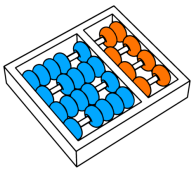
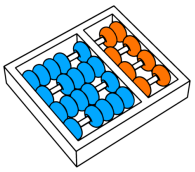


Bruno de Abreu Iizuka

"Variabilidade em Tratamento de Exceções em Linha de Produtos de Software"

CAMPINAS
2012



Universidade Estadual de Campinas
Instituto de Computação

Bruno de Abreu Iizuka

"Variabilidade em Tratamento de Exceções em Linha de Produtos de Software"

Orientador(a): **Cecília Mary Fischer Rubira**

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Computação da Universidade Estadual de Campinas para obtenção do título de Mestre em Ciência da Computação.

Este exemplar corresponde à versão final da Dissertação defendida por Bruno de Abreu Iizuka, sob orientação de Cecília Mary Fischer Rubira.

Assinatura do Orientador(a)

CAMPINAS
2012

FICHA CATALOGRÁFICA ELABORADA POR
ANA REGINA MACHADO - CRB8/5467
BIBLIOTECA DO INSTITUTO DE MATEMÁTICA, ESTATÍSTICA E
COMPUTAÇÃO CIENTÍFICA - UNICAMP

Iizuka, Bruno de Abreu, 1985-
Ii1v Variabilidade em tratamento de exceções em linha de produtos de software / Bruno de Abreu Iizuka. – Campinas, SP : [s.n.], 2012.

Orientador: Cecília Mary Fischer Rubira.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

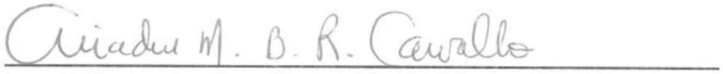
1. Engenharia de software. 2. Tratamento de exceções. 3. Engenharia de linha de produto de software. 4. Software - Arquitetura. I. Rubira, Cecília Mary Fischer,1964-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em inglês: Variability of exception handling in software product line
Palavras-chave em inglês:
Software engineering
Exception handling
Software product line engineering
Software architecture
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Cecília Mary Fischer Rubira [Orientador]
Patrick Henrique da Silva Brito
Ariadne Maria Brito Rizzoni Carvalho
Eliane Martins
Ivan Luiz Marques Ricarte
Data de defesa: 22-11-2012
Programa de Pós-Graduação: Ciência da Computação

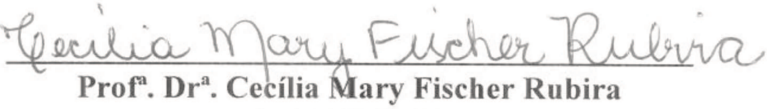
TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 22 de Novembro de 2012, pela Banca examinadora composta pelos Professores Doutores:



Profª. Drª. Ariadne Maria Brito Rizzoni de Carvalho
IC / UNICAMP

Prof. Dr. Patrick Henrique da Silva Brito
IC / UFAL



Profª. Drª. Cecília Mary Fischer Rubira
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Variabilidade em Tratamento de Exceções em Linha de Produtos de Software

Bruno de Abreu Iizuka

22 de Novembro de 2012

Banca Examinadora:

- Cecília Mary Fischer Rubira (Orientadora)
- Patrick Henrique da Silva Brito
  Instituto de Computação - Universidade Federal de Alagoas
- Ariadne Maria Brito Rizzoni Carvalho
  Instituto de Computação - Universidade Estadual de Campinas
- Eliane Martins
  Instituto de Computação - Universidade Estadual de Campinas (suplente interno)
- Ivan Luiz Marques Ricarte
  Faculdade de Engenharia Elétrica e de Computação - Universidade Estadual de Campinas (suplente externo)

Abstract

Nowadays, many efforts are being made to achieve a higher degree of reuse during the development of systems. Software Product Lines (SPL) is an approach to improve software reuse. A PLA provides a global view of the variabilities of a SPL, while it embodies the concepts and advantages of the traditional software architecture. Due to its variabilities, a PLA is harder to evolve than a conventional software architecture.

Exception handling is a well known technique to detect and treat errors in software systems. However, despite its popularity, its design and implementation are constituted of very complex tasks that do not receive the adequate attention from the existing development processes.

Separation of concerns is one of the overarching goals of exception handling in order to keep separate normal and exceptional behaviour of a software system. In the context of a software product line (SPL), this separation of concerns is also important for designing software variabilities related to normal and exceptional behaviour, such as the choice of different handlers depending on the set of selected features.

The main goal of this work is to present a method to specify and implement the variability of exception handling in SPL components-based. The method MVTE (Variability of Exception Handler Method) is a combination of methods known in the literature (PLUS and UML Components) and models COSMOS* and COSMOS*-VP. To validate the method MVTE, it was studied two empirical studies, and to measure their quality it was used the metrics impact change, coupling between modules and diffusion over concerns.

Resumo

Atualmente, muitos esforços vêm sendo feitos para se obter um maior grau de reutilização durante o desenvolvimento de sistemas. Linha de Produtos de Software (LPS) é uma abordagem que promove a reutilização de software. A Arquitetura de Linha de Produtos (ALP) provê uma perspectiva global das variabilidades da linha, ao passo que engloba os conceitos tradicionais de uma arquitetura de software. Devido as variabilidades de software de uma ALP, a evolução arquitetural é ainda mais complexa, do que quando comparado com evolução de arquiteturas de software convencionais.

Tratamento de exceções é uma técnica bastante conhecida para a detecção e tratamento de erros em sistemas de software. Porém, apesar da sua popularidade, o seu projeto e a sua implementação são constituídos de tarefas muito complexas que não recebem uma atenção adequada dos processos de desenvolvimento existentes.

Separação de interesses é um dos objetivos do tratamento de exceções para separar o comportamento normal e excepcional do sistema de software. No contexto de uma LPS, a separação de interesses é importante para o design das variabilidades de software relacionadas as estratégias do comportamento normal e do comportamento excepcional, como a escolha de diferentes tratadores de exceções por diferentes características.

O objetivo principal deste trabalho é apresentar um método para especificar e implementar a variabilidade de tratamentos de exceções em LPS baseadas em componentes. O método MVTE (Método de Variabilidade de Tratamento de Exceções) é uma combinação de métodos já conhecidos na literatura (PLUS e UML Components) e os modelos COSMOS* e COSMOS*-VP. Para validar o método MVTE foram utilizados dois estudos empíricos, e para medir a sua qualidade foram utilizadas as métricas de impacto de mudanças, acoplamento entre módulos e difusão de interesses.

Agradecimentos

Eu gostaria de agradecer primeiramente a Deus, por todas as portas abertas e fechadas, pelas experiências que Ele me proporcionou nessa vida.

Aos meus pais, Roberto e Elizabeth, por todo amor, dedicação, incentivo e puxões de orelha que foram fundamentais para me tornar a pessoa que sou hoje. Ao meu irmão, Victor, pelas alegrias, amor e paciência durante todo este tempo em que ele sempre foi e será meu melhor amigo.

A minha namorada, futura esposa, Marcela, a quem amo muito e me ajudou a superar este desafio e outros desafios que existiram e estão por vir para nos trazer um excelente futuro. Obrigado por ter me apoiado com os prazos e não me deixando desanimar nos momentos críticos. Agradeço também, aos meus sogros, Dona Nair e Seu Moritani, pelo apoio, confiança, carinho e paciência.

Agradeço, especialmente, a minha orientadora Professora Cecília M. F. Rubira, pelo auxílio, ensinamentos, correções, críticas sempre sinceros. Ajudando-me muito durante essa jornada.

A todos os amigos que fiz durante o período de cumprir disciplinas e trabalhos realizados no meu primeiro ano de mestrado, pelas brincadeiras e bate-papos que tivemos neste período. Aos meus amigos do LSD, que em momentos de tensão conseguimos descontrair falando sobre assuntos diversos e sempre divertidos. Outro agradecimento especial para a Amanda e o Leonardo, que sempre me auxiliaram corrigindo e criticando do começo ao fim desta dissertação, me mostrando alternativas de caminhos para a conclusão da mesma, e pelas nossas conversas divertidas que ajudam nos momentos de stress.

Aos meus amigos, que estudaram comigo durante a graduação, e que sempre estiveram prontos nos momentos necessários para me ouvir e me fazer rir e divertir.

A todos os professores e funcionários do IC que proporcionaram a estrutura para concluir esta dissertação.

Por fim, agradeço aos professores da banca, de qualificação e final, que contribuíram para uma pesquisa e um resultado final com qualidade.

Lista de Acrônimos e Siglas

ALP Arquitetura de Linha de Produto

AO Orientação a Aspectos

AO-CNH Versão da LPS E-Gov-CNH implementada em Aspectos

AO-MM Versão da LPS MobileMedia implementada em Aspectos

AOSD *Aspect-Oriented Software Development*

API *Application Program Interface*

CASE *Computer-Aided Software Engineering*

CNH Carteira Nacional de Habilitação

Corba *Common Object Request Broker Architecture*

COSMOS* *COmponent System MOdel for Software architectures*

COSMOS*-MM Versão da LPS MobileMedia implementada em COSMOS*

COTS *Common-Off-The Shelf*

CPF Cadastro de Pessoa Física

CTE Contexto de Tratamento da Exceção

DBC Desenvolvimento Baseado em Componentes

E-gov Aplicações de Governo Eletrônico

E-gov-CNH LPS de Governo Eletrônico para obtenção da CNH

LOC *Lines of Code*

LPS Linha de Produtos de Software

MDCE Metodologia para Definição do Comportamento Excepcional

MMAPI *Mobile Media Programming Interface Application*

MTE Mecanismos de Tratamento da Exceção

MVC *Model-View Controler*

MVTE Método de Variabilidade de Tratamento de Exceções

OO Orientação a Objetos

OO-MM Versão da LPS MobileMedia implementada somente em Orientação a Objetos

PLUS *Product Line UML-based Software Engineering*

POA Programação Orientada a Aspectos

RG Registro Geral

RIC Registro de Identificação Civil

SED *Software Engineering and Dependability Research Group*

SMS *Short Message Service*

UML *Unified Modeling Language*

Unicamp Universidade Estadual de Campinas

VP *Variation Point*

VP-CNH Versão da LPS E-Gov-CNH implementada usando o MVTE

VP-MM Versão da LPS MobileMedia implementada usando o MVTE

WMA *Windows Media Audio*

XPI *Crosscutting Program Interface*

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Atualmente os sistemas de software vêm sendo utilizados por pessoas em várias áreas, e cada vez mais elas desejam que seu funcionamento esteja em perfeita condição. Porém, as empresas, que desenvolvem sistemas de software, vêm desejando realizar entregas mais rápidas dos sistemas de software e com custo reduzido. Nesse contexto, são encontrados alguns problemas, são eles: a alta complexidade dos sistemas, o tamanho dos sistemas e as necessidades dos clientes estão sempre aumentando. Deste modo, a linha de produtos de software (LPS), o desenvolvimento baseado em componentes (DBC) [46] e o desenvolvimento centrado em arquiteturas [49] têm sido muito utilizados para tentar amenizar esses problemas, como discutido a seguir.

Linha de Produtos de Software (LPS) define um conjunto de produtos de software com alto grau de similaridade entre si, que atendem as necessidades específicas de um segmento de mercado ou missão, e que são desenvolvidos de forma prescritiva a partir de um conjunto de artefatos básicos [15]. Variabilidade de software é a capacidade de um sistema de software ou artefato ser alterado, customizado ou configurado, para ser utilizado em um contexto específico. Um alto grau de variabilidade permite a utilização do artefato de software em um contexto mais amplo, o que torna o artefato mais reutilizável. Em linha de produtos [15], por exemplo, os artefatos de software devem ser flexíveis o suficiente de modo a permitir que detalhes de implementação de um produto específico possam ser postergados para fases posteriores do desenvolvimento. Estas decisões de projeto postergadas são denominadas de pontos de variação, que são os locais do artefato de software em que uma decisão de projeto pode ser tomada; e variantes são alternativas de projeto associadas a este ponto.

Desenvolvimento Baseado em Componentes (DBC) é um paradigma de desenvolvimento em que os sistemas de software são desenvolvidos a partir da composição de blocos interoperáveis e reutilizáveis chamados de componentes de software [46]. O desenvolvimento baseado em componentes pode apoiar o desenvolvimento em LPS através da se-

paração entre especificação de componente e implementação de componente, o que reduz o acoplamento entre as partes, favorecendo a reutilização e a consequente construção de produtos de software a partir de componentes reutilizáveis [26].

A arquitetura de software, através de um alto nível de abstração, define o sistema em termos de seus componentes arquiteturais, que representam unidades abstratas do sistema; a interação entre essas entidades, que são materializadas explicitamente através dos conectores; e os atributos e funcionalidades de cada um [44]. Por conhecerem o fluxo interativo entre os componentes do sistema, é possível nos conectores, estabelecer protocolos de comunicação e coordenar a execução dos serviços que envolvam mais de um componente do sistema, assim é possível atingir os requisitos de qualidade.

Com o uso de conectores entre os componentes do sistema, o software consegue evoluir facilmente, caso contrário seu uso torna-se menos satisfatório [37]. O gerenciamento da evolução é uma tarefa árdua no contexto atual. Em sistemas com alto grau de reutilização, lidar com a evolução é particularmente difícil. Em parte, esse problema se deve principalmente a dois fatores [7]: (i) falta de controle das variabilidades de um componente; e (ii) falta de rastreabilidade entre os artefatos reutilizados em diferentes sistemas.

1.1 Problema

A existência de um alto acoplamento entre os comportamentos normal e excepcional dificulta o projeto e desenvolvimento de LPS com requisitos de confiabilidade, pois dificulta a adaptação do comportamento excepcional de acordo com as necessidades específicas de cada produto derivado da ALP (Arquitetura de Linha de Produtos). Além disso, o espalhamento do tratamento excepcional em todo o código não contribui para as futuras alterações ou melhorias que devem ocorrer durante o ciclo de vida da LPS.

Na Figura 1.1 mostra um trecho de uma ALP de um sistema de hotel, onde é representado os componentes de pagamento. Os componentes de Pagamento são `Cartao_Mgr`, `Boleto_Mgr`, `Pagamento_Mgr` e `Persistence`. Os componentes `Cartao_Mgr` e `Boleto_Mgr` possuem um ponto de variação arquitetural, ou seja, é um ponto na arquitetura da LPS, onde é possível fazer uma escolha de qual componente será usado, neste caso `Cartao_Mgr` ou `Boleto_Mgr`. Neste exemplo, o tratamento excepcional está todo espalhado nos componentes, que implementam o sistema. Caso seja necessário, evoluir ou melhorar tanto o tratamento excepcional (comportamento excepcional) quanto o comportamento normal será difícil, pois os comportamentos estão acoplados e espalhados pelos componentes da ALP.

Portanto, manter os comportamentos excepcional e normal da ALP separados é muito importante. Essa separação, além de promover a adaptabilidade e a reutilização, tanto do comportamento normal, quanto do excepcional, ainda pode possibilitar a adoção de

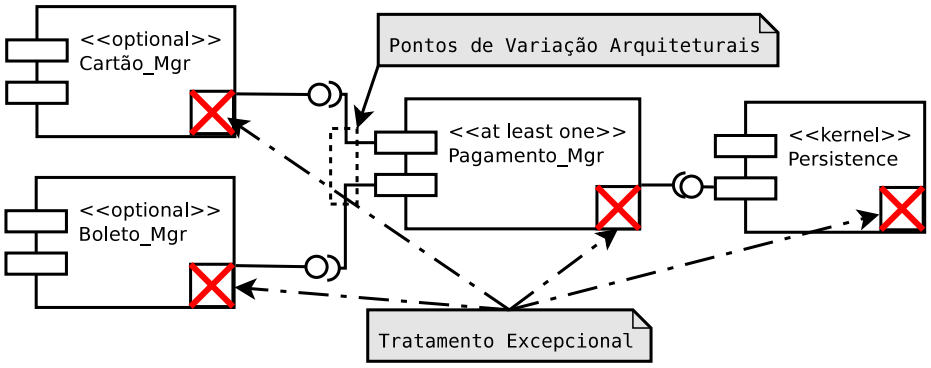


Figura 1.1: Definição do Problema em Componentes.

estratégias de tratadores de exceções que podem ser facilmente associados e desassociados. Assim, podemos projetar estratégias adaptáveis de tratamento de exceções que variam seu comportamento para atender as necessidades específicas de cada produto derivado da linha. Para as variabilidades que possam existir no comportamento excepcional são consideradas a escolha de diferentes tratadores de exceções, de acordo com as necessidades e os recursos específicos de cada produto da LPS.

Outro problema que pode ser encontrado em uma ALP baseada em componentes é o desejo de querer evoluir ou melhorar o comportamento excepcional que pode ser variável. Para que uma ALP baseada em componentes, que apresente variabilidades no comportamento excepcional, possa ser especificada é necessária a criação de um modelo que considere o tratamento de exceções de uma maneira global em nível arquitetural, e que possua as abstrações necessárias para a criação de tratadores de exceção variáveis em uma ALP, ficando, assim, mais fácil a evolução e a especificação de diferentes tratamentos através de refinamentos sucessivos, desde a arquitetura até a implementação.

O principal objetivo dessa dissertação é definir estratégias para que seja implementada variabilidade no tratamento de exceções em uma LPS no modelo arquitetural. Permitindo, assim, a separação do comportamento excepcional e normal presentes no modelo arquitetural, facilitando não só as evoluções do comportamento normal, quanto também, a do comportamento excepcional que podem ocorrer durante o ciclo de vida da linha de produtos. E, também, a rastreabilidade de ambos comportamentos em todo o modelo arquitetural e na implementação da LPS.

1.2 Solução Proposta

A solução proposta permite criar uma arquitetura de software baseada em componentes incluindo pontos de variação relacionados com diferentes opções de tratadores de exceção,

que podem ser selecionados durante as instanciações de produtos, fornecendo um tratamento excepcional diferente para cada um dos diferentes produtos. O método proposto, chamado MVTE (Método de Variabilidade de Tratamento de Exceções), auxilia a criação da linha de produtos começando com modelos de casos de uso até implementação da linha de produtos, ajudando assim a evoluir seu comportamento excepcional e sua instanciação em diferentes produtos. A solução é baseada no uso de conectores variáveis, que foram propostos por Dias [19], realizando a ligação entre o comportamento normal e o seu comportamento excepcional. Estes conectores, chamados de *Conectores-VP*, são baseados no conceito de conectores para implementar a variabilidade da arquitetura. Ao mesmo tempo que os *Conectores-VP* são empregados para ligar componentes, eles também encapsulam as decisões de variabilidade a partir dos componentes.

O MVTE foi proposto para que todas as pessoas interessadas na LPS consigam definir e entender o tratamento excepcional, que pode ser variável, desde os primeiros requisitos até o desenvolvimento e a fase de testes da LPS. Muitas vezes, o tratamento excepcional só é definido pelos desenvolvedores que detectam a possibilidade de erros durante o processo de codificação do sistema. Com o MVTE, esse tratamento excepcional já será definido nos levantamentos de requisitos.

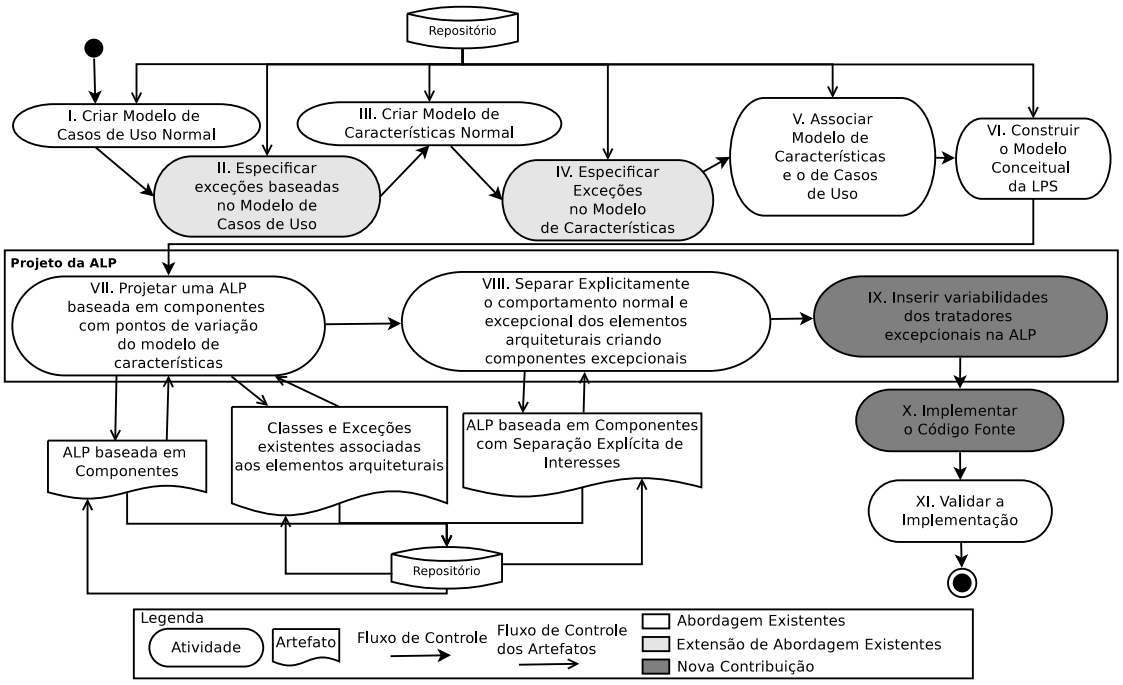


Figura 1.2: Diagrama do Método MVTE em UML.

Para o melhor acompanhamento de todos os interessados no desenvolvimento da LPS, o método é dividido em onze atividades, são elas: Atividade I - *Criar Modelo de Casos*

de Uso Normal, Atividade II - *Especificar Exceções baseadas no Modelo de Casos de Uso*, Atividade III - *Criar Modelo de Características Normal*, Atividade IV - *Especificar Exceções no Modelo de Características* Atividade V - *Associar o modelo de Características e o de Casos de uso*, Atividade VI - *Construir o Modelo Conceitual da LPS*, Atividade VII - *Projetar uma ALP baseada em componentes com pontos de variação do modelo de características*, Atividade VIII - *Separar Explicitamente o comportamento normal e excepcional dos elementos arquiteturais criando componentes excepcionais*, Atividade IX - *Inserir Variabilidades dos tratadores excepcionais na ALP*, Atividade X - *Implementar o Código Fonte*, Atividade XI - *Validar a Implementação*. Cada subseção representa uma atividade da Figura 1.2.

1.2.1 'Criar Modelo de Casos de Uso Normal' e 'Especificar Exceções baseadas no Modelo de Casos de Uso'

Essas duas atividades são as primeiras atividades do método MVTE. Elas contribuem para uma primeira identificação dos requisitos existentes na LPS, sendo que eles podem ser requisitos que compoem o comportamento normal ou o excepcional. A atividade "Criar Modelo de Casos de Uso Normal" com a variabilidade da LPS é uma atividade do método PLUS [25]. Esta atividade, conforme seu próprio nome diz, cria o modelo de casos de uso do comportamento normal da LPS de acordo com os requisitos definidos pelos usuários. A atividade "Especificar Exceções baseadas no Modelo de Casos de Uso" estende a atividade anterior para mostrar nos casos de uso as exceções que são lançadas por cada caso de uso com comportamento normal suas diferentes opções de tratadores excepcionais. Ambas atividades são documentadas em forma de tabela que são apresentadas nas Seções 3.1 e 3.2.

1.2.2 'Criar Modelo de Características Normal' e 'Especificar Exceções no Modelo de Características'

Nestas atividades é construído o modelo de características, que representa as variabilidades existentes na LPS. Assim, a atividade "Criar Modelo de Características Normal" permite a identificação das características (funções) do comportamento normal que a LPS irá prover e as variabilidades existentes. Essa atividade é outra atividade do método PLUS [25]. A atividade "Especificar Exceções no Modelo de Características" é responsável por adicionar a variabilidade relacionada ao tratamento excepcional no modelo de características, assim, após criado esse novo tipo de característica o modelo será capaz de representar o comportamento normal e o comportamento excepcional com suas variabilidades.

1.2.3 'Associar o modelo de Características e o de Casos de uso'

Para facilitar a rastreabilidade em todo o projeto da LPS é realizada uma associação dos modelos de caso de uso e do modelo de características. Assim, a próxima atividade, que é a "Associar o modelo de Características e o de Casos de uso" apoia essa rastreabilidade entre os dois modelos, consequentemente, melhorando a evolução do sistema.

1.2.4 'Construir o Modelo Conceitual da LPS'

"Construir o Modelo Conceitual da LPS" é uma atividade do método PLUS [25]. Para criar o modelo conceitual, temos as classes, os seus atributos, as suas operações e os relacionamentos entre as classes que serão modeladas. Durante a modelagem do problema, a ênfase é em determinar as classes que são entidades, seus atributos, e seus relacionamentos que definem o problema. O modelo representa classes que são do comportamento normal e do excepcional. Essa construção facilita o entendimento do sistema para os desenvolvedores, mostrando um modelo que é mais próximo de sua visão.

1.2.5 'Projetar uma ALP baseada em Componentes com Pontos de Variação do Modelo de Características'

A atividade ('Projetar uma ALP baseada em componentes com pontos de variação do modelo de características') tem por objetivo prover uma visualização em alto nível da estrutura do sistema mantendo a rastreabilidade do modelo de características.

Nesta atividade, são identificados os elementos arquiteturais (componentes e conectores) do modelo de característica. Após a sua identificação eles são associados com as exceções e as classes, e finalmente os elementos arquiteturais são conectados baseando-se nas dependências existente no modelo conceitual da LPS.

Ao executar esta atividade, é necessário refatorar a arquitetura de software com dois objetivos: (i) componentizar a ALP para especificar os pontos de variação arquitetural relacionadas ao tratamento excepcional, e (ii) reduzir o número de elementos arquiteturais para melhorar o entendimento do sistema.

1.2.6 'Separar Explicitamente o comportamento normal e excepcional dos elementos arquiteturais criando componentes excepcionais' e 'Inserir Variabilidades dos tratadores excepcionais na ALP'

Nas atividades "Separar Explicitamente o comportamento normal e excepcional dos elementos arquiteturais criando componentes excepcionais" e "Inserir Variabilidades dos

tratadores excepcionais na ALP" será separado explicitamente o comportamento normal e excepcional dos elementos arquiteturais. Estas atividades consistem na definição explícita dos componentes excepcionais, que tratam as exceções. O objetivo desta separação de interesses é agrupar os componentes que tratam as exceções não permitindo que elas fiquem espalhadas por todo o projeto, deste modo facilita a evolução e a implementação das variabilidades.

1.2.7 'Implementar o Código Fonte' e 'Validar a Implementação'

Na atividade "Implementar o Código Fonte" , algumas regras devem ser seguidas: (i) uso do modelo de implementação de componentes COSMOS* [24] e COSMOS*-VP [19]; (ii) gerar o código fonte da estrutura arquitetural; (iii) encapsular as classes associadas dos elementos arquiteturais; (iv) montar a configuração arquitetural.

Essa atividade provê um guia para especificar os conectores dos pontos de variação arquitetural, chamados *Conectores-VP* e inseri-los no código fonte. Os *Conectores-VP* inserem os pontos de variação do tratamento excepcional que já é conhecido durante o desenvolvimento da ALP.

"Validar a Implementação" é uma atividade para assegurar que o sistema esteja funcionando corretamente como os documentos de requisitos estejam pedindo. Esta atividade deve ser assegurada durante todo o processo de desenvolvimento do sistema, porém essa atividade não é tratada com ênfase neste trabalho.

1.3 Trabalhos Relacionados

Alguns trabalhos foram encontrados tendo temas similares como é o caso dos artigos de Adachi et al. [5, 4] e da dissertação de mestrado de Brito [9]. O artigo de Adachi et al. [5] e a dissertação de Brito [9] é sobre o tratamento de exceções. Já o outro artigo do Adachi et al. [4] é sobre o gerenciamento da variabilidade de software usando aspectos.

Adachi et al. [5] propõem e avaliam um conjunto de três estratégias heurísticas recomendadas para o uso de tratamento de exceções. O objetivo das heurísticas propostas é encontrar fragmentos de códigos que implementam o tratamento de exceções e recomendar uma lista desses fragmentos classificados de acordo com a relevância do contexto para os desenvolvedores. Além disso, essa heurística auxilia os desenvolvedores no processo de descoberta das ações relevantes para os tratamentos de exceções.

Brito [9] propõe um método para auxiliar a modelagem do comportamento excepcional de sistemas baseados em componentes, chamado MDCE+. O MDCE+ é voltado para o desenvolvimento de sistemas confiáveis e é centrado na arquitetura. O foco na arquitetura de software contribui para uma melhor definição e análise do fluxo de exceções entre

os componentes do sistema. Essa maneira estruturada de detectar e tratar exceções no contexto da ocorrência de falhas é particularmente relevante para sistemas que apresentam requisitos de confiabilidade. Este método não trata da variabilidade de exceções que podem acontecer em Linha de Produtos de Software.

Adachi et al. [4] propõem uma abordagem para gerenciar variabilidades em especificações arquiteturais de LPS. A abordagem combina uma linguagem de descrição arquitetural orientada a aspectos com uma linguagem de modelagem de variabilidades. É apresentada a aplicação da abordagem na linha de produtos de software *MobileMedia*.

1.4 Organização deste Trabalho

Este documento foi dividido em cinco capítulos, organizados da seguinte forma:

- **Capítulo 2 - Fundamentos e Conceitos** - Este capítulo apresenta os fundamentos e os conceitos que serão utilizados neste trabalho.
- **Capítulo 3 - Método MVTE** - Este capítulo apresenta o método proposto para inserirmos variabilidade de tratamento de exceções em Linha de Produtos de Software. O método começa desde a fase dos casos de uso até a fase de implementação do sistema, inserindo este tipo de tratamento na arquitetura da linha de produtos.
- **Capítulo 4 - Estudos Empíricos** - Neste capítulo são apresentados dois estudos empíricos para avaliar o método MVTE. Um deles é o E-Gov-CNH, que é um sistema de obtenção da licença para dirigir no Brasil, e o outro é o MobileMedia, que é um sistema de gerenciamento de mídias em celulares.
- **Capítulo 5 - Conclusões** - Neste capítulo são apresentadas as conclusões deste trabalho. Além disso, são apresentadas as contribuições e alguns direcionamentos para trabalhos futuros.

Capítulo 2

Fundamentos de Reuso de Software e Tolerância a Falhas

Este capítulo apresenta os fundamentos teóricos necessários para desenvolver o trabalho. Estes conceitos se separam em oito seções. As Seções 2.1 a 2.7 apresentam fundamentos relacionados a reuso de software. Nestas seções se encontram os seguintes tópicos: Linha de Produtos de Software (Seção 2.1), Arquitetura de Software (Seção 2.2), Desenvolvimento baseado em Componentes (Seção 2.3), Arquitetura de Linha de Produtos de Software baseadas em Componentes (Seção 2.4), Modelo de Implementação COSMOS* (Seção 2.5), COSMOS*-VP (Seção 2.6) e Programação Orientada a Aspectos (Seção 2.7). Em seguida, são apresentados os fundamentos relacionados à tolerância a falhas. Os fundamentos são apresentados nas seguintes Seções: Tolerância a Falhas (Seção 2.8), Tratamento de Exceções (Seção 2.9), Uma Arquitetura Confiável Baseada em Tratamento de Exceções (Seção 2.10), Componente Tolerante a Falhas Ideal (Seção 2.11).

2.1 Linha de Produtos de Software

Atualmente, muitos esforços estão sendo feitos para se obter um alto grau de reutilização durante o desenvolvimento de sistemas. Linha de Produtos de Software (LPS) é uma abordagem moderna pra promover a reutilização de artefatos de software. Uma LPS é definida por um conjunto de produtos de software com alto grau de similaridade entre si, que atendem as necessidades específicas de um segmento de mercado ou missão, e que são desenvolvidos de forma prescritiva a partir de um conjunto de ativos centrais[1] [15].

As próximas seções (Seção 2.1.1, 2.1.2, 2.1.3 e a 2.4) apresentam características importantes para o desenvolvimento de uma Linha de Produtos de Software.

[1] Inglês: *core assets*.

2.1.1 Engenharia de Linha de Produtos

Engenharia de Linha de Produtos é o enfoque que trata os produtos de software desenvolvidos por uma empresa como membros de uma mesma família de produtos que compartilham várias funcionalidades em comum. O gerenciamento de funcionalidades comuns e variáveis da família de produtos não é uma tarefa simples, e para isso a engenharia de linhas de produto se divide em dois ciclos de vida de desenvolvimento concorrentes: engenharia de família e engenharia de aplicação [15].

A *engenharia de família* analisa os produtos, já existentes e planejados, de uma empresa em relação às suas funcionalidades comuns e variáveis, que são então utilizadas para construir uma infraestrutura para a linha de produtos, como um repositório de artefatos reutilizáveis. Componentes reutilizáveis e a Arquitetura da Linha de Produtos (ALP) são exemplos de artefatos de infraestrutura.

A *engenharia de aplicação* usa a infraestrutura criada para derivar produtos específicos [39]. Por exemplo, para derivar um produto, a ALP e os componentes reutilizáveis são selecionados para criar uma configuração arquitetural, derivada da ALP, contendo as funcionalidades requeridas pelo produto.

A engenharia de família e a engenharia de aplicação são fortemente integradas. Enquanto que a engenharia de família cobre o desenvolvimento de artefatos comuns reutilizáveis previamente planejados, a engenharia de aplicação deriva produtos a partir destes artefatos. Se houver dificuldades na derivação de um novo produto, gerada a partir do não planejamento de algum artefato necessário, por exemplo, a engenharia de aplicação deve fornecer um *feedback* à engenharia de família. O *feedback* pode conter também informações que tornem mais fáceis a derivação de produtos.

2.1.2 Modelo de Características

Característica[2] é uma propriedade de sistema que é relevante para alguma parte interessada[3] [17]. Na definição inicialmente proposta por Kang et al. [32], são atributos de um sistema que afetam diretamente o usuário final. Além das Características serem um importante conceito em LPS, elas podem ser usadas para identificar funcionalidades comuns a todos os produtos e funcionalidades variáveis de uma Linha de Produtos de Software [25].

As características podem ser classificadas em mandatórias, opcionais ou alternativas. As características comuns ou mandatórias de uma LPS são aquelas que estão presentes em todos os produtos da LPS. Dessa forma, identificam o núcleo da LPS, ou seja, o conjunto de funcionalidades que serão providas por todos os produtos derivados da LPS. Identificar

[2]Inglês: *Feature*
[3]Inglês: *Stakeholder*

uma característica como variável consiste em definir que tal característica, de acordo com as especificações de cada produto, pode ser provida ou não por um produto derivado da LPS.

Gomaa [25] define três tipos de características:

- **Mandatórias:** São as características que devem ser providas por todos os produtos derivados da linha. Essas características também podem ser chamadas de características comuns. Pode-se agrupar todos os requisitos comuns em apenas uma característica mandatória ou identificá-los separadamente em várias características mandatórias. Isso irá depender do enfoque que se queira dar para o modelo de características. Por exemplo, se todos os requisitos comuns são bem entendidos pode-se concentrar apenas nas características variáveis, escolhendo representar todos os requisitos comuns em apenas uma característica mandatória. Entretanto, quando os requisitos não são bem entendidos, ou deseja-se identificar as características antes de defini-las como mandatórias ou não, a segunda abordagem é a mais apropriada. Gomaa define a utilização do estereótipo `«kernel»` para representar características mandatórias em diagramas de características.

- **Opcionais:** São características providas apenas por alguns dos produtos da linha. Todas as características opcionais podem depender de características mandatórias, uma vez que as mandatórias sempre estarão presentes em um produto. Características opcionais são identificadas, em diagramas de características, através do estereótipo `«optional»`.

- **Alternativas:** Duas ou mais características podem ser alternativas entre si. Assim como as opcionais, elas podem assumir que as características mandatórias serão providas. As características alternativas, juntamente com as opcionais, podem ser referenciadas como características variáveis ou não-mandatórias, já que sua presença varia conforme cada produto. Características alternativas são identificadas utilizando o estereótipo `«alternative»`.

As características de uma LPS, ou de um sistema de software, são comumente documentadas em um *Diagrama de características*, representando a decomposição hierárquica das características. No diagrama, além de identificadas através de seu tipo, as características podem ser agrupadas de acordo com suas restrições de seleção. Tais restrições limitam o conjunto de possíveis combinações de características variáveis (opcionais ou alternativas) a serem providas por um dado produto da LPS. As características de um diagrama podem ser agrupadas em:

- **Zero ou uma do grupo de características:** Em uma LPS, dado um grupo de características opcionais apenas uma deve ser escolhida para compor um produto,

ou seja, as características desse grupo são mutuamente exclusivas. Gomaa define que grupos de características desse tipo devem ser identificados utilizando o estereótipo `«zero-or-one-of-feature-group»`.

- **Apenas uma do grupo de características:** E outro caso em que as características do grupo são mutuamente exclusivas. Porém, neste caso obrigatoriamente uma das características do grupo deve estar presente em cada um dos produtos da LPS. Grupos de características desse tipo são identificadas, em diagramas de características, utilizando o estereótipo `«exactly-one-of-feature-group»`.

- **Pelo menos uma do grupo de características:** Desse grupo, podemos escolher uma ou mais características, mas tempos a restrição de que pelo menos uma delas devem compor cada produto da LPS. O estereótipo `«at-least-one-of-feature-group»` é usado para identificar grupos de características desse tipo.

- **Zero ou mais do grupo de características:** Desse grupo podemos escolher quantas características desejarmos. Embora esse grupo pareça desnecessário, uma vez que não há restrição de seleção de características, pode ser vantajoso agrupar um conjunto de características que dependam de alguma outra característica, por exemplo. Grupos de características desse tipo são identificadas, em diagramas de características, utilizando o estereótipo `«zero-or-more-of-feature-group»`.

Para fazer os diagramas de característica presente neste trabalho, utilizamos o método apresentado em Gomaa[25]. O método de Gomaa foi escolhido por apresentar semelhanças com o UML, assim será mais fácil compreende-lo ao perceber que está familiarizado com a representação gráfica do método. Neste trabalho, visando a simplificação da representação gráfica de diagramas de características, removemos o sufixo `of feature group` dos estereótipos utilizados para identificar tipos de grupos de características. Por exemplo, ao invés de utilizar o estereótipo `«zero-or-one-of-feature-group»` para identificar um grupo de características como sendo do tipo **Zero ou uma do grupo de características**, utilizamos o estereótipo `«zero-or-one »`.

A Figura 2.1 representa parte de um diagrama de características de uma LPS de aplicações para Hotéis. As características `Hotel`, `Reserva` e `Simples`, que são responsáveis por reservar quartos individualmente, são mandatórias, enquanto as demais são características variáveis. Nesse exemplo, todas as características variáveis estão agrupadas sobre algum tipo de grupo de características. Por exemplo, as características alternativas `Localmente` e `Remotamente` estão dentro do grupo `Log` do tipo **zero-ou-uma**. Portanto, elas são mutuamente exclusivas, ou seja, apenas uma das características não funcionais de *log* pode ser selecionada. Nesse caso, um dado produto fará *logging* de sua execução apenas localmente ou remotamente. O grupo de características `Coletiva` do tipo

zero-ou-mais, que agrupa as características `Turistas` e `Conferências`. Desse grupo podemos selecionar zero, uma ou as duas características. As características referentes a pagamento com cartão e com boleto, `Cartão` e `Boleto`, respectivamente, estão agrupadas por `Pagamento`, que é do tipo **pelo-menos-uma**. Então, ao menos uma das características responsáveis por realizar pagamentos deve ser provida por todos os produtos da linha.

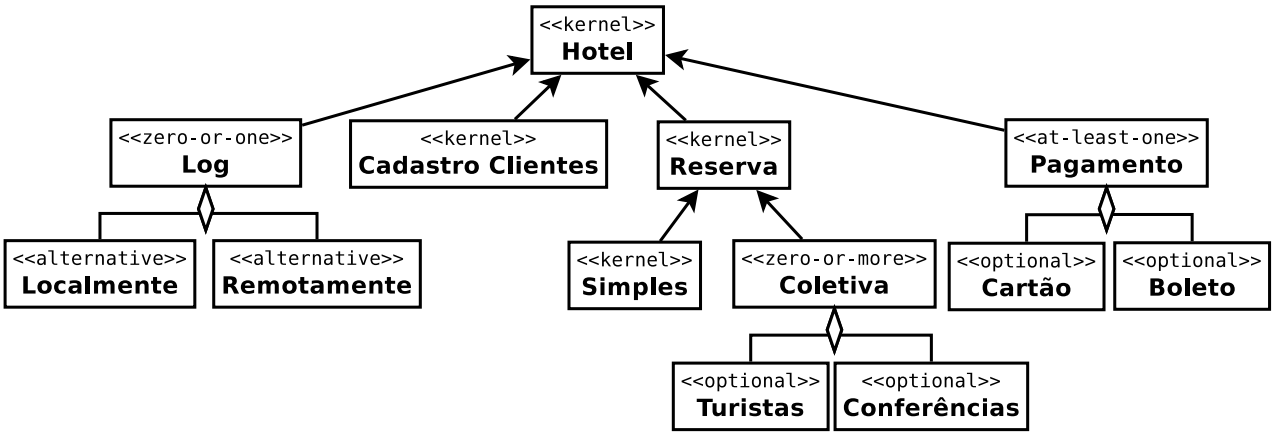


Figura 2.1: Parte de um diagrama de características de uma LPS de Hotel.

2.1.3 Variabilidade de Software

O conceito de *variabilidade de software* é fundamental no contexto de reutilização e evolução de software. *Variabilidade de software* é a capacidade de um sistema de software ou artefato ser modificado ou configurado, para ser utilizado em um contexto específico [33]. Um alto grau de variabilidade permite a utilização do artefato de software em um contexto mais amplo, o que torna o artefato mais reutilizável. É possível antecipar alguns tipos de variabilidade e construir sistemas que facilitem este tipo de variabilidade. Em LPS [15], por exemplo, os artefatos de software devem ser flexíveis o suficiente de modo a permitir que detalhes de implementação de um produto específico possam ser postergados para fases posteriores do desenvolvimento.

As decisões de projeto postergadas são denominadas *pontos de variação*. Um ponto de variação é o local do artefato de software em que uma decisão de projeto pode ser tomada, e *variantes* são alternativas de projeto associadas a este ponto [31]. Artefatos de diferentes níveis de abstração do sistema podem conter pontos de variação. Como por exemplo, classes, componentes, arquiteturas, etc. podem conter pontos de variação.

A Figura 2.2 apresenta um exemplo de ponto de variação em um diagrama de classes. A classe `Hotel` é composta por `Cliente`, `Reserva` e `Pagamento`. A classe `Pagamento` tem

duas classes agregadas, `Cartão` e `Boleto`. Há um ponto de variação, quer permite variar o tipo de pagamento a ser criado. O ponto de variação permite criar pagamentos com cartão, selecionando a classe `Cartão`, ou pagamento com boletos, selecionando `Boletos`.

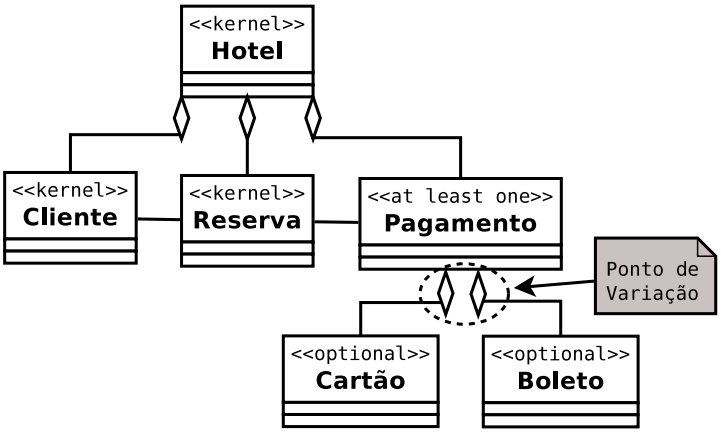


Figura 2.2: Parte de um diagrama de classes com um ponto de variação.

Uma parte da arquitetura de um sistema de vendas ilustrativo com ponto de variação é apresentada na Figura 2.3. A figura segue as definições de Gomaa [25] para identificar componentes alternativos e mandatórios. Na arquitetura há o componente `Cadastro` (mandatório) responsável pelo cadastramento de clientes adicionando novos clientes ao sistema. Há duas opções para a forma com que os clientes são persistidos pelo sistema. O sistema pode ser capaz de persistir diretamente em arquivos do Sistema de Arquivos, utilizando o componente `SistemaArquivos`, ou pode usar um Banco de Dados para persistir seus clientes, através do componente `BancoDeDados`. Essa escolha, entre qual componente será responsável por persistir os dados, representa um ponto de variação arquitetural do sistema.

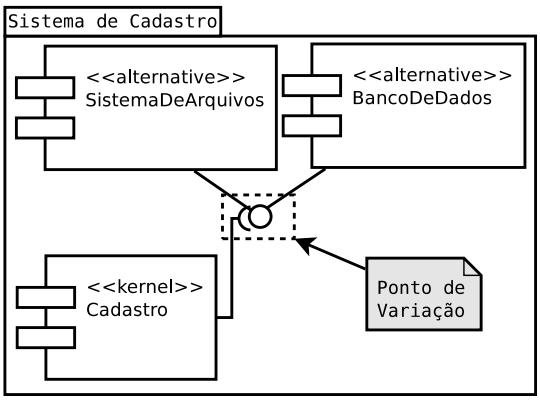


Figura 2.3: Parte de uma arquitetura baseada em componentes com um ponto de variação.

Analogamente com o que acontece com as características de um sistema ou LPS, os pontos de variação também podem apresentar restrições de seleção. Por exemplo, nas figuras 2.2 e 2.3 os pontos de variação são do tipo mutuamente exclusivo, ou seja, apenas uma de suas alternativas variantes pode ser selecionada para compor um produto. Um carro não pode conter dois tipos de motores ao mesmo tempo, e o Sistema de Pontos de Venda deve persistir seus dados através de apenas um dos mecanismos possíveis. Além de afetar um mesmo ponto, distintos pontos de variação podem ser afetados por uma restrição. As restrições de seleção podem ser dos tipos: (i) **restrição exclusiva**, a seleção de uma das variantes em um ponto de variação impede que alguma outra variante seja selecionada em outro ponto, ou no mesmo; (ii) **restrição inclusiva**, a seleção de uma das variantes em um ponto de variação acarreta na seleção de uma determinada variante em outro ponto, ou no mesmo.

O *tempo de resolução de variabilidades*[4] indica o tempo em que deve ser selecionada uma das variantes associadas a um ponto de variação. Na classificação proposta por Anastasopoulos e Gacek [23], o tempo de resolução de variabilidade pode ser: (i) tempo de pré-compilação; (ii) tempo de compilação; (iii) tempo de ligação; (iv) tempo de execução; e (v) tempo de atualização, após execução.

O tempo de resolução de um ponto de variação está diretamente ligado com a técnica utilizada para implementar a variabilidade contida na ponto. A seguir listamos algumas das mais populares técnicas de implementação de variabilidades utilizadas atualmente.

2.1.4 Técnicas de Implementação de Variabilidade

Existem diversas técnicas de implementação de variabilidades, como agregação/delegação, herança, parametrização, sobrecarga, reflexão computacional, programação orientada a aspectos, entre outras. Cada uma traz características específicas que as tornam menos ou mais eficazes, na implementação de variabilidades, de acordo com o nível de abstração da variabilidade em questão. Em um contexto baseado em componentes, os diferentes tipos de variabilidades podem ser classificados, de acordo com seu nível de abstração, em:

- Variabilidades Arquiteturais: São variabilidades cuja resolução irá afetar a arquitetura, ou a configuração arquitetural, em si. Por exemplo, a necessidade de escolha de um componente arquitetural entre dois ou mais componentes similares para compor uma determinada configuração arquitetural.

- Variabilidades de Componentes: São variabilidades internas aos componentes. Suas decisões tomadas com relação a esse tipo de variabilidade apenas afeta componentes, não afeta a arquitetura do sistema.

[4]Inglês: *variability binding time*

Nos trabalhos de Anastasopoulos e Gacek [23] e Jacobson, Griss e Jonsson [29] os autores realizaram trabalhos de identificação e comparação de técnicas de implementação de variabilidades. Os principais critérios utilizados para a comparação foram os diferentes tipos de variabilidades e o tempo de resolução das variabilidades fornecido pela técnica. Algumas das técnicas que foram comparadas são:

- Agregação/Delegação: é uma técnica de orientação a objetos que possibilita que um objeto provenha funcionalidades, delegando requisições a outros objetos agregados a ele. Variabilidades podem ser tratadas através dessa técnica implementando a funcionalidade comum, ou mandatória, no objeto "todo" da agregação e as funcionalidades variáveis em objetos variantes que serão agregados ao "todo". Dessa forma, para variar o comportamento da agregação basta variar sua a composição, selecionando diferentes agregados. Para variabilidades simples a agregação é uma boa alternativa. Entretanto, conforme o tamanho da variabilidade aumenta, a complexidade da agregação cresce muito rapidamente. Quando o tamanho ou a complexidade da variabilidade é grande, pode haver a necessidade que objetos agregadores agreguem não objetos simples, mas outra agregação, que por sua vez, agregue outra agregação, e assim por diante. Nesses casos a rastreabilidade entre as variabilidades de um nível mais alto, variabilidades nos requisitos do sistema por exemplo, e o código fonte é bastante comprometida [23]. O que pode acarretar em um aumento significativo do esforço necessário durante a evolução do sistema. A agregação tipicamente requer que a variabilidade seja decidida em tempo de compilação, porém utilizando-a combinadamente com outras técnicas, como carga dinâmica de classes, que outros tempos de resolução possam ser obtidos.

- Herança: herança é uma técnica de orientação a objetos que possibilita que funcionalidades padrões sejam implementadas em superclasses, e suas extensões sejam implementadas em subclasses. Existem diversos tipos de herança, porém focaremos apenas na chamada herança padrão, ou herança simples de classes. Nela uma subclasse pode introduzir novos parâmetros e métodos e sobrescrever métodos já existentes na superclasse estendida. Nessa técnica, variabilidades são implementadas através da implementação de funcionalidades mandatórias nas superclasses e funcionalidades variáveis (opcionais ou alternativas) nas subclasses. Assim como a agregação, quando a variabilidade em questão é complexa, ou seja, envolve muitas classes, o esforço para implementá-la utilizando herança torna-se muito grande [23]. Variabilidades implementadas utilizando herança devem ser resolvidas antes da compilação do sistema.

- Parametrização: a ideia da programação parametrizada é criar uma biblioteca de componentes parametrizados que poderão variar seu comportamento dependendo

do conjunto de valores escolhidos para seus parâmetros. Um tipo interessante de parametrização é a chamada parametrização dinâmica, na qual o comportamento de um componente é definido em tempo de execução, através de definição do valores de parâmetros também em tempo de execução. Por meio de diferentes subtipos dessa técnica, todos os tempos de resolução podem ser utilizados [23]. Dentre seus subtipos, o polimorfismo paramétrico é muito popular. Um típico exemplo de polimorfismo paramétrico é a implementação de uma classe 'Pilha' genérica capaz de empilhar qualquer tipo de elementos. Porém, apenas é possível empilhar um tipo de elemento por vez. O tipo empilhada a cada momento é definido por meio de parâmetros. Variabilidades implementadas utilizando-se de polimorfismo paramétrico devem ser resolvidas no máximo em tempo de compilação.

- Carga Dinâmica de Classes: é um mecanismo utilizado na linguagem em Java onde, ao invés de se carregar todas as classes necessárias de uma só vez, durante o processo de inicialização do sistema, as classes são carregadas assim que necessário. Dessa forma, cada uma das possíveis variantes de uma variabilidade é implementada em uma classe distinta. É possível, então, decidir em tempo de execução qual das classes alternativas deve ser carregada para prover a funcionalidade variável. Portanto, pode se postergar a resolução da variabilidade até a execução do sistema. Anastasopoulos e Gacek [23] apontam-na como uma técnica interessante para o contexto de LPS, pois, por meio dela, é possível que um produto descubra em tempo de execução qual o contexto no qual está inserido, e decidir de qual a melhor classe a carregar para prover cada uma das funcionalidades variáveis. Cada classe é selecionada dentre o conjunto de classes que implementam a mesma funcionalidade.

- Compilação Condicional: é um mecanismo que provê controle sobre quais trechos de códigos devem ser incluídos ou excluídos da compilação. A principal vantagem que Compilação Condicional oferece ao contexto de LPS é a possibilidade de manter no mesmo código todas as possibilidades de derivação de produtos. Como pode-se escolher quais trechos de código compilar, uma classe que apresente variabilidades é implementada incluindo todas as suas variantes no mesmo código, como se a classe provesse todas as suas variabilidades de uma só vez. Porém, cada trecho responsável por uma alternativa deve ser envolvido por uma marcação. Dessa forma, é dito ao compilador quais trechos que devem ser compilados informando os marcadores que os envolvem. A principal desvantagem da Compilação Condicional é o emaranhamento entre código que implementa funcionalidades mandatórias e códigos de funcionalidades opcionais e alternativas. Devido à necessidade de se decidir quais trechos de código devem ser compilados, dentre os responsáveis por prover as funcionalidades variáveis, as variabilidades devem ser resolvidas no máximo em tempo

de compilação.

- Reflexão Computacional: é uma técnica complexa que pode ser usada para interceptar e modificar o comportamento de operações básicas do modelo de objetos, como a criação de instâncias ou a invocação de operações dos objetos. A técnica é bastante interessante para implementar requisitos não funcionais, como tolerância a falhas, de forma transparentes às funcionalidade de um sistema [10]. No contexto de LPS, Reflexão Computacional pode ser usada para implementar as variabilidades da linha de maneira transparente à suas partes comum. Essa separação entre código variável e código comum pode facilitar futuras evoluções da linha. Devida a sua complexidade, a técnica é indicada para implementação de variabilidades arquiteturais. Uma das suas principais vantagens é a possibilidade de que a utilizando conjuntamente com carga dinâmica de classes é possível realizar reconfigurações arquiteturais em tempo de execução. Existem diversas abordagens de Reflexão Computacional permitindo que variabilidades sejam resolvidas em diferentes tempos, desde em tempo de compilação até em tempo de execução [23].

- Programação Orientada a Aspectos (POA): oferece mecanismos para a modularização de interesses transversais de um sistema de software [34]. Os novos módulos que modularizam esses interesses, outrora espalhados por todo o sistema, são tipicamente chamados de aspectos. Pode-se utilizar aspectos para modularizar variabilidades de uma LPS [21]. Dessa forma, as funcionalidades comuns são implementadas da maneira convencional, e as funcionalidades variáveis através de aspectos. As funcionalidades comuns e variáveis são posteriormente combinadas, em tempo de combinação para compor o produto final. O tempo de combinação varia de acordo com a linguagem orientada a aspectos utilizada. Neste caso, o tempo de resolução de variabilidades depende do tempo de combinação usado pela linguagem de POA utilizada. POA é mais bem detalhada na Seção 2.7.

Devido ao fato de não haver uma técnica que atenda a todos os aspectos específicos de cada um dos cenários encontrados na prática, diferentes contextos de variabilidades, que podem ocorrer em um único sistema ou LPS, devem ser analisados de maneira individual a fim de se identificar a técnica que melhor se encaixa [23]. A combinação de várias técnicas é considerada pelos autores a melhor abordagem.

2.2 Arquitetura de Software

A *arquitetura de software*, através de um alto nível de abstração, define o sistema em termos de seus **componentes arquiteturais**, que representam unidades abstratas do

sistema; a interação entre essas entidades, que são materializadas explicitamente através dos **conectores**; e os atributos e funcionalidades de cada um [44]. O modo como os elementos arquiteturais de um sistema ficam arranjados é conhecido como **configuração**.

Essa visão estrutural da arquitetura de um sistema proporciona benefícios importantes, que são imprescindíveis para o desenvolvimento de sistemas de software complexos. Os principais benefícios são: (i) a organização do sistema como uma composição de componentes lógicos; (ii) a antecipação da definição das estruturas de controle globais; (iii) a definição da forma de comunicação e composição dos elementos do projeto; e (iv) o auxílio na definição das funcionalidades de cada componente projetado. Além disso, uma propriedade arquitetural representa uma decisão de projeto relacionada a algum requisito não-funcional do sistema, que quantifica determinadas qualidades do seu comportamento, como confiabilidade, reusabilidade e manutenabilidade [6, 44].

A importância da arquitetura fica ainda mais clara no contexto do desenvolvimento baseados em componentes. Isso acontece porque, na composição de sistemas, os componentes interagem entre si para oferecer as funcionalidades desejadas. Além disso, devido à diferença de abstração entre a arquitetura e a implementação de um sistema, um processo de desenvolvimento baseado em componentes deve apresentar uma distinção clara entre os conceitos de componente arquitetural, que é abstrato e não é necessariamente instanciável, e componente de implementação, que representa a materialização de uma especificação em alguma tecnologia específica e deve necessariamente ser instanciável.

2.3 Desenvolvimento baseado em Componentes

A ideia de utilizar o desenvolvimento baseado em componentes (DBC) na produção de software é datada em 1976 [38]. O interesse pelo DBC só foi intensificado vinte anos depois, após a realização do Primeiro Workshop Internacional em Programação Orientada a Componentes, o WCOP'96.

As grandes pressões sofridas na indústria de software por produtos de melhor qualidade e por prazos mais curtos vêm ajudando na popularização do DBC. No DBC, um sistema é construído a partir da composição de componentes de software que já foram previamente especificados, construídos, testados e empregados, o que concede um ganho de produtividade e qualidade no software produzido.

A reutilização de componentes existentes garante esse aumento da produtividade na construção de novos sistemas. E o aumento da qualidade é uma consequência do fato dos componentes utilizados já terem sido empregados e testados em outros contextos. Algumas vezes se o componente for mal empregado e mal testado, ele pode facilitar a replicação de erros.

Ainda não existe um consenso geral sobre a definição de um componente de software,

que é a unidade básica de desenvolvimento do DBC, apesar da sua popularização. Porém, um aspecto muito importante é sempre ressaltado na literatura: um componente deve encapsular dentro de si seu projeto e sua implementação, além de oferecer interfaces bem definidas para o meio externo. O baixo acoplamento decorrente dessa política proporciona muitas flexibilidades, tais como: (i) facilidade de montagem de um sistema a partir de componentes COTS[5]; e (ii) facilidade de substituição de componentes que implementam interfaces equivalentes.

Uma definição complementar de componentes, adotada na maioria dos trabalhos publicados atualmente, foi proposta em 1998 [46]. Segundo Szyperski, um componente de software é uma unidade de composição com interfaces especificadas através de contratos e dependências de contexto explícitas. Sendo assim, além de explicitar as interfaces com os serviços oferecidos (**interfaces providas**), um componente de software deve declarar explicitamente as dependências necessárias para o seu funcionamento, através de suas **interfaces requeridas**.

Além dessa distinção clara entre interfaces providas e requeridas, os componentes seguem três princípios fundamentais, que são comuns à tecnologia de objetos [11]:

1. **Unificação de dados e funções:** Um componente deve encapsular o seu estado (dados relevantes) e a implementação das suas operações oferecidas, que acessam esses dados. Essa ligação estreita entre os dados e as operações ajuda a aumentar a coesão do sistema;

2. **Encapsulamento:** Os clientes que utilizam um componente devem depender somente da sua especificação, nunca da sua implementação. Essa separação de interesse[6] reduz o acoplamento entre os módulos do sistema, além de melhorar a sua manutenção;

3. **Identidade:** Independentemente dos valores dos seus atributos de estado, cada componente possui um identificador único que o difere dos demais.

A fim de contextualizar o componente em relação aos diferentes níveis de abstração que ele pode ser analisado, um componente pode ser observado através de diferentes pontos de vista [11]. Em outras palavras, dependendo do papel que o interessado[7] exerce no processo de desenvolvimento, a abstração como o componente é analisado pode variar. A Tabela 2.1 descreve os diferentes pontos de vista de um componente [11].

[5]Inglês: *Common-Off-The Shelf*
[6]Inglês: *Separation of Concerns*
[7]Inglês: *Stakeholder*

Tabela 2.1: Pontos de Vista de um Componente

Ponto de Vista	**Descrição**
Especificação	é constituído da especificação de uma unidade de software que descreve o comportamento de um componente. Esse comportamento é definido como um conjunto de interfaces providas e requeridas.
Plataforma	São definidas restrições que o componente deve seguir a fim de ser utilizado em alguma plataforma especifíca de desenvolvimento. As principais plataformas comerciais existentes são Enterprise Java Beans, Corba, COM+.
Implentação	é a realização de uma especificação. Sendo assim, a implementação de um componente está para a sua especificação, assim como uma classe está para a sua interface. Esse ponto de vista é compartilhado por todos os componentes já implementados em alguma linguagem de programação. Ele representa componentes prontos para serem instalados, como por exemplo, os componentes COTS.
Instalação	é uma instalação ou cópia de um componente implementado. Esse ponto de vista é utilizado na análise do ambiente de execução. Neste caso, pode ser feita uma analogia às classes localizadas no *classpath*. O *classpath* é a lista de diretórios onde a máquina virtual procura as classes a serem instanciadas.
Instanciação	é uma instância de um componente instalado. Esta é a visão de execução de componente, com os seus dados encapsulados e um identificador único. É semelhante ao conceito de objetos no paradigma de orientação a objetos.

2.3.1 UML Components

O *UML Components* é um processo de desenvolvimento de sistemas baseado em componentes. Ele é um processo simples e de fácil utilização prática. O *UML Components* possui uma arquitetura de quatro camadas pré-definida (Figura 2.4). E as camadas são as seguintes:

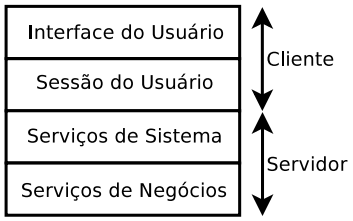


Figura 2.4: Arquitetura do *UML Components* [11].

- **Interface do Usuário**: camada que apresenta as informações para o usuário e que captura as informações inseridas;
- **Sessão do Usuário**: camada que gerencia o diálogo do usuário na sessão;

- **Serviços do Sistema**: camada de representação externa do sistema, provendo acesso aos seus serviços;
- **Serviços de Negócio**: camada com a implementação do núcleo dos negócios, informações e regras.

Na Figura 2.5 é exibido as fases do desenvolvimento do processo do *UML Components*. O desenvolvimento é distribuído em seis fases, que são: (i) especificação de requisitos[8]; (ii) especificação dos componentes[9]; (iii) provisionamento dos componentes[10]; (iv) montagem do sistema[11]; (v) testes[12]; (vi) implantação.

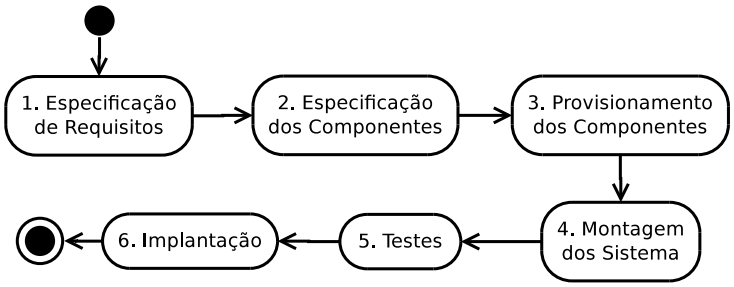


Figura 2.5: Fases do *UML Components* [11].

2.3.2 Especificação de Requisitos

Na fase de especificação dos requisitos, as funcionalidades do sistema são representadas através de `casos de uso`. Além disso, é especificado o modelo conceitual do negócio, que representa as entidades básicas da lógica do mesmo. Dada a sua importância, esses artefatos são considerados as principais saídas desta fase. O modelo conceitual do negócio representa o domínio do problema. Deste modo, ele necessita ser entendido e firmado entre os interessados no sistema. O propósito principal desse modelo é proporcionar um vocabulário comum entre os envolvidos no projeto. Além disso, esse modelo é a base para a identificação dos componentes da camada de negócio.

Os casos de uso são utilizados como notação para representar os requisitos funcionais do sistema. O modelo inicial de casos de uso pode ser construído com as principais funcionalidades do sistema. Após o início da especificação, esse modelo pode ser refinado para contemplar os demais requisitos funcionais.

[8]Inglês: *requirements definition*
[9]Inglês: *specification*
[10]Inglês: *provisioning*
[11]Inglês: *assembly*
[12]Inglês: *test*

Apesar de deixar bem claro o papel desta fase, inclusive descrevendo os artefatos que devem ser produzidos nela, o *UML Components* não detalha as atividades da sua execução.

2.3.3 Especificação de Componentes

A fase de especificação dos componentes é a fase do desenvolvimento que o *UML Components* mais entra em detalhes. Como o próprio nome indica, essa fase é responsável por especificar os componentes do sistema. Essa especificação acontece tomando como entrada os artefatos produzidos na fase de especificação de requisitos: (i) modelo conceitual do negócio; e o (ii) modelo de casos de uso.

Os principais artefatos de saída da fase de especificação dos componentes são três: (i) especificação das interfaces (refinamento das assinaturas); (ii) especificação dos componentes (interfaces providas e requeridas); e (iii) a arquitetura do sistema, como consequência da definição das dependências entre os componentes. Como pode ser visto na Figura 2.6 para produzir esses artefatos, o *UML Components* divide essa fase em três estágios: (i) identificação dos componentes; (ii) interação entre os componentes; e (iii) especificação final. A seguir, é explicado o papel de cada um dos estágios de especificação.

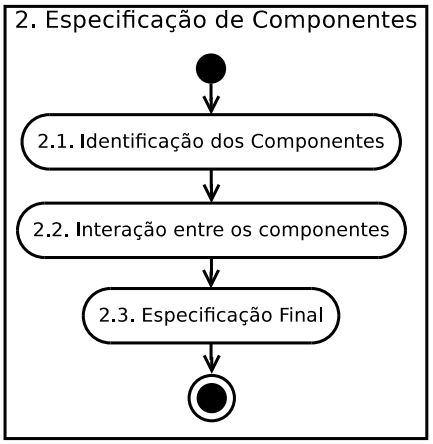


Figura 2.6: Especificação dos Componentes do *UML Components* [11].

1. O estágio de `identificação dos componentes` recebe como entrada o `modelo conceitual do negócio` e a `especificação dos casos de uso` do sistema, artefatos da fase de especificação de requisitos. O objetivo dessa etapa é identificar um conjunto inicial de interfaces. Além disso, essas interfaces devem ser classificadas em duas categorias: (i) interfaces de sistema, relativas à funcionalidade do sistema; e (ii) interfaces de negócio, que conterão as operações básicas do negócio.

Devido às características já citadas de cada grupo de interfaces, as interfaces de sistema são identificadas a partir dos casos de uso e as interfaces de negócio a partir do modelo conceitual fornecido. Em seguida, para cada uma dessas interfaces, é criado um componente. Devido à classificação da interface, o componente já pode ser posicionado na arquitetura, nas camadas correspondentes à classificação da interface (sistema ou negócio). Em relação às interfaces de sistema, além da identificação da interface propriamente dita, neste estágio também são identificadas as primeiras operações. Apesar disso, são definidos apenas os seus nomes; a assinatura completa é refinada apenas durante o próximo estágio, de interação entre os componentes.

2. Durante a `interação entre os componentes`, a preocupação é voltada para o detalhamento das estruturas identificadas anteriormente. Por esse motivo, nesse estágio o projetista examina como cada operação do sistema será implementada. Em outras palavras, a preocupação é saber quais componentes do sistema deverão interagir para o serviço ser implementado. Para isso, o *UML Components* utiliza diagramas de colaboração UML. Através dessas colaborações, as operações requeridas do negócio são descobertas e as assinaturas das operações de sistema podem ser refinadas. Com a definição das dependências entre os componentes do sistema, a estrutura da arquitetura vai sendo definida. No final da execução do estágio de interação entre os componentes, a arquitetura do sistema já está definida.

3. Finalmente, o estágio de `especificação final` dos componentes é onde as especificações são refinadas e, se possível, representadas de uma maneira formal. Durante a especificação final, a arquitetura do sistema não deve variar. Dessa forma, o detalhamento da especificação só deve ser feito no momento em que a arquitetura do sistema já está considerada estável. A definição formal de assertivas é uma tarefa custosa, do ponto de vista de esforço de trabalho, e por isso deve-se evitar retrabalho nessa tarefa.

Com a especificação dos componentes finalizada, é chegada a hora de providenciar cada um dos componentes para em seguida compor o sistema, testá-lo e entregá-lo ao cliente para utilização. Apesar da importância dessas fases, o *UML Components* não detalha a sua execução. As subseções seguintes descrevem o papel de cada uma dessas fases do desenvolvimento.

2.3.4 Provisionamento de Componentes

A etapa de provisionamento deve garantir a concretização dos componentes especificados. Essa concretização pode ocorrer das seguintes formas: (i) reutilização de componentes já existentes; e (ii) implementação de componentes novos. Normalmente, os componentes da

camada de negócio são candidatos à reutilização, uma vez que representam características básicas para sistemas de um mesmo domínio. Já os componentes da camada de sistema oferecem implementações específicas, peculiares às necessidades de um sistema em particular. Por esse motivo, apesar de também poderem ser reutilizados, as chances para isso são reduzidas. Para cada uma das formas de concretização, há diferentes atividades a serem realizadas. Para o caso de reutilização de componentes, devido às possíveis inconsistências entre as interfaces especificadas e reutilizadas, pode ser necessário implementar adaptadores [44]. No caso dos componentes implementados, é necessário especificar as suas estruturas internas. Para isso, deve-se utilizar algum modelo de componentes que possibilite a sua especificação através das estruturas existentes nas linguagens de programação, tais como objetos e classes.

2.3.5 Montagem do Sistema

A fase de montagem é responsável pela concretização da configuração da arquitetura do sistema. Ela consiste na integração dos componentes para construção da aplicação como um todo. Basicamente, existem duas atividades durante esta fase: (i) a construção dos conectores, que implementam um código de mapeamento entre os componentes do sistema; e (ii) a construção do programa principal, que deve conter o código de instanciação dos componentes, dos conectores, além da "ligação" entre eles, que é efetuada dinamicamente.

2.3.6 Testes

Teste é a atividade de executar um software com o objetivo de mostrar a não existência de falhas. Apesar de não comprovar a ausência de falhas, a sua execução aumenta consideravelmente a confiabilidade do sistema [41]. Porém, para uma maior eficácia, os testes devem ser considerados desde as fases iniciais do desenvolvimento [41] [44]. Apesar da importância dos testes, o processo *UML Components* se restringe ao desenvolvimento, não contendo atividades sistemáticas de testes. Em parte isso é uma vantagem, uma vez que possibilita a utilização conjunta com processos de teste existentes. De uma forma geral, o papel de um processo de testes é realizar verificações entre o sistema e os requisitos especificados. Dessa forma, os testes podem auxiliar na obtenção de respostas para as seguintes perguntas: (i) os requisitos estão corretos? (ii) os requisitos estão consistentes (não-contraditórios entre si)? (iii) o sistema atende minimamente a todos os requisitos? Como pode ser percebido, as respostas a essas questões são de fundamental importância para a garantia da qualidade do sistema.

2.3.7 Implantação

Após o seu desenvolvimento e uma garantia maior da satisfação dos seus requisitos, o sistema pode ser implantado para sua utilização. A conclusão da fase de implantação não significa que o ciclo de desenvolvimento do sistema foi finalizado. Pelo contrário, muito provavelmente serão necessários alguns ciclos de manutenções corretivas para que o sistema possa ser considerado estável [41]. Além disso, mesmo com a estabilidade do sistema, nada impede que novas mudanças de requisitos sejam solicitadas por parte do cliente. As atividades de evolução da especificação do sistema, decorrentes dessas mudanças de requisitos, são conhecidas como manutenções perfectivas (ou evolutivas) [41] [44].

2.4 Arquitetura de Linha de Produtos de Software baseadas em Componentes

A *Arquitetura de Linha de Produtos* (ALP) é um dos principais artefatos do desenvolvimento de uma LPS. Enquanto abstrai os detalhes do sistema, uma ALP provê uma visão global do sistema, que é um importante mecanismo para identificar características comuns e variáveis em termos de elementos arquiteturais e suas configurações, bem como, para definir um planejamento estratégico para a reutilização de software [8]. Em uma ALP, as características comuns dos elementos arquiteturais e suas configurações são reutilizadas em diferentes produtos, enquanto que variabilidades são resolvidas através de decisões de projeto tomadas para cada produto derivado da linha.

O DBC pode apoiar o desenvolvimento de uma LPS, através da criação de uma ALP baseada em componentes, criando uma separação entre a especificação e a implementação dos componentes da LPS, o que reduz o acoplamento entre as partes, favorecendo a sua reutilização [14]. Além disso, ALPs baseadas em componentes permitem a especificação de *pontos de variação* arquiteturais e a redução de elementos arquiteturais, diminuindo a complexidade da arquitetura.

ALP é um dos artefatos de infraestrutura criados pela engenharia da família, ao longo do ciclo de desenvolvimento de uma nova LPS. Após sua criação, a ALP é maciçamente utilizada para a derivação dos produtos. É através dela, durante a engenharia de aplicações, que é guiada a determinação de quais componentes devem ser reutilizados na criação de um produto. A configuração arquitetural de cada um dos produtos da linha deve ser uma derivação da ALP, ou seja, ela deve ser composta pela junção das configurações arquiteturais de todos os possíveis produtos da linha.

Visando que a configuração arquitetural derivada de um produto criado satisfaça as restrições de seleção dos pontos de variação arquiteturais, alguns cuidados devem ser tomados durante a criação do novo produto. Por exemplo, ao decidir pelo provimento

de uma funcionalidade opcional, através da seleção de um componente não-mandatório, é necessário verificar se essa decisão não implica na seleção de algum outro componente ou na exclusão de um já selecionado.

Na Figura 2.7 é apresentado um exemplo ilustrativo de parte da ALP de uma LPS de reserva de quartos em hotel. Dois componentes mandatórios estão presentes na ilustração, `Persistence` e `Pagamento_Mgr`, responsáveis por persistir e gerenciar pagamentos, respectivamente. Há dois componentes alternativos, `Cartao_Mgr` e `Boleto_Mgr`, responsáveis por prover operações específicas para os tipos de pagamentos de cartão e boleto, respectivamente. Um componente opcional chamado `Log_Mgr`, responsável por implementar a funcionalidade de envio de mensagens de log, também faz parte da ALP. A ALP ilustrada contém dois pontos de variação, um representando a decisão entre qual dos componentes alternativos deve ser selecionado, e outro identificando a necessidade de tomar a decisão de selecionar ou não o componente `Log_Mgr` para compor um produto.

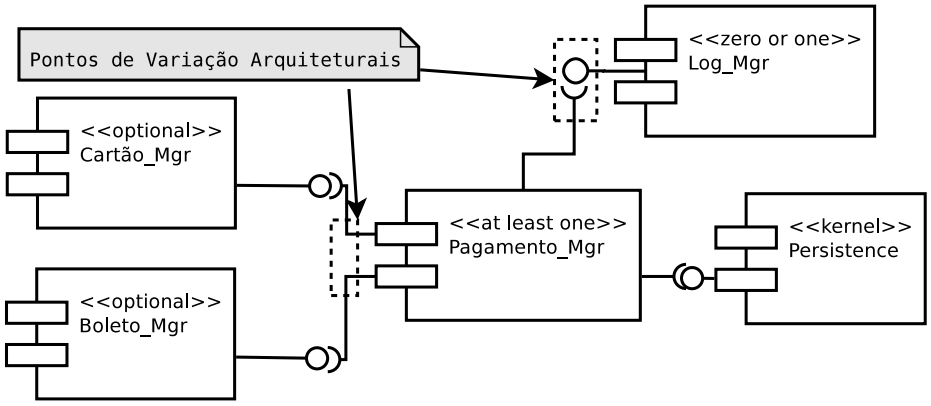


Figura 2.7: Parte de uma ALP baseada em componentes.

É importante mencionar que em alguns casos, para a criação de pontos de variação arquiteturais, é necessária que alguns componentes apresentem variabilidades em seu comportamento interno, criando pontos de variação internos. Por exemplo, para dar suporte ao ponto de variação envolvendo o componente `Log_Mgr` é preciso criar um ponto de variação interno ao componente `Pagamento_Mgr`. Esse novo ponto é responsável por eliminar a dependência entre os componentes, pois, quando `Log_Mgr` não é selecionado, a interface requerida pelo `Pagamento_Mgr`, satisfeita por `Log_Mgr`, deve ser suprimida.

Através da tomada de diferentes decisões nos pontos de variação arquiteturais, pode-se criar produtos distintos. Por exemplo, escolhendo selecionar `Boleto_Mgr` ao invés de `Cartao_Mgr`, e escolhendo não selecionar `Log_Mgr`, é derivado o produto *HotelBoleto* (Figura 2.8(a)). Selecionando `Boleto_Mgr` e `Log_Mgr` deriva-se o produto *HotelBoletoLog* (Figura 2.8(b)). E ainda, selecionando `Cartao_Mgr` ao invés de `Boleto_Mgr` tem-se o produto *HotelCartao* (Figura 2.8(c)). Note que, nesse caso, o componente `Log_Mgr` não pode

ser selecionado uma vez que ele depende do componente `Boleto_Mgr`, o que caracteriza uma restrição de seleção entre os pontos de variação arquiteturais.

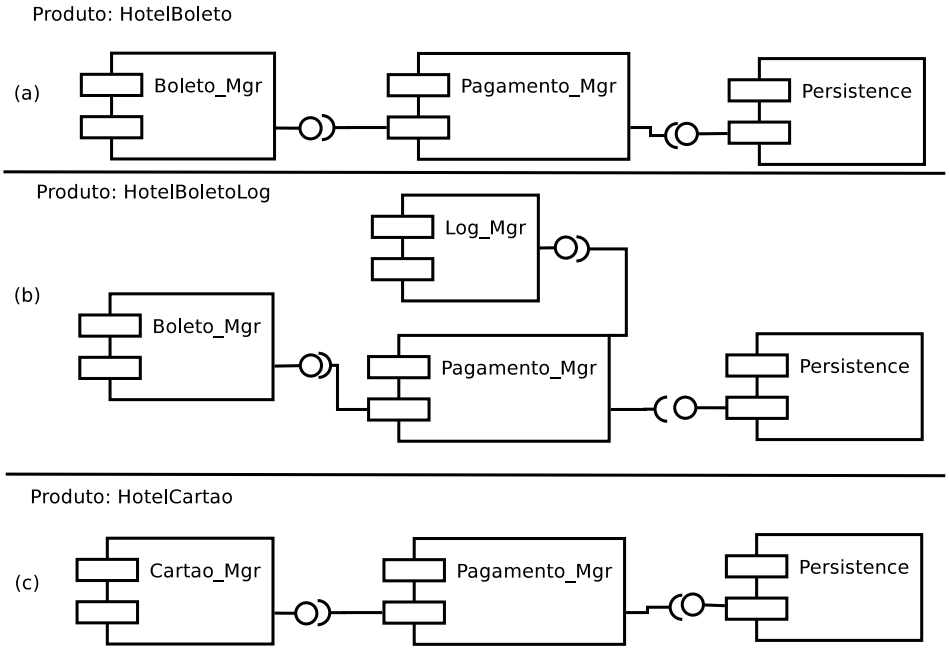


Figura 2.8: Diferentes configurações arquiteturais derivadas de uma ALP.

2.5 Modelo de Implementação COSMOS*

Para que os benefícios proporcionados pela arquitetura de software sejam efetivamente válidos, é necessário que as abstrações produzidas no projeto arquitetural do sistema sejam consideradas nas várias fases do desenvolvimento: análise, projeto arquitetural, projeto, implementação, testes e distribuição [12]. Porém, as linguagens de programação atuais não oferecem a abstração necessária para o mapeamento direto das estruturas básicas da arquitetura para o código [18]. Sendo assim, se faz necessário o uso de um modelo capaz de viabilizar a implementação dos componentes, conectores e configurações.

O modelo COSMOS* [18], leia-se cosmos estrela, é um modelo genérico e independente de plataforma, que utiliza estruturas disponíveis na maioria das linguagens de programação orientadas a objetos, para a implementação de componentes de software. Os principais objetivos do COSMOS* são (i) garantir que a implementação do sistema esteja em conformidade com a sua arquitetura e (ii) facilitar a evolução dessa implementação.

Para oferecer esses objetivos, o modelo COSMOS* incorpora um conjunto de diretivas de projeto que facilita a evolução dos componentes implementados. Essas diretivas

incluem: (i) materialização dos elementos arquiteturais, isto é, componentes, conectores e configurações; (ii) separação entre a especificação e a implementação dos componentes, garantindo que apenas a especificação seja pública; (iii) declaração explícita das dependências entre os componentes, através de interfaces requeridas; e (iv) baixo acoplamento entre as classes da implementação.

O modelo COSMOS* define três submodelos que tratam diferentes aspectos do desenvolvimento de um sistema baseado em componentes: (i) o modelo de especificação define a visão externa de um componente através da especificação das suas interfaces providas e requeridas; (ii) o modelo de implementação define como o componente deve ser implementado internamente; e (iii) o modelo de conector especifica as conexões entre os componentes, que são materializadas explicitamente através de conectores. Cada um desses modelos possui padrões de modelagem bem definidos que possibilitam a geração automática de códigos-fonte através de alguma ferramenta CASE[13]. O projeto Bellatrix [47] é um ambiente, constituído de uma série de *plugins* para a plataforma Eclipse [3], voltado para o desenvolvimento baseado em componentes, que guia o desenvolvedor seguindo as atividades propostas pelo processo *UML Components*. Utilizando-o é possível especificar arquiteturas de software baseadas em componentes graficamente e, a partir dessa representação gráfica, mapeá-las para componentes e conectores COSMOS* implementados em código fonte Java. A Figura 2.9 mostra uma configuração composta de dois componentes (`A` e `B`) e uma visão do mapeamento correspondente no modelo COSMOS*. Essa visão de implementação é detalhada na Figura 2.10, que representa a implementação do componente `A`.

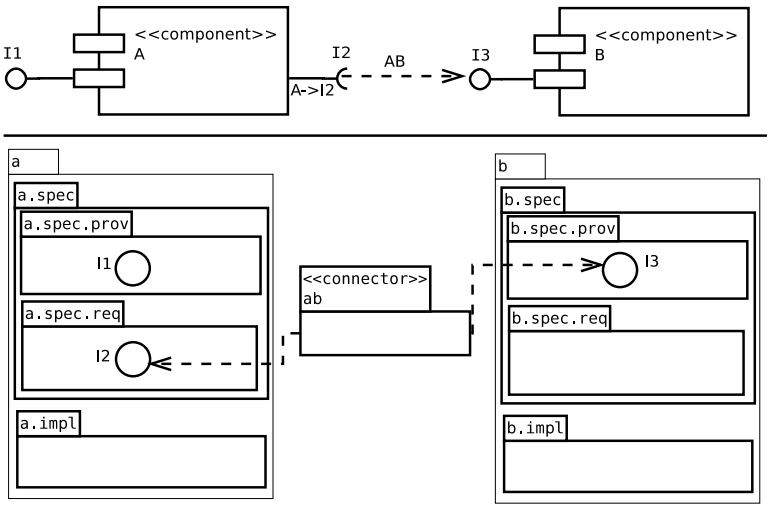


Figura 2.9: Modelo de Especificação do COSMOS*

[13]Inglês: *Computer Aided Software Engineering*

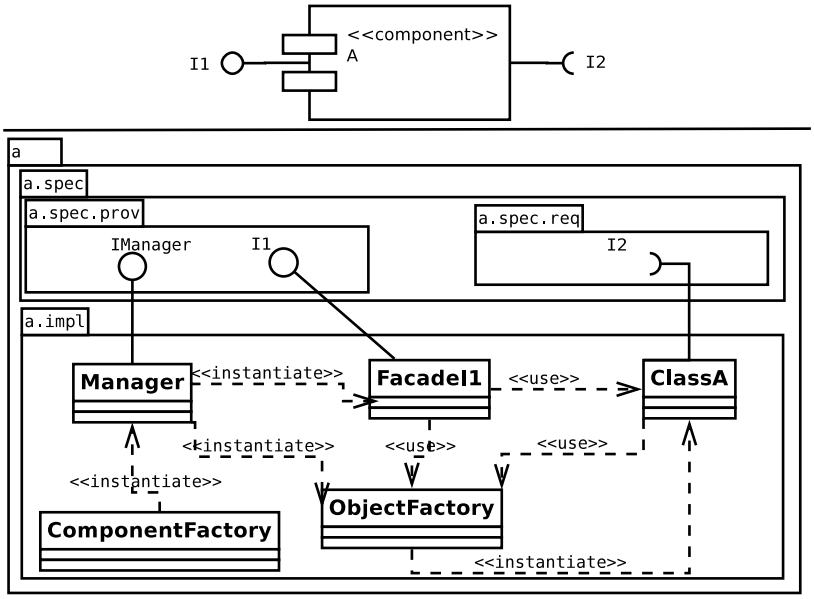


Figura 2.10: Modelo de Implementação do COSMOS*

Na Figura 2.10, é possível perceber mais claramente a separação explícita entre a especificação e a implementação do componente. Nesse modelo, as interfaces providas do componente pertencem ao pacote `spec.prov`, enquanto suas dependências são declaradas no pacote `spec.req`. Todas essas interfaces, sejam elas providas ou requeridas, possuem visibilidade pública. O modelo de implementação, por sua vez, tem o papel de materializar os serviços oferecidos pelo componente, utilizando as operações declaradas nas suas interfaces requeridas. A implementação do conector `AB` é apresentada na Figura 2.11.

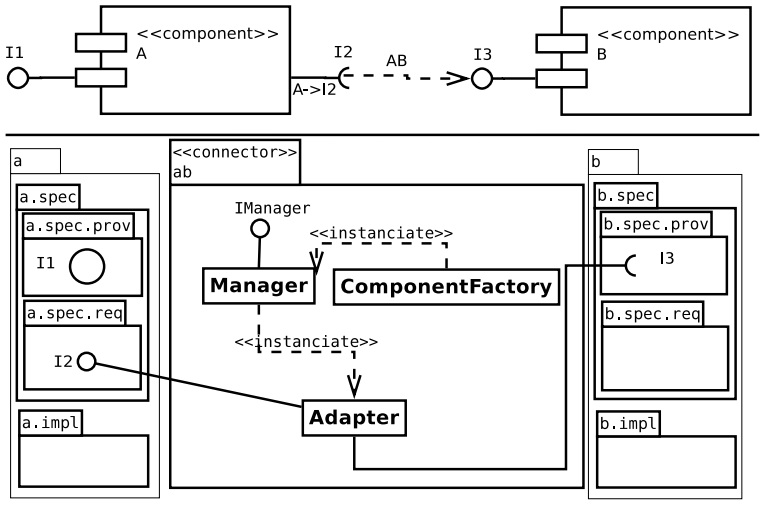


Figura 2.11: Modelo de Conectores do COSMOS*

A Tabela 2.2 detalha o papel das principais entidades presentes no modelo COSMOS*.

Tabela 2.2: Principais Entidades do Modelo COSMOS*

Entidade	Descrição
`IManager`	é uma interface provida pré-definida pelo modelo. Essa interface oferece as operações relativas à montagem e utilização do componente, isto é, operações de captura das instâncias que materializam as interfaces providas e de definição das instâncias que materializam as interfaces requiridas.
`Manager`	é a classe que materializa (Inglês: realises) a interface `IManager`.
`ComponentFactory`	Esta é a classe responsável pela instanciação do componente, mais especificamente, pela instanciação da classe `Manager`. Por esse motivo, a `ComponentFactory` é a única classe com visibilidade pública no modelo de implementação (pacote `impl`).
`Facade(s)`	Cada classe `Facade` materializa o comportamento de uma interface provida pelo componente (exceto `IManager`). A associação de cada `Facade` com a sua respectiva interface provida é feita já nos construtores da classe `Manager`, através do método `setProvidedInterface()`, especificado em `IManager`.
`ObjectFactory`	A classe `ObjectFactory` é responsável pela instânciação das demais classes necessárias para a implementação do componente. O seu papel é facilitar a evolução interna do componente, através da redução do acoplamento entre suas classes.

2.6 COSMOS*-VP

O modelo COSMOS*-VP, proposto por Dias [19] é uma extensão do modelo COSMOS* [24], apresentado na Seção 2.5. Seu principal objetivo é especificar e implementar variabilidades em ALPs baseadas em componentes. Para isso COSMOS*-VP integra modernas abordagens de programação orientada a aspectos ao contexto de modelos de implementação de componentes. A combinação das abordagens foca em facilitar a evolução de ALPs componentizadas, através da mitigação dos problemas do forte acoplamento de componentes aspectuais e o problema de espalhamento de pontos de variação, causadores de instabilidades arquiteturais durante a evolução das ALPs.

O modelo COSMOS*-VP apoia a especificação e implementação de *Pontos de Variação Explícitos* através da utilização do modelo chamado Modelo de Pontos de Variação Explícitos. O modelo foi criado para refinar o modelo de conectores do COSMOS* visando a modularização de pontos de variação, impedindo-os de serem implementados espalhadamente por vários elementos em uma ALP. E ainda, facilita a identificação de pontos de variação arquiteturais e aqueles que existem dentro dos componentes da ALP.

Tabela 2.3: Elementos do Modelo de Ponto de Variação Explícitos

Elemento	Descrição
Interface de Requisição	são um conjunto de interfaces públicas externamente ao Ponto de Variação Explícito. As interfaces de requisição serão utilizadas pelos elementos do nível base para requisitar funcionalidades não-mandatórias ao Ponto.
Interfaces de Delegação	são um conjunto de interfaces utilizadas pelo Ponto de Variação Explícito para delegar funcionalidades não-mandatórias aos elementos não-mandatórios conectados a ele, os quais irão prover a funcionalidade requisitada.
Adapter	é um padrão de projeto [] entre as interfaces de requisição e as interfaces de delegação de um Ponto de Variação Explícito. Tem como objetivo, além de adaptar as interfaces, modularizar as regras de decisões e a implementação de suporte de decisões de um Ponto de Variação Explícito, permitindo assim que o problema do espalhamento do ponto de variação seja mitigado.

2.6.1 *Ponto de Variação Explícito*

Um *Ponto de Variação Explícito* é um intermediário entre os elementos base da aplicação, ou seja, aqueles que implementam as características mandatórias da LPS e os elementos não-mandatórios. Assim, a utilização de Ponto de Variação Explícito separa a implementação dos pontos de variação da implementação das características de uma LPS. Essa separação diminui o número de componentes da ALP que devem ser alterados para evoluir o modelo de características de uma LPS.

A Tabela 2.3 lista os elementos que compreendem um Ponto de Variação Explícito. As *Interfaces de Requisição* são usadas pelos elementos mandatórios para requisitar funcionalidades não-mandatórias ao ponto. Uma interface de requisição atua como uma interface provida do Ponto de Variação Explícito. As *Interfaces de Delegação* são utilizadas pelo ponto para delegar, aos elementos não-mandatórios, as funcionalidades requisitadas a ele. Uma interface de delegação pode ser considerada como uma interface requerida do Ponto de Variação Explícito. O principal elemento do Ponto de Variação Explícito é o *Adapter*. Além de realizar a conexão entre as interfaces de requisição e as interfaces de delegação, conectando assim os elementos mandatórios aos não-mandatórios, o *Adapter* tem duas outras funções. Primeira, ele tem a função de adaptar pequenas diferenças entre as interfaces que conecta. Segundo, ele é responsável por modularizar a implementação de suporte de decisões do Ponto de Variação Explícito, permitindo que, ao invés de espalhado, o ponto de variação seja totalmente modularizado em um único elemento.

A Figura 2.12 mostra exemplos de Pontos de Variação Explícitos com diferentes possíveis combinações no número de interfaces. Um Ponto de Variação Explícito pode conter apenas uma interface de requisição e uma interface de delegação (Figura 2.12(a)). Esse é o caso mais simples, que ocorre quando um Ponto de Variação Explícito provê a caracterís-

tica de um componente opcional a apenas um outro componente. A Figura 2.12(b) mostra um exemplo de Ponto de Variação Explícito com duas ou mais interfaces de requisição e uma interface de delegação. Esse tipo de Ponto de Variação é utilizado quando a característica implementada pelo componente não-mandatório é requerida por dois ou mais componentes. Na Figura 2.12(c) é mostrado um exemplo de Ponto de Variação Explícito em que o inverso ocorre. Neste caso, o Ponto é utilizado para prover as características de dois, ou mais, componentes não-mandatórios a apenas um componente. A Figura 2.12(d) mostra o caso n pra m, em que o Ponto de Variação é utilizado para conectar dois ou mais componentes não-mandatórios a dois ou mais outros componentes.

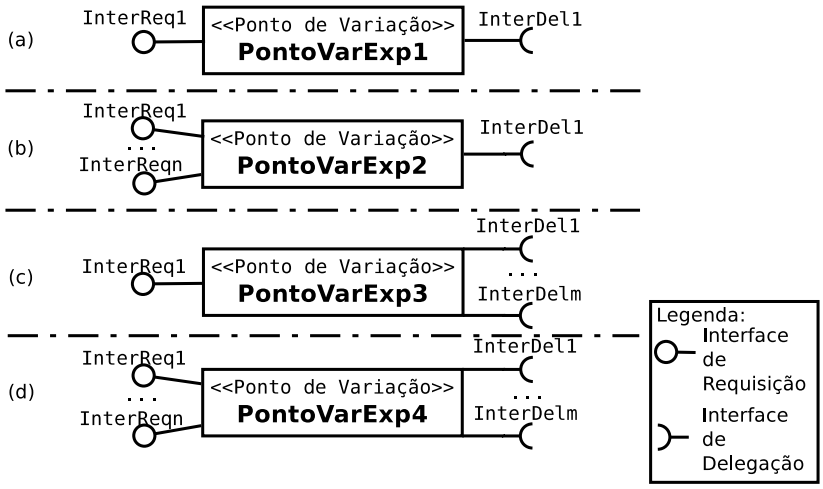


Figura 2.12: Exemplos de Ponto de Variação com diferentes combinações no número de interfaces.

2.6.2 Modelo de `Conectores-VP` do COSMOS*-VP

O modelo de Pontos de Variação Explícitos refina o modelo dos conectores COSMOS* para criar o modelo dos `Conectores-VP`, visando a especificação e a implementação dos pontos de variação arquiteturais explícitos. Com o uso deste conector, a modularização de variabilidades arquiteturais e a criação de uma visão global das variabilidades da LPS serão facilitadas. Os `Conectores-VP` devem ser utilizados em cada ponto de variação arquitetural da ALP, para que todas as variabilidades da ALP e todos os produtos criados a partir delas sejam identificados.

Além de modularizar pontos de variação arquiteturais em elementos explícitos, COSMOS*-VP tem como objetivo manter desacoplados os elementos arquiteturais de uma ALP, sejam eles mandatórios, opcionais ou alternativos. A utilização de componentes aspectuais para modularizar características não-mandatórias de uma linha ajuda a manter

a estabilidade arquitetural da ALP. Então, para mediar as conexões de um componente não-mandatório, implementado utilizando aspectos, um `Conector-VP` combina os conceitos do Modelo de Pontos de Variação Explícitos aos conceitos de conector aspectual. Dessa forma, enquanto modulariza um ponto de variação arquitetural, um `Conector-VP` mantém desacoplados os componentes aspectuais relacionados com o ponto modularizado. A próxima seção detalha como `Conectores-VP` podem ser criados para modularizar pontos de variação arquiteturais, e conectar componentes não-mandatórios implementados utilizando aspectos de maneira desacoplada.

O modelo de conector do modelo COSMOS* foi estendido pela adição dos elementos do modelo de Pontos de Variação Explícitos, **Interface de Requisição**, **Interface de Delegação** e **Adapter**. E, os elementos do modelo de Pontos de Variação Explícitos foram mapeados para serem utilizados em um contexto Orientado a Aspectos, pois os `Conectores-VP` são utilizados para realizar a conexão de componentes aspectuais não-mandatórios ao nível básico da ALP. Assim, as interfaces de requisição de um `Conector-VP` são na verdade aspectos que interceptam os componentes que requerem funcionalidades não-mandatórias em um ponto de variação arquitetural. As interfaces de delegação são aspectos que estendem os aspectos abstratos dos componentes não-mandatórios relacionados ao ponto.

A seguir, será realizado um exemplo sobre o mapeamento e o processo de criação de um `Conector-VP`. Como mostra a Figura 2.13, um componente opcional (`B_Mgr`) provê sua característica utilizando um aspecto abstrato, `BAbsAsp`, e um componente mandatório, `A_Mgr`, especifica os locais onde requer a característica opcional em uma XPI pública, `XPI_A` são conectados de maneira desacoplada utilizando um `Conector-VP`. Como um conector aspectual, o `Conector-VP` estende o aspecto abstrato do componente opcional e configura os adendos estendidos para interceptar os pontos de corte da XPI do componente mandatório. A criação desse `Conector-VP`, necessário para modularizar o ponto de variação arquitetural que ocorre na conexão dos dois componentes, deve seguir quatro passos básicos:

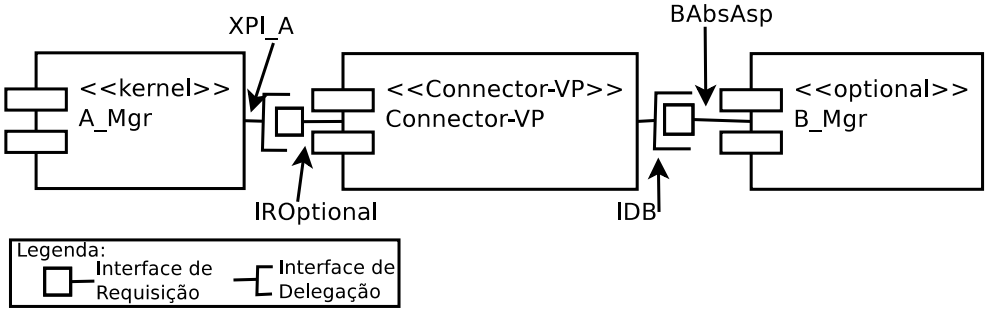


Figura 2.13: Exemplos de um `Conector-VP`.

1. **Passo 1 - Criação de Interfaces de Requisição:** como nesse caso há apenas um componente mandatório, `A_Mgr`, que requer a característica opcional, apenas uma interface de requisição deve ser criada. A interface, `IROptional`, a ser criada na verdade é um aspecto que intercepta os pontos de corte, especificados pela XPI, `XPI_A` do componente mandatório, onde a característica opcional deve ser provida.

2. **Passo 2 - Criação de Interfaces de Delegação:** como há apenas um componente opcional, `B_Mgr`, apenas uma interface de delegação, `IDB`, deve ser criada para delegar requisições para o único componente opcional. Devido ao fato do componente opcional ser implementado utilizando aspectos, e portanto ele prove sua característica através de um aspecto abstrato, a interface de delegação a ser criada na verdade é um aspecto que estende o aspecto abstrato do componente opcional.

3. **Passo 3 - Implementação Interna:** para conectar os aspectos que atuam como interfaces de requisição aos aspectos de interfaces de delegação, uma interface convencional deve ser criada para cada interface de delegação. Cada nova interface criada, internamente ao `Conector-VP`, deve conter um método para cada adendo da interface de delegação correspondente. Nesse exemplo, há apenas uma interface de delegação, portanto, só uma interface interna é criada. O conjunto de interfaces internas é chamado de `Adapter`, pois como media a conexão entre os aspectos o mesmo pode ser utilizado para adaptar algumas imperfeições entre eles, por exemplo, a ordem dos atributos de dois adendos conectados.

4. **Passo 4 - Conexão das Interfaces:** esse é o passo mais complicado da criação de um `Conector-VP`, e é o único que não pode ser totalmente automatizado. Esse passo é o responsável por determinar como as requisições originadas da interface de requisição serão delegadas para o componente opcional, através da interface de delegação. Em um caso extremamente simples, é necessário apenas configurar os adendos concretizados pela interface de delegação para interceptar cada um dos métodos correspondentes da interface de adapter, e configurar a interface de requisição para invocar tais métodos. Dessa forma, o fluxo de controle fluirá diretamente dos pontos de corte especificados pela XPI do componente mandatório para os adendos do aspecto abstrato do componente opcional.

   Entretanto, geralmente, os pontos de variação arquiteturais possuem determinadas regras de decisão para a seleção e utilização de componentes opcionais, que implicam na criação de algum tipo de implementação para testar essas regras, ou garantir que as configurações arquiteturais resultantes das decisões tomadas sejam consistentes. Por exemplo, em um caso bastante simples, quando há mais de um componente opcional provendo características similares, pode ser necessário que os valores dos

parâmetros das requisições sejam verificados para se determinar para qual componente opcional as requisições devem ser delegadas, a cada momento. Esse tipo de implementação é a chamada implementação de infraestrutura de apoio de decisões, e deve ser implementada dentro dos aspectos que atuam como interfaces de requisição.

Um exemplo de `Conector-VP` é apresentado na Figura 2.14(a). Ela ilustra a visão arquitetural do `Conector-VP`, chamado `Conector-VP`, responsável por mediar a conexão entre o componente mandatório `A_Mgr` e os componentes opcionais `B_Mgr` e `C_Mgr`. `A_Mgr` usa a interface de requisição `IROptional` para realizar requisições ao `Conector-VP`. As requisições de `A_Mgr` são delegadas para os componentes opcionais utilizando as interfaces de delegação `ID_B` e `ID_C`. De acordo com a regras de seleção para o ponto de variação arquitetural, pelo menos um dos componentes opcionais deve ser selecionada para compor os produtos. Portanto, `Conector-VP` pode ser configurado para delegar requisições somente para `B_Mgr`, somente para `C_Mgr`, ou para ambos.

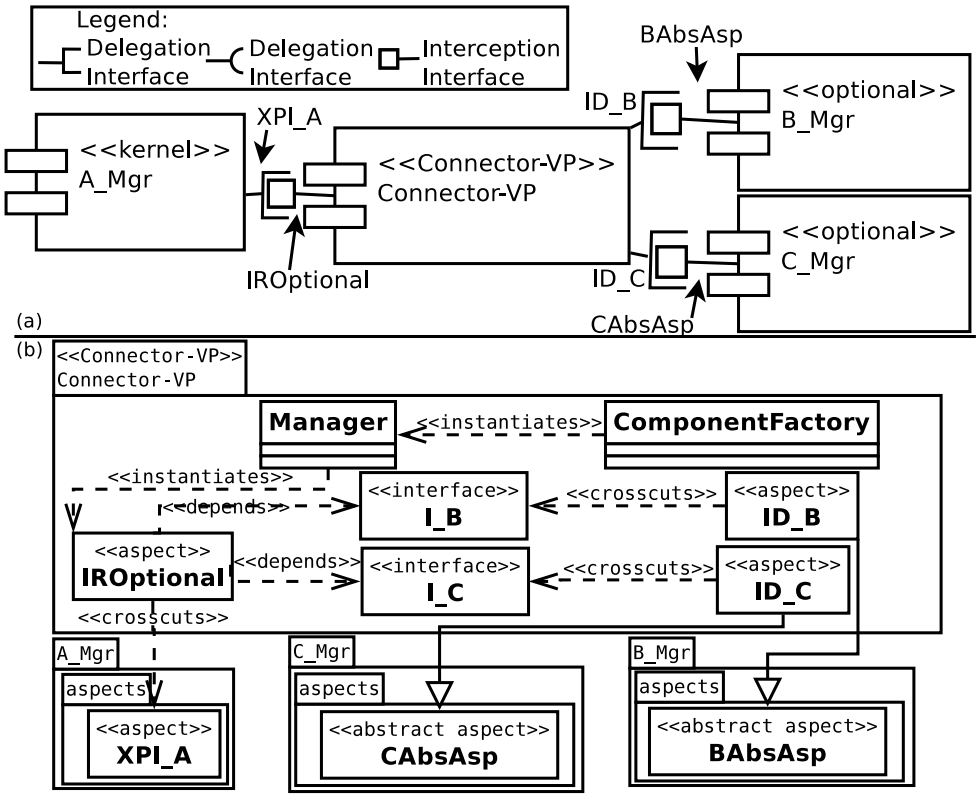


Figura 2.14: Exemplos de Ponto de Variação Arquitetural modularizado pelo `Conector-VP`.

Figura 2.14(b) mostra como um `Conector-VP` pode ser implementado utilizando AspectJ. Os métodos de classes e os pacotes requeridos `spec` e `impl` de `B_Mgr` e `C_Mgr`

foram omitidos para tornar a figura mais clara. `Conector-VP` emprega os aspectos `ID_B` e `ID_C`, que atuam como interfaces de delegação, para estender os aspectos abstratos, `B_AbsAsp` e `C_AbsAsp`, dos componentes `B_Mgr` e `C_Mgr`, respectivamente, e emprega o aspecto `IROptional`, que atua como uma interface de requisição, para interceptar os pontos de corte do componente `A_Mgr` especificados em sua XPI chamada `XPI_A`. Para prover as características implementadas pelos componentes opcionais ao componente `A_Mgr`, duas interfaces são criadas entre `IROptional` e os aspectos `ID_B` e `ID_C`. `IROptional` para delegar para `B_Mgr` faz uso da interface interna `I_B`, a qual tem cada um dos seus métodos interceptado por um adendo de `ID_B`. Analogamente, `IROptional`, para delegar requisições para `C_Mgr`, invoca os métodos da interface interna `I_C`, os quais são interceptados por `ID_C`.

`Conectores-VP` realizam o processo de delegação que pode ser implementado de diversas maneiras, dependendo da complexidade dos aspectos abstratos, das XPIs conectadas e da complexidade das possíveis decisões a serem tomadas no ponto de variação arquitetural modularizado. Devido ao uso de interfaces de requisição, interfaces de delegação e adapter, um `Conector-VP` é capaz de modularizar toda a implementação de suporte de decisão de um ponto de variação arquitetural, seja ela complexa ou não.

2.7 Programação Orientada a Aspectos

Programação Orientada a Aspectos (POA) oferece mecanismos para a modularização de interesses transversais de um sistema de software. Interesses transversais são chamados dessa forma, pois são implementados de maneira espalhada por diversos módulos do sistema. POA, através da separação de interesses, aumenta a modularidade de um sistema impedindo que códigos fontes correspondentes a distintos interesses sejam implementados entrelaçadamente em um módulo.

2.7.1 AspectJ

AspectJ [35] é uma extensão da linguagem Java, que une classes e aspectos para aumentar a qualidade da implementação de um sistema. AspectJ utiliza-se dos conceito de separação de interesses para tornar o sistema mais modular. Interesses que são bem modularizados com mecanismos de orientação a objetos tradicionais são implementados utilizando-se classes Java. Ao passo que, módulos de AspectJ chamados de *aspectos*[14] são utilizados para modularizar interesses transversais. Classes e aspectos são integrados durante o processo de combinação[15] para compor o sistema final.

[14] Inglês: *aspects*
[15] Inglês: *weaving process*

Os aspectos combinam três principais mecanismos: *pontos de junção*[16], *pontos de corte*[17] e *adendo*[18]. Os pontos de junção são os pontos de execução de um sistema que podem ser identificáveis por um aspecto. Chamadas de métodos e definições de valores a atributos de classe são exemplos de pontos de junção. Os pontos de corte são construções, definidas em aspectos, que selecionam pontos de junção e coleta o contexto em tais pontos. Por fim, um adendo é uma construção, utilizada nos aspectos, que intercepta os fluxo de execução do sistema nos pontos identificados pelos pontos de corte. Um adendo define um trecho de código que pode ser executado antes, depois ou ao redor de um ponto de junção. Através do adendo, o comportamento normal de um sistema é alterado.

A Figura 2.15 apresenta um exemplo de AspectJ. Nela temos a classe `Reserva` que define o método `reservaQuarto()`. O aspecto `Logging` define o ponto de corte `method` que captura a execução do método `reservaQuarto()` da classe `Reserva`. O adendo, então, define o trecho de código a ser executado antes de cada execução do método `reservaQuarto()`.

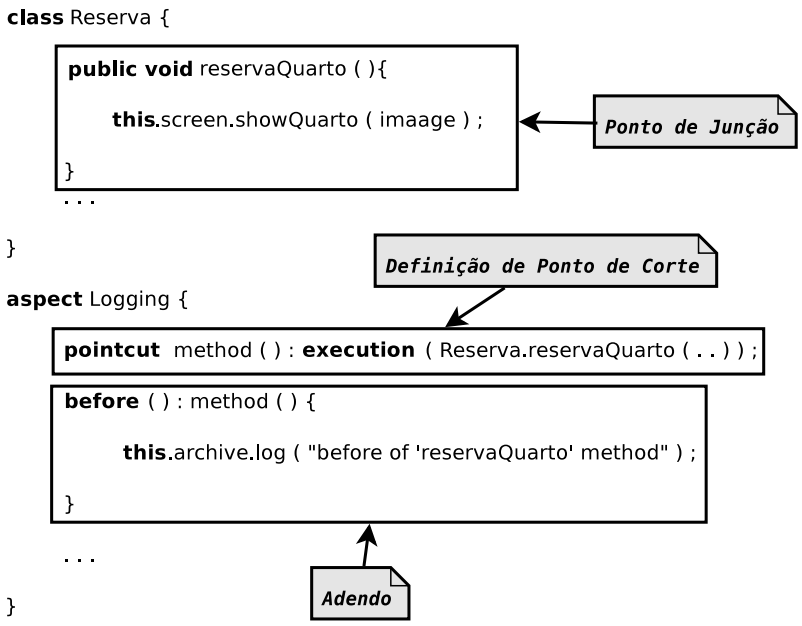


Figura 2.15: Exemplo de aspecto utilizando o AspectJ

A funcionalidade de *Log* implementada no aspecto `Logging` pode ser generalizada para ser executa antes de cada método do sistema. Dessa forma, seria implementado um interesse transversal de uma maneira bastante modular. Pois, a funcionalidade seria implementada em apenas um módulo, em vez de espalhada por todo o código do sistema.

[16] Inglês: *join points*
[17] Inglês: *pointcuts*
[18] Inglês: *advice*

A separação de interesses provida por AspectJ não apenas melhora a qualidade da implementação do sistema, como também facilita a presença de variabilidades. Por exemplo, no exemplo assim pode-se escolher criar instâncias do sistema que não apresentem a funcionalidade implementada pelo aspecto `Logging`, para isso basta não incluí-lo no processo de combinação. Outra maneira de criar variabilidades é através da implementação de diferentes versões de `Logging`, cada uma implementando a funcionalidade de uma maneira distinta. Então, durante o processo de combinação seria escolhida qual versão seria combinada às classes básicas para compor o sistema.

2.7.2 XPIs e Aspectos Abstratos

Crosscutting Programming Interfaces (XPIs), proposto por Griswold et al. [27], permite que o acoplamento entre aspectos e os elementos interceptados seja diminuído. Para isso, a abordagem dos XPIs cria uma separação entre a especificação de pontos de corte, dos elementos do nível base do sistema e os aspectos que os interceptam.

Um componente que espera ser interceptado define um conjunto de pontos de corte que poderá ser utilizado para interceptá-lo, e o torna público através do XPI. Assim, o aspecto que irá interceptá-lo deve fazer uso das XPIs públicas, e é proibido de interceptá-lo em outros pontos de junção se não aqueles definidos pelos pontos de corte especificados pelas XPIs. Dessa maneira, a quebra de encapsulamento é evitada, pois o componente aspectual é impedido de manter detalhes internos do componente interceptado.

Para desacoplar totalmente um componente aspectual dos componentes que ele intercepta, é necessário determinar que os aspectos dos componentes aspectuais sejam abstratos. Aspectos abstratos não necessitam, previamente, que seus adendos especifiquem os locais do nível base que irão interceptar. Para isso, os seus adendos utilizam pontos de cortes abstratos, os quais não definem pontos de junção concretos. Isso evita que detalhes da implementação dos componentes interceptados sejam mantidos pelos aspectos de um componente aspectual, eliminando assim a dependência direta entre um componente aspectual e o conjunto de componentes interceptados por ele.

A Figura 2.16 mostra um exemplo de componentes que especificam aspectos abstratos e XPIs, e são conectados por um conector aspectual. O componente aspectual `AComp` provê suas funcionalidades por meio de um aspecto abstrato, o `ConcernA`. O componente `BComp` especifica os pontos de corte onde espera que seja interceptado em uma XPI, a `XPIBClass`. A conexão entre os componentes é realizada pelo conector aspectual `Conn_AB`, que estende `ConcernA` e intercepta a classe `BClass` do componente `BComp`, utilizando os pontos de corte especificados em `XPIBClass`. É importante notar que os componentes COSMOS*-VP, como os ilustrados na Figura 2.16, têm uma estrutura similar a de um componente COSMOS*, diferencia-se apenas através do pacote *aspects*. Outro

fato que deve ser lembrado é que, embora ilustrado detalhadamente na figura, o pacote de implementação `impl` é um pacote privado e, portanto, seus elementos não são visíveis externamente.

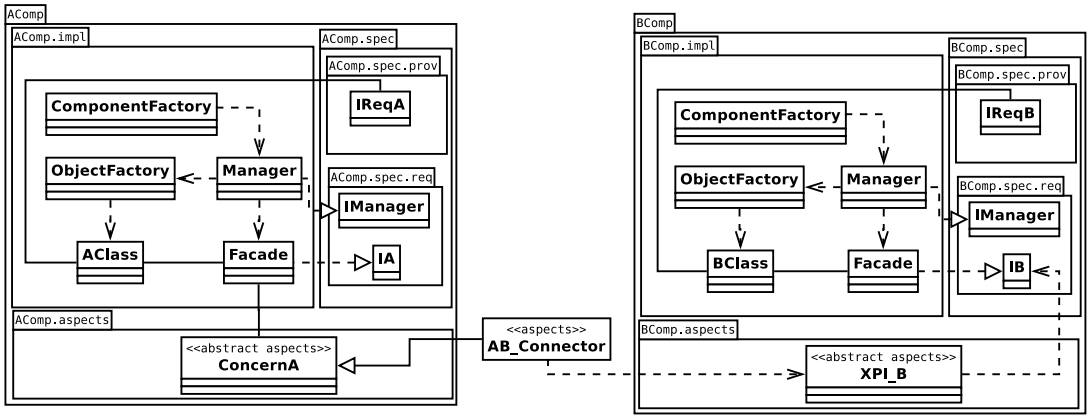


Figura 2.16: Componentes usando XPI e Aspectos Abstratos.

Nos Códigos 2.1 e 2.2 é exemplificado como XPIs são usadas para definir pontos de corte de um componente. No código 2.1 está ilustrado a implementação da classe `BClass`, a qual pertence ao pacote `impl` do componente `BComp`. A classe `BClass` define o método `doSomething`. O ponto de corte `executionOfDoSomething` (código 2.2), especificado pela XPI `XPIBClass`, define que a execução do método `doSomething` pode ser interceptada. Note que a XPI `XPIBClass` apenas especifica um ponto que pode ser interceptado por algum aspecto, ela não determina qual irá interceptá-lo. Os aspectos que desejam interceptar a execução do método `doSomething` necessitam referenciar o ponto de corte `executionOfDoSomething`. Dessa forma, a XPIs permitem que os componentes, que interceptam o componente `BComp`, não precisem quebrar o encapsulamento do componente, mantendo detalhes de sua implementação.

Código 2.1: Exemplo de implementação de uma classe

```
1 package br.unicamp.ic.cosmosxpi.CompB.impl;
2
3 class BClass{
4     void doSomething(){
5     ...
6     }
7
8 }
```

Código 2.2: Exemplo de XPI especificando um ponto de corte de um componente

```
1 package br.unicamp.ic.cosmosxpi.CompB.aspects;
```

```
2
3  public aspects XPIBClass{
4      public pointcut executionOfDoSomething() :
5          execution (
6              void br.unicamp.ic.cosmosxpi.CompB.impl.BClass.doSomething()
7          );
8
9  }
```

No código 2.3 detalha o aspecto abstrato `ConcernA` do componente aspectual `AComp`. `ConcernA` define o ponto de corte abstrato `adviseSomewhere`, sem argumentos. `adviseSomewhere` pode ser utilizado para interceptar qualquer componente da arquitetura, uma vez que é abstrato e não identifica nenhum ponto de junção especificamente. Dessa forma, não há nenhuma dependência direta entre o componente aspectual `CompA` e os componentes interceptados por ele.

Para que o aspecto abstrato de `AComp` intercepte a classe do componente `BComp`, um conector aspectual deve ser criado. O conector aspectual `Conn_AB`, detalhado no código 2.4, concretiza o aspecto abstrato `ConcernA`, do componente `AComp`, e utiliza o ponto de corte especificado em `XPIBClass` para interceptar a classe `BClass` de `BComp`. Dessa forma, após cada execução do método `doSomething`, o adendo especificado em `ConcernA` irá interceptá-lo, provendo assim a funcionalidade implementada pelo componente `AComp`.

Código 2.3: Exemplo de um aspecto abstrato

```
1  package br.unicamp.ic.cosmosxpi.AComp.aspects;
2
3  public abstract aspect ConcernA {
4      public abstract pointcut adviseSomewhere ( );
5          after ( ) : adviseSomewhere ( ) {
6          ...
7          }
8  }
```

Código 2.4: Exemplo de um conector aspectual

```
1  package br.unicamp.ic.cosmosxpi.ex ;
2
3  import br.unicamp.ic.cosmosxpi.ex.AComp.aspects.ConcernA ;
4  import br.unicamp.ic.cosmosxpi.ex.BComp.aspects.XPIBClass ;
5
6  public aspect Conn_AB extends ConcernA{
7      public pointcut adviseSomewhere() : XPIBClass.executionOfDoSomething();
8  }
```

2.8 Tolerância a Falhas

2.8.1 Falha, Erro e Defeito

Falha[19] é um evento que causa *erros*[20]. Caso o estado errôneo do sistema não seja tratado em um tempo determinado, ocorrerá um *defeito*[21], que se manifestará pela não execução ou algum tipo de mudança indesejada no serviço especificado. Como pode ser visto na Figura 2.17, além de indicar um estado errôneo irreversível, a ocorrência de um defeito realimenta o ciclo e pode gerar novas falhas. Esse fenômeno é conhecido como propagação da falha e deve ser contido através do seu confinamento.

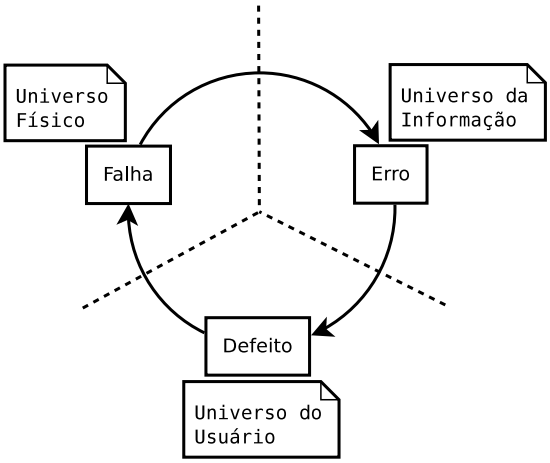


Figura 2.17: Ciclo da ocorrência de Falhas, Erros e Defeitos.

Existem várias classificações para falhas na literatura técnica [36] [1] [30]. Normalmente, essas classificações agrupam as falhas em falhas físicas, que se referem a falhas de componentes de hardware, e falhas humanas. Falhas humanas compreendem falhas de projeto, que são criadas durante as fases de desenvolvimento do software e falhas de interação, que por sua vez, podem ser acidentais ou intencionais durante o uso do software.

No princípio, o conceito de falhas se relacionava apenas às falhas de origem física, por exemplo, a mudança de um bit na memória por interferência eletromagnética. Com o aumento da complexidade dos sistemas de software, falhas de projetos se tornaram frequentes e quase inevitáveis.

As falhas também podem ser classificadas segundo a sua duração. Nessa perspectiva, as falhas são distribuídas em três grupos: (i) *transientes*, quando existe a chance das falhas ocorrem mais de uma vez, mas não obrigatoriamente, por exemplo, a invasão do

[19] Inglês: *fault*
[20] Inglês: *error*
[21] Inglês: *failure*

sistema por um hacker; (ii) *intermitente*, quando a ocorrência da falha se repete, mas não necessariamente em períodos definidos, por exemplo, a manifestação de um vírus presente no sistema; e (iii) *permanentes*, que são caracterizadas pela sua presença constante no sistema, por exemplo, falhas de especificação e implementação.

É importante observar que uma falha pode resultar em um, mais de um ou nenhum erro. Esse fato, aliado à impossibilidade de se conhecer todas as causas de uma falha dificultam seu o processo de detecção. Outro fator que complica a detecção de uma falha é o fato de várias falhas distintas poderem acarretar em um mesmo erro. Por esse motivo, a identificação do erro não resulta naturalmente na identificação da falha responsável por ele. Desta forma, faz-se necessário uma análise rigorosa do contexto de manifestação dos erros para que seja possível inferir suas causas e em seguida tratá-las.

2.8.2 Fases de Tolerância a Falhas

Existem várias propostas para a divisão das fases de implementação de tolerância a falhas. Uma das propostas mais utilizadas é a classificação em quatro fases de aplicação [36]: (i) detecção do erro[22], (ii) confinamento e avaliação[23], (iii) recuperação de erros[24], e (iv) tratamento de falhas[25].

Uma falha, primeiro, se manifesta como um erro para então ser detectada. A detecção do erro consiste na verificação da consistência do estado do sistema. De forma complementar à detecção, a fase de confinamento e avaliação tem o objetivo de conhecer e isolar as entidades inconsistentes do sistema. Após conhecer os seus componentes errôneos, é necessário recuperar o estado normal do sistema. A recuperação do erro pode ocorrer de duas maneiras: (i) recuperação por retrocesso[26], que consiste na restauração de um estado anteriormente válido; e (ii) recuperação por avanço[27], que consiste na construção de um novo estado válido a partir das informações contextuais disponíveis. Duas técnicas bastante utilizadas para a recuperação de estados errôneos são a definição de marcos de execução[28] e o tratamento de exceções. Essas técnicas, são utilizadas respectivamente para implementar técnicas de recuperação por retrocesso e recuperação por avanço. A Seção 2.9 detalha a segunda abordagem.

A última fase sugerida para as atividades de tolerância a falhas é reservada para o tratamento de falhas propriamente dito. O objetivo desta fase é identificar e eliminar as causas dos erros do sistema, de maneira que eles não voltem a acontecer. Uma técnica

[22]Inglês: *error detection*
[23]Inglês: *damage assessment*
[24]Inglês: *error recovery*
[25]Inglês: *fault treatment*
[26]Inglês: *backward error recovery*
[27]Inglês: *forward error recovery*
[28]Inglês: *checkpoints*

muito utilizada para essa tarefa é a de reconfiguração arquitetural através do uso de componentes redundantes [28]. Com a disponibilidade de componentes críticos sobressalentes, é possível, por exemplo, reconfigurar a arquitetura do sistema de forma a evitar o envio de mensagens aos componentes falhos. Esse tipo de correção de erro é conhecido como mascaramento da falha[29].

Vale a pena ressaltar a diferença entre componentes replicados e componentes redundantes. No contexto do desenvolvimento de software, o conceito de réplica significa que existe outra instância de um mesmo componente. Enquanto isso, um componente redundante, além de ser materializado por uma instância distinta, possui estrutura interna igualmente diferente, já que se trata de outro componente, projetado possivelmente por uma equipe distinta de desenvolvedores (diversidade de projeto[30]). Em termos de tolerância a falhas, componentes redundantes são mais expressivos, apesar do seu reflexo negativo no custo do sistema.

Apesar das técnicas de tolerância a falhas se basearem na distribuição e na redundância de elementos [36], muitos fatores devem ser levados em conta ao se empregar redundância, seja ela de componentes, de projeto ou temporal. Os principais fatores são os custos de finanças, de tempo, de sobrecarga e de espaço. A utilização desses recursos deve ser analisada baseado no seu impacto nas fases de projeto, implementação e utilização do sistema. Deve ser feita uma análise da relação entre esses custos e as consequências causadas por problemas relativos ao mal funcionamento ou à indisponibilidade dos serviços prestados pelo sistema.

Uma maneira de se implementar técnicas de tolerância a falhas é utilizando a estrutura oferecida pelos mecanismos de tratamento de exceções [20] das linguagens de programação. Esses mecanismos oferecem um arcabouço para a criação de tratadores adequados para cada tipo de exceção, possibilitando o tratamento de possíveis inconsistências, ou até mesmo a continuação da execução das funcionalidades do sistema.

Este trabalho possui foco no tratamento de exceções que será apresentado na seção a seguir (Seção 2.9).

2.9 Tratamento de Exceções

O **comportamento normal** de um sistema é responsável por oferecer os serviços documentados nos seus requisitos. Porém existem algumas circunstâncias que não permitem a correta execução desses serviços. Estas circunstâncias espera-se que ocorram raramente, programadores referem-se a elas como **exceções** [16]. Apesar de se esperar uma ocorrência rara, na prática não é o que se observa. Devido à grande influência do meio externo

[29] Inglês: *fault masking*
[30] Inglês: *design diversity*

nos sistemas computacionais, as situações excepcionais são bastante frequentes [16].

Tratamento de exceções [26] é um mecanismo conhecido para a implementação de tolerância a falhas em sistemas de software. Os Mecanismos de Tratamento de Exceções (MTE) capacitam os desenvolvedores a definirem o **comportamento excepcional** dos sistemas, que consiste tanto das atividades de detecção e lançamento das exceções, quanto dos seus tratadores correspondentes. Quando um erro é detectado, uma exceção é gerada, ou **lançada** e o MTE da linguagem de programação ativa automaticamente o tratador mais próximo que se adéqua ao tipo da exceção. Dessa forma, se a mesma exceção puder ser lançada em diferentes partes do programa, é possível que diferentes tratadores sejam executados. Por esse motivo, o local de lançamento da exceção também é conhecido como Contexto de Tratamento da Exceção (CTE). Um CTE é uma região de um programa onde exceções de um mesmo tipo são tratadas de maneira uniforme. Cada CTE possui um conjunto de tratadores associados a ele e cada tratador, por sua vez, está associado a um tipo de exceção em particular. Uma exceção lançada dentro de um CTE pode ser **capturada** por um dos seus tratadores. Quando a exceção é tratada, o sistema pode retomar o seu comportamento normal de execução; caso contrário, a exceção é **sinalizada** para o CTE de nível imediatamente superior. O Código 2.5 ilustra o funcionamento de um MTE. Apesar da Exceção E1 ter sido lançada no CTE2 (Linha 4), ela só é capturada no nível imediatamente superior (CTE1, Linha 13). Por esse motivo, essa exceção é tratada pelo tratador T3.

Código 2.5: Funcionamento de um Mecanismos de Tratamento de Exceções

```
1  try { //CTE1 (bloco de contexto excepcional 1 )
2    // ... código implementado
3    try { //CTE2 (bloco de contexto excepcional 2 )
4      throw new E1 ( ); // lançamento de uma exceção do tipo E1
5    }
6    catch ( E2 e ) { // Tratador1 é T1
7    // tratamento para exceções do tipo E2
8    }
9    catch ( E3 e ) { // Tratador2 é T2
10   // tratamento para exceções do tipo E3
11   }
12 }
13 catch ( E1 e ) { // Tratador3 é T3
14 // tratamento para exceções do tipo E1
15 }
```

2.10 Uma Arquitetura Confiável Baseada em Tratamento de Exceções

A arquitetura apresentada nesta seção é uma arquitetura genérica que auxilia a modelagem do tratamento de exceções [9]. A Figura 2.18 apresenta os quatro componentes que a compõem e os seus relacionamentos. As responsabilidades de cada um desses componentes arquiteturais foram classificadas em dois tipos:

1. Dependentes da aplicação (DA). Essas responsabilidades são relacionadas às funcionalidades que normalmente são específicas para cada aplicação como, o lançamento de exceções. Outro exemplo poderia ser a especificação de colaborações, que depende dos requisitos do sistema.

2. Independentes da aplicação (IA). Essas responsabilidades incluem funcionalidades básicas do gerenciamento de exceções e por isso podem ser utilizadas em qualquer sistema. Por exemplo, facilidades para gerenciamento de informações contextuais das exceções e o desvio do fluxo de controle no momento do lançamento de uma exceção.

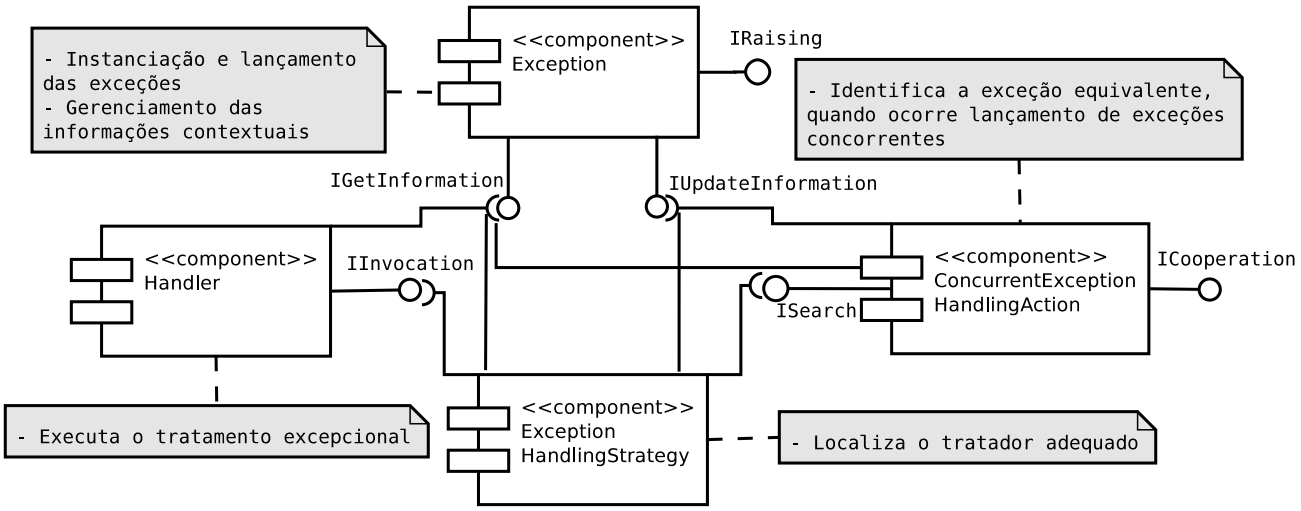


Figura 2.18: Arquitetura de Componentes Genérica de Tratamento de Exceções.

A seguir, cada um dos componentes arquiteturais presentes na Figura 2.18 e suas respectivas interfaces são apresentados brevemente. Adiante, nas seguintes subseções eles são detalhados.

- Exception. Este componente possui duas responsabilidades principais: (i) modelagem e instanciação das classes de exceções, sejam elas locais ou cooperativas (DA); e (ii) gerenciamento das informações contextuais das exceções (IA).

- Handler. As principais responsabilidades deste componente são: (i) modelagem dos tratadores excepcionais (DA); e (ii) invocação dos tratadores (IA).

- Exception Handling Strategy. Devido ao papel genérico que este componente desempenha, as suas responsabilidades são todas independentes da aplicação (IA). São elas: (i) busca por tratadores; e (ii) desvio do fluxo de controle após o lançamento de uma exceção.

- Concurrent Exception Handling Action. Este componente trata dos aspectos complementares necessários para o tratamento de exceções concorrentes. As suas principais responsabilidades são: (i) especificação das colaborações (DA); (ii) sincronização dos participantes; e (iii) resolução da exceção equivalente às exceções concorrentes. A especificação das colaborações consiste na identificação de quais os componentes serão participantes de cada colaboração; a sincronização dos componentes é o processo de garantia da permanência de todos os participantes, até que a colaboração termine; e por fim a identificação da exceção equivalente consiste na implementação de uma das técnicas existentes.

Como pode ser visto na Figura 2.18, o componente `Exception` implementa os serviços relacionados a um repositório de informações contextuais das exceções, tornando essas informações disponíveis para os outros componentes. Esses componentes podem interagir com o componente `Exception` tanto para obter as informações, quanto para atualizá-las. O componente `ExceptionHandlingStrategy` tem um papel central na arquitetura e interage com todos os outros: captura as informações contextuais de exceções (componente `Exception`), antes de selecionar o tratador adequado. Após encontrar o tratador, ele solicita a sua execução (componente `Handler`). Quando exceções concorrentes são lançadas durante uma cooperação, o componente `ConcurrentExceptionHandlingAction` é acionado. Após a descoberta da exceção equivalente, o componente `ExceptionHandlingStrategy` entra mais uma vez em ação, afim de localizar os diferentes tratadores dos componentes participantes que devem ser executados.

Componente `Exception`

Para aumentar o poder de modelagem de sistemas confiáveis, os desenvolvedores de sistemas tolerantes a falhas baseados em tratamento de exceções devem estar aptos a especificar tanto exceções simples, quanto compostas. Qualquer uma dessas exceções pode

ser lançada pelos seus componentes normais em tempo de execução. Esses componentes normais devem armazenar as informações contextuais dessas exceções, como por exemplo o local do seu lançamento, uma vez que elas são úteis para a escolha do tratador mais adequado.

Para o componente `Exception`, as exceções devem ser representadas através de classes que encapsulam suas próprias informações contextuais. Além disso, esse componente implementa três interfaces: (i) `IUpdateInformation`, responsável pela atualização das informações contextuais das exceções; (ii) `IGetInformation`, que disponibiliza as operações para a consulta das informações contextuais; e (iii) `IRaising`, que contém as operações de instanciação e lançamento da exceção. Essas três interfaces providas pelo componente `Exception` são públicas, isto é, disponíveis para todos os componentes do sistema.

Componente `Handler`

Outra necessidade dos desenvolvedores que utilizam os mecanismos de tratamento de exceções para implementar técnicas de tolerância a falhas é a implementação de tratadores excepcionais. Esses tratadores podem se referir tanto a exceções locais, quanto a exceções tratadas de maneira colaborativa. Além de possibilitar a definição dos tratadores, a infraestrutura da arquitetura de software deve oferecer uma separação de interesse entre os tratadores, que implementam o comportamento excepcional do sistema, e os componentes normais, que implementam as funcionalidades especificadas.

Dessa forma, o fato da arquitetura possuir um componente com o papel específico de tratar exceções possibilita uma separação explícita entre os comportamentos normal e excepcional do sistema. O componente `Handler` implementa a interface `IInvocation`. O papel dessa interface é possibilitar a execução dos tratadores excepcionais por parte do componente `ExceptionHandlingStrategy`. Dessa forma, essa interface possui visibilidade pública, onde cada um dos tratadores excepcionais especificados é materializado como uma operação provida.

Os principais benefícios decorrentes da utilização do componente `Handler` são:

- Expressividade. Os tratadores das exceções podem ser tanto locais, quanto arquiteturais, podendo envolver mais de um componente do sistema como parte de um tratamento coordenado mais elaborado.

- Legibilidade e Manutenibilidade. Esta abordagem proporciona uma separação explícita entre as atividades normais do sistema e as relativas à recuperação de erros.

- Reusabilidade. A adoção de componentes normais e excepcionais possibilita a visualização dos dois comportamentos de maneira ortogonal entre si. Essa separação de interesse, além de possibilitar a reutilização de tratadores entre os componentes

de um mesmo sistema, ela torna os componentes normais mais propícios a serem reutilizados.

Componente `ExceptionHandlingStrategy`

O papel do componente `ExceptionHandlingStrategy` é desviar o fluxo de execução do componente no momento em que uma exceção é lançada. Esse fluxo deve ser direcionado para o tratador mais adequado para a exceção e o contexto em questão. Dessa forma, o papel exercido por esse componente se mostra essencial no contexto de qualquer sistema que envolva tratamento excepcional, seja ele crítico ou não.

A identificação do tratador ideal em um fluxo excepcional é uma tarefa recursiva, que inicia no local de lançamento da exceção e segue propagando para o cliente imediato do serviço solicitado, isto é, para o componente que solicitou o serviço, seguindo recursivamente. Essa busca só é finalizada quando o componente `ExcepcionHandlingStrategy` localiza um tratador adequado para a exceção, ou quando ele atinge o final da pilha de requisições (cliente iniciador da chamada). No primeiro caso, o tratador excepcional deve ser executado. No segundo, a exceção provoca a parada repentina do sistema sem que haja a execução de um tratamento prévio. Uma boa prática de programação é evitar a existência de exceções sem tratadores associados, principalmente no contexto de sistemas críticos, onde até mesmo o processo de parada deve ser controlado.

O componente `ExceptionHandlingStrategy` implementa a interface pública `ISearch`. Esta interface possibilita que o componente `ConcurrentExceptionHandlingAction` execute o serviço de busca para obter e executar o tratador de cada participante da colaboração, dados um tipo de exceção e as suas informações de contexto. A principal informação contextual levada em consideração nesse momento é o ponto da execução onde a exceção foi lançada.

Componente `ConcurrentExceptionHandlingAction`

Ao desenvolver sistemas reais baseados em componentes concorrentes, percebe-se que a possibilidade de ocorrência de mais de uma exceção concomitantemente deve ser considerada. Essas exceções devem ser tratadas de maneira colaborativa, sendo necessário para isso a implementação de colaborações e de funções de resolução. O componente `ConcurrentExceptionHandlingAction` implementa uma única interface é pública (`ICooperation`), que oferece as operações necessárias para a implementação dessas duas funcionalidades.

Os principais benefícios da utilização do componente `ConcurrentExceptionHandlingAction` são:

- Uniformidade. Apesar das suas características próprias, a estratégia de tratamento de exceções concorrentes é uma extensão consistente das estratégias convencionais existentes.

- Simplicidade. O fato de uma colaboração ser controlada de maneira centralizada (um único coordenador) proporciona uma maior facilidade de entendimento e implementação. Apesar da simplicidade proporcionada, a centralização é uma vulnerabilidade em termos de confiabilidade, uma vez que representa um ponto único de falhas. Uma maneira de amenizar essa questão é a utilização de coordenadores redundantes. Porém, há uma relação de compromisso estreita entre o número de componentes redundantes e o custo final do sistema. Além disso, o envio de mensagens para os participantes da colaboração deve necessariamente ser intermediado pelo coordenador, o que proporciona uma implementação simples para garantir o confinamento do erro. Porém, dependendo da natureza da aplicação, isso pode se tornar um "gargalo" e comprometer o desempenho.

- Controle da complexidade. A complexidade geral do sistema pode ser melhor controlada com a utilização de colaborações aninhadas, isto é, coordenadores que são participantes de outras colaborações.

2.11 Componente Tolerante a Falhas Ideal

O conceito de componente tolerante a falhas ideal, ou simplesmente componente ideal foi introduzido por Anderson e Lee [36] para denominar um componente tolerante a falhas cujo comportamento é categorizado em dois tipos: comportamento normal e comportamento excepcional (ou anormal). Com o comportamento normal, o componente executa adequadamente os serviços, produzindo respostas corretas. Entretanto, na ocorrência de alguma falha, o componente assume o comportamento excepcional, tentando tratá-la de maneira adequada.

A divisão do comportamento é refletida na sua estrutura, como ilustrado na Figura 2.19. A estrutura do componente ideal é dividida em duas partes: uma define o seu comportamento normal, enquanto a outra é responsável pelo tratamento de exceções, implementando o comportamento excepcional [36]. Um componente ideal pode ser constituído de outros componentes ideais, caracterizando uma estruturação recursiva.

As exceções ocorridas em um componente ideal podem ser classificadas como internas ou externas. As exceções externas, por sua vez, podem ser exceções de defeito, quando indicam a incapacidade de fornecer um serviço, ou de interface, decorrentes de requisições incorretas por parte dos clientes. Na ocorrência de qualquer um dos tipos, a execução normal do componente é interrompida e é ativado o tratador específico para a exceção

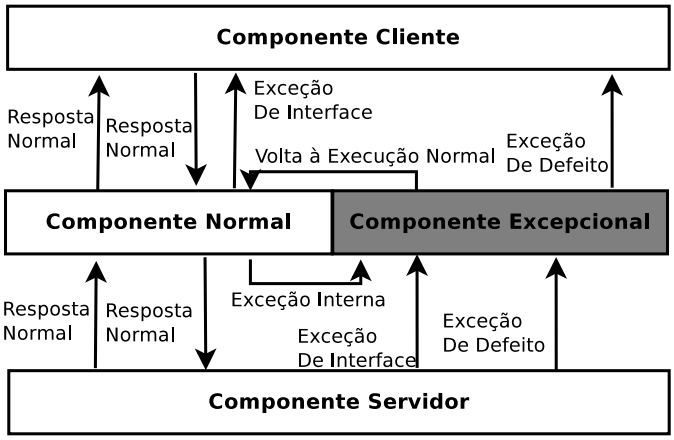


Figura 2.19: Estrutura do Componente Tolerante a Falhas Ideal.

ocorrida. Caso seja possível, após a execução do tratamento adequado, o fluxo de execução deve retornar ao componente normal. Caso não seja possível a recuperação do estado normal do sistema, é retornada uma exceção de defeito à sua camada superior, que pode tratá-la ou continuar propagando. Porém, devido à importância das informações contextuais, quanto mais próximo do local de origem do erro, mais eficiente pode ser o tratamento [36].

2.12 Resumo

Este capítulo apresentou os fundamentos necessários que são usados no método MVTE. Primeiramente foram apresentados os conceitos de LPS na Seção 2.1, que é o tipo de software em que o MVTE é utilizado. Além disso, na Seção 2.2, foi ressaltada a importância da arquitetura de software para o desenvolvimento de sistemas. Entre os fundamentos de DBC apresentados na Seção 2.3, foi possível ver quais as características necessárias para que um processo de desenvolvimento possa se adequar ao desenvolvimento de sistemas baseados em componentes. Em seguida, na Seção 2.4 é apresentado os conceitos que unem a arquitetura de software, as LPS e, também, o desenvolvimento baseado em componentes, o que auxilia no reuso e na evolução de LPS. Nas Seções 2.5 e 2.6 foram apresentados os modelos COSMOS, que proporciona um mapeamento das estruturas da arquitetura de software para o código, facilitando o entendimento e a execução das atividades de manutenção, e o COSMOS*-VP, que apresenta conectores que implementam a variabilidade apresentada nas LPS. O COSMOS*-VP utiliza da Programação Orientada a Aspectos (Seção 2.7) para a implementação da variabilidade na LPS. Como o MVTE é um método para inserir a variabilidade de tratamentos excepcionais nas LPS, os conceitos de tolerância a falhas (Seção 2.8), tratamento de exceções (Seção 2.9) e componente tole-

rante a falhas (Seção 2.11) são apresentados para entender o princípios básicos do MVTE. Sendo assim, após apresentados todos os conceitos para o MVTE, o método é apresentado no Capítulo 3.

Capítulo 3

Método MVTE

Este capítulo apresenta o método MVTE (Método de Variabilidade de Tratamento de Exceções) para criar uma arquitetura para linha de produtos de software com a separação entre o comportamento normal e excepcional, sendo que o tratamento excepcional é variável facilitando, assim, a evolução do comportamento normal e excepcional. O método baseia-se em uma junção de métodos propostos e conhecidos na literatura, como o método PLUS [25], o UML Components [11], o modelo de componentes COSMOS* [24] e COSMOS*-VP [19], com algumas alterações sugeridas e mencionadas nas seções a seguir.

As seções explicam cada uma das atividade contidas na Figura 3.1 para facilitar o seu entendimento.

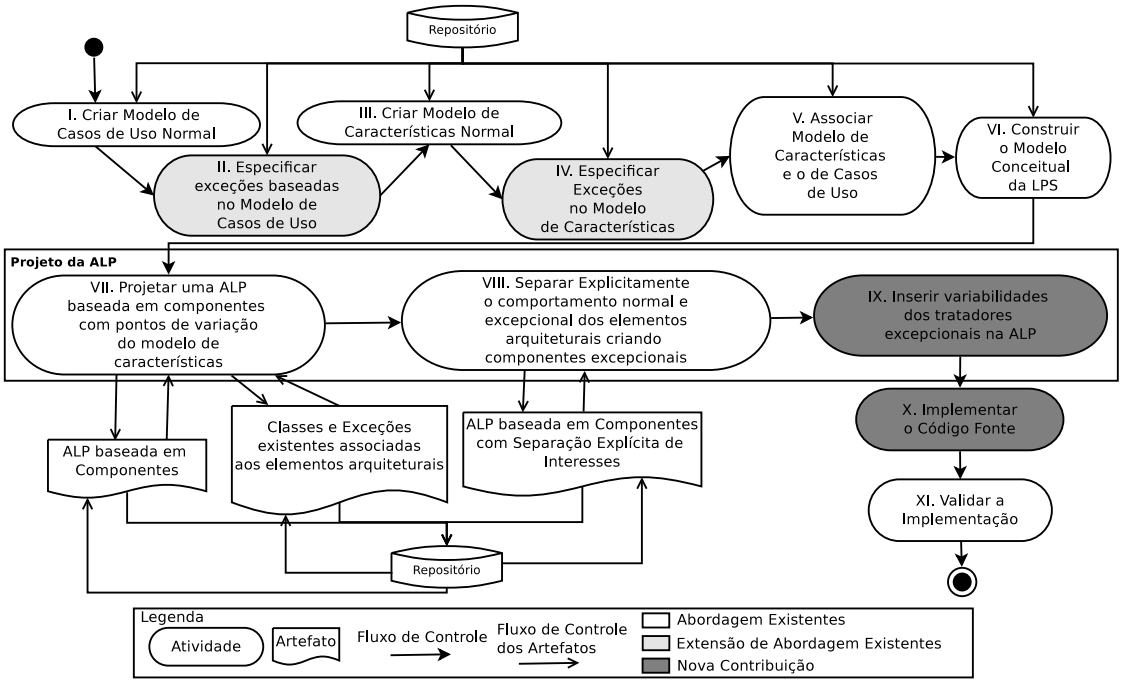


Figura 3.1: Diagrama do Método MVTE em UML.

3.1 Criar Modelo de Casos de Uso Normal

A atividade *Criar Modelo de Casos de Uso Normal* com variabilidade de uma LPS é uma atividade do método PLUS [25]. Todos os requisitos de software especificados para o sistema devem ser fornecidos pelo usuário. De acordo com Gomaa [25], para uma LPS é importante capturar os requisitos funcionais comuns presentes na LPS, tanto os mandatórios, quanto os opcionais e os alternativos. Por este motivo é necessário termos diferentes tipos de casos de uso, como: (i) **Caso de Uso Mandatório**, que é necessário para todos os produtos gerados pela LPS; (ii) **Caso de Uso Opcional**, que é necessário para somente alguns membros da LPS; e (iii) **Caso de Uso Alternativo**, onde diferentes casos de uso são necessários por diferentes produtos da LPS.

No *Modelo de Casos de Uso*, requisitos funcionais são definidos em termos dos **atores**, que são usuários do sistema, e **casos de uso**. Um **ator** participa do caso de uso, e o **caso de uso** define uma sequência de interações entre os atores e o sistema. Cada caso de uso define o comportamento funcional de uma parte do sistema sem revelar sua estrutura interna, como uma caixa preta.

Para um sistema simples, todos os casos de uso e atores são necessários. No entanto, quando uma linha de produtos está sendo modelada, geralmente, apenas alguns casos

de uso e atores são necessários para um produto de uma LPS. Os casos de uso de uma LPS usam os seguintes *UML Stereotypes*, que são um mecanismo do UML usado para diferenciar tipos de elementos de modelo: (i) `«kernel»`, usado para caso de uso que é sempre necessário; (ii) `«optional»`, usado para caso de uso que é algumas vezes necessário; e (iii) `«alternative»`, usado para caso de uso que seja necessário fazer uma escolha.

A Figura 3.2 mostra um exemplo de caso de uso que utiliza os *stereotypes* `«kernel»`, `«optional»` e `«alternative»` interagindo com os `Cliente` e `Hotel`. O `Cliente` inicia qualquer um dos casos de uso, `Cadastro Cliente`, `Reserva`, `Pagamento Boleto` e `Pagamento Cartão` e o `Hotel` irá receber a mensagem transmitida por qualquer um dos casos de uso.

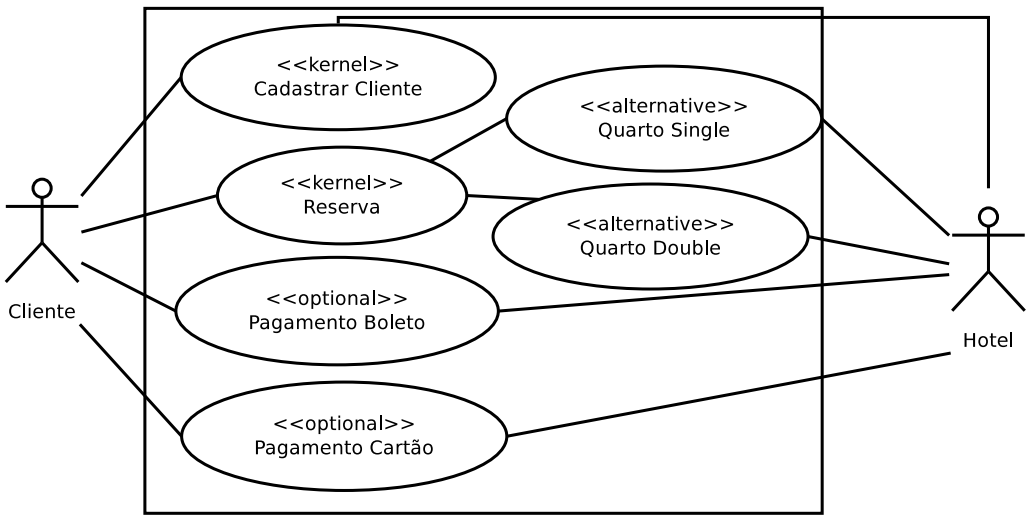


Figura 3.2: Exemplo de Caso de Uso `kernel`, `optional` e `alternative`.

A documentação dos casos de uso é feita segundo a Tabela 3.1.

Tabela 3.1: Documentação do Caso de Uso

Seção	**Descrição**
Nome do Caso de Uso	Cada Caso de uso deve receber um nome para ser associado.
Categoria de Reuso	Esta seção especifica qual o tipo de caso de uso: `kernel`, `optional` ou `alternative`
Resumo	Esta seção descreve brevemente o caso de uso.
Atores	Esta seção informa quais são os atores que irão participar do caso de uso.
Dependência	Esta seção informa quais são os outros caso de uso, ou seja, aqueles casos de uso que são ligados por `includes` ou `extends`. Esta seção é opcional.
Pré-condições	Esta seção especifica uma ou mais condições que devem ser realizadas antes do início do caso de uso.
Descrição	Esta seção mostra uma descrição da sequência principal do caso de uso, que é a mais usual das interações entre o ator e o sistema. A descrição é sempre feita como uma ação inicial do ator, seguido pela resposta do sistema.
Descrições Alternativas	Esta seção descreve um ramo alternativo da sequência principal, podendo existir muitas descrições alternativas.
Pontos de Variação	Esta seção define os locais na descrição do caso de uso onde diferentes funcionalidades podem ser adicionadas por diferentes produtos da LPS.
Pós-condições	Esta seção identifica as condições que devem sempre ser verdadeiras ao fim do caso de uso.
Questões	Esta seção documenta quais são as questões sobre o caso de uso que devem ser discutidas com os usuários.

Um exemplo da documentação dos casos de uso relacionada ao hotel (Cadastrar Cliente) é feita segundo a Tabela 3.2.

Um exemplo da documentação do caso de uso relacionado ao hotel (Pagamento) é apresentada na Tabela 3.3.

Na LPS é necessário descrever variações que são tratadas diferentemente por diferentes produtos da linha de produtos. A variabilidade dos casos de uso é tratada através dos *pontos de variação*, que é o local no caso de uso onde uma mudança pode ser realizada. A *variação* neste contexto é uma situação que pode ser tratada diferentemente por diferentes produtos da linha de produtos.

A documentação das variações dos casos de uso é feita segundo a Tabela 3.4.

Tabela 3.2: Exemplo de Documentação do Caso de Uso (Cadastrar Cliente)

Seção	**Descrição**
Nome do Caso de Uso	Cadastrar Clientes.
Categoria de Reuso	`kernel`
Resumo	Caso de uso relacionado ao cadastro de clientes no sistema.
Atores	Cliente e Hotel.
Dependência	Não há dependência.
Pré-condições	O cliente não deve estar cadastrado no sistema.
Descrição	1. Cliente entra no sistema e acessa a tela de cadastro. 2. Cliente insere seus dados. 3. Sistema verifica se o cliente já está cadastrado, utilizando o CPF. 4. Caso o cliente não esteja cadastrado, o sistema valida a operação e o insere no sistema. 5. O sistema retorna uma mensagem de cadastro realizado com sucesso.
Descrições Alternativas	1. Sistema verifica se o cliente já está cadastrado, utilizando o CPF. 2. Caso o cliente esteja cadastrado, o sistema invalida a operação e não o insere no sistema. 3. O sistema retorna uma mensagem de cadastro não realizado e informa que o CPF já pertence a alguém.
Pontos de Variação	A mensagem de cliente não cadastrado pode ser enviada localmente (tela do cliente) ou remotamente (administrador).
Pós-condições	O cliente deve estar cadastrado no sistema, permitindo, assim, fazer a reserva de quartos.
Questões	Não há questões.

Tabela 3.3: Exemplo de Documentação do Caso de Uso (Pagamento)

Seção	**Descrição**
Nome do Caso de Uso	Pagamento.
Categoria de Reuso	`at-least-one`
Resumo	Caso de uso relacionado ao pagamento no sistema.
Atores	Cliente e Hotel.
Dependência	Não há dependência.
Pré-condições	O cliente deve ter realizado uma reserva no sistema.
Descrição	1. Após o cliente reservar o quarto no sistema, será exibido qual o tipo de pagamento que o cliente deseja. 2. Cliente seleciona entre boleto ou cartão (caso estejam disponíveis pela LPS). 3. Cliente preenche os dados que restam. 4. Sistema realiza a impressão do boleto ou o pagamento pelo cartão. 5. O sistema retorna uma mensagem de boleto impresso ou de pagamento realizado com sucesso.
Descrições Alternativas	Não há descrições alternativas.
Pontos de Variação	A mensagem de cliente não cadastrado pode ser enviada localmente (tela do cliente) ou remotamente (administrador).
Pós-condições	Pagamento realizado com sucesso.
Questões	Cliente deve escolher qual o produto do hotel, ou seja, com o pagamento do tipo boleto e/ou cartão.

Tabela 3.4: Documentação das Variações do Caso de Uso

Seção	Descrição
Nome	Cada variação de Caso de uso deve receber um nome para ser associado.
Tipo de Funcionalidade	Esta seção especifica quais os tipos de pontos de variação que serão inseridos: `optional`, `mandatory alternative` ou `optional alternative`.
Número das Linhas	Esta seção mostra em quais linhas na descrição do caso de uso que a variação será inserida.
Descrição	Esta seção mostra uma descrição da funcionalidade inserida pelo ponto de variação.

Um exemplo da documentação das variações dos casos de uso (Tipos de Pagamento) é feito segundo a Tabela 3.5.

Tabela 3.5: Exemplo de Documentação das Variações do Caso de Uso (Tipos de Pagamento)

Seção	Descrição
Nome	Tipos de Pagamento.
Tipo de Funcionalidade	`optional`
Número das Linhas	2
Descrição	O pagamento será escolhido entre boleto e cartão, caso os dois estejam na LPS.

Durante a criação desse modelo de caso de uso, o foco é baseado no comportamento normal do sistema. Na Seção 3.2, a seguir, é apresentado a extensão do modelo de casos de uso para o comportamento excepcional.

3.2 Especificar Exceções baseadas no Modelo de Casos de Uso

A atividade *Especificar Exceções baseadas no Modelo de Casos de Uso* é um refinamento da Atividade I (Seção 3.1). Esta atividade irá especificar e representar os possíveis tratadores de exceções variáveis para um determinado caso de uso. Para diferenciar os casos de uso do comportamento normal do comportamento excepcional, Shui et al. [43] propõem o uso do *stereotypes* `«handler»`; assim, os desenvolvedores identificam o comportamento excepcional com maior facilidade. Deste modo, no modelo de casos de uso não serão somente identificados os casos de uso do comportamento normal de uma LPS, mas também os casos de uso do comportamento excepcional variáveis, que serão indicados pelos *stereotypes*: (i) `«kernel handler»`, usado para caso de uso de tratadores excepcionais

que é sempre necessário; (ii) «optional handler», usado para caso de uso de tratadores excepcionais que é algumas vezes necessário; e (iii) «alternative handler», usado para caso de uso de tratadores excepcionais em que seja necessário fazer uma escolha. Assim, poderão ser vistas desde o principio do desenvolvimento as possíveis exceções que deverão ser encontradas e tratadas pela LPS.

No modelo de casos de uso, os tratadores de exceções («handler») devem ser associados a algum caso de uso de comportamento normal. Esta associação é similar ao padrão UML «extends»; ela especifica qual será o comportamento do caso de uso de comportamento normal quando afetado por uma exceção e como o tratador de exceção deverá agir ao encontrá-la.

Ao ocorrer uma exceção, o comportamento normal entra em espera ou é abandonado, e a interação do tratamento de exceção é iniciada. O tratador pode temporariamente usar o sistema para cuidar da exceção e então voltar para o comportamento normal. Neste caso, Shui et al. [43] nomeiam a relação como «interrupt & continue». Os casos em que o comportamento excepcional não puder ser tratado facilmente, e consequentemente ocorrer uma falha, são chamados de «interrupt & fail».

A exceção que ativará o tratador de exceções é mostrada como um comentário da ligação entre os casos de uso do comportamento normal e do comportamento excepcional no modelo de casos de uso. A Figura 3.3 mostra um exemplo de caso de uso com o comportamento normal e excepcional em uma LPS. No exemplo, o Caso de Uso 1 é obrigatório para todos os produtos (kernel) e está associado a um tratador excepcional Excep. Caso de Uso 4, que é obrigatório, e será ativado quando a exceção Exceção1 ocorrer e ao detectá-la o sistema não irá parar («interrupt & continue»). O Caso de Uso 2 é obrigatório para todos os produtos e está associado a dois tratadores excepcionais Excep. Caso de Uso 5 e Excep. Caso de Uso 6, que são alternativos, e será ativado quando a exceção Exceção2 ocorrer e ao detectá-la o sistema não irá parar («interrupt & continue») e escolherá um dos dois tratadores. O Caso de Uso 3 é obrigatório para todos os produtos e está associado a um tratador excepcional Excep. Caso de Uso 7, que é opcional, e será ativado quando a exceção Exceção3 ocorrer e ao detectá-la o sistema irá parar («interrupt & fail»).

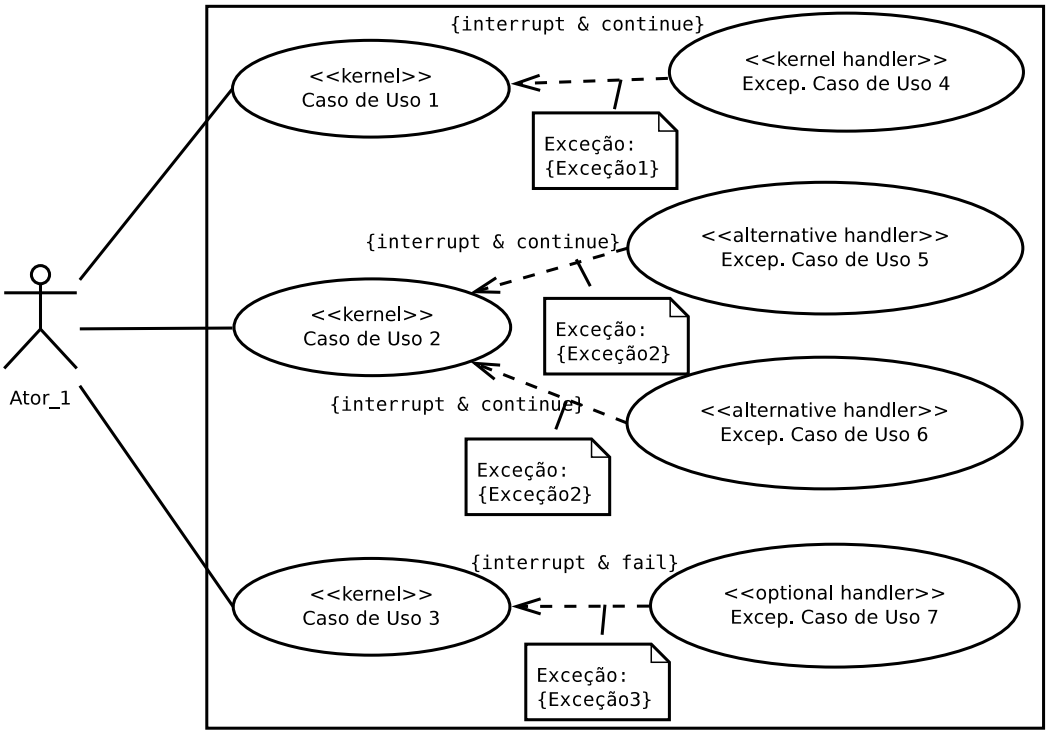


Figura 3.3: Exemplo de Caso de Uso Excepcional com variabilidade.

Para documentar os casos de uso tratadores de exceções é usada a Tabela 3.6.

Tabela 3.6: Documentação dos Casos de Uso Tratadores de Exceções

Seção	**Descrição**
Nome Trat. Exc.	Cada caso de uso deve receber um nome para ser associado.
Descrição	Esta seção descreve o tratador de exceções.
Tipo de Funcionalidade	Esta seção especifica quais os tipos de pontos de variação que serão inseridos: `optional handler`, `kernel handler` ou `alternative handler`.
Número das Linhas	Esta seção mostra em quais linhas na descrição do caso de uso que a exceção pode ocorrer.
Atividade do Tratador	Esta seção descreve como será tratado a exceção ocorrida.
Pós-Condições	Esta seção descreve o conjunto de pós-condições que deve ser verdadeira no fim do tratamento.

Um exemplo de documentação dos casos de uso dos tratadores de exceções (CPF já cadastrado) é usado na Tabela 3.7 e na Figura 3.4.

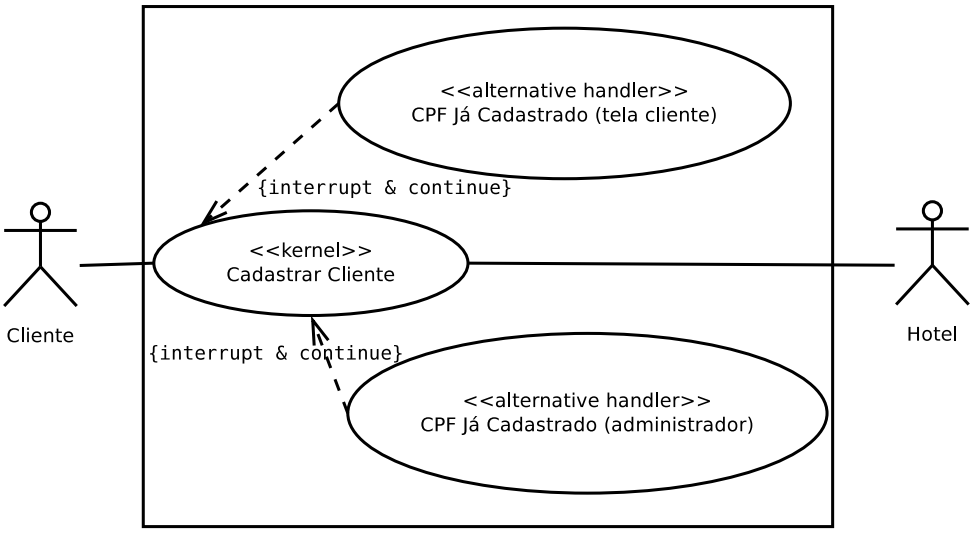


Figura 3.4: Exemplo de Caso de Uso Excepcional com variabilidade.

Tabela 3.7: Documentação dos Casos de Uso Tratadores de Exceções

Seção	**Descrição**
Nome Trat. Exc.	CPF já Cadastrado.
Descrição	A mensagem de cliente não cadastrado pode ser enviada localmente (tela do cliente) ou remotamente (administrador).
Tipo de Funcionalidade	`alternative handler`.
Número das Linhas	4.
Atividade do Tratador	Ao ocorrer a exceção uma mensagem deverá ser enviada ao cliente ou ao administrador.
Pós-Condições	O sistema não deve parar, e continuar com as próximas operações permitindo a alteração do CPF, caso tenha ocorrido um erro de digitação.

3.3 Criar Modelo de Características Normal

A atividade *Criar Modelo de Características Normal* de uma LPS é uma atividade do método PLUS [25]. Em uma LPS, uma *característica* é um requisito que é provido por um ou mais membros da linha de produtos. Em particular, as *características* são usadas para diferenciar os membros da linha de produtos, e assim, determinar e definir as funcionalidades comuns e variáveis da LPS.

Especificando um sistema, este não é considerado completo a menos que possua todos os requisitos satisfeitos, assim o sistema deve prover todas as suas características. Se um sistema for desenvolvido em fases, a ordem em que as características são introduzidas deve ser considerada, pois algumas características dependem de outras e estas devem ser

introduzidas anteriormente para garantir que a característica tenha a sua sequência.

Analisando as características da LPS, uma importante distinção é feita entre as características *mandatórias*, *opcionais* e *alternativas*. As características *mandatórias* são um subconjunto das características da linha de produtos que devem ser providas por todos os produtos. As características *opcionais* são aquelas que são necessárias em apenas alguns membros da linha de produtos. As características *alternativas* são necessárias quando há uma escolha para selecionar um dado membro da linha de produtos. As características opcionais e alternativas descrevem as variabilidades na linha de produtos e determina as características de um dado membro da linha de produtos. Assim, na análise de características da linha de produtos, a ênfase é na análise das características opcionais e alternativas.

Características relacionadas podem ser agrupadas em *Grupos de Características*. Cada tipo de grupo de características é identificado por *stereotypes*. Existem diferentes *stereotypes* para as categorias dos Grupos de Características, como `«exactly-one-of feature group»`, `«zero-or-one-of feature group»`, `«at-least-one-of feature group»` e `«zero-or-more-of feature group»`(Seção 2.1.2).

3.4 Especificar Exceções no Modelo de Características

A atividade *Especificar Exceções no Modelo de Características* é um refinamento da Atividade III (Seção 3.3). Esta atividade irá especificar e representar os possíveis tratadores de exceções variáveis para uma determinada característica. Para diferenciar as características do comportamento normal do comportamento excepcional, são utilizados os *stereotypes* `«handler»`, assim os desenvolvedores identificaram o comportamento excepcional com maior facilidade. Deste modo, no modelo de característica não identificará somente as características do comportamento normal de uma LPS, mas, também, as do comportamento excepcional variáveis, que serão indicados pelos *stereotypes*: (i) `«kernel handler»`, usado para tratadores excepcionais que será sempre necessário; (ii) `«optional handler»`, usado para tratadores excepcionais que será algumas vezes necessário; e (iii) `«alternative handler»`, usado para tratadores excepcionais que seja necessário fazer uma escolha. Assim, poderá ser visto desde o principio do desenvolvimento as possíveis exceções que deverão ser encontradas e tratadas pela LPS.

Os grupos de características também são identificados por diferentes *stereotypes* que mostram quais são os grupos de comportamento excepcional e os de comportamento normal presentes no diagrama de característica. Para identificar os diferentes *stereotypes* para as categorias dos Grupos de Características dos tratadores excepcionais serão utilizados os seguintes *stereotypes*, `«exactly-one-of feature group handler»`, `«zero-or-one-of feature group handler»`, `«at-least-one-of feature group handler»` e `«zero-`

-or-more-of feature group handler».

O exemplo da LPS do Hotel apresentado na Seção 2.1.2 foi alterado para ter as características que serão tratadores de exceções variáveis.

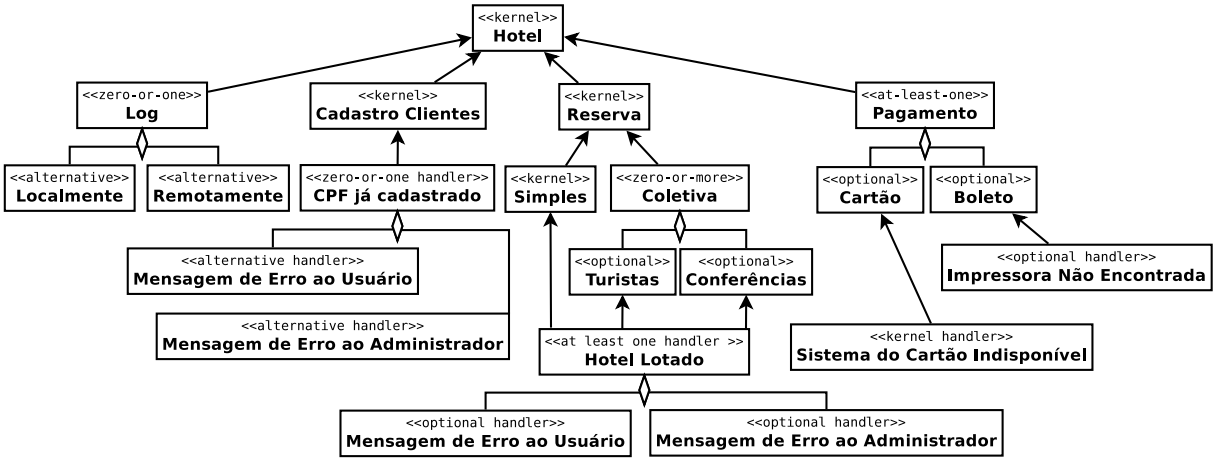


Figura 3.5: Exemplo de Modelo de Características com tratadores excepcionais variáveis da LPS Hotel.

No modelo de características da LPS do Hotel temos a característica `Reserva` que é dividida em `Simples`, `Coletiva Turistas` e `Coletiva Conferências`. Para esta característica temos o grupo de característica de tratadores excepcionais `«at-least-one» Hotel Lotado`, que possui as características `Mensagem de Erro ao Usuário` e `Mensagem de Erro ao Administrador` sendo que ambas são `«optional handler»`. Para a característica `Cadastro Clientes` temos um grupo de características de tratadores excepcionais `«zero-or-one handler» CPF já cadastrado` que pode ser escolhido entre as características `Mensagem de Erro ao Usuário` e `Mensagem de Erro ao Administrador`, sendo que ambas são `«alternative handler»`. Para o `Pagamento` em `Cartão` temos a característica `«kernel handler» Sistema de Cartão Indisponível` que é obrigatório este tipo de tratamento para o `Cartão`. Já para a característica `Boleto`, a característica tratadora de exceções será `«optional handler» Impressora não encontrada`.

3.5 Associar o Modelo de Características e o de Casos de Uso

A atividade *Associar o modelo de Características e o de Casos de uso* é realizada para apoiar a rastreabilidade do modelo de casos de uso para o modelo de características e, consequentemente, melhorar a evolução do sistema. Rastreabilidade é a associação entre

os artefatos, no caso o modelo de Características e o de Casos de uso, para facilitar o sistema e ter um reuso consistente [40]. Os artefatos são inter-relacionados pelas associações de rastreabilidade para assegurar as definições das partes comuns e das variabilidades das LPS pelos artefatos. Estas associações suportam a evolução consistente da linha de produtos. Por exemplo, se um artefato é alterado, os outros artefatos associados a ele também serão afetados pela mudança para manter a consistência.

A documentação explícita das variabilidades permite que a rastreabilidade da variabilidade seja melhorada. Este tipo de associação realizada pela rastreabilidade é necessária, por exemplo, para realizar de forma eficiente a engenharia de requisitos. E além disso, a associação da rastreabilidade facilita a implementação de mudanças.

Para gerenciar a sobreposição e diferenciar potencialmente as definições das variabilidades com o modelo de casos de uso, foi utilizado o modelo ortogonal de variabilidade, proposto por Pohl et al [40]. O modelo ortogonal de variabilidade documenta a variabilidade da LPS e define a rastreabilidade ligando as variantes e os pontos de variações e suas definições nos requisitos correspondentes as variabilidades. O modelo ortogonal de variabilidade permite associar uma variante com diferentes definições de variabilidade do modelo de caso de uso. Acompanhando as associações da rastreabilidade, o analista pode determinar como uma dada variante é realizada no modelo de casos de uso e verificar se as diferentes definições das variabilidades são consistentes.

Na Figura 3.6 representa a associação parcial do diagrama de caso de uso com o modelo de características. No exemplo da Figura 3.6 o caso de uso `Pagamento Cartão` e `Pagamento Boleto` está associado com as características `Cartão` e `Boleto`, respectivamente. Cada caso de uso será associado a uma característica para mostrar a rastreabilidade que esses dois documentos possuem.

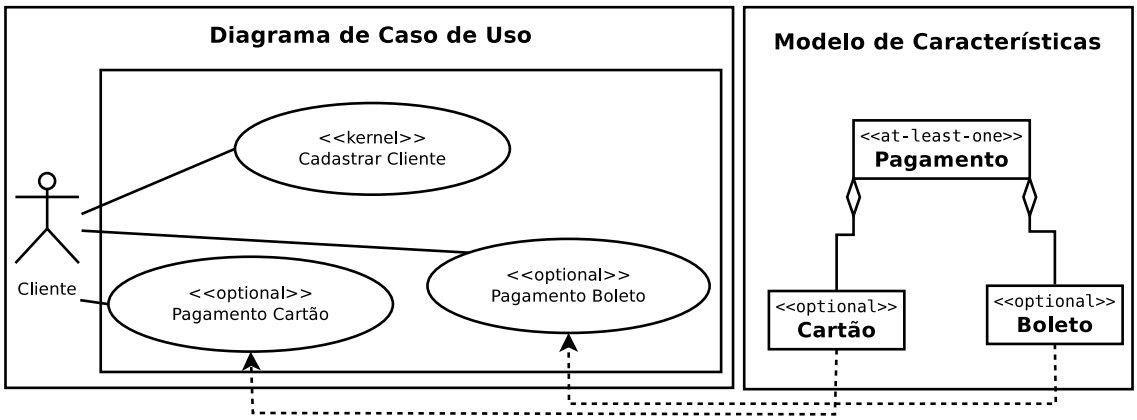


Figura 3.6: Exemplo de Associação entre o Modelo de Características e o Diagrama de Caso de Uso.

3.6 Construir o Modelo Conceitual da LPS

A *Construir o Modelo Conceitual da* (Atividade VI) é outra atividade baseada no método PLUS [25] é importante para a modelagem de uma LPS porque ela oferece uma noção da existência das partes comuns e das variáveis de uma família de produto. O modelo conceitual descreve a estrutura estática da linha de produtos sendo modelada. Ele é usado para modelar as associações entre as classes, assim como a modelagem da hierarquia usada na linha de produtos.

Na modelagem conceitual em um sistema, todas as classes são necessárias. No entanto, em uma LPS, uma classe é necessária em pelo menos um membro da linha de produtos mas não é necessária por todos os membros da linha de produtos. Uma classe que é requerida por todos os membros da linha de produtos é referida como `kernel class`. Uma `optional class` é uma classe que é requerida por alguns membros da família, mas não por todos. Diferentes membros da família do produto necessitam de uma versão alternativa de classes. Estas classes são chamadas de `variant class`. Dentro de um grupo de `variant classes`, uma delas deve ser designada como *default*. Dependendo da característica da linha de produtos, alguns variantes devem poder coexistir em um mesmo membro da linha de produtos. Em outras linha de produtos, os variantes são mutuamente exclusivos para um membro da família. A hierarquia de generalização pode ser usada para modelar as diferentes variações das classes, que são usadas por diferentes membros da família.

Ao modelar uma LPS, cada classe pode ser categorizada de acordo com a sua característica de reuso. Os *stereotypes* `kernel`, `optional` e `variant` são usados para diferenciar essas características da linha de produtos. Na UML 2.0, um elemento da modelagem pode ser descrito com mais de um stereotype. Então um stereotype pode ser usado para representar a característica de reuso enquanto um diferente stereotype descreve uma característica em particular da classe, como por exemplo `entity` ou `interface`. Sendo que cada uma dessas características são independentes, ou seja, uma classe pode ser `kernel` e `entity`.

Na modelagem do domínio do problema, a fase inicial é modelar as classes físicas e as classes do tipo *entity*. `Classes Físicas` são classes que possuem características de objetos físicos, ou seja, objetos que podem ser tocados. Nestas classes incluem-se dispositivos físicos, usuários, sistemas externos e temporizadores. `Classes Entity` são classes conceituais de dados que possuem uma longa vida. Elas são particularmente predominantes em sistemas de informação, como aplicações bancárias.

Todo o processo de criação das classes e suas associações, generalizações e especializações é feita baseando-se na UML 2.0 [2].

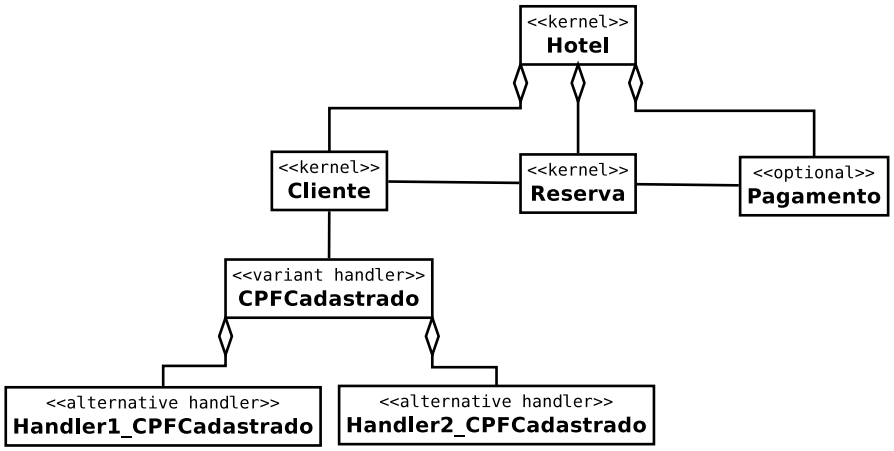


Figura 3.7: Exemplo de Diagrama de Classe em UML.

3.7 Projeto da ALP

Nesta Seção serão apresentadas as atividades VII, VIII e IX em conjunto, pois as três irão projetar a ALP com o comportamento normal e excepcional separados facilitando a evolução e a reusabilidade dos elementos arquiteturais.

A atividade *Projetar uma ALP baseada em componentes com pontos de variação do modelo de características* (Atividade VII) é baseada no método UML Components [11] apresentada na Seção 2.3.1. Ela tem como objetivo prover uma visualização em alto nível da estrutura do sistema mantendo a rastreabilidade com o modelo de características. Do modelo de características e de classes existentes da LPS, dois artefatos são produzidos: uma arquitetura de linha de produtos baseada em componentes, e as classes e exceções existentes relacionadas com cada elemento arquitetural.

Nesta atividade, primeiramente, são identificados os elementos arquiteturais do modelo de características. A seguir, são associadas as classes e as exceções com os elementos arquiteturais. Finalmente, são conectados os componentes baseados nas dependências entre as classes. Para executar essa atividade, é necessário refatorar a arquitetura de software com dois objetivos: (i) componetizar a ALP para especificar os pontos de variações arquiteturais relacionados aos tratamentos de exceções, e; (ii) reduzir o número de elementos arquiteturais para melhorar o entendimento do sistema.

Ao projetar a arquitetura de software baseada em componentes foi utilizado o processo do *UML Components* [11] com algumas adaptações. A primeira delas é o estilo da arquitetura em camadas, sugerido pelo processo, foi alterada para o estilo arquitetural MVC, que é adotado originalmente pelo ALP. E a segunda, ao invés de usarmos o modelo de casos de uso como entrada para identificarmos os componentes, é utilizado o modelo de características da LPS.

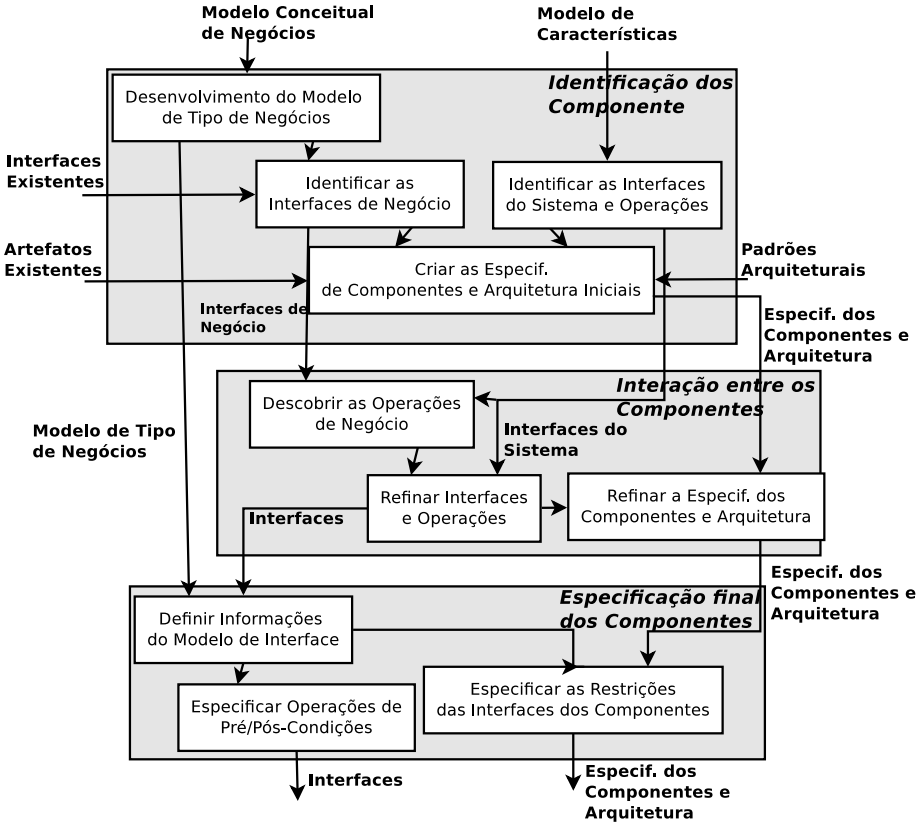


Figura 3.8: Figura Adaptada do UML Components [11] dos estágios de `identificação dos componentes`, `interação entre os componentes` e `especificação final dos componentes` do nível de especificação.

Na Figura 3.8, o estágio de `identificação dos componentes` recebe como entrada o `modelo conceitual do negócio` e o `modelo de características` do sistema, artefatos obtidos da fase de especificação de requisitos e da atividade de criação do modelo de característica, respectivamente. O objetivo dessa etapa é criar uma arquitetura inicial e as especificações iniciais dos componentes para o sistema. Com as especificações iniciais dos componentes criadas, os componentes já podem ser posicionados na arquitetura nas suas camadas correspondentes de acordo com o modelo arquitetural MVC.

A atividade *Separar Explicitamente o comportamento normal e excepcional dos elementos arquiteturais criando componentes excepcionais* (Atividade VIII) separa o comportamento normal e excepcional criando componentes excepcionais obtidos através do UML Components [11].

No estágio de `identificação dos componentes` (Figura 3.8) com o uso do modelo de característica contendo as características excepcionais, pode-se identificar os componentes que serão componentes excepcionais na arquitetura da linha. O objetivo dessa separação

de interesses é evitar o espalhamento do tratamento de exceções em diversos componentes, assim o tratamento de exceções possuíra componentes específicos para facilitar sua evolução e implementação das variabilidades relacionadas a ele.

Na fase de `interação entre os componentes`, ocorre o detalhamento das estruturas identificadas anteriormente. Por esse motivo, nesse estágio o projetista examina como cada operação do sistema será implementada, ou seja, é decidido como cada componente será combinado para que o serviço seja implementado corretamente. No final da execução da interação entre os componentes, a arquitetura do sistema já está definida.

Durante esta fase é o momento em que ocorre a atividade *Inserir Variabilidades dos tratadores excepcionais na ALP* (Atividade IX), e é necessário realizar alguns passos para que os componentes normais sejam conectados com os excepcionais: (i) especificar as variabilidades excepcionais. Ao especificá-las pode-se definir em quais locais da arquitetura que será utilizado os `Conectores-VP` para obter as variabilidades excepcionais; (ii)definir os `Conectores-VP`, XPIs e os aspectos abstratos (Seção 2.7.2 entre os componentes normais e excepcionais (Seção 2.6). Com essa definição é possível unir os componentes normais e excepcionais provendo a variabilidade excepcional mostrada no modelo de características. Usando a combinação de componentes, aspectos e XPIs, o projeto do componente pode especificar os locais onde as interceptações são permitidas, tornando esses locais públicos para o uso do XPI.

Finalmente, a fase de `especificação final dos componentes` é onde as especificações são refinadas e, se possível, representadas de uma maneira formal. O detalhamento da especificação só deve ser feito no momento em que a arquitetura do sistema já está considerada estável. Com a especificação dos componentes finalizada, é chegada a hora de providenciar cada um dos componentes para em seguida compor o sistema, testá-lo e entregá-lo ao cliente para utilização. Apesar da importância dessas fases, o *UML Components* não detalha a sua execução.

As outras atividades não sofrem alteração e, portanto, seguem o mesmo processo do UML Components (Seção 2.3.1).

3.8 Implementar o Código Fonte

A atividade *Implementar o Código Fonte* é necessário seguir algumas regras: (i) uso do modelo de implementação de componentes COSMOS*, descrito na Seção 2.5; (ii) gerar o código fonte da estrutura arquitetural; (iii) encapsular as classes com os elementos arquiteturais; (iv) realizar a configuração arquitetural.

Para inserir as variabilidades dos tratadores de exceção provê um guia para especificar os conectores aspectuais para os pontos de variação arquiteturais, chamados *Conectores-VP*, descrito na Seção 2.6, e inserir o código fonte. Os *Conectores-VP* inserem os pontos

de variação do tratamento excepcional que já é conhecida no desenvolvimento da ALP.

3.9 Validar a Implementação

A atividade *Validar a Implementação* é uma atividade obrigatória para assegurar que o sistema esteja funcionando corretamente de acordo com o documento de requisitos. Esta atividade deve ser realizada durante todo o processo de desenvolvimento do sistema, porém, como o *UML Components* o foco do método é o desenvolvimento do sistema e não propor a melhor maneira de validação para um sistema.

3.10 Resumo

Este capítulo apresentou o método MVTE que é a principal contribuição deste trabalho. Este método foi criado para especificar a variabilidade no tratamento excepcional em LPS baseado em componentes. O método utiliza os métodos (PLUS e UML Components) e modelos (COSMOS* e COSMOS*-VP), com algumas alterações, já conhecidos na literatura. Ele possui as seguintes atividades: *Criar Modelo de Casos de Uso Normal*, *Especificar Exceções baseadas no Modelo de Casos de Uso*, *Criar Modelo de Características Normal*, *Especificar Exceções no Modelo de Características*, *Associar o modelo de Características e o de Casos de uso*, *Construir o Modelo Conceitual da LPS*, *Projetar uma ALP baseada em componentes com pontos de variação do modelo de características*, *Separar Explicitamente o comportamento normal e excepcional dos elementos arquiteturais criando componentes excepcionais*, *Inserir Variabilidades dos tratadores excepcionais na ALP*, *Implementar o Código Fonte* e *Validar a Implementação*. Com essas atividades o MVTE auxilia a evolução da LPS e ajuda na rastreabilidade do comportamento normal e excepcional durante todo o desenvolvimento da LPS.

Capítulo 4

Estudos Empíricos

Neste capítulo são apresentados dois estudos empíricos que avaliam os benefícios e desvantagens da utilização do método MVTE. O método propõe o uso do modelo de implementação de componentes COSMOS*-VP na especificação e implementação de ALPs componentizadas inserindo-o nas variabilidades do comportamento excepcional. O objetivo dos estudos empíricos é analisar qualitativamente e quantitativamente a estabilidade de ALPs com o modelo COSMOS*-VP. Estabilidade arquitetural é a capacidade que uma arquitetura tem de evoluir mantendo suas propriedades modulares [21], isto é, quanto mais estável uma ALP, menor serão os impactos sofridos por ela durante evoluções da LPS. Na Seção 4.1 é apresentado o primeiro estudo empírico da utilização do modelo COSMOS*-VP nas variabilidades do comportamento excepcional. Nesse estudo empírico, o modelo foi utilizado para a especificação e implementação da ALP da linha de aplicações de governo eletrônico (E-Gov) para a automatização do processo de obtenção da Carteira Nacional de Habilitação (CNH). A Seção 4.3 apresenta o segundo estudo empírico, onde o método proposto foi utilizado na especificação e implementação da ALP da linha de aplicações de manipulação de mídias para dispositivos móveis chamada MobileMedia [50]. Cada um dos estudos empíricos foi organizado nos seguintes tópicos: (i) descrição do estudo empírico; (ii) planejamento do estudo empírico; (iii) execução do estudo empírico; e (iv) resultados obtidos.

4.1 E-Gov-CNH

4.1.1 Descrição do Estudo Empírico

Neste estudo empírico, para avaliar a aplicabilidade do MVTE, foi utilizada a LPS voltada para *e-Gov* chamada `E-Gov-CNH`. A LPS foi desenvolvida, em Java, pelos alunos de pós-graduação do *Software Engineering and Dependability Research Group* (SED) do Instituto

de Computação da Universidade Estadual de Campinas (Unicamp), e simula a automatização do processo de obtenção de Carteira Nacional de Habilitação (CNH).

A LPS tem em sua maioria características opcionais, possibilitando que, em um possível cenário de implantação, os diversos departamentos estaduais de trânsito utilizem diferentes estratégias gradativas de automatização do processo de obtenção de CNH. É possível derivar da linha desde um produto extremamente simples, que apenas cadastra novos candidatos a condutores e agenda a pré-inscrição deles ao processo de obtenção da CNH, até produtos que automatizam quase totalmente o processo.

Um dos principais objetivos do método proposto é facilitar a evolução do comportamento excepcional das ALPs. Este estudo empírico tem como foco detalhar qualitativamente os benefícios que o método traz para a evolução de ALPs. Para isso, o impacto arquitetural causado por dois cenários de evolução da linha é discutido detalhadamente, comparando os resultados obtidos utilizando o modelo proposto, que usa o COSMOS*-VP, com os resultados da utilização combinada de COSMOS* e aspectos. A utilização do COSMOS*-VP refina o modelo de conectores COSMOS* visando a modularização de pontos de variação, não permitindo que eles sejam implementados espalhadamente por vários elementos arquiteturais. Já com a utilização do COSMOS* e do aspectos, os pontos de variação não ficam modularizados nos elementos arquiteturais; eles podem ficar espalhados entre esses elementos, o que não auxilia a evolução ou manutenção de elementos existentes.

4.1.2 Planejamento do Estudo Empírico

Para avaliar os benefícios da utilização dos conceitos empregados pelo MVTE, duas ALPs de `E-Gov-CNH` são utilizadas neste estudo empírico, uma especificada e implementada usando COSMOS*-VP, que será chamada daqui por diante de `VP-CNH`, e uma implementada utilizando COSMOS* e aspectos, que será chamada de `AO-CNH` devido ao fato de suas variabilidades serem implementadas utilizando componentes aspectuais. Para contrastar a estabilidade arquitetural provida pela utilização de COSMOS*-VP, foram comparados os impactos arquiteturais causados nas duas ALPs em dois cenários de evolução.

O produto mais simples da linha, que pode ser criado selecionando apenas as características do núcleo mandatório da linha, provê o cadastramento de dados de novos candidatos a condutores e o agendamento da pré-inscrição deles no processo de obtenção de CNH. Ainda relacionado ao cadastramento de dados, a linha tem como características opcionais, a validação dos dados cadastrais. As validações de CPF, RG e endereço, que são características opcionais independentes entre si, são realizadas por meio dos protótipos de serviços web `ReceitaFederalServ`, `PoliciaCivilServ` e `SERASAServ`. Cada uma dessas validações possui uma característica alternativa para o tratamento excepcional de

`CPF Inválido`, `RG Inválido` e `Endereço Inválido`. Caso o cadastramento seja feito por um usuário com permissão de administrador, será exibido um tipo de mensagem direta para o administrador; se o usuário possuir permissão de usuário comum, será exibido outro tipo de mensagem.

Além dessas características alternativas, um produto pode ser derivado contendo o agendamento de exames médicos e psicotécnicos, o agendamento de exames teóricos e o práticos, que são características opcionais independentes entre si. Utilizando um produto que provê a característica opcional *Exames Preliminares*, o usuário, além de poder agendar exames médicos e psicotécnicos, pode buscar os locais dos exames usando filtros, que selecionam as clínicas mais próximas de seu endereço, ou as que têm horários disponíveis à noite, por exemplo. O agendamento dos exames também possui características excepcionais variáveis, podendo ser escolhidas pelo tipo de usuário (administrador ou usuário comum) que entra no sistema.

E por último, um produto que possua pelo menos uma das características opcionais de agendamento de exames é possível selecionar as características de pagamento *Pagamento-Boleto* e *Pagamento-Cartão*, ou apenas uma delas. Para cada característica de pagamento é utilizada uma característica excepcional alternativa, que é escolhida pelo tipo de usuário que entrará no sistema. Na Figura 4.1 mostra o modelo de características antes das evoluções escolhidas acontecerem.

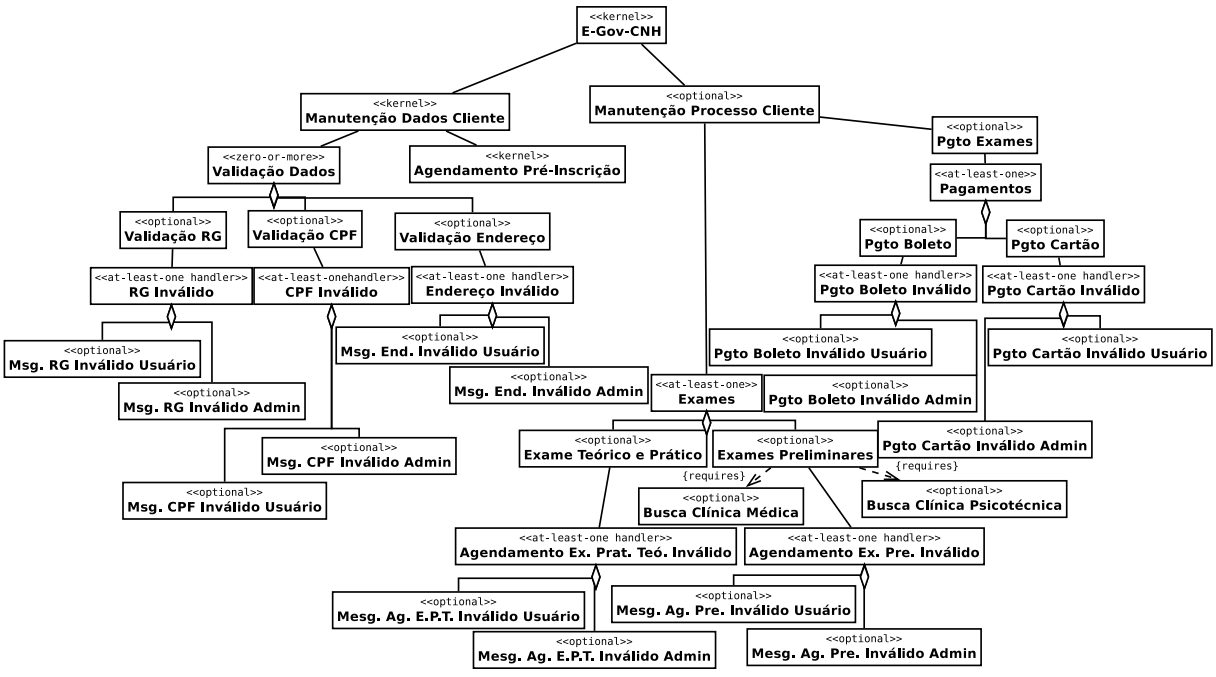


Figura 4.1: Modelo de Características do E-Gov-CNH antes das evoluções.

As evoluções, sofridas pelas duas ALPs envolvidas neste estudo empírico, foram criadas baseadas em exemplos de cenários de evolução reais descritos por Svahnberg e Bosch [45]. Os cenários utilizados foram: (i) divisão de característica; e (ii) aperfeiçoamento de característica. Os cenários são descritos a seguir:

- Divisão de característica. A versão V2 da ALP é criada após o cenário de evolução divisão de característica, que divide a característica opcional `Exames Preliminares` nas características opcionais `Exames Médicos` e `Exame Psicotécnicos`. O componente opcional que implementava essa característica é dividido, causando impacto arquitetural na ALP. A divisão de implementações, que sob um ponto de vista conceitual, modularizam mais de uma funcionalidade ocorre com frequência [45]. Dessa forma, é permitido que elas sejam usadas separadamente, como o que ocorre com as característica Exames Médicos e Exames Psicotécnicos, que após a evolução, podem ser selecionadas individualmente. Deste modo, ocorre, também, a divisão da característica excepcional que era exclusiva dos Exames Preliminares e passam a ser duas características excepcionais para o Exame Médico e para o Exame Psicotécnico.

- Aperfeiçoamento de característica: O cenário mais comum de evolução em Linha de Produtos de Software é o aperfeiçoamento da implementação de uma característica modularizada por um componente [45]. Geralmente, é criada uma opção alternativa ao componente aperfeiçoado, em vez de apenas substituí-lo. O cenário de evolução aperfeiçoamento de característica, que acarreta na criação da versão V3 da ALP, é um exemplo. A implementação da característica `Gerenciamento de Dados` é evoluída para realizar o cadastramento de novos condutores que também possam ser identificados por meio do novo Registro de Identidade Civil (RIC), e não apenas através de CPF e RG. Além de transformar um componente mandatório em alternativo e criar mais um componente alternativo, essa evolução acarreta na criação da característica opcional `Validação de RIC`. Com a criação desta nova característica é necessário a criação da característica excepcional de `RIC Inválido` com suas alternativas dependendo do tipo de usuário que está no sistema.

A estabilidade arquitetural provida por cada uma das abordagens envolvidas nesse estudo empírico foi avaliada baseada em métricas convencionais de impacto de mudanças [42] e em um conjunto de métricas de modularidade [48]. Para melhorar a compreensão do impacto causado pelas evoluções, foram utilizadas as seguintes métricas de modularidade:

- Impacto de Mudanças nos módulos - A métrica de impacto de mudanças provê dados para a análise da estabilidade das ALPs. O impacto de mudanças causado por uma evolução é mensurado contando-se o número de elementos arquiteturais modificados e adicionados à ALP após a evolução.

- Acoplamento entre módulos - Mede o acoplamento eferente médio dos elementos arquiteturais de uma ALP. Foi utilizada a abordagem introduzida por Chidamber e Kemerer [13] para a contagem do número de elementos de que cada um dos elementos de uma ALP depende. Após essa contagem, foi calculada a média entre as quantidades de dependências de todos os elementos arquiteturais de cada uma das ALP.

- Difusão de Interesses sobre módulos - Para medir essa métrica, foi escolhida uma característica da LPS e em seguida foi analisada a implementação de cada módulo da arquitetura para identificar qual ou quais módulos eram responsáveis pela implementação da característica escolhida. [21]

4.1.3 Execução do Estudo Empírico

A execução do estudo empírico foi realizada conforme as atividades descrita na Figura 3.1.

- Atividade I e II - Nestas atividades, explicadas anteriormente nas Seções 3.1 e 3.2, é realizada a criação do modelo de Caso de Uso e as especificações das exceções nos casos de uso. Com essas atividades foi obtido o modelo de casos de uso na Figura 4.2.

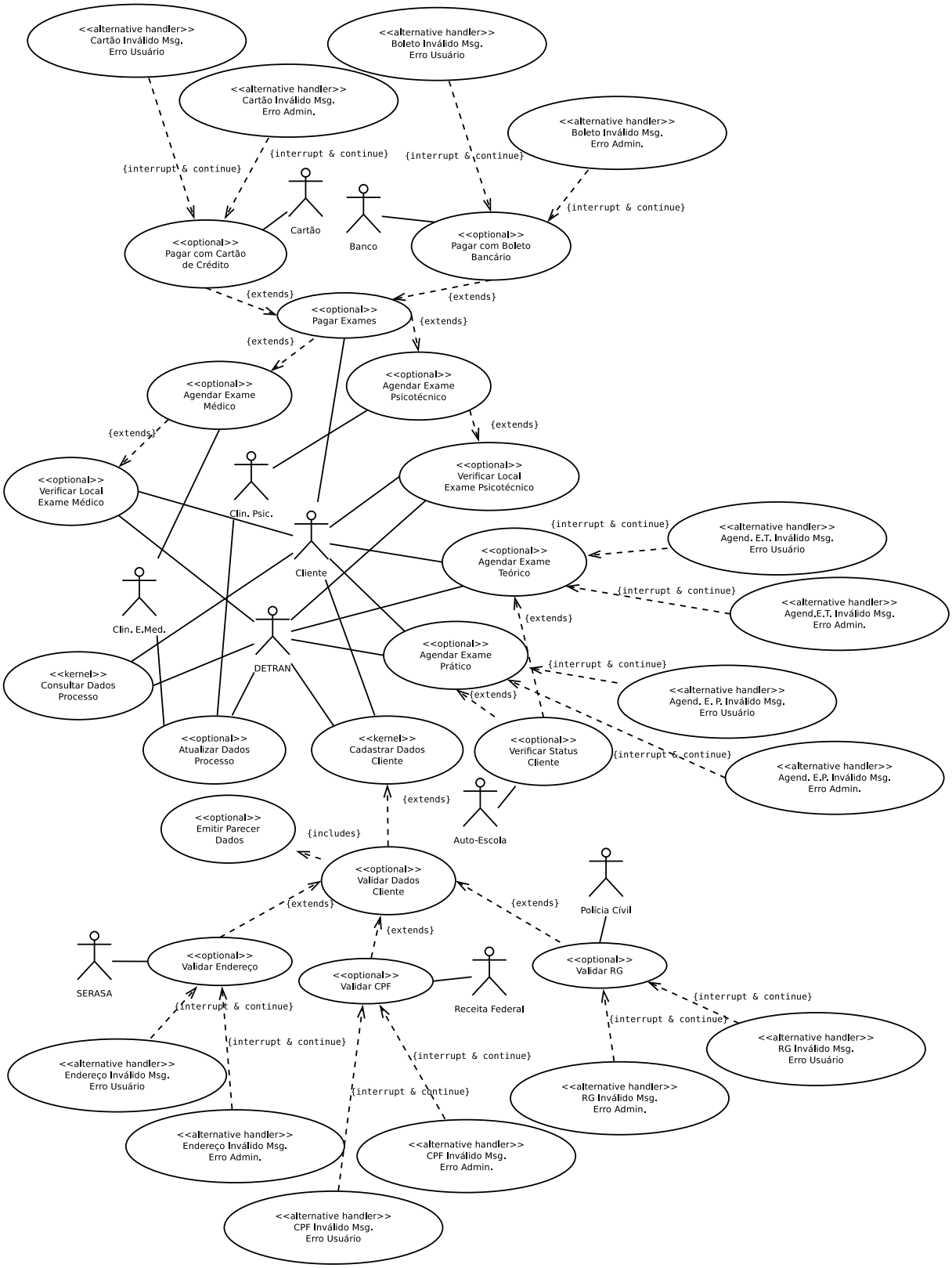


Figura 4.2: Modelo de Casos de Uso do E-Gov-CNH.

- Atividade III e IV - Nestas atividades é criado o modelo de característica com a variabilidade excepcional no modelo, como explicado nas Seções 3.3 e 3.4. A Figura 4.3 mostra o modelo de característica depois das evoluções que foram aplicadas no estudo de caso.

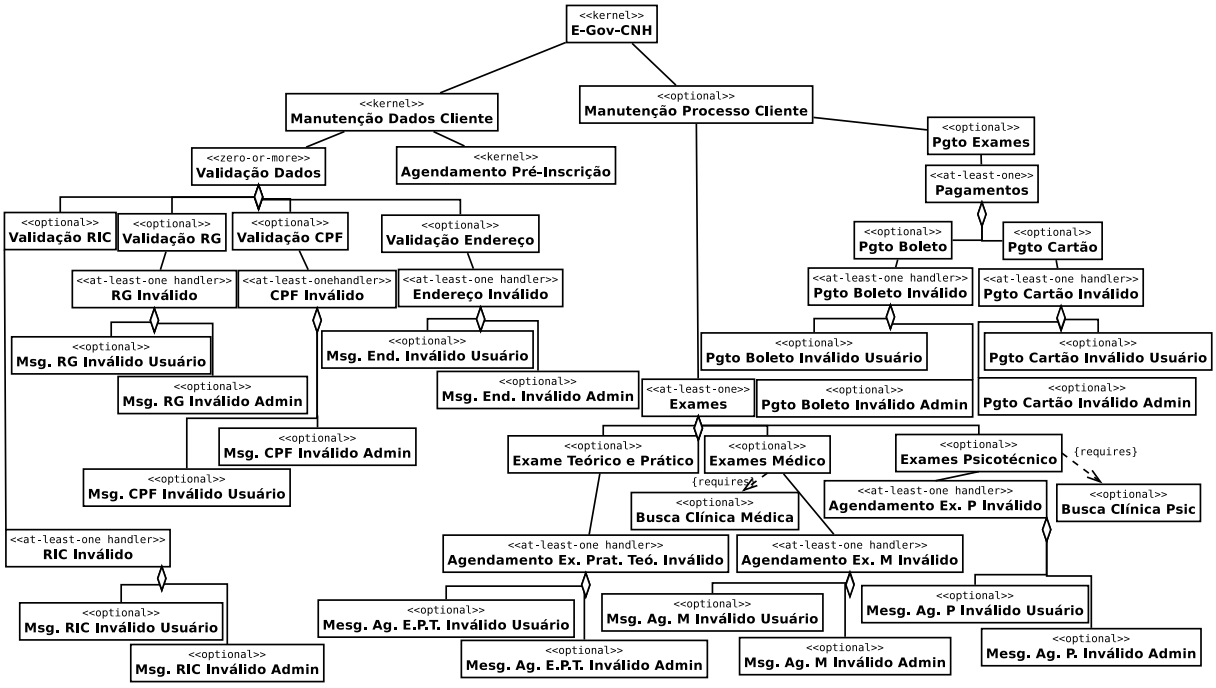


Figura 4.3: Modelo de Características do E-Gov-CNH depois das evoluções.

- Atividade V - Associação do modelo de Características e o de Casos de uso. A Figura 4.4 mostra um trecho da associação realizada entre o modelo de característica e o de caso de uso.

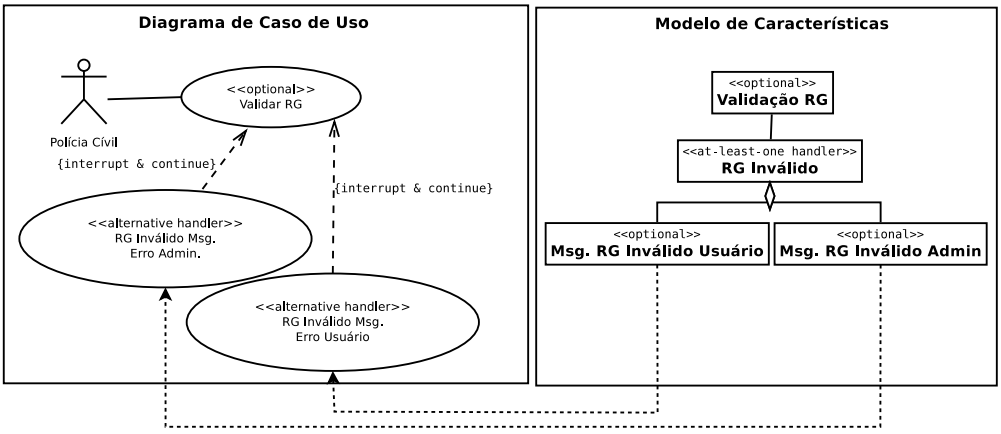


Figura 4.4: Associação do Modelo de Características e do Modelo de Casos de Uso do E-Gov-CNH.

- Atividade VI - Construção do Modelo Conceitual da LPS. A Figura 4.5 mostra o diagrama de classes da LPS E-Gov-CNH.

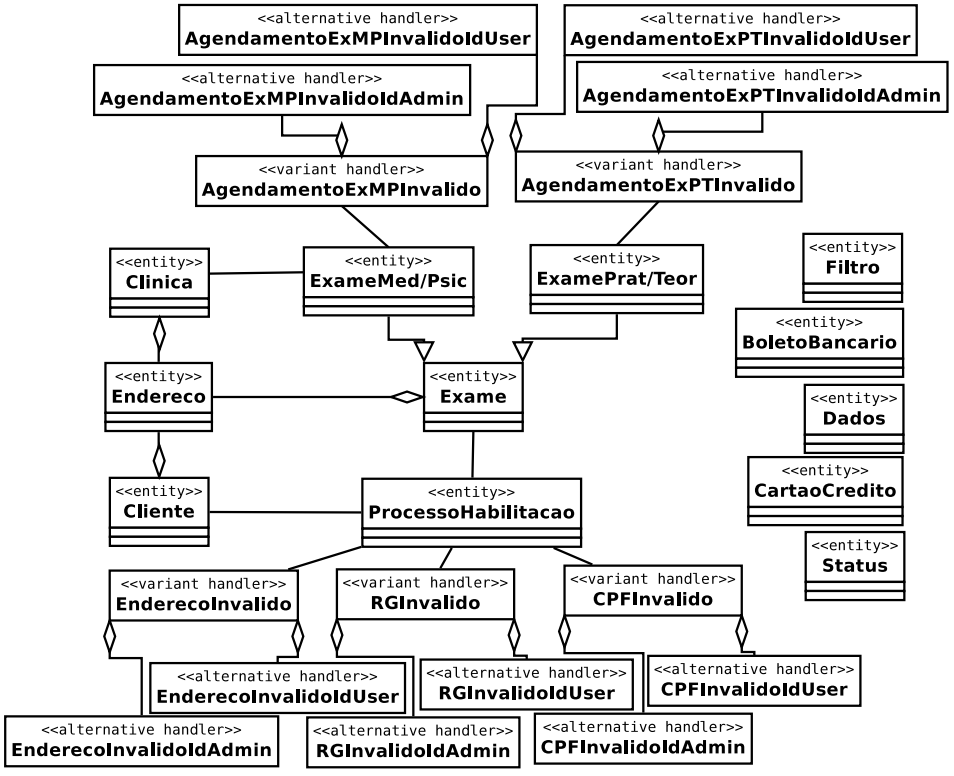


Figura 4.5: Diagrama de classes do E-Gov-CNH.

- Projeto da ALP - Atividades VII, VIII e IX. A Figura 4.6 mostra um trecho da

arquitetura da LPS do E-Gov-CNH.

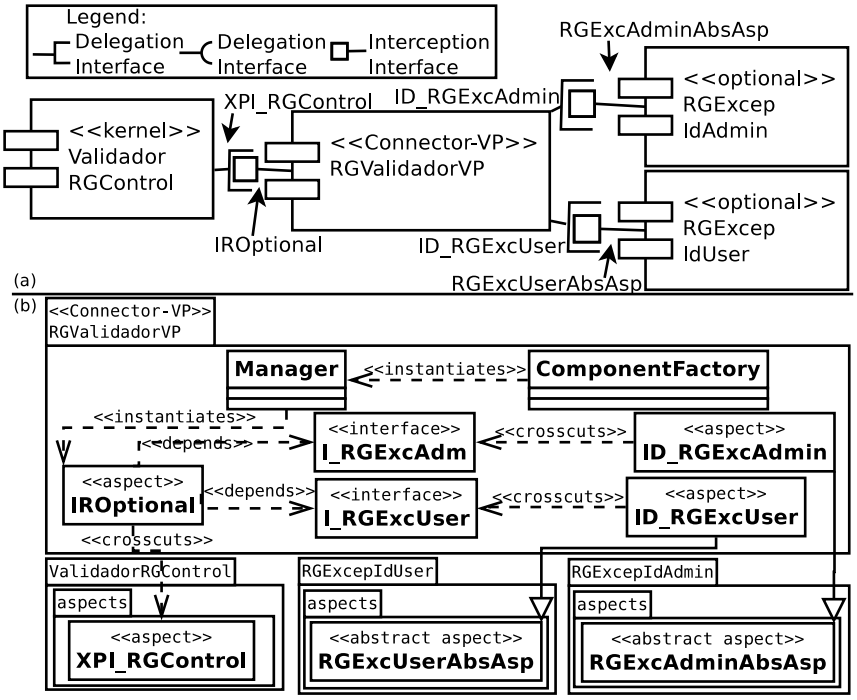


Figura 4.6: Trecho da Arquitetura do E-Gov-CNH - Validação do RG.

- Atividade X - As três versões das duas ALPs (`AO-CNH` e `VP-CNH`) são implementadas utilizando o COSMOS*. E ao implementar as variabilidades existentes nas duas ALPs é utilizado o COSMOS*-VP.
- Atividade XI - Durante essa atividade de validação, foi utilizado o JUnit, para verificar e validar os comportamentos excepcionais do sistema.

Depois de utilizado o método proposto foi necessário medir as métricas de impacto de mudanças, acoplamento e difusão de interesses sobre os módulos das três versões (V1, V2 e V3) das ALPs `AO-CNH` e `VP-CNH`. Em seguida, foi realizado o estudo comparativo dos resultados obtidos das duas ALPs.

4.2 E-Gov-CNH: Resultados Obtidos

4.2.1 Impacto de Mudanças nos Módulos

O impacto de mudanças sobre os elementos da ALP é medido através do número de componentes e conectores (módulos) afetados, ou seja, adicionados, modificados e removidos,

em cada evolução da ALP. Quanto maior o número de módulos afetados, maior será o impacto sofrido pela ALP. Assim, nesta seção será discutido o impacto de mudanças das ALPs `AO-CNH` e `VP-CNH` sofrido durante as evoluções propostas.

A Tabela 4.1 exibe o número de módulos afetados nas ALPs durante os cenários de evolução. Com o uso do método proposto, foi apresentada uma diminuição no número total de módulos afetados pelas evoluções. Na `AO-CNH` foram afetados 15 módulos; já no `VP-CNH` foram apenas 9 módulos afetados pelas evoluções.

Tabela 4.1: Número de Elementos Arquiteturais Afetados em cada Versão pelo Impacto de Mudanças.

		ALPs	
Versões	**Elementos**	**AO-CNH**	**VP-CNH**
V2	Adicionados	3	0
	Modificados	2	2
	Removidos	0	0
V3	Adicionados	5	5
	Modificados	4	2
	Removidos	1	0

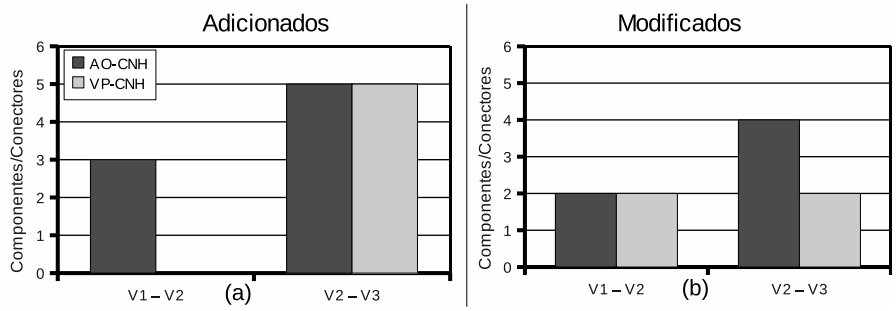


Figura 4.7: Gráfico dos Impactos de Mudanças do E-Gov-CNH.

Durante a divisão de característica da `AO-CNH` foram modificados dois módulos e três módulos foram adicionados, com a divisão da característica `Exames Preliminares` em duas características `Exames Médicos` e `Exames Psicotécnicos`. Com a inclusão dessas características foi necessário adicionar um tratamento excepcional para os `Exames Psicotécnicos` e a alteração nos módulos que ligam essas características. Já na `VP-CNH` na divisão de característica, não houve adição de módulos e dois módulos foram modificados. Nesse caso, o impacto arquitetural foi pequeno pois com o uso do COSMOS*-VP foi criado um ponto de variação interno a um componente evitando que ele fosse divi-

dido em dois componentes; assim, ocorreram somente as duas modificações nos módulos necessários.

No cenário de aperfeiçoamento de característica, onde é inserida uma nova forma de um novo candidato ser cadastrado no sistema, a LPS `AO-CNH` teve cinco módulos adicionados, quatro módulos modificados e um módulo removido. Os módulos adicionados foram aqueles em que ocorre a `Validação do RIC`, o seu tratamento excepcional variável e o novo tipo de candidato que foi inserido no sistema. Já os quatro módulos modificados são: `Validação RG`, `Validação CPF` e `Validação Endereço` e o tipo de candidato antigo. Essas modificações foram necessárias, pois todos esses módulos são necessários para que o novo tipo de candidato seja cadastrado no sistema. Na `VP-CNH`, os módulos adicionados são os mesmos cinco módulos para a `Validação do RIC`, seu tratamento excepcional e o novo tipo de candidato. Os módulos modificados são apenas dois módulos que fazem a análise dos dados, um deles é o conector que irá fazer a seleção entre o novo tipo de cliente e o antigo, e o outro `conector-vp` para mediar as conexões entre os validadores e os novos e antigos tipos de clientes.

Podemos perceber que na V2 da LPS, o uso do MVTE obteve uma pequena melhora no caso em que são adicionados novos elementos arquiteturais, pois ao invés de incluir novos elementos, foi somente necessário modificar dois elementos. Na versão V3 da LPS, a diferença se encontra no número de elementos modificados, que foi reduzida pela metade (de quatro da versão V2 para dois da versão V3). Em ambos os casos, o uso do MVTE trouxe uma não significativa melhora em relação ao uso de aspectos.

4.2.2 Acoplamento entre Módulos

O acoplamento entre módulos refere-se ao nível de interdependência entre partes de um sistema [13]. A estabilidade dessa métrica está relacionada com a estabilidade arquitetural, isto é, uma evolução que altere os elementos arquiteturais da ALP, também causam grandes alterações nos valores da métrica (valores bons ou ruins).

Tabela 4.2: Acoplamento Médio entre Módulos para cada versão da ALPs.

Versão	AO-CNH	VP-CNH
V1	2,62	2,53
V2	2,68	2,55
V3	2,75	2,53

A Tabela 4.2 e a Figura 4.8 apresentam os resultados para o acoplamento entre os módulos das duas ALPs (`AO-CNH` e `VP-CNH`). Nelas pode-se perceber que a utilização do COSMOS*-VP para a variabilidade do tratamento excepcional conseguiu alcançar

o objetivo de diminuir o acoplamento entre os módulos, pois a ALP `VP-CNH` apresenta um acoplamento menor entre seus módulos do que os apresentados pela ALP `AO-CNH`. Isto ocorreu pois os componentes aspectuais não dependem dos componentes que são interceptados. Na ALP `AO-CNH`, o acoplamento entre os módulos aumenta, pois o impacto arquitetural é causado pelas evoluções, tanto adicionando quanto alterando os módulos da ALP.

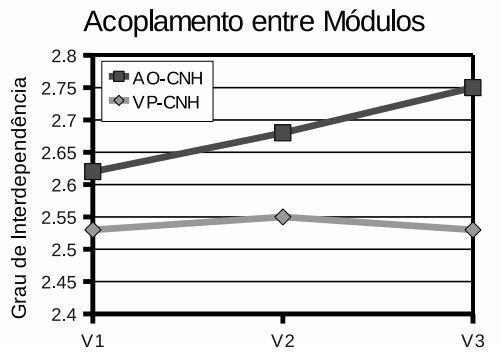


Figura 4.8: Gráfico de Acoplamento entre Módulos do E-Gov-CNH.

A melhora do acoplamento dos módulos ao utilizar o MVTE foi, em média, de cinco a seis porcento em relação ao uso dos aspectos. Esta melhora apresentada é relativamente baixa. O nível de interdependência não sofreu muita alteração entre o uso do MVTE e os aspectos.

4.2.3 Difusão de Interesses sobre Módulos

A difusão de um interesse é dada pelo número de módulos que implementam esse interesse sobre o número total de módulos da arquitetura de LPS. Logo, quanto mais difuso um interesse, menos modularizada é a arquitetura de LPS. Para que um módulo fosse incluído na lista de módulos que implementam um interesse, bastava ter uma única linha de código que fosse relacionada a esse interesse. Neste estudo empírico apresentado, o interesse que foi medido foi o tratamento de exceções que se encontra no código fonte das ALPs.

Tabela 4.3: Difusão de Interesses sobre Módulos para cada versão da ALPs.

Versão (%)	**AO-CNH**	**VP-CNH**
V1	24	20
V2	25	20
V3	27	18

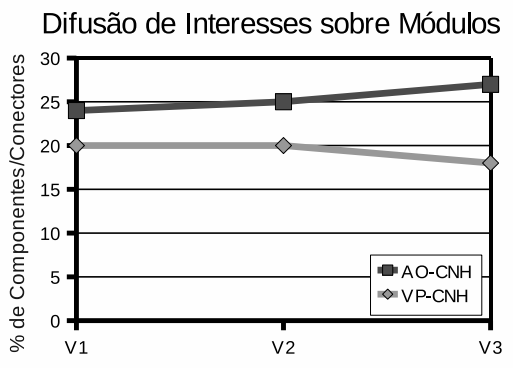


Figura 4.9: Gráfico de Difusão de Interesses sobre Módulos do E-Gov-CNH.

A Tabela 4.3 e a Figura 4.9 mostram os resultados obtidos durante a métrica de difusão de interesses. Deste modo, é possível verificar que as versões da ALP implementada com o COSMOS*-VP (`VP-CNH`) tiveram uma porcentagem menor de difusão do que a implementada em aspectos (`AO-CNH`), pois a implementação do tratamento excepcional com o COSMOS*-VP é totalmente separada do comportamento normal da ALP, facilitando as alterações que podem ou poderão ser realizadas. O MVTE apresenta melhora na difusão de interesses relacionados ao tratamento excepcional, pois com o seu uso, o tratamento excepcional é separado do comportamento normal da ALP, enquanto que o tratamento excepcional que é implementado em aspectos, o número de módulos que o implementam é maior, pois está mais espalhado no código fonte da ALP.

4.3 MobileMedia

4.3.1 Descrição do Estudo Empírico

Neste estudo empírico, que avalia o MVTE, foi utilizada uma linha de produtos desenvolvida pela Universidade de Lancaster de aplicações para dispositivos móveis, chamada `MobileMedia` [50]. Os produtos gerados pela LPS podem manipular fotos, músicas e vídeos em dispositivos com recursos limitados, como celulares, através da utilização de várias tecnologias baseadas na plataforma Java ME, por exemplo, SMS, WMA e MMAPI. MobileMedia possui 7 cenários de evolução, o que resultou em 8 diferentes versões. E, através de diferentes combinações de suas características, implementadas na última versão da linha, é possível derivar 168 diferentes produtos. A Figura 4.10 mostra o modelo de característica antes das evoluções que foram aplicadas no estudo empírico.

MobileMedia foi escolhida para compor esse estudo por ter sido utilizada como um exemplo de linha de produtos de software em outros trabalhos, como [21] e [22]. A

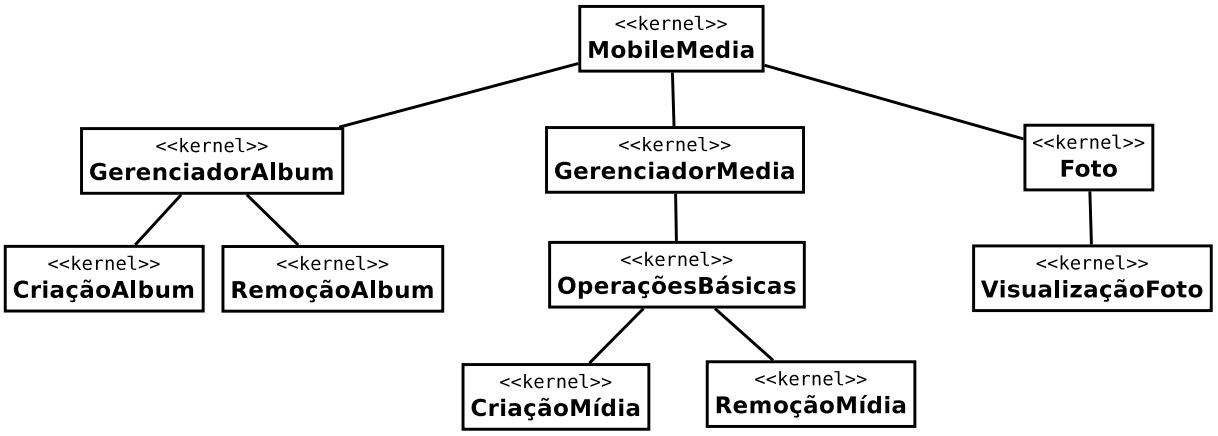


Figura 4.10: Modelo de Características do MobileMedia antes das evoluções.

linha possui duas diferentes implementações, criadas, evoluídas e desenvolvidas em Lancaster [21]. São elas: (i) `OO-MM`, implementação com 11 mil linhas de código, onde as variabilidades foram implementadas utilizando Compilação Condicional; e (ii) `AO-MM`, implementação contendo 12 mil linhas de código. Nessa versão, aspectos foram utilizados para implementar as variabilidades da ALP e na implementação do requisito não funcional e transversal de tratamento de exceções. As duas implementações da linha passaram pelos mesmos 7 cenários de evolução.

A Tabela 4.4 lista os 7 cenários de evolução pela qual a LPS MobileMedia passou. A coluna Versão identifica a versão resultante da evolução. A coluna Descrição apresenta uma descrição da evolução ocorrida. O tipo da evolução pela qual a linha passou está apresentada na coluna Tipo da evolução. Os cenários compreendem diferentes tipos de evolução, envolvendo características mandatórias, opcionais e alternativas, bem como a introdução de um requisito não funcional. O objetivo das evoluções é causar impactos significativos na ALP da linha. Dessa forma, MobileMedia serve como um importante estudo empírico para avaliar a estabilidade de ALPs especificadas e implementadas utilizando o modelo COSMOS*-VP qualitativamente, e sobretudo, quantitativamente.

4.3.2 Planejamento do Estudo Empírico

A LPS MobileMedia possui um conjunto de características heterogêneo, envolvendo características mandatórias, opcionais e alternativas. Para avaliar os benefícios do modelo COSMOS*-VP para a evolução de ALPs, a implementação `AO-MM` foi refatorada utilizando COSMOS*-VP, resultando na implementação `VP-MM`. A ALP da implementação refatorada utilizando COSMOS*-VP foi comparada com outras duas refatorações, `AO-MM` e `COSMOS*-MM`. Para a obtenção da implementação `AO-MM` nenhum novo modelo foi uti-

Tabela 4.4: Lista de Cenário de Evolução de MobileMedia

Versão	**Descrição**	**Tipo de Evolução**
V0	Núcleo da LPS	
V1	O tratamento de exceções foi incluído.	Inclusão de uma característica mandatória.
V2	A característica opcional adicionada permite que o usuário organize as fotos de acordo com o número de visualizações de cada uma. A característica mandatória adicionada permite que o usuário edite o rótulo das fotos.	Inclusão de uma característica mandatória e uma opcional.
V3	A característica opcional adicionada permite que usuários identifiquem quais são as suas fotos favoritas.	Inclusão de uma característica opcional.
V4	A característica opcional adicionada permite que usuários armazenem múltiplas cópias de fotos.	Inclusão de uma característica opcional.
V5	A característica opcional adicionada permite enviar e receber fotos aravés de mensagens SMS.	Inclusão de uma característica opcional.
V6	A característica de gerenciamento de fotos foi evoluída para característica para mídias de uma forma geral. Dessa forma foi possível criar o conceito de mídias alternativas. Agora a aplicação pode gerenciar fotos e(ou) músicas. As características opcionais que permitem fotos favoritas, copiar fotos e organizar fotos ordenadamente devem prover suas funcionalidades também às músicas.	Evolução de uma característica mandatória em alternativa e adição de uma nova característica alternativa.
V7	A característica alternativa adicionada permite o gerenciamento de vídeos. Agora a aplicação pode gerenciar fotos e(ou) músicas e(ou) vídeos. As características opcionais devem ser providas também para os vídeos.	Inclusão de uma característica alternativa.

lizado, apenas combinamos a utilização POA para a implementação de variabilidades com o modelo de implementação de componentes COSMOS*. A intenção de incluir nessa comparação a implementação `COSMOS*-MM` é permitir realizar uma comparação dos benefícios trazidos pelos dois conceitos-chave de COSMOS*-VP separadamente. `COSMOS*-MM` utiliza apenas o novo modelo de especificação, que permite especificar componentes aspectuais desacoplados, enquanto que `VP-MM`, além do novo modelo de especificação, utiliza o modelo de pontos de variação explícitos, aplicado aos modelos de conectores e de implementação COSMOS*-VP.

As três implementações da LPS envolvidas nesse estudo empírico foram refatoradas da mesma implementação original da linha, a `AO-MM`, passaram pelos mesmos sete cenários de evolução descritos na Tabela 4.4, e, portanto, apresentam o mesmo diagrama de características. Entretanto, em todas versões (V0-V7), cada implementação apresenta uma ALP diferente das demais. Isso ocorre, principalmente, devido às diferenças da abordagem que cada modelo utiliza para conectar componentes aspectuais. Em `VP-MM` são utilizados `Conectores-VP` para conectá-los, em `COSMOS*-MM` são utilizados conectores aspectuais simples que estender aspectos abstratos ligando-os às XPIs, e por fim, em `AO-MM` não é utilizado nenhum tipo de conector para mediar a conexão de componentes aspectuais.

O tratamento excepcional para o MobileMedia possui uma variabilidade em todas as exceções encontradas no sistema. Para esta variabilidade será realizado de acordo com a escolha de configuração que deseja aplicar a ALPs, sendo que, ao encontrar uma exceção durante a execução do sistema pode ser impresso na tela do celular a mensagem da exceção correspondente, ou então, juntamente com a mensagem da exceção é disparado um alarme sonoro de erro pelo celular.

A estabilidade arquitetural provida por cada uma das abordagens envolvidas nesse estudo empírico foi avaliada com as mesmas utilizadas no primeiro estudo empírico, impacto de mudanças, acoplamento e difusão de interesses sobre módulos.

4.3.3 Execução do Estudo Empírico

A execução do estudo empírico foi realizada conforme as atividades da Figura 3.1, conforme será mostrado a seguir:

- Atividade I e II - Nestas atividades, explicada anteriormente nas Seções 3.1 e 3.2 é realizada a criação do modelo de Caso de Uso e as especificações das exceções nos casos de uso. Com estas atividades foi obtido modelo de casos de uso na Figura 4.11.

  Por ter uma limitação de página o caso de uso da Figura 4.11 só representa o comportamento normal que o sistema possuirá. Mas um caso de uso parcial representado na Figura 4.12 exibe dois tratamentos excepcionais variáveis no caso de uso, `Label Já Existe` e `Não Existe Album`.

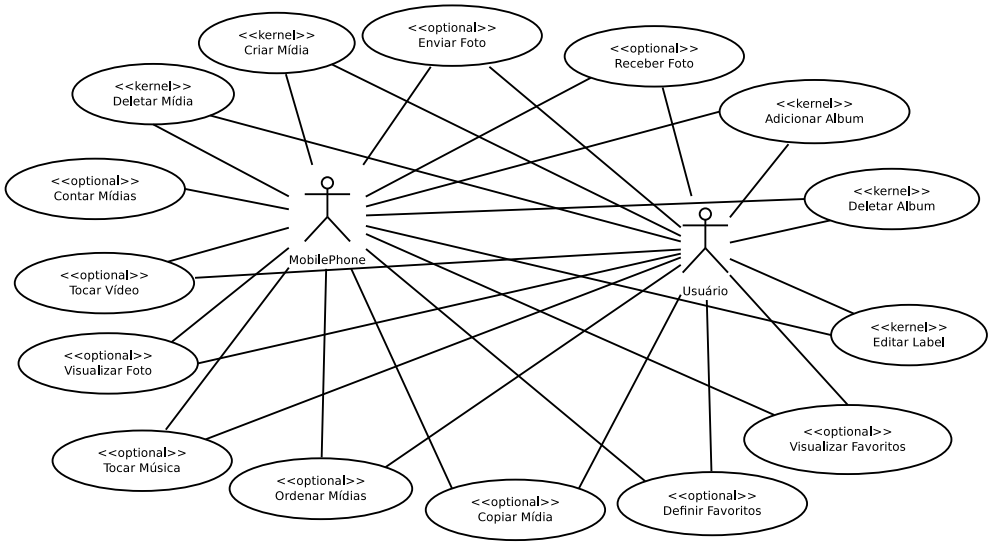


Figura 4.11: Modelo de Casos de Uso do MobileMedia.

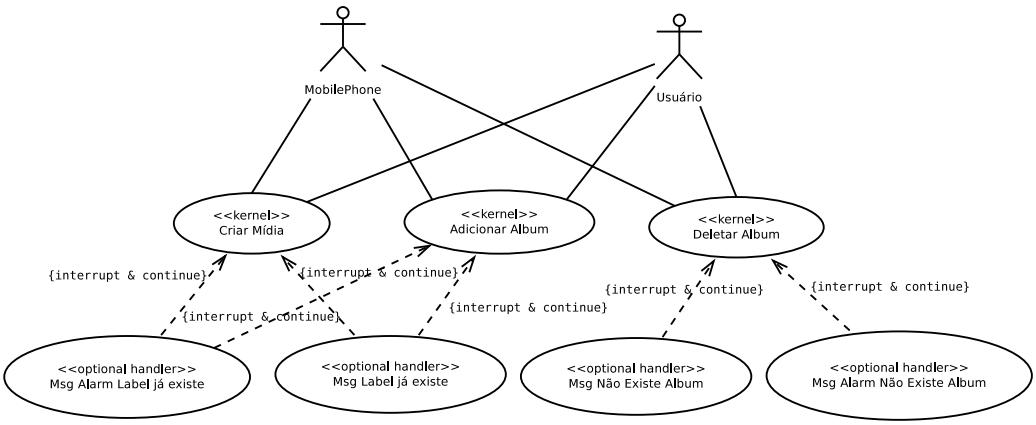


Figura 4.12: Modelo de Casos de Uso do MobileMedia.

- Atividade III e IV - Nestas atividades é criado o modelo de característica com a variabilidade excepcional no modelo, como explicado nas Seções 3.3 e 3.4. A Figura 4.13 mostra o modelo de característica depois das evoluções que foram aplicadas no estudo empírico.

  A Figura 4.13 representa o digrama de características do comportamento normal do MobileMedia, que por uma limitação de espaço da página ele não pode representar o comportamento normal e excepcional completo, assim a Figura 4.14 representa parcialmente duas das características excepcionais do MobileMedia.

- Atividade V - Associação do modelo de Características e do de Casos de uso. A

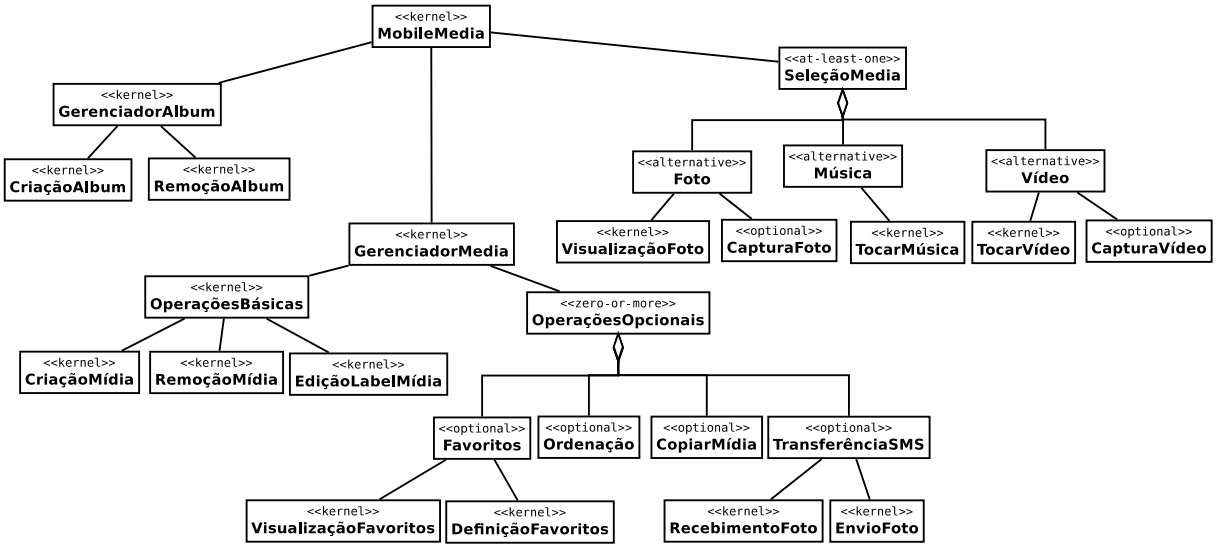


Figura 4.13: Modelo de Características do MobileMedia depois das evoluções.

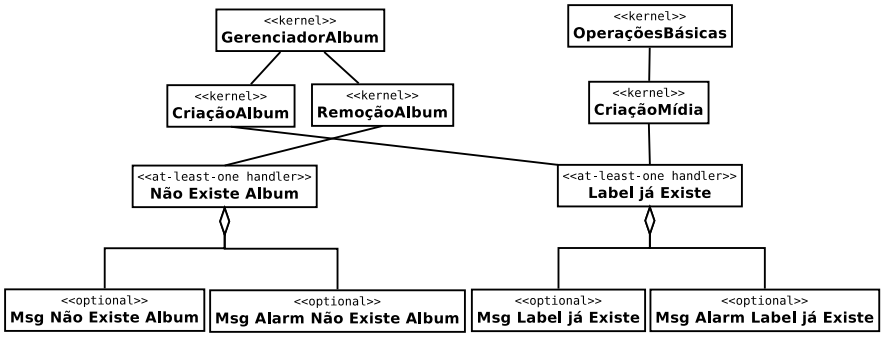


Figura 4.14: Modelo de Características do MobileMedia parcial com características excepcionais.

Figura 4.15 demonstra um trecho da associação realizada entre o modelo de característica e o de caso de uso.

- Atividade VI - Construção o Modelo Conceitual da LPS. A Figura 4.16 mostra o diagrama de classes da LPS MobileMedia.
- Projeto da ALP - Atividades VII, VIII e IX. A Figura 4.17 mostra um trecho da arquitetura da LPS do MobileMedia.

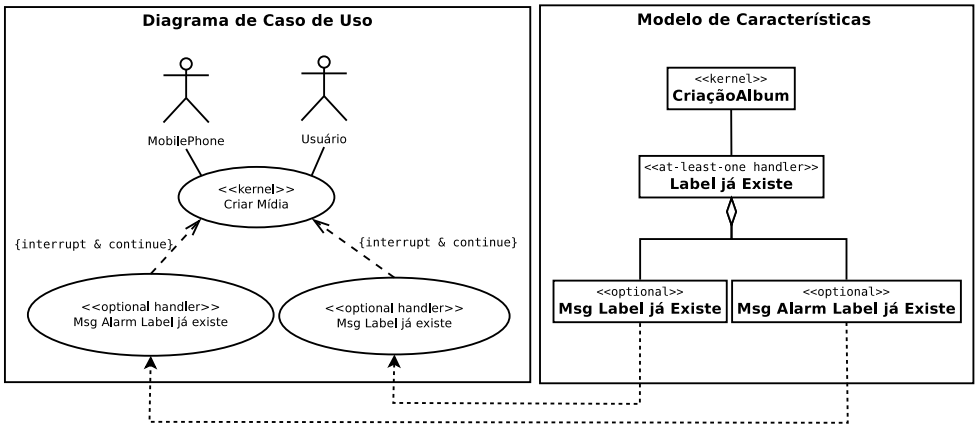


Figura 4.15: Associação do Modelo de Características e do Modelo de Casos de Uso do MobileMedia.

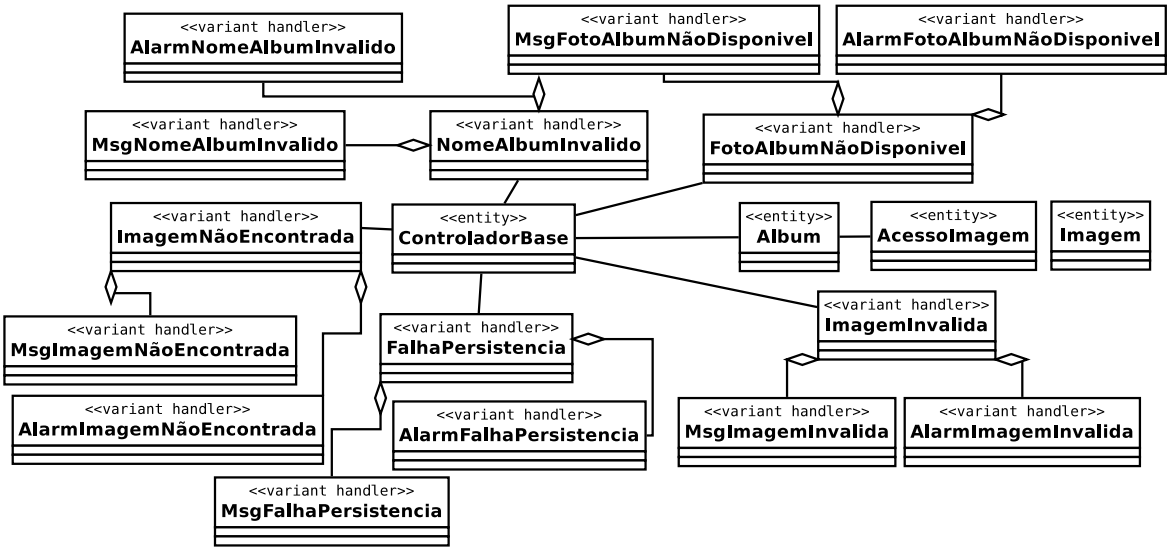


Figura 4.16: Diagrama de classes do MobileMedia.

- Atividade X - Uma ALP foi implementada utilizando Orientação a Objetos (`OO-MM`), outra ALP foi implementada com Aspectos (`AO-MM`), as outras duas ALPs foram implementadas com COSMOS* e COSMOS*-VP (`COSMOS*-MM` e `VP-MM`, respectivamente), sendo que cada uma dessas ALPs possuem oito versões cada uma.

- Atividade XI - Durante essa atividade de validação foi utilizado o JUnit, para verificar e validar os comportamentos excepcionais do sistema.

Após a utilização do método proposto, foi necessário obter as métricas de impacto de mudanças, acoplamento e difusão de interesses sobre os módulos das oito versões das

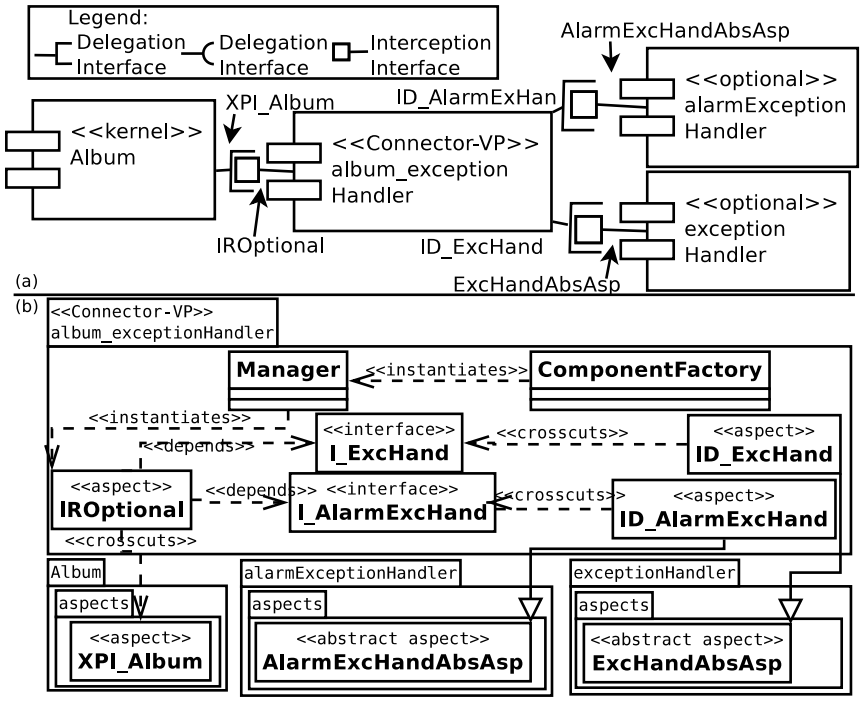


Figura 4.17: Trecho da Arquitetura do MobileMedia - Tratamento Excepcional do componente `Album`.

quatro ALPs. O estudo comparativo dos resultados obtidos das quatro ALPs realizado é apresentado a seguir.

4.4 MobileMedia: Resultados Obtidos

4.4.1 Impacto de Mudanças nos Módulos

O impacto de mudanças sobre os elementos da ALP é medido através do número de componentes e conectores (módulos) afetados, ou seja, adicionados, modificados e removidos, em cada evolução da ALP. Quanto maior o número de módulos afetados, maior será o impacto sofrido pela ALP. Assim, nesta seção será discutido o impacto de mudanças das ALPs do MobileMedia sofridos durante as evoluções propostas.

De acordo com a Tabela 4.5 e a Figura 4.18, verifica-se que durante as evoluções ocorridas da versão V0 para V1 e da V1 para V2, o número de módulos adicionados e modificados pela ALP `VP-MM` foi maior do que as outras. Isso ocorreu pois foram adicionadas características mandatórias - neste caso o tratamento excepcional da ALP - o que afetou todo a ALP, aumentando o número de módulos, tanto adicionados quanto os modificados.

Tabela 4.5: Número de Elementos Arquiteturais Afetados em cada Versão pelo Impacto de Mudanças.

		ALPs			
Versão	**Elementos**	**OO-MM**	**AO-MM**	**COSMOS*-MM**	**VP-MM**
V1	Adicionados	9	17	57	39
	Modificados	5	5	27	24
	Removidos	0	1	48	33
V2	Adicionados	2	3	9	31
	Modificados	7	11	25	16
	Removidos	1	1	1	0
V3	Adicionados	0	3	0	21
	Modificados	5	3	10	7
	Removidos	0	0	0	0
V4	Adicionados	5	7	89	91
	Modificados	8	12	36	25
	Removidos	0	0	58	12
V5	Adicionados	8	11	45	47
	Modificados	7	3	23	7
	Removidos	0	1	17	0
V6	Adicionados	17	33	123	60
	Modificados	10	18	57	16
	Removidos	7	12	57	0
V7	Adicionados	6	37	53	62
	Modificados	26	21	71	43
	Removidos	3	8	6	6

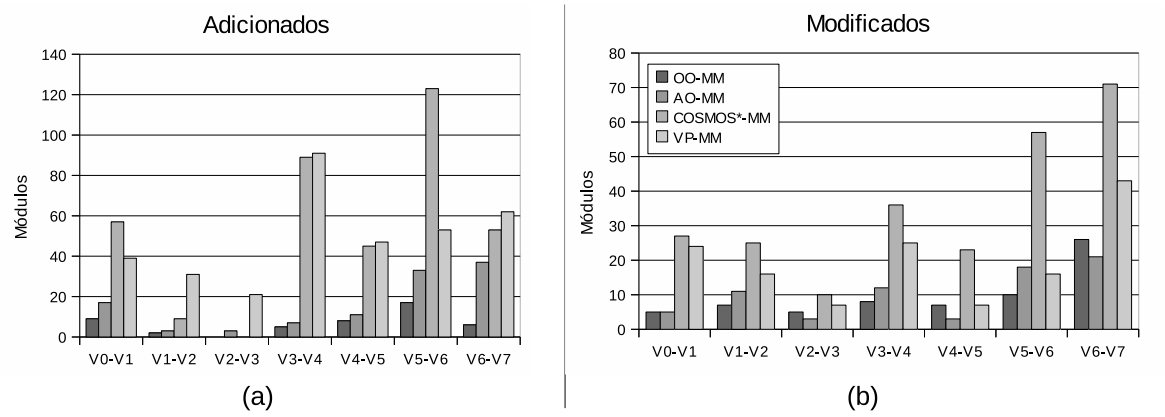


Figura 4.18: Gráfico dos Impactos de Mudanças do MobileMedia.

Nas evoluções das versões V3, V4 e V5 são adicionadas as características opcionais nas ALPs. Durante essas evoluções, percebemos que o número de adições de módulos realizadas foi maior na ALP VP-MM, pois além dos módulos inclusos o tratamento excepcional com variabilidade foi incluído. Já o número de modificações e remoções dos módulos a VP-MM obteve o menor resultado, pois as alterações necessárias não estavam espalhadas por todos os módulos do sistema.

A evolução das versões V5 para V6 e V6 para V7, incluíram características alternativas. A Tabela 4.5 e a Figura 4.18 mostram que o número de módulos modificados e adicionados na ALP VP-MM foi menor do que as outras implementações. Isso ocorreu pois a utilização do COSMOS*-VP facilitou a evolução da ALP, sendo que a implementação dos componentes foi isolada da implementação dos pontos de variação.

Com a Tabela 4.5, podemos perceber que na maioria das versões (excluindo as versões V1 e V6) o número de elementos adicionados aumenta consideravelmente entre as LPS implementadas (OO-MM, AO-MM, COSMOS*-MM e VP-MM). As maiores diferenças na adição de elementos acontecem nas LPS COSMOS*-MM e VP-MM em que o número é maior do que as outras implementações (OO-MM, AO-MM). Porém, com relação ao número de elementos modificados, se comparados somente entre COSMOS*-MM e VP-MM, a utilização do MVTE permitiu que a quantidade de elementos tivesse uma redução; mas, se compararmos com as implementações em OO-MM e AO-MM, o número de elementos modificados no MVTE (VP-MM) é maior. Assim, conclui-se que não é em todos os casos que o impacto de mudanças é satisfatoriamente melhor do que o apresentado em outras implementações.

4.4.2 Acoplamento entre Módulos

O acoplamento entre módulos refere-se ao nível de interdependência entre partes de um sistema [13]. A estabilidade dessa métrica está relacionada com a estabilidade arquitetural, isto é, uma evolução que altere os elementos arquiteturais da ALP, também causam grandes alterações nos valores da métrica (valores bons ou ruins).

A Figura 4.19 e a Tabela 4.6 apresentam os resultados para o acoplamento entre os módulos das quatro ALPs (OO-MM, AO-MM, COSMOS*-MM e VP-MM). Nelas pode-se observar que a utilização do COSMOS*-VP para a variabilidade do tratamento excepcional teve um grau de interdependência menor entre seus módulos do que as outras ALPs implementadas. As ALPs OO-MM e a AO-MM obtiveram os maiores graus de interdependência, pois os módulos existentes estão interligados fazendo com que eles sejam dependentes entre si. Já com a ALP VP-MM houve um menor grau de interdependência, pois os componentes aspectuais da ALP não dependem dos componentes que serão interceptados.

Tabela 4.6: Acoplamento Médio entre Módulos para cada versão da ALPs.

Versão	OO-MM	AO-MM	COSMOS*-MM	VP-MM
V0	0,867	1	0,867	0,867
V1	1,625	2,677	1,348	1,102
V2	1,76	2,939	1,449	1,257
V3	1,76	3,083	1,427	1,277
V4	2,43	3,79	1,369	1,183
V5	2,64	3,81	1,51	1,198
V6	2,956	4,071	1,686	1,254
V7	3,36	4,134	1,778	1,575

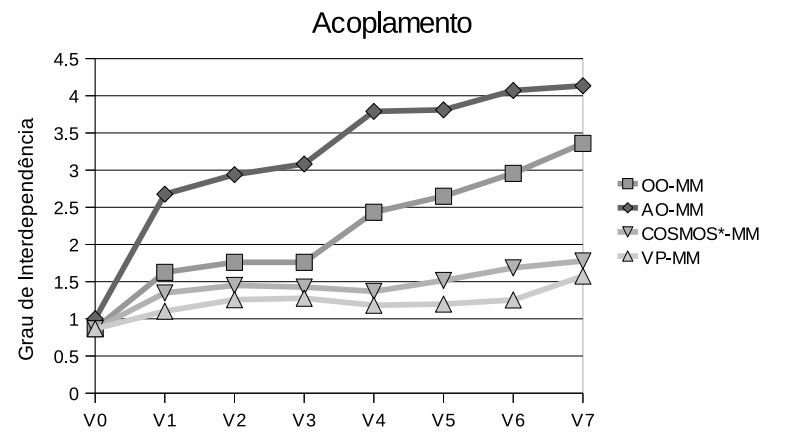


Figura 4.19: Gráfico de Acoplamento entre Módulos do MobileMedia.

4.4.3 Difusão de Interesses sobre Módulos

A difusão de um interesse é dada pelo número de módulos que implementam esse interesse sobre o número total de módulos da arquitetura de LPS. Logo, quanto mais difuso um interesse, menos modularizada é a arquitetura de LPS. Para que um módulo fosse incluído na lista de módulos que implementam um interesse, bastava ter uma única linha de código que fosse relacionada a esse interesse. O interesse medido nessa métrica para este estudo empírico foi o tratamento excepcional e variável nas ALPs. Deste modo, é possível verificar o quão espalhado está o tratamento excepcional na ALP.

A Tabela 4.7 e a Figura 4.20 mostram os resultados obtidos durante a métrica de difusão de interesses. Deste modo, é possível verificar que as versões da ALP implementadas com o COSMOS*-VP (`VP-MM`) tiveram uma porcentagem menor do que as outras implementada, pois a implementação do tratamento excepcional com o COSMOS*-VP é totalmente separada do comportamento normal da ALP, facilitando as alterações que

Tabela 4.7: Difusão de Interesses sobre Módulos para cada versão das ALPs do MobileMedia.

Versão (%)	OO-MM	AO-MM	COSMOS*-MM	VP-MM
V0	20	20	19	19
V1	62,5	74,19	41,09	37,8
V2	64	72,73	38,41	36,48
V3	60	69,44	37,68	37,22
V4	60	65,12	44,64	34,48
V5	59,46	62,26	34,59	32,25
V6	58,7	57,14	37,8	32,24
V7	66	49,48	41,98	40,09

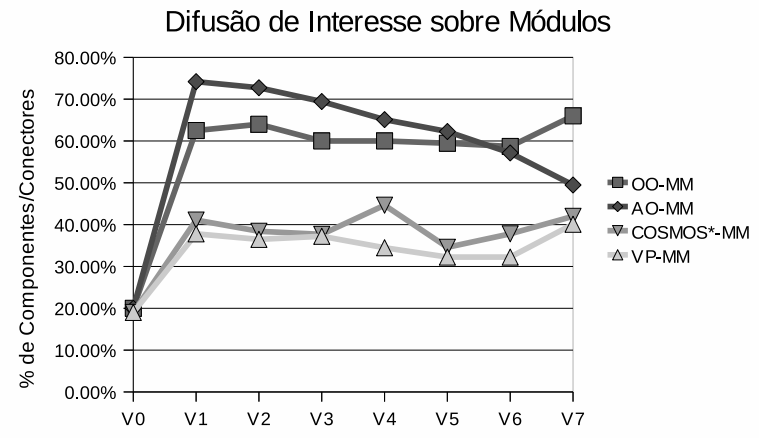


Figura 4.20: Gráfico de Difusão de Interesses sobre Módulos do MobileMedia.

podem ou poderão ser realizadas.

4.5 Resumo

Neste capítulo foram apresentados dois estudos empíricos que avaliam a aplicabilidade do método proposto MVTE. O objetivo dos estudos empíricos foi analisar qualitativamente e quantitativamente a estabilidade de ALPs especificadas e implementadas utilizando o método MVTE. Em um dos estudos empíricos, MVTE foi utilizado para ALP da linha de aplicações de governo eletrônico (E-Gov) para a automatização do processo de obtenção da Carteira Nacional de Habilitação (CNH). Nesse estudo, COSMOS*-VP foi comparado com uma abordagem combinada do modelo COSMOS* original e aspectos. No segundo estudo empírico, o método proposto foi utilizado na ALP da linha de aplicações de manipulação

de mídias para dispositivos móveis, chamada MobileMedia. A eficácia do MVTE nas ALPs componentizadas foi avaliada aplicando às ALPs diversos cenários de evolução reais. Em cada um dos cenários de evolução, métricas de impacto de mudanças, acoplamento entre módulos e difusão de interesses foram coletadas para cada uma das versões das ALPs envolvidas nos estudos.

Resumindo os resultados obtidos durante o primeiro estudo empírico (E-GoV-CNH), a ALP criada e evoluída utilizando a solução proposta, MVTE, sofreu um impacto arquitetural menor em seus elementos em relação a ALP criada e evoluída utilizando COSMOS* e aspectos. Além disso, as 3 versões da ALP implementada utilizando MVTE obtiveram um menor acoplamento entre seus elementos arquiteturais e, também, a difusão de interesses foi menor do que à apresentada na outra implementação.. Um ponto negativo seria que a diferença entre ambas as implementações, COSMOS*-VP e COSMOS não foi muito grande. A implementação com COSMOS*-VP foi, ligeiramente, melhor do que a com COSMOS*.

Com relação aos resultados obtidos durante o segundo estudo empírico, a ALP criada e evoluída utilizando MVTE durante o impacto de mudanças mostrou que este impacto varia tanto para melhor quanto para pior, dependendo do cenário que será evoluído. Já quanto o acoplamento entre módulos e a difusão de interesses, o método proposto `VP-MM` mostrou uma pequena melhora em relação às outras implementações (**OO-MM**,**AO-MM**,**COSMOS*-MM**).

Capítulo 5

Conclusões e Trabalhos Futuros

Este trabalho apresentou o MVTE, um método que irá auxiliar na construção da arquitetura baseada em componentes da LPS, que possui variabilidade no tratamento de exceções. O método auxilia a separação do comportamento normal do comportamento excepcional da LPS durante o desenvolvimento do sistema, facilitando futuras evoluções de ambos os comportamentos.

O método proposto facilita a visualização do comportamento excepcional em todo o desenvolvimento da LPS. O comportamento normal e o comportamento excepcional são visualizados totalmente separados nos diagramas de caso de uso, diagramas de características, diagramas de classes e a arquitetura da linha de produtos.

Com os resultados obtidos (Seções 4.2 e 4.4) é possível perceber que com o uso do MVTE houve uma melhora no impacto de mudanças, acoplamento e no espalhamento do código excepcional por toda ALP. Quanto ao impacto de mudanças, com as evoluções propostas foi possível verificar, que dependendo da evolução realizada, houve uma pequena melhora nos impactos causados por ela. Com o acoplamento pudemos perceber que, com o uso do método proposto, os módulos se tornaram mais interdependentes do que com as outras versões implementadas. Por fim o espalhamento do código medido pela difusão de características mostrou que o comportamento excepcional está concentrado, somente nos módulos que o trataram, facilitando uma visualização separada do que é o comportamento normal e o excepcional que existe na ALP.

5.1 Contribuições

As contribuições deste trabalho são:

- A principal contribuição deste trabalho foi o método proposto (MVTE), que auxilia a inclusão da variabilidade do tratamento excepcional em uma ALP, facilitando a

evolução da LPS.

- A separação do comportamento normal e excepcional da LPS durante todo o desenvolvimento da LPS, desde os casos de uso até a manutenção da LPS. Com esta separação é possível evoluir tanto o comportamento normal quanto o excepcional sem interferir em todo a LPS;
- A utilização do modelo COSMOS*-VP na variabilidade do comportamento excepcional. Anteriormente, o uso do COSMOS*-VP [19] era apenas utilizado junto ao comportamento normal;
- Validação prática do método proposto, através de dois estudos empíricos de LPS. Nos estudos empíricos as vantagens do método proposto foram comparadas com a utilização de outras maneiras para obtermos a variabilidade do comportamento excepcional.

5.2 Trabalhos Futuros

Com a conclusão deste trabalho, foram identificados alguns direcionamentos para pesquisas futuras. Esses direcionamentos visam principalmente o aperfeiçoamento do método proposto.

As sugestões para trabalhos futuros são:

- **MVTE para sistemas críticos**. Todo sistema crítico tem a necessidade de tratar seus erros com mais urgência. O método MVTE não preve o uso de variabilidade de tratamento de exceções em sistemas redundantes nas LPS.
- **Suporte Ferramental**. A construção de uma ferramenta CASE, que seja orientada ao método proposto. Assim, essa ferramenta deve auxiliar na construção de todos os diagramas necessários para o método.
- **Verificação e Validação de Testes**. O método proposto engloba a atividade de verificação e teste que não foi devidamente utilizada; foram realizados testes unitários para a validação da variação do tratamento excepcional. Neste caso, seria necessário um método para verificar e validar essa variação do tratamento excepcional com maior rigor.

5.3 Publicação e Apresentação

Durante o andamento deste trabalho, foram produzidas: uma publicação em um workshop internacional e uma apresentação de pôster em um outro workshop internacional.

Publicação:

- **Supporting the evolution of Exception Handling in Component-based Product Line Architecture, 5th International Workshop on Exception Handling, ICSE 2012 - 34th International Conference on Software Engineering**. (*Bruno de A. Iizuka, Amanda S. Nascimento, Leonardo P. Tizzei e Cecília M. F. Rubira*): Esta publicação apresenta o uso do COSMOS*-VP para variar o tratamento de exceções presentes em LPS baseada em componentes. Apresenta, também, um estudo empírico (E-Gov-CNH) para exemplificar e mostrar os benefícios do uso.

Apresentação:

- **Variability of Exception Handling in Software Product Lines, First Workshop on Exception Handling in Contemporary Software Systems, LADC 2011 - IV Latin-American Symposium on Dependable Computing**. (*Bruno de A. Iizuka, Amanda S. Nascimento e Cecília M. F. Rubira*): Este trabalho apresentou uma versão não final do método proposto apresentado no Capítulo 3.

Referências Bibliográficas

[1] *Dependable Computing and Fault Tolerance : Concepts and Terminology*, August 2002.

[2] *UML 2 Bible*. Wiley India Pvt. Limited, 2003.

[3] Eclipse ide, September 2012.

[4] Eiji Adachi, Thaís Batista, Uirá Kulesza, Ana Luisa Medeiros, Christina Chavez, and Alessandro Garcia. Variability management in aspect-oriented architecture description languages: An integrated approach. In *Proceedings of the 2009 XXIII Brazilian Symposium on Software Engineering*, SBES '09, pages 1–11, Washington, DC, USA, 2009. IEEE Computer Society.

[5] Eiji Adachi Barbosa, Alessandro Garcia, and Mira Mezini. Heuristic strategies for recommendation of exception handling code. In *SBES*, pages 171–180. IEEE Computer Society, 2012.

[6] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice, Second Edition*. Addison-Wesley Professional, Abril 2003.

[7] Kathrin Berg, Judith Bishop, and Dirk Muthig. Tracing software product line variability: from problem to solution space. In *Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*, SAICSIT '05, pages 182–191, Republic of South Africa, 2005. South African Institute for Computer Scientists and Information Technologists.

[8] Ivo Augusto Bertoncello, Marcelo Oliveira Dias, Patrick H. S. Brito, and Cecília M. F. Rubira. Explicit exception handling variability in component-based product line architectures. In *Proc. of the 4th int. works. on Exception handling*, pages 47–54. ACM, 2008.

[9] Patrick H. S. Brito. Um método para modelagem de exceções em desenvolvimento baseado em componentes. Master's thesis, IC - Unicamp, October 2005.

[10] Luiz Eduardo Buzato, Cecília M. F. Rubira, and Maria Lúcia Blanck Lisbôa. A reflective object-oriented architecture for developing fault-tolerant software. *J. Braz. Comp. Soc.*, 4(2), 1997.

[11] John Cheesman and John Daniels. *UML Components: A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2000.

[12] Feng Chen, Qianxiang Wang, Hong Mei, and Fuqing Yang. An architecture-based approach for component-oriented development. In *COMPSAC*, pages 450–458. IEEE Computer Society, 2002.

[13] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20:476–493, 1994.

[14] A. Childs, J. Greenwald, G. Jung, M. Hoosier, and J. Hatcliff. CALM and Cadena: metamodeling for component-based product-line development. *Computer*, 39(2):42–50, 2006.

[15] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 3rd edition, 2001.

[16] Flaviu Cristian. Exception handling. In *Dependability of Resilient Computers*, pages 68–97, 1989.

[17] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[18] Moacir C. da Silva Jr, Paulo A. de C. Guerra, and Cecília M. F. Rubira. A java component model for evolving software systems. *Automated Software Engineering, International Conference on*, 0:327, 2003.

[19] Marcelo Oliveira Dias, Leonardo P. Tizzei, Cecília M. F. Rubira, Alessandro F. Garcia, and Jaejoon Lee. Leveraging aspect-connectors to improve stability of product-line variabilities. In *VaMoS, 4th Int. Works. on Variability Modelling of Software-Intensive Systems*, pages 21–28, 2010.

[20] Gisele R. M. Ferreira. *Tratamento de exceções no desenvolvimento de sistemas confiáveis baseados em componentes*. IC, Unicamp, Brazil, December 2001.

[21] Eduardo Figueiredo, Nelio Cacho, Mario Monteiro, Uira Kulesza, Ro Garcia, Sergio Soares, Fabiano Ferrari, Safoora Khan, O Filho, and Francisco Dantas. Evolving software product lines with aspects: An empirical study on design stability. In *proc. of the ICSE*, 2008.

[22] Fernando Castor Filho, Nélio Cacho, Eduardo Figueiredo, Raquel Maranhão, Alessandro Garcia, and Cecília M. F. Rubira. Exceptions and aspects: the devil is in the details. In Michal Young and Premkumar T. Devanbu, editors, *SIGSOFT FSE*, pages 152–162. ACM, 2006.

[23] Critina Gacek and Michalis Anastasopoules. Implementing product line variabilities. *SIGSOFT Softw. Eng. Notes*, 26:109–117, May 2001.

[24] Leonel Aguilar Gayard, Cecília Mary Fischer Rubira, and Paulo Astério de Castro Guerra. COSMOS*: a COmponent System MOdel for Software Architectures. Technical Report IC-08-04, Institute of Computing, University of Campinas, 2008.

[25] Hassan Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[26] John B. Goodenough. Exception handling: issues and a proposed notation. *Commun. ACM*, 18(12):683–696, December 1975.

[27] William G. Griswold, Kevin Sullivan, Yuanyuan Song, Macneil Shonle, Nishit Tewari, Yuanfang Cai, and Hridesh Rajan. Modular Software Design with Crosscutting Interfaces. *IEEE Software*, 23(1):51–60, 2006.

[28] V. Issarny and J. Banatre. Architecture-based exception handling. *Hawaii International Conference on System Sciences*, 9:9058, 2001.

[29] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software reuse: architecture, process and organization for business success.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1997.

[30] Ingrid Jansch-Pôrto. Fundamentos de tolerância a falhas. *RITA*, 1(3):63–99, 1989.

[31] Jilles Van Gurp, Jan Bosch, and Mikael Svahnberg. On the notion of variability in software product lines. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, WICSA '01, pages 45–, Washington, DC, USA, 2001. IEEE Computer Society.

[32] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. 1990.

[33] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Gerard Jounghyun Kim, and Euiseob Shin. Form: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5:143–168, 1998.

[34] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*. SpringerVerlag, 1997.

[35] Ramnivas Laddad. *AspectJ in Action: Enterprise AOP with Spring Applications*. Manning Publications Co., Greenwich, CT, USA, 2nd edition, 2009.

[36] P. A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2nd edition, 1990.

[37] M. M. Lehman, J. F. Ramil, and D. E. Perry. On evidence supporting the feast hypothesis and the laws of software evolution. In *Proceedings of the 5th International Symposium on Software Metrics*, METRICS '98, pages 84–, Washington, DC, USA, 1998. IEEE Computer Society.

[38] Doug Mcilroy. Mass-produced software components. In J. M. Buxton, P. Naur, and B. Randell, editors, *Proceedings of Software Engineering Concepts and Techniques*, pages 138–155. NATO Science Committee, January 1969.

[39] D. Muthig, I. John, M. Anastasopoulos, T. Forster, J. Doerr, and K. Schmid. Gophone - a software product line in the mobile phone domain. 2004.

[40] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software product line engineering - foundations, principles, and techniques*. Springer, 2005.

[41] Roger Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, Inc., New York, NY, USA, 7 edition, 2010.

[42] Claudio Sant'anna, Alessandro Garcia, Christina Chavez, Carlos Lucena, and Arndt v. von Staa. On the reuse and maintenance of aspect-oriented software: An assessment framework. In *Proc. Brazilian Symp. on Software Engineering*, 2003.

[43] Aaron Shui, Sadaf Mustafiz, Jörg Kienzle, and Christophe Dony. Exceptional use cases. In *Proceedings of the 8th international conference on Model Driven Engineering Languages and Systems*, MoDELS'05, pages 568–583, Berlin, Heidelberg, 2005. Springer-Verlag.

[44] Ian Sommerville. *Software Engineering (7th Edition)*. Pearson Addison Wesley, 2004.

[45] Mikael Svahnberg and Jan Bosch. Evolution in software product lines: Two cases. *Journal of Software Maintenance*, 11(6):391–422, November 1999.

[46] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[47] Rodrigo Teruo Tomita, Fernando Castor Filho, Paulo A. de Castro Guerra, and Cecília M. F. Rubira. Bellatrix: An environment with arquitectural support for component-based development. In *Proc. of the Braz. Works. on Component-Based Development*, pages 43–48, 2004.

[48] S.S. Yau and J.S. Collofello. Design stability measures for software maintenance. *IEEE Transactions on Software Engineering*, 11:849–856, 1985.

[49] Zhang You-Sheng and He Yu-Yun. Architecture-based sw process model. *SIGSOFT Softw. Eng. Notes*, 28(2):16–, March 2003.

[50] Trevor J. Young and Trevor J. Young. Using aspectj to build a software product line for mobile devices. msc dissertation. In *University of British Columbia, Department of Computer Science*, pages 1–6, 2005.