passageiros) em agências da empresa viária. Não existe um valor mínimo de carga inicial e o cartão não é pago.

R9 - O vale-transporte operacional, confeccionado na cor verde, é adquirido pelos funcionários e operários da empresa viária. Não existe carga e a aquisição do cartão é gratuita. As passagens dos funcionários e operários é gratuita, havendo desconto de 100% no valor da passagem e não há restrição no uso do cartão.

R10 - O vale-transporte gratuidade, confeccionado na cor azul, é adquirido por passageiros com mais de 65 anos em agências da empresa viária. Deve ser apresentada uma certidão de nascimento ou certidão de casamento, assim como a carteira de identidade e o comprovante de residência. Não existe carga e a aquisição do cartão é gratuita. O desconto da passagem é de 100% e não há restrição no uso do cartão.

R11 - O vale-transporte avulso, confeccionado na cor cinza, é adquirido por qualquer passageiro em pontos de atendimento e em agências da empresa viária. Ele é aplicado a passageiros que pagam a passagem inteira (sem desconto no valor da passagem), profissionais liberais, turistas e trabalhadores domésticos. A aquisição do cartão é gratuita, sendo necessário fazer uma primeira carga equivalente a 20 passagens. É registrada a categoria do cartão, a data da compra e o valor carregado inicialmente. Depois disso, as recargas podem ser feitas em qualquer valor.

R12 - O sistema deve permitir a recarga dos cartões vale-transporte convencional e avulso. No caso do vale-transporte convencional, a recarga é feita pelas empresas que utilizam o benefício do vale-transporte no cartão de seus funcionários. Um representante da empresa usuária se desloca

para uma das agências da empresa viária e informa o valor a recarregar para cada um dos seus funcionários. O valor total é pago pelo representante e o valor informado para cada funcionário é armazenado no computador central para os registros de seus cartões. No caso do vale-transporte avulso, a recarga é feita pelos próprios passageiros em postos de atendimento ou em agências da empresa viária. O passageiro informa o valor a recarregar, paga o valor da carga e o valor é armazenado no computador central para o registro do cartão do passageiro.

R13 - Além dos cartões do tipo vale-transporte, existe o cartão estudantil. Ele também possui um chip e deve ser passado na leitora do validador para liberação da catraca, porém, ele não armazena o saldo no cartão. A passagem deve ser sempre paga de forma manual. Para liberar a catraca após o pagamento manual, o cobrador passa um cartão específico para esse caso na leitora, para que o dinheiro arrecadado seja registrado para a corrida atual do ônibus. O cartão estudantil possibilita um desconto de 50% no valor a ser pago pelo passageiro, sem restrição de quantidade mensal. O sistema incrementa um contador de pagamento com carteira de estudante para posterior verificação. A aquisição do cartão custa R\$ 10,00 para estudantes das instituições particulares e é gratuita para estudantes das instituições públicas. É registrada a categoria do cartão, a data da compra e a instituição de ensino do estudante.

R14 - Um passageiro pode possuir mais de um cartão, não havendo restrição na combinação de cartões que ele possua.

R15 - O sistema deve permitir ao passageiro conferir seu saldo pelo sistema Web da empresa viária. Para tal, ele precisa ter seu número de identificação do cartão.

R16 - O sistema deve permitir a inclusão de todos os usuários (passageiros) da empresa viária que utilizam algum cartão, com seus dados cadastrais (nome, endereço completo, telefones, e-mail, estado civil, local de trabalho, endereço comercial, data de nascimento C.P.F., R.G., data de emissão e órgão emissor).

R17 - O sistema deve permitir a inclusão de empresas usuárias que fornecem benefícios de vale-transporte eletrônico para seus funcionários, com todos os seus dados cadastrais que são: nome fantasia, razão social, C.N.P.J, endereço completo, telefones, e-mail, contato e endereço.

R18 - O sistema deve permitir a inclusão de linhas da empresa viária, contendo o código e nome da linha, o ponto inicial, o ponto final e os horários diários de partida do ponto inicial das diversas corridas.

R19 - O funcionário da empresa viária (cobrador) que trabalha em um ônibus deve registrar as suas corridas pelo uso de um cartão específico que deve ser passado na leitora do validador no momento da saída do ponto inicial da linha do ônibus e da chegada no ponto final da linha. Ao ser lido, o cartão possibilita que o computador central registre a corrida.

R20 - Para que o passageiro possa usar o ônibus sem utilizar um cartão e a catraca seja liberada, o passageiro deve fazer um pagamento manual ao cobrador. O cobrador passa então um cartão específico para essa função na leitora do validador, que permite registrar no sistema a arrecadação para a corrida atual do ônibus.

Requisitos de Interface Externa

R21 - A interface gráfica com o usuário das máquinas de consultas das agências da empresa deve seguir requisitos ergonômicos, definidos pela equipe responsável e verificados por meio de testes com usuários finais.

Requisitos de Performance

R22 - Mensagens de erro devem ser mostradas até 30 segundos após a interação com o usuário.

R23 - Se não houver resposta do computador central da empresa dentro de 2 minutos, uma mensagem de erro deve ser exibida, o passageiro deve pagar manualmente e seus dados devem ser anotados para futuros esclarecimentos.

R24 - A empresa viária pode processar viagens vindas de ônibus diferentes ao mesmo tempo.

Atributos

Disponibilidade

R25 - O sistema BET deve ficar disponível vinte e quatro horas por dia, com menos de 30 minutos sem funcionar por mês, incluindo paradas para manutenção.

Segurança

R26 - O sistema BET deve ser seguro. A única forma de débito do valor do cartão é passando o cartão na leitora do ônibus. As demais interações com o cliente devem ser autenticadas por uma senha.

Manutenção

R27 - Somente os administradores estão autorizados a conectar novos validadores aos ônibus.

Banco de Dados

R28 - O sistema BET deve ser capaz de ler os dados no formato da base de dados do computador central da empresa.

R29 - O registro de uma viagem deve ter todas as propriedades de transações em uma base de dados (atomicidade, consistência, isolamento e durabilidade).

Documento de Requisitos do Sistema BET de Campo Grande

Introdução

Propósito

Este documento descreve os requisitos de software para um sistema de controle de Bilhetes Eletrônico para Transporte, denominado BET, no município de Campo Grande. Destina-se ao projetista, desenvolvedor e administrador do BET.

Escopo

A função do BET é dar apoio computadorizado à rede de ônibus da cidade de Campo Grande.

Visão Geral

O restante deste documento está organizado como segue: inicialmente, definem-se alguns termos importantes para entendimento do documento na Seção B. A Seção B contém uma descrição geral do BET. A Seção B identifica os requisitos funcionais específicos, as interfaces externas e os requisitos de desempenho do BET.

Definições

- **Cartão:** Cartão eletrônico (tecnologia SmartCard) que permite a recarga de valores em dinheiro para serem utilizados no transporte municipal.
- **Passageiro:** Pessoa que utiliza o sistema de transporte para realizar viagens de ônibus (na própria cidade). Existem categorias que possuem desconto no valor da passagem, como por exemplo estudantes e domésticas. Outros pagam valor integral.
- **Ônibus:** Um veículo de transporte que possui uma leitora (dispositivo para leitura do cartão e comunicação com o sistema da empresa). Essa comunicação é on-line e ocorre via um sistema de radiofrequência (RFID) ou tecnologia semelhante. (Detalhes desta comunicação podem ser abstraídos para os propósitos deste projeto)
- **Linha:** Trecho percorrido pelo ônibus desde o ponto-inicial até o ponto-final. Cada linha tem diversos horários de saída do ponto-inicial e é identificada por um nome e um código.
- **Recarga:** O cartão é adquirido com número definido de passagens, convertidas em reais, e pode ser carregado mensalmente com um número máximo de passagens.
- **Validador:** É um equipamento instalado em todos os ônibus. Ela é programável e é composta de um visor de LCD para emitir pequenas mensagens ao passageiro, um atuador para liberar a catraca e um leitor de cartão.
- **Terminal:** Local de embarque e desembarque de passageiros de ônibus, tal que possa ser feita a integração entre diversas linhas sem que haja pagamento de uma nova passagem. A entrada em um terminal também pode ser feita pelo uso do validador.

Descrição Geral

Perspectiva do Produto

O sistema BET não trabalha independentemente. Ele depende das informações centralizadas na empresa.

Funções do Produto

O software deve permitir que a rede BET trabalhe de forma computadorizada. A empresa viária possui um computador central que mantém os dados de todos os passageiros, cartões, linhas, ônibus, viagens realizadas etc. O validador instalado em um ônibus ou em um terminal aceita um cartão como entrada e comunica-se com o computador da empresa viária por rádio para realizar o débito do valor correspondente no cartão do passageiro.

Existe um sistema de integração temporal de ônibus, que permite ao usuário pagar um único valor de passagem, desde que as viagens ocorram em ônibus que participem da linha de integração e estejam dentro de um intervalo de 60 minutos. O sistema deve cuidar do armazenamento da última viagem realizada.

Existe também um sistema de terminais de integração de ônibus, que permite ao usuário pagar um único valor de passagem, desde que as viagens ocorram em ônibus que participem dos terminais de integração e a mudança de ônibus seja feita dentro dos terminais, não sendo necessário passar o cartão novamente na leitora do validador.

Existem computadores espalhados nas agências da empresa viária que permitem ao passageiro fazer consulta do seu saldo no cartão e da última viagem realizada.

Características do Usuário

O **passageiro** interage com o sistema BET via uma leitora instalada no ônibus. É desejável que seja bem fácil usar a leitora mas, em caso de dúvida, deve haver algum tipo de suporte ao passageiro. Além disso, o passageiro pode usar terminais disponíveis nas agências da empresa de transportes para consultar as viagens realizadas no período.

O **atendente** é a pessoa responsável pelo registro de passageiros no sistema, assim como pela criação e carga de cartões para os passageiros.

O **administrador** é a pessoa responsável por manter o sistema, por exemplo, conectar uma nova leitora em um ônibus ou reiniciar a leitora quando necessário.

Requisitos Específicos

Requisitos Funcionais

R1 - O sistema deve permitir ao passageiro utilizar seu cartão para pagar uma viagem feita em ônibus. O cartão pode ser usado nas leitoras dos validadores existentes tanto nos ônibus quanto nos terminais. O cartão armazena um identificador do cartão físico.

R2 - Sempre que um cartão é inserido, o sistema deve verificar se se trata de um cartão válido por meio de acesso ao computador central. O validador envia uma mensagem contendo os identificadores do cartão e do ônibus para o computador central, que verifica a validade do cartão. Além disso, um funcionário da empresa (cobrador) estará dentro do ônibus para verificar se a foto impressa no cartão confere com o passageiro.

R3 - Um cartão é válido se a informação nele contida puder ser lida e repassada para o computador central, se sua data de validade não tiver expirado e se o passageiro tiver saldo disponível. Dessa forma, o valor referente a uma viagem é diminuído no registro do cartão no computador central.

R4 - Se o passageiro utilizou seu cartão em uma viagem há até 60 minutos, o sistema deve verificar se a viagem atual está em uma linha complementar à linha da viagem anterior e que está sendo feita a primeira integração temporal da viagem. Em caso positivo, não haverá débito no registro do cartão no computador central e é enviado uma mensagem para o validador para que a catraca seja liberada; caso contrário, o valor da passagem será debitado do total disponível no registro do cartão no computador central e o saldo restante deverá ser exibido no visor do validador, com a subsequente liberação da catraca.

R5 - Se o cartão do passageiro não tiver saldo suficiente para cobrir o valor da passagem e se não se tratar de uma viagem de integração, o sistema deve imprimir uma mensagem no visor do validador e um sistema manual deve ser usado, ou seja, o passageiro deve pagar o valor em dinheiro para o funcionário da empresa (o cobrador).

R6 - Após a passagem de um passageiro pela catraca, tenha sido ela liberada manualmente pelo funcionário da empresa ou pelo sistema, a catraca volta a ficar na posição travada. No caso dos terminais, após passar pela catraca, o passageiro pode optar por usar qualquer ônibus que tenha um ponto de parada no terminal. A entrada do ônibus em um terminal é feita por uma porta que dá acesso diretamente ao interior do ônibus, sem passar pela catraca, não havendo mais o uso de cartão ou pagamento manual de passagem.

R7 - O sistema deve permitir a recarga do cartão em agências da empresa viária distribuídas pela cidade ou pelo site na internet, pelo uso de um login e senha. O passageiro informa o valor a recarregar e o valor é transferido para seu cartão após confirmação do pagamento.

R8 - O sistema tem quatro tipos de cartões: o cartão eletrônico temporal convencional, classicamente utilizado pelos usuários do benefício do vale-transporte; o cartão eletrônico temporal estudantil, utilizado pelos estudantes do município; o cartão eletrônico temporal gratuidade, fornecido aos passageiros maiores de 65 anos; e o cartão eletrônico temporal avulso, fornecido a passageiros comuns que desejam possuir um cartão para facilitar o pagamento das passagens.

R9 - O cartão eletrônico temporal convencional é adquirido por empresas usuárias da cidade para seus funcionários (passageiros) em agências da empresa viária. Não existe um valor mínimo de carga inicial e o cartão não é pago.

R10 - O cartão eletrônico temporal estudantil é adquirido pelos estudantes do município de Campo Grande, deve ser apresentado um comprovante da entidade de estudo e a carteira de identidade (no caso de não possuir, a carteira do responsável deve ser apresentada). A aquisição do cartão é gratuita. As passagens dos estudantes têm um desconto de 50% no valor a ser pago pelo passageiro normal, sem restrição de quantidade mensal de uso do cartão.

R11 - O cartão eletrônico temporal gratuidade é adquirido pelos passageiros com mais de 65 anos em agências da empresa viária. Deve ser apresentada uma certidão de nascimento ou certidão de casamento, assim como a carteira de identidade e o comprovante de residência. Não existe carga e a aquisição do cartão é gratuita. O desconto da passagem é de 100% e não há restrição no uso do cartão.

R12 - O cartão eletrônico temporal avulso é adquirido por qualquer passageiro em pontos de venda, em agência da empresa viária e no seu site na internet. Ele é aplicado a passageiros que pagam a passagem inteira (sem desconto no valor da passagem). A aquisição do cartão é gratuita, sendo necessário fazer uma primeira carga para tornar o cartão válido para uso. É registrada a categoria do cartão, a data da compra e o valor carregado inicialmente. As recargas podem ser feitas para qualquer valor.

R13 - O sistema deve permitir a recarga dos cartões eletrônicos convencional, estudantil e avulso em agências da empresa viária e no seu site na internet. No caso do cartão convencional, a recarga é feita pelas empresas usuárias (pessoa jurídica) que utilizam o benefício do vale-transporte no cartão de seus funcionários (passageiros). A empresa informa o valor a recarregar e o valor é transferido para o cartão do passageiro. No caso do cartão avulso, a recarga é feita pelos próprios passageiros. O passageiro informa o valor a recarregar e o valor é transferido para seu cartão após confirmação do pagamento.

R14 - Um passageiro pode possuir mais de um cartão, não havendo restrição na combinação de cartões que ele possua.

R15 - O sistema deve permitir ao passageiro conferir seu saldo e sua última viagem em agências da empresa viária. Para tal, ele precisa ter seu número de identificação do cartão e uma senha.

R16 - O sistema deve permitir a inclusão de passageiros da empresa viária, com todos os seus dados cadastrais que são: nome, endereço completo, bairro, cidade, estado, cep, sexo, estado civil, telefone, data de nascimento, C.P.F. e R.G..

R17 - O sistema deve permitir a inclusão de empresas usuárias que fornecem benefícios de vale-transporte eletrônico para seus funcionários, com todos os seus dados cadastrais que são: nome fantasia, razão social, C.N.P.J, endereço completo, telefones, e-mail, contato, endereço, porte.

R18 - O sistema deve permitir a inclusão de linhas da empresa viária, contendo o código e nome da linha, o ponto inicial, o ponto final e os horários diários de partida do ponto inicial.

R19 - O sistema deve permitir a inclusão de ligações entre linhas, permitindo saber quais podem ser considerados complementares ou integradas umas às outras.

R20 - O sistema permite apenas uma integração temporal para cada passagem, portanto, mesmo que ainda esteja no intervalo de tempo de 60 minutos, após já efetuar uma viagem integrada, não se pode fazer outra.

R21 - O funcionário da empresa viária (cobrador) que trabalha em um ônibus deve registrar as suas corridas pelo uso de um cartão específico que deve ser passado na leitora do validador no momento da saída do ponto inicial da linha do ônibus e da chegada no ponto final da linha. Ao ser lido, o cartão possibilita que o computador central registre a corrida.

R22 - Para que o passageiro possa usar o ônibus sem utilizar um cartão e a catraca seja liberada, o passageiro deve fazer um pagamento manual ao cobrador. O cobrador passa então um cartão específico para essa função na leitora do validador, que permite registrar no sistema a arrecadação para a corrida atual do ônibus.

Requisitos de Interface Externa

R23 - A interface gráfica com o usuário dos terminais de consultas das agências da empresa deve seguir requisitos ergonômicos, definidos pela equipe responsável e verificados por meio de testes com usuários finais.

Requisitos de Performance

R24 - Mensagens de erro devem ser mostradas até 30 segundos após a interação com o usuário.

R25 - Se não houver resposta do computador central da empresa dentro de 2 minutos, uma mensagem de erro deve ser exibida, o passageiro deve pagar manualmente e seus dados devem ser anotados para futuros esclarecimentos.

R26 - A empresa viária pode processar viagens vindas de ônibus diferentes ao mesmo tempo.

Atributos

Disponibilidade

R27 - O sistema BET deve ficar disponível vinte e quatro horas por dia.

Segurança

R28 - O sistema BET deve ser seguro. A única forma de débito do valor do cartão é por meio da inserção do cartão na leitora do ônibus. As demais interações com o cliente devem ser autenticadas por uma senha.

Manutenção

R29 - Somente os administradores estão autorizados a conectar novas leitoras aos ônibus.

Banco de Dados

R30 - O sistema BET deve ser capaz de ler os dados no formato da base de dados do computador central da empresa viária.

R31 - O registro de uma viagem deve ter todas as propriedades de transações em uma base de dados (atomicidade, consistência, isolamento e durabilidade).

Documento de Requisitos do Sistema BET de São Carlos

Introdução

Propósito

Este documento descreve os requisitos de software para um sistema de controle de Bilhetes Eletrônico para Transporte, denominado BET, no município de São Carlos. Destina-se ao projetista, desenvolvedor e mantenedor do BET.

Escopo

A função do BET é dar apoio computadorizado à rede de ônibus de uma determinada cidade.

Visão Geral

O restante deste documento está organizado como segue: inicialmente, definem-se alguns termos importantes para entendimento do documento na Seção C. A Seção C contém uma descrição geral do BET. A Seção C identifica os requisitos funcionais específicos, as interfaces externas e os requisitos de desempenho do BET.

Definições

- **Cartão:** Cartão eletrônico (tecnologia SmartCard) que permite a recarga de valores em dinheiro para serem utilizados no transporte municipal.
- **Passageiro:** Pessoa que utiliza o sistema de transporte para realizar viagens de ônibus (na própria cidade). Existem categorias que possuem desconto no valor da passagem, como por exemplo estudantes e domésticas. Outros pagam valor integral.
- **Ônibus:** Um veículo de transporte que possui uma leitora (dispositivo para leitura do cartão e comunicação com o sistema da empresa). Essa comunicação é on-line e ocorre via um sistema de radiofrequência (RFID) ou tecnologia semelhante. (Detalhes desta comunicação podem ser abstraídos para os propósitos deste projeto)
- **Linha:** Trecho percorrido pelo ônibus desde o ponto-inicial até o ponto-final. Cada linha tem diversos horários de saída do ponto-inicial e é identificada por um nome e um código.
- **Recarga:** O cartão é adquirido com número definido de passagens, convertidas em reais, e pode ser carregado mensalmente com um número máximo de passagens.
- **Leitora:** É um equipamento instalado em todos os ônibus. Ela é programável e é composta de um visor de LCD para emitir pequenas mensagens ao passageiro e um atuador para liberar a catraca.

Descrição Geral

Perspectiva do Produto

O sistema BET não trabalha independentemente. Ele depende das informações centralizadas na empresa.

Funções do Produto

O software deve permitir que a rede BET trabalhe de forma computadorizada. A empresa de transportes possui um computador central que mantém os dados de todos os passageiros, cartões, linhas, ônibus, viagens realizadas etc. A leitora instalada nos ônibus aceita um cartão como entrada e comunica-se com o computador da empresa por rádio para realizar o débito do valor correspondente no cartão do passageiro. Existe um sistema de integração de ônibus, que permite ao usuário pagar um único valor de passagem, desde que as viagens ocorram em ônibus que participem da linha de integração e esteja dentro de um intervalo de 90 minutos. O sistema deve cuidar da armazenagem das viagens e medidas de segurança. Existem também terminais espalhados nas agências

da empresa de transportes equipados com monitores de vídeo, impressora e Internet, que permitem ao passageiro fazer consulta sobre suas viagens.

Características do Usuário

O **passageiro** interage com o sistema BET via uma leitora instalada no ônibus. É desejável que seja bem fácil usar a leitora mas, em caso de dúvida, deve haver algum tipo de suporte ao passageiro. Além disso, o passageiro pode usar terminais disponíveis nas agências da empresa de transportes para consultar as viagens realizadas no período.

O **mantenedor** é a pessoa responsável por manter o sistema, por exemplo, conectar uma nova leitora em um ônibus ou reiniciar a leitora quando necessário.

Requisitos Específicos

Requisitos Funcionais

R1 - O sistema deve permitir ao passageiro utilizar seu cartão para pagar uma viagem feita em ônibus da empresa.

R2 - Sempre que um cartão é inserido, sistema deve verificar se se trata de um cartão válido. Além disso, um funcionário da empresa estará dentro do ônibus para verificar se a foto impressa no cartão confere com o passageiro.

R3 - Um cartão é válido se a informação nele contida puder ser lida, se sua data de validade não tiver expirado e se o passageiro tiver saldo de viagens disponível.

R4 - Se o passageiro utilizou seu cartão em uma viagem há menos de 90 minutos, o sistema deve verificar se a viagem atual está em uma linha complementar à linha da viagem anterior. Em caso positivo, não haverá débito e a catraca será liberada; caso contrário, o valor da passagem será debitado do total disponível no cartão e o saldo restante deverá ser exibido no visor da leitora, com a subsequente liberação da roleta.

R5 - Se o cartão do passageiro não tiver saldo suficiente para cobrir o valor da passagem e se não se tratar de uma viagem de integração, o sistema deve emitir um sinal sonoro, imprimir uma mensagem no visor da leitora e um sistema manual deve ser usado, ou seja, o passageiro deve pagar o valor em dinheiro para o funcionário da empresa (o cobrador).

R6 - Após a passagem de um passageiro pela catraca, tenha sido ela liberada manualmente pelo funcionário da empresa ou pelo sistema, a roleta volta a ficar na posição travada.

R7 - O sistema deve permitir a recarga do cartão em agências da empresa distribuídas pela cidade. O passageiro informa o valor a recarregar, limitado a um valor máximo mensal para cada categoria, e o valor é transferido para seu cartão.

R8 - O valor do desconto segue as seguintes categorias de passageiros: A) estudantes e B) empregados doméstica(o)s podem carregar até 50 passagens por mês, com desconto de 50% sobre o valor da passagem; C) trabalhadores registrados em carteira podem carregar até 50 passagens por mês, com 30% de desconto; D) demais usuários não possuem restrição, pois não obtêm desconto no valor da passagem (pagam o valor integral) e podem comprar até 60 em cada compra.

R9 - Um passageiro pode possuir mais de um cartão, mas apenas para as seguintes combinações: AD, AC, BD e CD.

R10 - O sistema deve permitir ao passageiro conferir seu saldo ou imprimir um extrato de suas viagens no período, pela Internet ou no escritório da empresa. Para tal, ele precisa ter seu número de identificação do cartão e uma senha.

R11 - O sistema deve permitir a inclusão de passageiros da empresa, com todos os seus dados cadastrais que são: nome, endereço completo, telefones, e-mail, estado civil, local de trabalho, endereço comercial, data de nascimento C.P.F., R.G., data de emissão e órgão emissor.

R12 - O sistema deve permitir a inclusão de linhas da empresa, contendo o código e nome da linha, o ponto inicial, o ponto final e os horários diários de partida do ponto inicial.

R13 - O sistema deve permitir a inclusão de ligações entre linhas, permitindo saber quais podem ser considerados complementares ou integradas umas às outras.

R14 - O sistema deve permitir a compra de um cartão da empresa pelo cliente, sendo registrada a categoria de cartão (A, B, C ou D), a data da compra e o valor carregado inicialmente.

Requisitos de Interface Externa

R15 - A interface gráfica com o usuário dos terminais de consultas das agências da empresa deve seguir requisitos ergonômicos, definidos pela equipe responsável e verificados por meio de testes com usuários finais.

Requisitos de Performance

R16 - Mensagens de erro devem ser mostradas até 30 segundos após a interação com o usuário.

R17 - Se não houver resposta do computador central da empresa dentro de 2 minutos, uma mensagem de erro deve ser exibida, o passageiro deve pagar manualmente e seus dados devem ser anotados para futuros esclarecimentos.

R18 - A empresa pode processar viagens vindas de ônibus diferentes ao mesmo tempo.

Atributos

Disponibilidade

R19 - O sistema BET deve ficar disponível vinte e quatro horas por dia.

Segurança

R20 - O sistema BET deve ser seguro. A única forma de débito do valor do cartão é por meio da inserção do cartão na leitora do ônibus. As demais interações com o cliente devem ser autenticadas por uma senha.

Manutenção

R21 - Somente os mantenedores estão autorizados a conectar novas leitoras aos ônibus.

Banco de Dados

R21 - O sistema BET deve ser capaz de ler os dados no formato da base de dados do computador central da empresa.

R22 - O registro de uma viagem deve ter todas as propriedades de transações em uma base de dados (atomicidade, consistência, isolamento e durabilidade).

Alguns Artefatos da LPS-BET

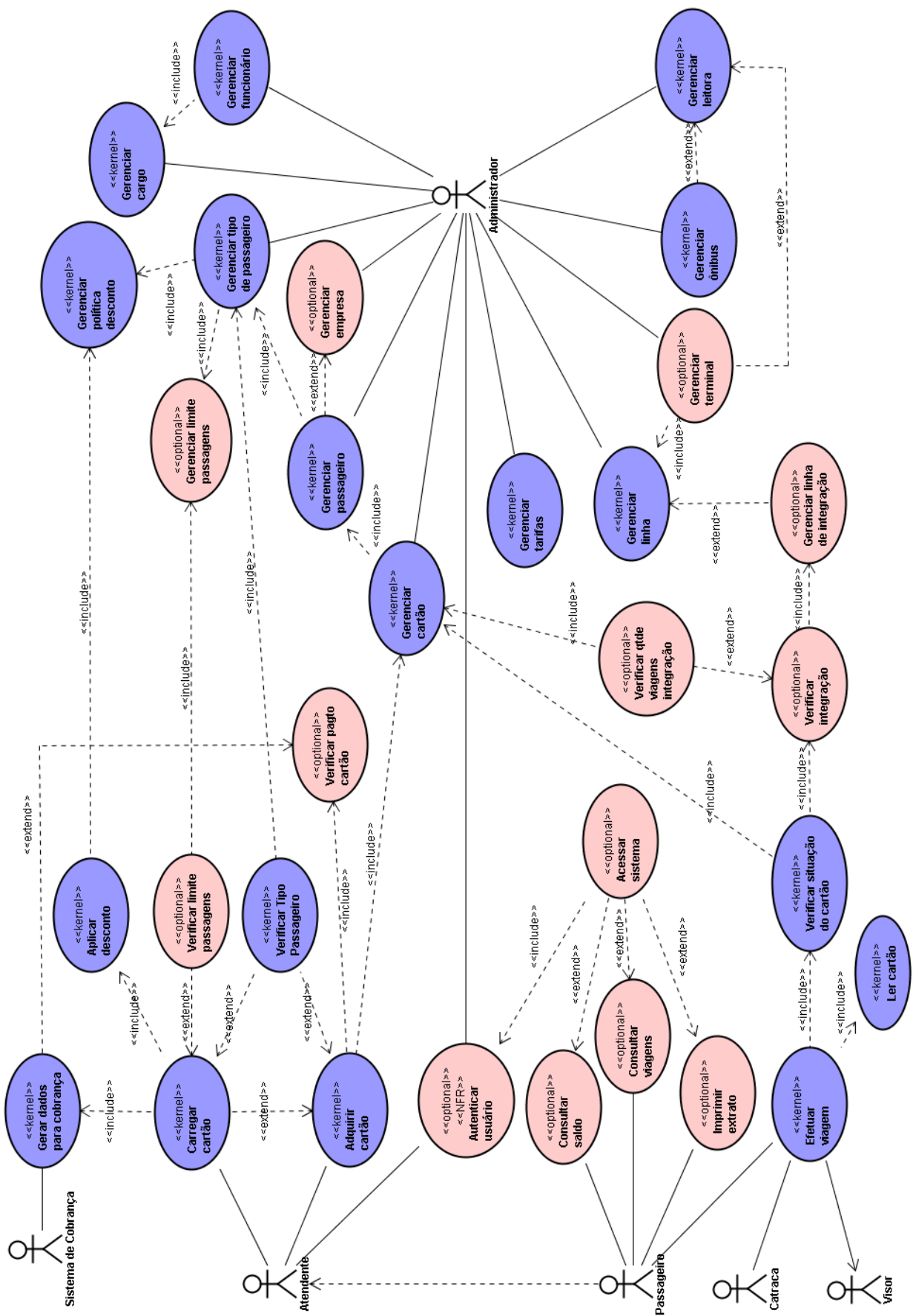


Figura D.1: Diagrama de Casos de Uso da LPS-BET

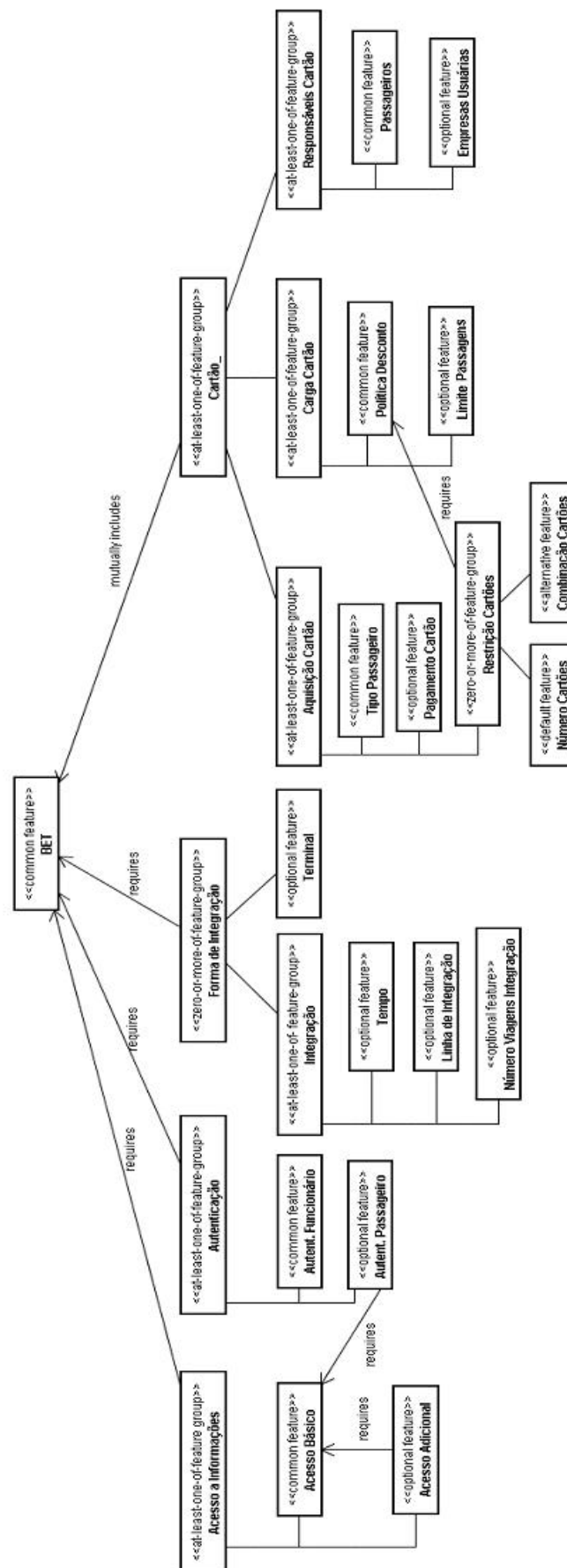


Figura D.2: Diagrama de Características para a LPS-BET

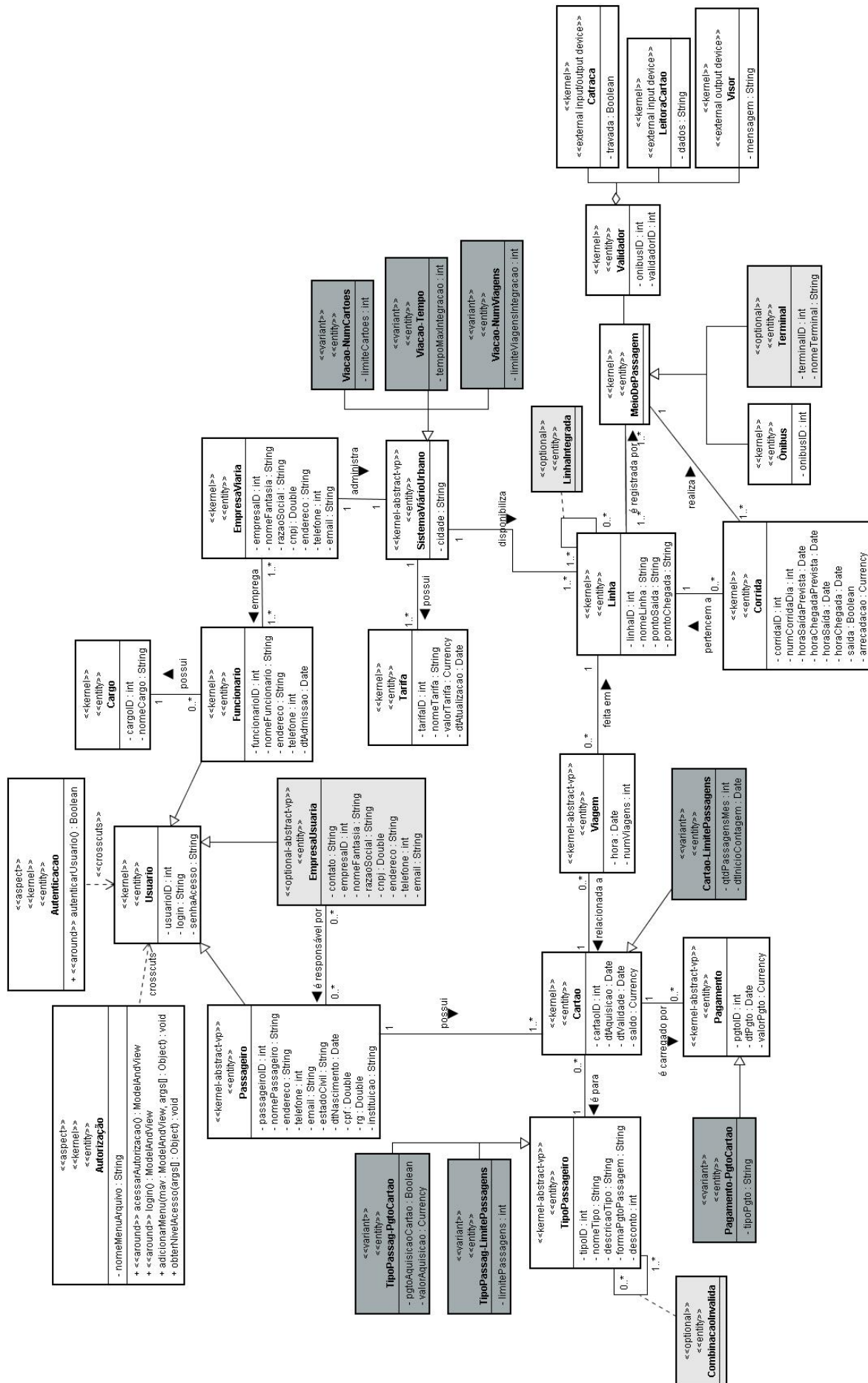


Figura D.3: Diagrama de Classes da LPS-BET

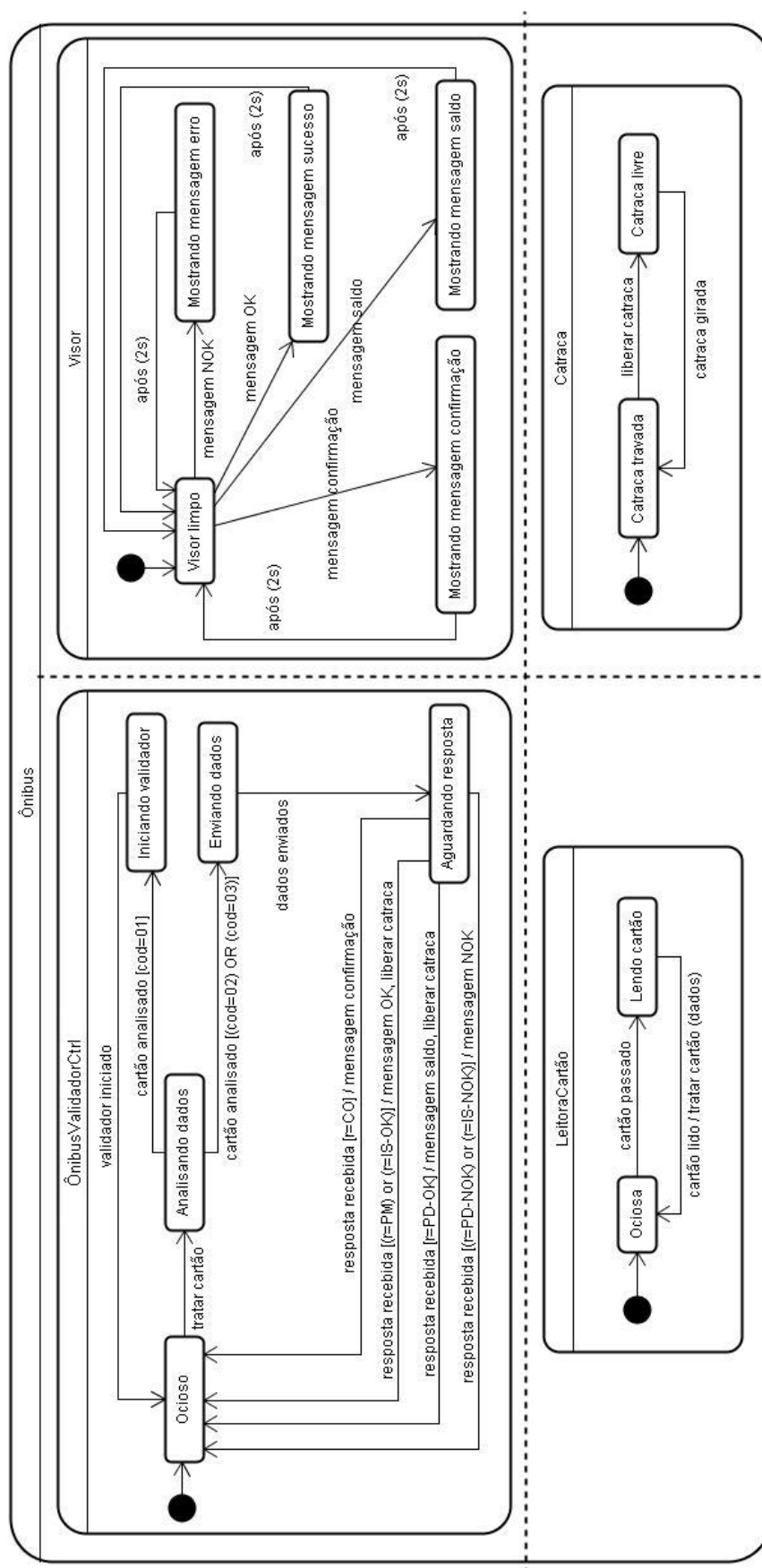


Figura D.4: Diagrama de Estados para os componentes do Ônibus

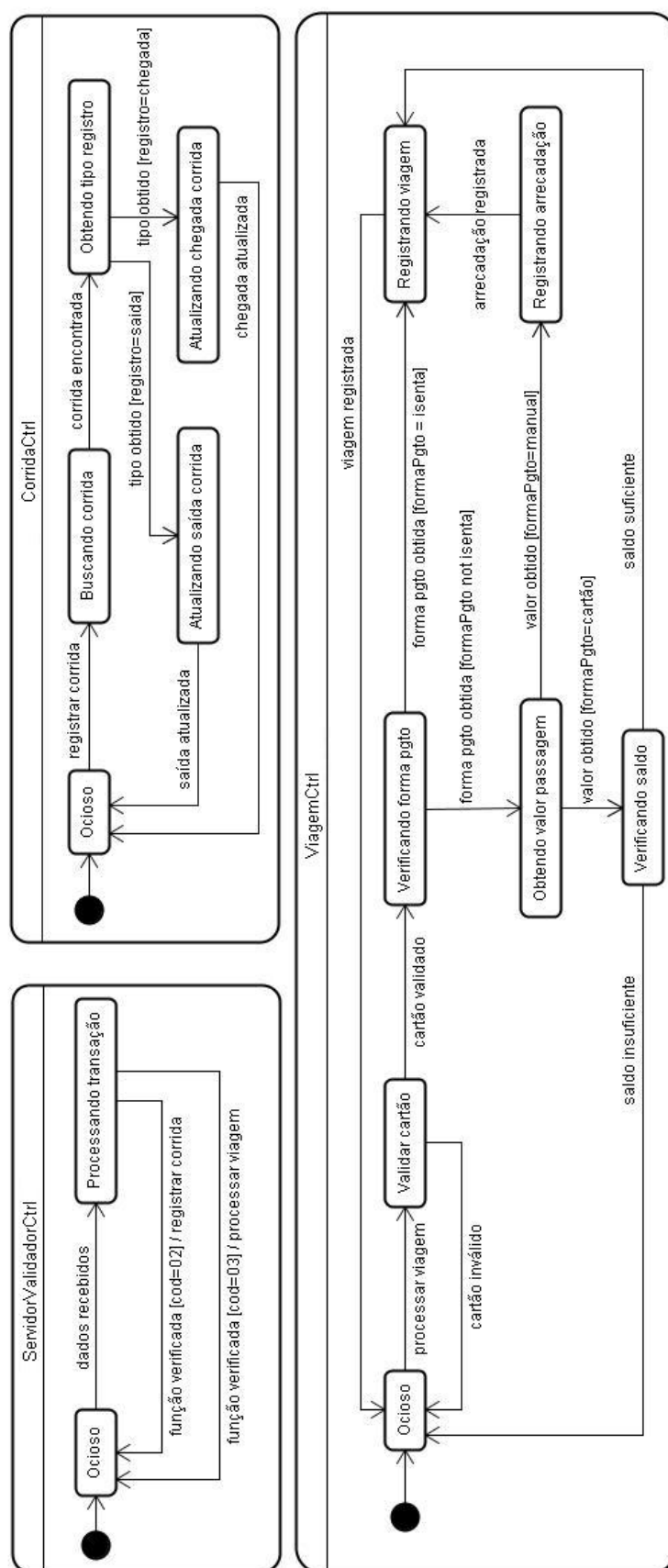


Figura D.5: Diagrama de Estados para os componentes relacionados ao ValidadorServidorCtrl