Luciana Akemi Burgareli

GERENCIAMENTO DE VARIABILIDADE DE LINHA DE PRODUTOS DE SOFTWARE COM UTILIZAÇÃO DE OBJETOS ADAPTÁVEIS E REFLEXÃO

Tese apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do título de Doutor em Engenharia Elétrica

São Paulo

2009

Luciana Akemi Burgareli

GERENCIAMENTO DE VARIABILIDADE DE LINHA DE PRODUTOS DE SOFTWARE COM UTILIZAÇÃO DE OBJETOS ADAPTÁVEIS E REFLEXÃO

Tese apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do título de Doutor em Engenharia Elétrica

Área de Concentração: Sistemas Digitais

Orientadora: Profa. Dra. Selma Shin Shimizu Melnikoff
Co-Orientador: Prof. Dr. Mauricio Gonçalves Vieira Ferreira

São Paulo

2009

Este exemplar foi revisado e alterado em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, 26 de agosto de 2009.

Assinatura do autor _____

Assinatura do orientador _____

FICHA CATALOGRÁFICA

Burgareli, Luciana Akemi
Gerenciamento de variabilidade de linha de produtos de software com utilização de objetos adaptáveis e reflexão / L.A. Burgareli. -- ed.rev. -- São Paulo, 2009.
p. 246

Tese (Doutorado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1. Linhas de produtos de software 2. Processo de software 3. Reúso de software 4. Padrões de software I. Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II. t.

DEDICATÓRIA

À minha mãe querida Tomoco Ohira Burgareli.
Mãe dedicada, zelosa e amiga.
Mulher sábia, honesta e trabalhadora.
Ser admirável, sobretudo, por sua sensatez.

AGRADECIMENTOS

Agradeço a **Deus**.
Pela dádiva da vida.
Por me presentear com a luz do dia, com o perfume das flores, com o canto dos pássaros e com a beleza do mar.
Por me conceder saúde, família, amigos e oportunidades de estudo e trabalho.
Por Sua infinita generosidade e ainda, por me permitir concluir esta tese.

Agradeço aos meus pais **Jurandir Roberto Burgareli** e **Tomoco Ohira Burgareli**.
Por todo carinho e cuidado dedicados à minha criação.
Pelos esforços e sacrifícios realizados e por tudo que renunciaram para me proporcionar sempre o melhor.
Por parte das suas vidas que a mim destinaram e ainda destinam.

Agradeço ao meu marido **Fábio Adoniran Vieira Pimentel**.
Pelo amor, respeito, paciência e companheirismo.
Por compreender minha ausência durante este período de estudos.
Por me consolar nos momentos difíceis.
Por sua alegria e bondade, e por me ensinar a ser uma pessoa melhor.

Agradeço aos meus orientadores **Selma Shin Shimizu Melnikoff** e **Mauricio Gonçalves Vieira Ferreira**.
Pelos inúmeros conhecimentos a mim transmitidos.
Por encorajar-me em todas as etapas deste trabalho.
Pela paciência, amizade e confiança.

Agradeço a todas as pessoas que torceram e contribuíram de forma direta ou indireta para a realização deste trabalho, em especial, à minha querida irmã **Juliana Saemi Burgareli,** às grandes amigas **Rovedy Aparecida Busquim e Silva** e **Martha Adriana Dias Abdala** e aos amigos: **Abel de Lima Nepomuceno, Adilson de Jesus Teixeira, Carlos Henrique Netto Lahoz, Carmen Silva Monteiro Roque, Manuel Martinez Gamallo, Mário Sisido, Nanci Naomi Arai, Pedro Haruo Takahashi e Renato Rosa da Silva**.

E Deus disse:

“Jamais o deixarei, jamais o abandonarei.”

Antigo Testamento

RESUMO

A abordagem de linha de produtos de software oferece benefícios ao desenvolvimento de software como economia, qualidade e desenvolvimento rápido, pois se baseia em reuso de arquitetura de software mais planejado e direcionado a um domínio específico. Neste contexto, o gerenciamento da variabilidade é uma questão chave e desafiadora, já que esta atividade auxilia a identificação, projeto e implementação dos novos produtos derivados da linha de produtos de software.

O objetivo deste trabalho é definir um processo de gerenciamento de variabilidade de linha de produtos de software. Este processo, denominado GVLPS, identifica a variabilidade, extraindo as variantes a partir de diagramas de casos de uso e modelando-as através de *features*, especifica a variabilidade identificada e utiliza como suporte, na criação de variantes, um mecanismo de variabilidade baseado em modelos de objetos adaptáveis e em reflexão. A aplicação do processo é realizada através de um estudo de caso sobre o software de um veículo espacial hipotético, o Lançador de Satélites Brasileiro (LSB).

Palavras Chave: Linha de produtos de software. Variabilidade. Modelos de objetos adaptáveis. Reflexão. Lançador de Satélites Brasileiro.

ABSTRACT

The Software Product Line approach offers benefits such as savings, large-scale productivity and increased product quality to the software development because it is based on software architecture reuse which is more planned and aimed to a specific domain.

The management of variability is a key and challenging issue, since this activity helps identifying, designing and implementing new products derived from software products line.

This work defines a process for the variability management of software product line, called GVLPS. After modeling the variability, extracting the variants from use case diagrams and features, the next step is to specify the variability that was identified. Finally, the proposed process uses a variability mechanism based on adaptive object model and reflection as support in the creation of variants.

The proposed process uses as case study the software system of a hypothetic space vehicle, the Brazilian Satellites Launcher (LSB).

Keywords: Software Product Line. Variability. Adaptive object model. Reflection. Brazilian Satellites Launcher.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ACIS	*Association for Computer and Information Science*
ADL	*Architecture Description Language*
AEB	Agência Espacial Brasileira
AIAA	*American Institute of Aeronautics and Astronautics*
API	*Application Programming Interface*
ASWEC	*Australian Software Engineering Conference*
CASE	*Computer Aided Software Engineering*
CLA	Centro de Lançamento de Alcântara
CORBA	*Common Object Request Broker Architecture*
COTS	C*ommercial Off-The-Shelf*
CTA	Comando Geral de Tecnologia Aeroespacial
CTA	Centro Técnico Aeroespacial
CTD	Controlador Digital
DARTS	*Design Approach for Real-Time Systems*
DFD	Diagrama de Fluxo de Dados
ECBS	*Engineering of Computer Based Systems*
EERC	*Eastern European Regional Conference*
ESPLEP	*Evolutionary Software Product Line Engineering Process*
FAST	*Family-Oriented Abstraction, Specification and Translation*
FMECA	*Failure Mode, Effects and Criticality Analysis*
FODA	*Feature-Oriented Domain Analysis*
FSPLP	*Framework for Software Product Line Practice*
FTA	*Fault Tree Analisys*
GOOD	*Grammar-oriented Object Design*
GVLPS	Gerenciamento de Variabilidade de Linha de Produtos de Software
IAC	*International Astronautical Congress*
IAE	Instituto de Aeronáutica e Espaço
IAE	Instituto de Atividades Espaciais
IAF	*International Astronautical Federation*
IDE	*Integrated Development Environment*
IESE	*Institute for Experimental Software Engineering*

INPE	Instituto Nacional de Pesquisas Espaciais
KobrA	*Komponentenbasierte Anwendungsentwicklung*
LSB	Lançador de Satélites Brasileiro
MDA	*Model Driven Architecture*
MEC	Missão Espacial Completa
MECB	Missão Espacial Completa Brasileira
MSI	Módulo Sensor Inercial
OMG	*Object Management Group*
PABX	*Private Automatic Branch Exchange*
PIM	*Platform Independent Model*
PNAE	Programa Nacional de Atividades Espaciais
PSM	*Platform Specific Model*
PuLSE	*Product Line Software Engineering*
RTTI	*Run-Time Type Identification*
RUP	*Rational Unified Process*
SAE	Seminário de Atividades Espaciais
SEI	*Software Engineering Institute*
SERA	*Software Engineering Research, Management and Applications*
SOAB	Software Aplicativo de Bordo
SpaceOps	*Space Operations*
SVGA	*Super Video Graphics Array*
UML	*Unified Modeling Language*
VLS	Veículo Lançador de Satélites
VLSS	Veículo Lançador de Satélites a Propelente Sólido

SUMÁRIO

"A coragem consiste em
fazer o que você não se atreve."

Eddie Rickenbacker

1 INTRODUÇÃO

Este capítulo apresenta a motivação do trabalho, o seu objetivo, a justificativa através de trabalhos relacionados, a metodologia que norteou o desenvolvimento do trabalho e a estrutura da tese.

1.1 Motivação

A linha de produtos de software é uma das abordagens promissoras de reuso de arquitetura de software. Esta abordagem oferece benefícios ao desenvolvimento de software como economia, qualidade e desenvolvimento rápido e, por isso, está sendo amplamente adotada pela comunidade de engenharia de software.

O objetivo da linha de produtos de software é apoiar o desenvolvimento sistemático de um conjunto de sistemas com software similares, através do entendimento e controle da sua *commonality*, que é a característica que todos os sistemas devem compartilhar, e da variabilidade, que representa a capacidade de um sistema ser alterado e customizado (KIM et al., 2006).

No desenvolvimento da linha de produtos de software destacam-se duas atividades principais: (1) Engenharia de Domínio que engloba análise, projeto e implementação da *commonality* e da variabilidade, que serão utilizadas para gerar os produtos resultantes da linha de produtos de software, e (2) Engenharia de Aplicação, onde os produtos de software são construídos através do reuso dos artefatos construídos durante a Engenharia de Domínio (POHL, BÖCKLE, LINDEN, 2005), (NORTHROP, CLEMENTS, 2007).

Além destas atividades, também é importante considerar o gerenciamento de variabilidade, pois fornece recurso para a manipulação sistemática das partes

variáveis dos sistemas. Ainda, a necessidade pelo desenvolvimento de novos produtos causa a evolução da linha de produtos de software e isso resulta em um aumento da variabilidade, o que reforça a necessidade desta atividade. O gerenciamento da variabilidade da linha de produtos de software tem, portanto, o objetivo de identificar, projetar, implementar e rastrear a flexibilidade na linha de produtos de software (VOELTER, GROHER, 2007).

1.2 Objetivo

O objetivo deste trabalho é definir um processo de gerenciamento de variabilidade de linha de produtos de software, denominado GVLPS, que oferece um mecanismo de variabilidade baseado em modelos de objetos adaptáveis e em reflexão.

As principais atividades deste processo consistem em identificar, especificar e implementar a variabilidade através de um mecanismo que apóie a criação de variantes.

A Identificação da Variabilidade é feita através dos requisitos de sistema, expresso pelo modelo de casos de uso, e através do modelo de *features*, que descreve o domínio em que o sistema está inserido.

A Especificação da Variabilidade é realizada através da determinação de certas restrições à variabilidade, visando facilitar sua implementação por algum mecanismo. Assim, nesta atividade são tomadas decisões sobre sua cardinalidade e sobre o momento de seleção de suas variantes (tempo de resolução ou *binding time*).

A Implementação da Variabilidade é feita através do mecanismo baseado em um conjunto de modelos de objetos adaptáveis, que representam suas características como um metadado, e em reflexão, característica em que um sistema atua em si próprio alterando sua estrutura e comportamento. Este tipo de mecanismo foi adotado porque aumenta a flexibilidade para a variação do produto, já que permite

que o sistema se adapte dinamicamente em tempo de execução aos novos requisitos do usuário.

O estudo de caso é realizado através da aplicação do processo GVLPS ao software de um veículo espacial fictício, denominado Lançador de Satélites Brasileiro (LSB). Pretende-se, com isso, apresentar a análise e a escolha das soluções reutilizáveis de software para habilitar a derivação de software de outros veículos.

Espera-se ainda, que este trabalho sirva de embrião para a criação de uma linha de produtos de software para a família de veículos lançadores de satélite.

1.3 Justificativa

O gerenciamento de variabilidade de linha de produtos de software é um processo que apóia a Engenharia de Domínio e a Engenharia de Aplicação. Através dele, a variabilidade pode ser identificada e projetada durante a Engenharia de Domínio; o seu mecanismo de variabilidade fornece apoio para gerar os diferentes sistemas de software na fase da Engenharia da Aplicação. Por isso, o gerenciamento da variabilidade é considerado uma das questões mais importantes para o sucesso do desenvolvimento da linha de produtos de software, tanto para a Engenharia de Domínio quanto para a Engenharia de Aplicação.

Existem várias abordagens para a implementação de linha de produtos de software (NORTHROP, CLEMENTS, 2007), (GOMAA, 2005), (SPC, 1993), (BAYER et al., 1999), (MATINLASSI, 2004), (HARSU, 2002), (ATKINSON, BAYER, MUTHIG, 2000); entretanto, o gerenciamento de variabilidade ainda é considerado um grande desafio, porque não é uma tarefa trivial, já que muitos fatores influenciam em como as decisões serão tomadas para realizar as suas atividades (KIM et al., 2006), (GOMAA, SHIN, 2007).

Poucos trabalhos relacionados ao gerenciamento de variabilidade de linha de produtos de software foram encontrados na literatura. Alguns dos mais relevantes são apresentados a seguir.

Kim et al. (2006) apresentam um processo para gerenciamento de variabilidade de linha de produtos de software dividido em: (1) análise de escopo - para identificar requisitos; (2) análise de domínio - para especificar o domínio através de casos de uso e modelos de decisão que distinguem entre *commonality* e variabilidade; e (3) arquitetura da linha de produtos de software - para construção dos artefatos. Este processo é totalmente voltado para as atividades de Engenharia de Domínio e não faz menção aos mecanismos de variabilidade.

Van Gurp, Bosch e Svahnberg (2001) dividem seu processo para gerenciamento de variabilidade em: (1) identificação da variabilidade, (2) atribuição de restrições à variabilidade, (3) implementação da variabilidade e (4) gerenciamento de variantes. Este processo abrange tanto as atividades de Engenharia de Domínio, quanto as de Engenharia de Aplicação, entretanto, não define ou implementa qualquer mecanismo de variabilidade específico para seu processo.

Oliveira Junior (2005) se baseou no trabalho de Van Gurp, Bosch e Svahnberg (2001) para criar seu processo de gerenciamento de variabilidade e definiu as sub-atividades das atividades do processo apresentado, elaborando um modelo de rastreamento para identificar a variabilidade e delimitando-a, através de definição da multiplicidade e dos tempos de resolução dos pontos de variação. O autor se limita a citar opções de mecanismos de variabilidade para futuras implementações.

A escolha adequada e a automatização do mecanismo de variabilidade constituem os pontos principais da abordagem de linha de produtos de software e são questões que ainda merecem investigação apropriada na área acadêmica (THAO, MUNSON, NGUYEN, 2008), (GOMAA, SHIN, 2007), (OLIVEIRA et al., 2005).

Desta forma, a complexidade destas atividades aliadas à pouca existência de publicações com estudos direcionados (SVAHNBERG, BOSCH, 2000), (OLIVEIRA JUNIOR, 2005), faz com que exista uma necessidade de se conduzir mais pesquisas na área de gerenciamento e mecanismos de variabilidade (NORTHROP, CLEMENTS, 2007).

Ainda, no contexto das atividades profissionais da autora, a linha de produtos de software mostrou-se uma tecnologia promissora para, eventualmente, organizar o

conhecimento relacionado com o software de bordo dos veículos lançadores de satélites.

O primeiro Veículo Lançador de Satélites (VLS) foi desenvolvido pelo Instituto de Aeronáutica e Espaço (IAE), e tinha como objetivo colocar em órbitas de baixa altitude, entre 200 e 1000Km, satélites de Coleta de Dados assim como, de Sensoriamento Remoto, lançados a partir do Centro de Lançamento de Alcântara (CLA) (REIS FILHO, 1995).

Em quase 15 anos de desenvolvimento do software do VLS, a equipe de desenvolvimento tem passado por grande rotatividade de pessoal qualificado. Infelizmente, poucos desenvolvedores se mantiveram no projeto, desde a sua concepção. Desta forma, é necessário manter o conhecimento do projeto através de documentação clara e modelos atualizados do lançador, para que não haja dificuldades em transferir as informações do projeto aos novos integrantes da equipe, a fim de que os mesmos possam entender os artefatos gerados pelos desenvolvedores originais.

Além disso, este tipo de projeto costuma ser de longa duração e problemas surgem devido à obsolescência das tecnologias utilizadas, como metodologias e ferramentas CASE (*Computer Aided Software Engineering*). O software do VLS foi originalmente projetado utilizando as tecnologias do início dos anos de 1990, como análise estruturada, utilizando *TeamWork*, uma ferramenta de modelagem gráfica originalmente desenvolvida pela Cadre. Atualmente, se torna cada vez mais difícil manter uma infra-estrutura para estes ambientes, dificultando assim a captura, edição e atualização dos modelos e documentos por eles gerados. Assim, surge a necessidade de uma nova abordagem de desenvolvimento, utilizando-se tecnologias mais recentes, melhorando o entendimento e auxiliando a utilização e a atualização dos modelos pela equipe.

A manutenção do conhecimento sobre o assunto se torna importante, pois o Comando Geral de Tecnologia Aeroespacial - CTA e a Agência Espacial Brasileira - AEB (AEB, 2006) anunciaram em 2005, o programa Cruzeiro do Sul para o desenvolvimento de cinco novos lançadores - Alfa, Beta, Gama, Delta e Epsilon (MORAIS, 2005). Este programa prevê uma evolução gradativa dos lançadores para alcance de melhores desempenhos e de maiores capacidades para o transporte de

carga útil. A perspectiva de novos projetos, como o Cruzeiro do Sul, também leva à necessidade de desenvolvimento mais planejado de software, com elaboração de modelos e artefatos de software que possam ser reusados pelos novos veículos lançadores de satélites.

Atualização de modelos e documentação, substituição de tecnologias de desenvolvimento antigas e perspectivas de novos projetos justificam a pesquisa de novas abordagens para o desenvolvimento de software para o VLS que privilegiem reuso e assegurem qualidade e economia para o desenvolvimento de novas versões de lançadores de satélites.

Na estratégia atual, para acompanhar a evolução do VLS é necessário desenvolver ou adaptar modelos e artefatos de software específico para cada versão do lançador, o que é uma atividade custosa.

Assim, percebe-se a necessidade de um processo de desenvolvimento de software que permita aumentar o reuso do software do VLS, evitando esforços repetitivos para desenvolver software para veículos na mesma família.

Espera-se que o processo de variabilidade de linha de produtos de software - GVLPS possa beneficiar o desenvolvimento do software do VLS, aumentando a flexibilidade de alteração e reuso dos artefatos e código.

1.4 Metodologia

Este trabalho foi desenvolvido realizando-se as seguintes atividades:

1. Pesquisa sobre reuso de software;
2. Estudo sobre a abordagem de linha de produtos de software e gerenciamento da variabilidade;
3. Estudo sobre modelos de objetos adaptáveis e reflexão;

4. Estudo detalhado do software de bordo do VLS, através da documentação e código;

5. Modelagem do software do LSB através da *Unified Modeling Language* (UML);

6. Definição do processo GVLPS através de suas atividades e seus artefatos;

7. Definição e construção do protótipo do mecanismo de variabilidade baseado em modelos de objetos adaptáveis e de reflexão;

8. Aplicação do processo GVLPS ao software do LSB.

Inicialmente, foi realizada uma pesquisa sobre reuso de software, quando se estudou principalmente a abordagem *Model Driven Architecture* (MDA), criada pela *Object Management Group* (OMG). Nesta abordagem, o desenvolvimento de sistema de software se baseia na separação dos modelos independentes de plataforma (*Platform Independent Model* - PIM) dos modelos com detalhes específicos de plataforma (*Platform Specific Model* - PSM) (OMG, 2009). É uma técnica que, além de visar a reutilização, valoriza os modelos, colocando-os como pontos chave no desenvolvimento do software. A abordagem de linha de produtos de software foi selecionada para este trabalho, porque apresenta uma abrangência maior, de forma que um sistema de software especificado com MDA pode ser considerado um caso particular de linha de produtos de software.

Foi realizado um estudo através de uma pesquisa bibliográfica na área de linha de produtos de software e gerenciamento de variabilidade. Esta pesquisa abrangeu trabalhos sobre processos relacionados, ferramentas utilizadas e aplicação de linha de produtos de software em sistemas aeroespaciais (GIMENES, TRAVASSOS, 2002), (HABLI, KELLY, HOPKINS, 2007), (KEEPENCE, MANNION, 1999), (CARDOSO et al., 2008) e (McCOMAS et al., 2000).

A fase seguinte de estudo abrangeu trabalhos sobre modelos de objetos adaptáveis, abordando os padrões utilizados, e sobre as técnicas reflexivas. Alguns exemplos

também foram implementados para melhor entendimento da técnica e familiarização com mecanismos reflexivos apresentados pela linguagem de programação selecionada para o desenvolvimento do protótipo.

Uma pesquisa bibliográfica foi realizada buscando trabalhos acadêmicos direcionados ao software de bordo do VLS. Através deles, foi possível analisar o software do VLS sob vários aspectos, como por exemplo, sob a aplicação do paradigma orientado a objetos (ANDRANDE, 2000), e sob abordagens de qualidade (LEMES, 1997) e confiabilidade (REIS FILHO, 1995) sugeridas ao software.

Além disso, foram obtidas informações do software do VLS através da sua documentação, da análise do seu código e de uma série de entrevistas realizadas com os especialistas de veículos espaciais do IAE. A partir destas informações foi possível construir os modelos estáticos e dinâmicos do LSB utilizando a UML. A confecção destes modelos consolidou o entendimento do software do veículo.

Em seguida foi definido o processo GVLPS através das atividades e dos artefatos. Paralelamente a esta definição, foi implementado um protótipo do mecanismo de variabilidade onde foram utilizadas técnicas de modelos de objetos adaptáveis e reflexivas.

Finalmente, o processo GVLPS foi aplicado a um estudo de caso. Como o projeto do VLS é sigiloso, foi definido um veículo espacial fictício, o LSB, cujo software foi utilizado para este estudo. Com isso, foi possível avaliar as características do processo e sugerir novos trabalhos.

1.5 Organização do Trabalho

Os demais capítulos deste trabalho estão sucintamente descritos a seguir:

O Capítulo 2 apresenta um breve histórico sobre a atividade espacial no Brasil, com o objetivo de fornecer subsídios ao estudo de caso. É apresentada a Missão Espacial Completa Brasileira para mostrar o contexto das atividades do IAE. O Veículo Lançador de Satélites do IAE é descrito através dos requisitos do seu

software de bordo e de outras abordagens sugeridas ao projeto. O capítulo se encerra com um resumo sobre o atual programa espacial brasileiro, o Programa Cruzeiro do Sul.

O Capítulo 3 é destinado a apresentar uma introdução à linha de produtos de software. São descritas as atividades necessárias ao desenvolvimento desta abordagem e as iniciativas mais relevantes como método de desenvolvimento de linha de produtos de software.

O Capítulo 4 descreve conceitos sobre a variabilidade. São também apresentadas as atividades necessárias a um processo de gerenciamento de variabilidade, uma importante atividade para o desenvolvimento da linha de produtos de software. O capítulo se encerra com resumo e discussão sobre várias técnicas que podem ser empregadas para a atividade de implementação da variabilidade.

O Capítulo 5 introduz conceitos sobre reflexão e modelos de objetos adaptáveis que são utilizados no desenvolvimento do mecanismo de variabilidade do processo GVLPS. O capítulo apresenta a arquitetura de modelos de objetos adaptáveis, assim como os padrões de projetos necessários para a construção desta arquitetura.

O Capítulo 6 apresenta o processo GVLPS, através das atividades e artefatos de entrada e saída que compõe o processo proposto, e exemplifica os artefatos gerados por estas atividades. O capítulo introduz ainda, o mecanismo de variabilidade, baseado em modelos de objetos adaptáveis e em reflexão, utilizado pelo processo GVLPS.

O Capítulo 7 descreve a aplicação do processo GVLPS ao LSB; mostra como os passos do processo apresentados no capítulo anterior, devem ser seguidos para se gerenciar a variabilidade do LSB, detalha a construção dos artefatos resultantes, assim como, o projeto do mecanismo de variabilidade para o processo GVLPS.

O Capítulo 8 apresenta o protótipo do mecanismo de variabilidade baseado em modelos de objetos adaptáveis e em reflexão para o processo GVLPS. O ambiente de desenvolvimento é apresentado e é detalhado como os padrões de modelos de objetos adaptáveis e as técnicas de reflexão são utilizadas na implementação do

protótipo. Também é descrito como as variantes são selecionadas e criadas através do protótipo do mecanismo de variabilidade proposto.

Finalmente, no Capítulo 9, são apresentados as contribuições, os trabalhos futuros, as conclusões e retrospectiva do trabalho.

"Mais que seu objetivo,
o que importa é a força que
você tem para lutar por ele."

2 ATIVIDADES ESPACIAIS NO BRASIL

Este capítulo destina-se a apresentar os tópicos relevantes das atividades espaciais no Brasil, para contextualizar o estudo de caso deste trabalho. São apresentados a missão espacial brasileira, o Veículo Lançador de Satélites juntamente com seus requisitos do software de bordo e outras abordagens sugeridas neste projeto. O capítulo apresenta ainda, um resumo sobre o atual programa espacial brasileiro, o Programa Cruzeiro do Sul.

As atividades espaciais contribuíram de maneira significativa ao país, resultando em proveito para indústria e sociedade seja, através das missões de satélites, cargas úteis sub-orbitais e balões para observação da Terra, ou através das inovações decorrentes dos esforços empregados no desenvolvimento de tecnologia para atender as necessidades da área de veículos lançadores (AEB, 2006). Além disso, a construção de veículos espaciais, além de garantir e preservar a autonomia de acesso ao espaço, também possibilita a exploração comercial de serviços de lançamento (AEB, 2005).

Como benefícios indiretos dos desafios impostos às atividades espaciais, a utilização das tecnologias advindas deste setor tem possibilitado o avanço tecnológico em diversas áreas industriais, destacando-se a indústria metalúrgica, de materiais compostos, eletrônica e robótica (OLIVEIRA,1998).

Visando a obtenção destes benefícios citados e a necessidade da capacitação no domínio da tecnologia espacial, o Brasil iniciou, na década de 60, suas primeiras atividades espaciais, através do lançamento de pequenos foguetes de sondagem.

No final da década de 70, iniciou-se a Missão Espacial Completa Brasileira (MECB), um programa integrado que visava projeto, desenvolvimento, construção e operação de satélites, de fabricação nacional, a serem lançados por um foguete também projetado e construído no país (IAE, 2006a).

Coube então ao antigo Instituto de Atividades Espaciais (IAE), atual Instituto de Aeronáutica e Espaço (IAE), a responsabilidade da construção e do lançamento do primeiro Veículo Lançador de Satélites (VLS).

Em 1997, foi dado início ao plano de qualificação em vôo do VLS, que consistia de lançamentos de 4 protótipos (COMANDO DA AERONÁUTICA, 2004). Desde então, três tentativas de lançamento, em 1997, 1999 e 2003, destes protótipos foram realizadas, sendo que nenhuma destas obteve sucesso. Dias antes da última tentativa de lançamento em 2003, um grave acidente ocorreu, resultando na morte de 21 profissionais. Após este fato, o projeto VLS passou por muitas revisões e reestruturações e atualmente, neste ano de 2009, o Veículo Lançador de Satélites continua em fase de desenvolvimento.

2.1 Missão Espacial Completa Brasileira – MECB

Em agosto de 1977, durante o Primeiro Seminário de Atividades Espaciais (SAE), ocorrido na cidade do Rio de Janeiro, estabeleceu-se o desenvolvimento de uma ambiciosa proposta, a realização de uma Missão Espacial Completa (MEC), integrada por três projetos específicos (MONTENEGRO,1997):

(1) Veículo Lançador de Satélites – cujo desenvolvimento ficou a cargo do antigo Instituto de Atividades Espaciais do Centro Técnico Aeroespacial (CTA) -

> "...o Seminário optou pelo desenvolvimento de um veículo para lançar satélites de 100 a 120 Kg, em órbita circular situada entre 500 e 700 Km..." (SAE apud MONTENEGRO,1997).

(2) Satélite – cujo projeto foi atribuído ao Instituto Nacional de Pesquisas Espaciais (INPE) -

> "...a comissão recomenda, como passo inicial, o desenvolvimento de satélites de 100 a 120 Kg para órbita circular entre 500 e 700 Km. Ressalte-se que um satélite

> com estas características possibilita várias aplicações, como as científicas, as de meteorologia, as de sensoriamento remoto..." (SAE apud MONTENEGRO,1997).

(3) Segmento Terrestre -

> "...o desenvolvimento de estações de rastreio, telemetria e comando, e as instalações de montagem nos locais de lançamento..." (SAE apud MONTENEGRO,1997).

A partir daí, passaram a ser definidas as especificações e finalidades da missão para que, em novembro de 1979, durante o Segundo Seminário de Atividades Espaciais, realizado na cidade de São José dos Campos, a Comissão Brasileira de Atividades Espaciais, o INPE e o IAE/CTA apresentassem, em conjunto, a Missão Espacial Completa Brasileira (MECB) (COMANDO DA AERONÁUTICA, 2004).

Assim, em 1980 foi aprovada a proposta da MECB; suas principais informações estão resumidamente descritas a seguir:

(1) a proposta consiste de desenvolvimento, construção e operação de satélites de fabricação nacional, previstos para duas missões básicas – coleta de dados e sensoriamento remoto - a serem colocados em órbitas baixas por um Veículo Lançador de Satélites a Propelente Sólido (VLSS) projetado e construído no país e lançado de uma base situada em território brasileiro (IAE, 2006a), (MONTENEGRO,1997);

(2) os satélites de Coleta de Dados teriam aproximadamente 100Kg, sendo colocados em órbita circular quase equatorial, a cerca de 700km de altura; os de Sensoriamento Remoto teriam cerca de 150Kg e seriam colocados em órbita circular hélio-síncrona quase polar a cerca de 650Km de altura (MONTENEGRO,1997);

(3) o VLSS, que viria ser o VLS, deveria satisfazer às duas missões básicas de satelização, anteriormente descritas;

(4) o Instituto de Atividades Espaciais, que mais tarde se uniria a um outro instituto do CTA, o Instituto de Pesquisas e Desenvolvimento,

formando o atual Instituto de Aeronáutica e Espaço, ficou responsável pelo desenvolvimento do VLS e da Base de Lançamentos de foguetes, que resultou no Centro de Lançamento de Alcântara (COMANDO DA AERONÁUTICA, 2004), (MONTENEGRO,1997);

(5) ao INPE coube o desenvolvimento de dois satélites de coleta de dados ambientais e dois satélites de sensoriamento remoto (COMANDO DA AERONÁUTICA, 2004), (MONTENEGRO,1997);

(6) como outros objetivos, pode-se ainda citar (MONTENEGRO,1997):

a. adquirir a capacitação tecnológica e industrial para a produção de bens de qualidade compatível com os critérios exigidos pelos programas espaciais;

b. permitir ao país colocar em órbita, satélites como de mapeamento geológico, de agricultura, de análise ambiental, etc, contribuindo para seus programas de interesse e estabelecendo competência na área espacial;

c. gerar bens para a economia de desenvolvimento do país, tais como, ligas de aço de ultra-alta-resistência, técnicas especiais de tecelagem, tecnologia de materiais compostos, hélices leves, etc.

2.2 Veículo Lançador de Satélites

O Veículo Lançador de Satélites (VLS) é um veículo lançador convencional, cujo objetivo é colocar em órbitas de baixa altitude, entre 200 e 1000Km, satélites de Coleta de Dados assim como os de Sensoriamento Remoto, com massas entre 100 e 200Kg, lançados a partir do Centro de Lançamento de Alcântara (REIS FILHO, 1995).

O VLS é composto por quatro estágios, um compartimento para transporte de carga útil (satélite), seções (baias ou módulos) para alojamento de instrumentação e

equipamentos diversos, quatro redes elétricas funcionais - Serviço, Telemedidas, Segurança e Controle - e um conjunto de 244 pirotécnicos.

A divisão do veículo em estágios, ou seja, em partes que contêm combustível e/ou módulos com equipamentos, é necessária para auxiliar o veículo a atingir a altitude necessária. Cada estágio do veículo é responsável por gerar uma nova propulsão em momentos distintos de uma seqüência pré-determinada. Após a queima do combustível contido nestes estágios, os mesmos são descartados, aliviando assim, o peso do veículo.

Uma descrição detalhada da configuração e equipamentos de cada estágio pode ser obtida em (COMANDO DA AERONÁUTICA, 2004).

A Rede Elétrica de Serviço tem como objetivo comandar os eventos principais de vôo, tais como a separação de estágio, ignições, ativação de subsistema. Além disso, esta rede tem a função de fornecer potência elétrica para todos os equipamentos do veículo e condicionar os sinais dos sensores para a Rede Elétrica de Telemedidas.

A Rede Elétrica de Telemedidas tem como função coletar, codificar e transmitir, para o solo, as informações e medidas de diversos parâmetros do vôo.

A Rede Elétrica de Segurança tem como objetivo facilitar a localização do veículo pelos radares em solo e de receber o telecomando de destruição do veículo, no caso de sua trajetória evoluir de forma perigosa para as áreas consideradas de proteção.

A Rede Elétrica de Controle tem como função, realizar a navegação e o controle de atitude do veículo desde a sua decolagem, até pouco antes do vôo do quarto estágio. Esta rede é composta pelos Sensores Inerciais, Controlador Digital (CTD), Atuadores Elétricos e Condicionadores de Sinais. O CTD é constituído pelo Computador de Bordo onde é instalado o Software Aplicativo de Bordo (SOAB) (REIS FILHO, 1995). O SOAB permite a execução das decisões de controle e de sequência de eventos do veículo.

Na decolagem, o veículo possui 19,4 metros de comprimento e massa de 49,7 toneladas, sendo 41 toneladas de combustível propelente sólido, responsável pela

propulsão de todos os estágios do veículo (COMANDO DA AERONÁUTICA, 2004), (IAE, 2006b). A configuração do VLS pode ser observada na Figura 2.1.

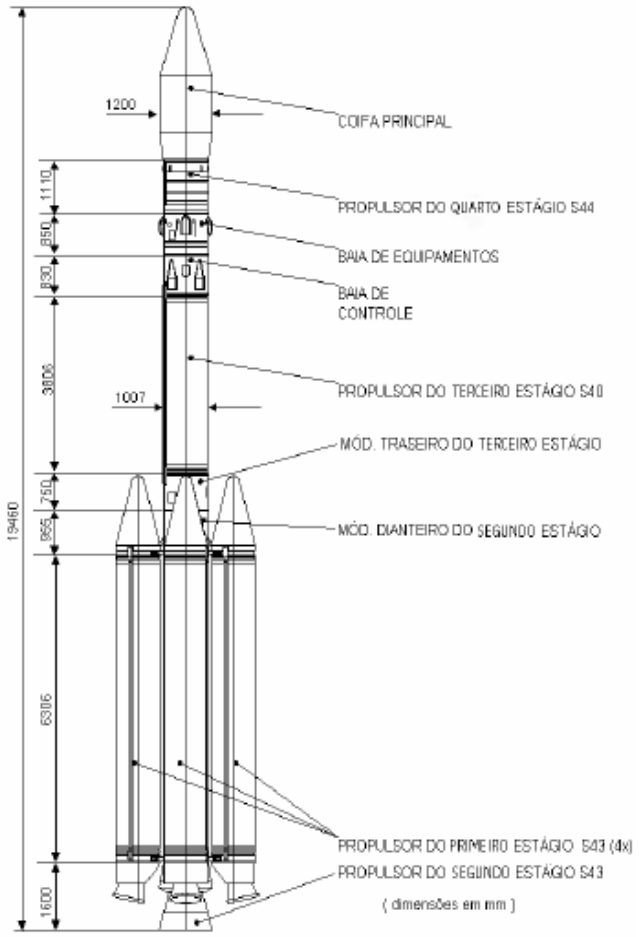


Figura 2.1- Configuração do VLS
Fonte: (COMANDO DA AERONÁUTICA, 2004)

Para possibilitar a injeção do satélite na órbita prevista, cada um dos estágios tem por objetivo prover um incremento de velocidade ao veículo. A coifa, além de proporcionar ao veículo uma forma aerodinâmica adequada, protege o satélite desde a fase de preparação do lançamento até o final da travessia da atmosfera mais densa pelo veículo (REIS FILHO, 1995), (IAE,2006b).

A seqüência de eventos que ocorrem no VLS, durante a sua missão, pode ser observada na Figura 2.2.

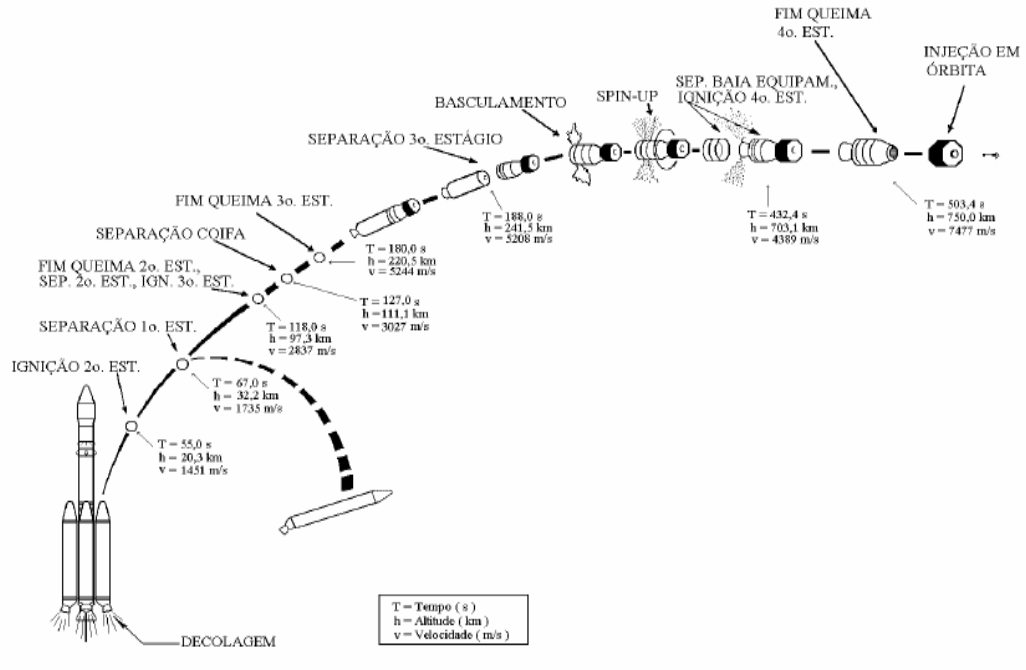


Figura 2.2 - Perfil típico da missão VLS
Fonte: (COMANDO DA AERONÁUTICA, 2004)

No instante inicial do lançamento, para a decolagem do veículo, os quatro motores do primeiro estágio são acionados simultaneamente. A ignição do segundo estágio ocorre antes do fim de queima do primeiro estágio para não prejudicar o controle efetivo do veículo, entre o final de queima dos quatro motores do primeiro estágio e a separação dos mesmos. Após o fim de queima do segundo estágio e sua separação, o terceiro estágio é acionado. No início do vôo do terceiro estágio, ocorre a separação da coifa de proteção do satélite. Após o fim de queima do terceiro estágio, o motor vazio do terceiro estágio e a baia de controle de rolamento são separados do veículo. Segue-se a manobra de basculamento, que visa posicionar o conjunto do quarto estágio mais o satélite na atitude desejada. Após a orientação do veículo, este é colocado em rotação pelo sistema impulsor de rolamento (s*pin-up*) e, em seguida é feita a separação da baia de controle, ativando o acendimento do quarto estágio. Ao final da queima do quarto estágio, dá-se a separação do satélite e sua injeção em órbita (IAE, 2007).

2.2.1 O Software de Bordo do Veículo

O SOAB é o software de tempo real de aplicação embarcado no CTD. Seu desenvolvimento iniciou-se em 1994, tendo como objetivo principal habilitar o CDT a desempenhar todas as suas funções nas Redes Elétricas de Controle e Serviço. Ao SOAB, cabe as seguintes funções (REIS FILHO, 1995), (OLIVEIRA,1998):

- controlar, em tempo real, o vôo dos três primeiros estágios;
- realizar a guiagem do terceiro estágio;
- realizar a manobra do conjunto do quarto estágio e baia de equipamentos;
- comandar a seqüência de eventos até a separação da baia de equipamentos;
- manter armazenado um perfil de atitude pré-calculado no solo para uso dos algoritmos do sistema de controle;
- processar as informações inerciais necessárias aos algoritmos de controle do veículo;
- monitorar a pressão dos quatro *boosters* do primeiro estágio e a aceleração específica;
- fornecer informações para a Rede Elétrica de Telemetria;
- manter comunicação com o Banco de Controle em solo;
- monitorar e comandar eventos internos;
- medir o tempo decorrido a partir da decolagem;
- iniciar-se, de modo a obter a máxima segurança para a sua operação;
- iniciar o Módulo Sensor Inercial (MSI);

- prover suporte para a realização de simulações do vôo na fase de desenvolvimento e de integração dos sistemas de bordo, assim como entre os sistemas de bordo e os de solo;
- fornecer meios para a simulação do vôo do veículo;
- fornecer meios para testar toda a Rede Elétrica de Controle e os subsistemas físicos na fase anterior ao vôo;
- testar o hardware do CTD isoladamente;
- testar o CTD integrado às Redes Elétricas de Controle, de Serviço e de Telemedidas, em parte ou todas elas.

Ainda, segundo Reis Filho (1995), o SOAB deve fornecer meios para avaliação dos resultados dos algoritmos de controle até o final do vôo do VLS.

2.2.2 Abordagens Utilizadas para Desenvolvimento do SOAB

Esta seção tem como objetivo mostrar o contexto do desenvolvimento do SOAB. O SOAB foi originalmente projetado para atender a um conjunto de quatro protótipos de um mesmo projeto de veículo lançador. Por isso, não houve um planejamento do software para reuso decorrente de alterações de requisitos, ou para o desenvolvimento de projetos similares. Além disso, na época, os envolvidos no projeto não previram a longa duração do desenvolvimento do projeto. Isto ocasionou obsolescência das tecnologias utilizadas como metodologias e ferramentas CASE, e dificuldades para a manutenção do conhecimento devido à diminuição da equipe de desenvolvimento.

O processo de desenvolvimento de software do SOAB foi baseado no paradigma estruturado. Os aspectos de tempo real foram incorporados aos modelos de análise através da utilização do método de Hatley-Pirbhai (HATLEY,PIRBHAI, 1990), que é especificamente para sistemas de tempo real. O principal artefato foi o diagrama de fluxo de dados (DFD), com recursos para representar as características de tempo

real, tais como eventos e sincronização entre as transformações de dados e de controle.

A transição entre as fases de análise e projeto foi realizada com auxílio do método *Design Approach for Real-Time Systems* (DARTS) (GOMAA,1984). Este método também é estruturado e possui um procedimento para definir as tarefas, que constituem o sistema, e as interfaces entre elas a partir dos diagramas de fluxos de dados.

O detalhamento das tarefas foi realizado através dos diagramas de estrutura, que foram construídos utilizando-se o método de Page-Jones (PAGE-JONES, 1988).

Para a implementação do SOAB, foi utilizada a linguagem C.

Para gerenciar o desenvolvimento do SOAB, foi adotado o padrão DoD-Std-2167A (DEFENSE SYSTEM SOFTWARE DEVELOPMENT, 1988). Este padrão é amplamente utilizado no desenvolvimento de software de natureza crítica, como os de defesa e os espaciais, devido aos procedimentos bem definidos que o mesmo apresenta (REIS FILHO, 1995). Um outro padrão também adotado foi o padrão IEEE-Std-983 (IEEE, 1986) para orientar as atividades de garantia da qualidade do SOAB.

A equipe de desenvolvimento do SOAB optou por utilizar o paradigma de ciclo de vida clássico. Este paradigma requer uma abordagem sistemática, seqüencial ao desenvolvimento do software, que se inicia no nível do sistema e avança ao longo da análise, projeto, codificação, teste e manutenção (PRESSMAN, 1995).

Devido à sua complexidade e suas características de software crítico de tempo real, o SOAB serviu como estudo de caso em vários trabalhos acadêmicos relacionados a processo de desenvolvimento, qualidade e confiabilidade.

Uma abordagem de qualidade e confiabilidade foi desenvolvida para o SOAB por Reis Filho (1995). Neste trabalho, o autor realiza um estudo detalhado do SOAB relacionado à segurança e aplica técnicas de análise de risco ao software como a *Failure Mode, Effects and Criticality Analysis* (FMECA) e *Fault Tree Analisys* (FTA).

Lemes (1997) apresenta o processo de elaboração do Plano de Garantia da Qualidade para o projeto do SOAB. Alguns resultados decorrentes da aplicação do plano elaborado no desenvolvimento do SOAB também podem ser encontrados no trabalho citado.

O trabalho de Andrade (2000) focou o emprego do *Rational Unified Process* (RUP) ao SOAB, onde vários modelos dinâmicos e estáticos em UML foram gerados para o software.

2.3 Programa Cruzeiro do Sul

A fim de aprimorar a tecnologia de propulsão de foguetes com o desenvolvimento de novos motores, o CTA e a AEB anunciaram, em outubro de 2005, o desenvolvimento de uma nova família de veículos lançadores com capacidade para transportar satélites e plataformas espaciais de pequeno, médio e grande porte a órbitas baixas, médias e de transferência geoestacionária (CTA, 2006).

A nova família de veículos lançadores, a ser desenvolvida através do Programa Cruzeiro do Sul (MORAIS, 2005), é composta pelos veículos Alfa, Beta, Gama, Delta e Epsilon, referentes às cinco estrelas da constelação Cruzeiro do Sul. Esta família atenderá tanto as missões espaciais propostas no Programa Nacional de Atividades Espaciais (PNAE) da AEB (AEB, 2005), como também as missões de clientes internacionais.

O programa, que prevê uma evolução gradativa dos seus veículos lançadores para alcance de melhores desempenhos e de maiores capacidades para o transporte de carga útil, terá como um de seus maiores desafios o desenvolvimento e a fabricação de motores a propulsão líquida de médio e grande porte, uma tecnologia ainda não completamente dominada pelo Brasil. Os motores com sistema de propulsão líquida são mais sofisticados que motores de propulsão sólida, utilizados atualmente no VLS.

Além desta inovação tecnológica, uma outra característica comum e importante da família Cruzeiro do Sul é a otimização de eventos na trajetória de lançamento dos veículos lançadores. Isto leva a uma maior simplicidade da configuração dos veículos, como por exemplo, a diminuição do número de motores no primeiro estágios do veículo Beta e Gama, permitindo maior confiabilidade no lançamento.

O veículo lançador Alfa será uma evolução direta do VLS, constituído pelo primeiro e segundo estágios do veículo atual e por um propulsor a propelente líquido como estágio superior. Terá capacidade de colocar satélites de até 250Kg em órbitas equatoriais de 750Km de altitude.

O veículo lançador Beta terá um motor com capacidade para 40 toneladas de propelente sólido no primeiro estágio e dois propulsores a propelente líquido no segundo e terceiro estágios. Terá capacidade para lançar satélites de 800 Kg em órbitas equatoriais de 800Km de altitude.

Os três outros veículos lançadores Gama, Delta e Epsilon, terão capacidade para colocar em órbita satélites geoestacionários, que são aqueles de maior porte e que operam em maiores altitudes, de 36 mil quilômetros, para garantir o apontamento fixo sobre a Terra.

Este programa que será desenvolvido em parceria com a Rússia, possui um prazo de execução de 17 anos (até 2022) e possibilitará ao Brasil a independência no transporte espacial de satélites de pequeno a grande porte (MORAIS, 2005). A Figura 2.3 ilustra a família de veículos lançadores de satélites do Programa Cruzeiro do Sul, sendo da esquerda para direita: Alfa, Beta, Gama, Delta e Epsilon.

Figura 2.3 - Família de veículos do Programa Cruzeiro do Sul
Fonte: (CTA, 2006)

2.4 Considerações Finais

Por ser tão abrangente e desafiador, o setor espacial tem atraído uma variedade de estudos e pesquisas, o que pode ser comprovado através do crescente número de trabalhos apresentados nas conferências internacionais do setor, assim como, dos resultados alcançados por muitas pesquisas de institutos e agências espaciais espalhadas pelo mundo.

O desenvolvimento de veículos lançadores brasileiros é um projeto ambicioso que visa inserir o Brasil no seleto grupo de países que dominam a conquista do espaço e obter os benefícios providos por este avanço tecnológico. Pode-se observar, no decorrer deste capítulo, que o software para controle de um sistema desta natureza é considerado complexo e exige um processo definido, envolvendo metodologias e normas adequadas em seu longo período de desenvolvimento.

A perspectiva de novos programas para desenvolvimento de família de veículos lançadores de satélite, como o Cruzeiro do Sul, leva a uma necessidade de um desenvolvimento mais planejado de software, com elaboração de um conjunto de

artefatos para que estes possam ser reusados nos novos veículos lançadores de satélites.

A abordagem da linha de produtos de software busca minimizar estas dificuldades, pois sua proposta é fornecer meios para um desenvolvimento que visa um reuso de forma organizada e eficiente.

Como já citado, devido ao projeto VLS ser de natureza sigilosa, o VLS não será utilizado diretamente como objeto de estudo neste trabalho. Foi definido um software de um veículo lançador fictício, denominado Lançador de Satélites Brasileiro – LSB, para ser utilizado como estudo de caso na aplicação do processo GVLPS. Os benefícios identificados com a aplicação do processo GVLPS ao software do LSB, tais como, flexibilidade de alteração e extensão, e reuso dos artefatos e código, podem constituir uma base para a implantação de uma linha de produtos de software para veículos lançadores reais.

As inúmeras pesquisas empregadas no setor espacial resultam em benefícios para diversas áreas. Este fato, aliado à certa familiaridade da autora neste assunto, por atuar profissionalmente no contexto espacial, motivaram a escolha da área espacial como estudo de caso deste trabalho.

"Utilize os talentos que tiver:
haveria silêncio nos bosques
se só os pássaros que cantam bem,
cantassem."

Henry Van Dyke

3 LINHA DE PRODUTOS DE SOFTWARE

Este capítulo introduz a abordagem de linha de produtos de software. A definição dos conceitos relacionados e as principais atividades necessárias ao desenvolvimento de linha de produtos de software são apresentadas. São descritas, a seguir, as iniciativas de vários autores e instituições para métodos de desenvolvimento de linha de produtos de software. O capítulo também aponta os benefícios e os custos que devem ser considerados no emprego desta abordagem.

3.1 Introdução à Linha de Produtos de Software

A linha de produtos compreende um conjunto de produtos que juntos se destinam a um segmento de mercado específico ou a uma missão particular (NORTHROP, CLEMENTS, 2007).

A idéia de linha de produtos é explorar as características comuns que os produtos similares possam ter, para agilizar sua construção. Esta idéia não é recente e vem sendo utilizada por diversas empresas em diferentes setores.

Entretanto, o conceito de linha de produtos de software é relativamente novo e pode ser considerado uma das mais promissoras abordagens de reuso de arquitetura que está sendo amplamente adotada pela comunidade de reuso (NORTHROP, 2002). O aumento pelo interesse em linha de produtos de software se deve principalmente ao fato dos desenvolvedores perceberem que podem se beneficiar muito mais com um reuso de arquitetura de software mais planejado e direcionado a um domínio específico, ao invés de continuar reusando componentes de software individuais de forma oportunista, acidental e adaptando-os a inúmeras situações.

Além disso, o desenvolvimento de software baseado na abordagem de linha de produtos de software difere-se do processo tradicional de software porque considera uma família de sistemas de software e não cada sistema individualmente.

Segundo Gomaa (2005), a linha de produtos de software consiste de uma família de sistemas de software que possui características comuns (*commonalities*) e variabilidade (*variability*).

As características comuns são aquelas compartilhadas por todas as aplicações da linha de produtos de software e a variabilidade representa a capacidade de um sistema de ser mudado ou customizado (KIM et al., 2006). Nas características comuns está incluída ainda, a funcionalidade comum para todos os membros da linha de produtos de software, assim como, na variabilidade está incluída a funcionalidade que é fornecida por alguns, mas não por todos os membros da linha de produtos de software (GOMAA, 2005).

A partir das características comuns e variabilidade é gerado um conjunto de ativos principais (*core assets*) que consiste de itens reusáveis, tais como, requisitos, modelos, componentes, arquitetura, planos de trabalho, descrição de processo, planos de testes, casos de testes, entre outros. Este conjunto é a base para a linha de produtos de software e os seus itens podem ser reusados por diferentes membros da família de produtos.

Um outro conceito importante na abordagem da linha de produtos de software é a *feature*. Uma *feature* é um aspecto, qualidade ou característica de um software ou de sistemas que é preeminente e visível ao usuário ou a outro sistema (CZARNECKI, EISENECKER, 2005).

Conhecendo-se o conceito sobre *core assets* e *feature* pode-se entender uma das definições mais clássicas encontrada na literatura sobre linha de produtos de software, estabelecida por Northrop e Clements (2007) do *Software Engineering Institute* (SEI):

> Uma linha de produtos de software é um conjunto de sistemas com uso intensivo de software que compartilham um conjunto de *features* comuns e gerenciáveis, que satisfazem às necessidades específicas de um segmento

> de mercado particular ou missão e que são desenvolvidos a partir de um conjunto de *core assets* de uma forma pré-estabelecida.

3.2 Desenvolvimento da Linha de Produtos de Software

Segundo Northrop e Clements (2007), o desenvolvimento da linha de produtos de software pode ser dividido em três atividades principais:

(1) Engenharia de Domínio: também conhecida como Desenvolvimento de *Core Asset,* é a atividade que engloba as fases de análise do domínio, projeto e implementação do *core asset*. Durante estas etapas, o *core asset*, constituído por características comuns e variabilidade, é construído para ser usado na produção de outros produtos na linha de produtos de software.

(2) Engenharia de Aplicação: também denominada de Desenvolvimento de Produtos, é a atividade em que os produtos de software são construídos através do reuso dos *core assets* desenvolvidos durante a atividade de Engenharia de Domínio.

(3) Gerenciamento: engloba as atividades de gerenciamento nas áreas técnica e organizacional. O Gerenciamento Técnico é responsável por supervisionar as atividades relacionadas à Engenharia de Domínio e de Aplicação, o Gerenciamento Organizacional auxilia e determina uma estratégia de produção.

O SEI utiliza os termos Desenvolvimento de *Core Asset* e Desenvolvimento de Produtos, entretanto, neste trabalho, optou-se por adotar Engenharia de Domínio e Engenharia de Aplicação respectivamente para estas atividades, já que são as nomenclaturas adotadas pela maioria dos trabalhos encontrados na literatura. Contudo, manteve-se a utilização das definições do SEI para estes termos.

A Figura 3.1 ilustra as atividades que constituem o desenvolvimento da linha de produtos de software. Os círculos, fazendo um movimento de rotação, significam que as três atividades são essenciais, interligadas e altamente interativas, podendo ocorrer em qualquer ordem (NORTHROP, CLEMENTS, 2007).

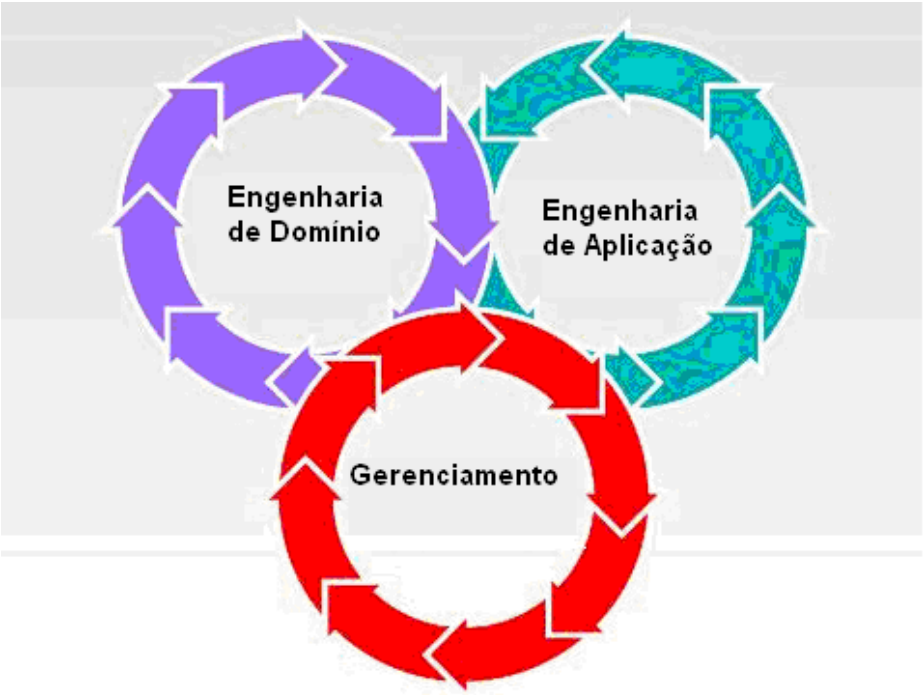


Figura 3.1- Atividades principais para linha de produtos de software
Fonte Adaptada: (NORTHROP, CLEMENTS, 2007)

As Seções 3.2.1 a 3.2.3 explicam em detalhes cada uma destas atividades.

3.2.1 Engenharia de Domínio

No âmbito da engenharia de software, o termo domínio é definido como uma parte especializada do conhecimento, uma área de experiência ou uma coleção de funcionalidade relacionada (NORTHROP, CLEMENTS, 2007).

O objetivo da Engenharia de Domínio é realizar as atividades de coletar, organizar e armazenar experiências passadas no desenvolvimento de sistemas ou partes do sistema em um domínio específico na forma de *core assets* e providenciar meios adequados para reusá-los ao construir novos sistemas (CZARNECKI, EISENECKER, 2005).

Estas atividades podem ser organizadas através de um processo em que as características comuns e a variabilidade são definidas e desenvolvidas (POHL, BÖCKLE, LINDEN, 2005), (NORTHROP, CLEMENTS, 2007).

A Figura 3.2 mostra as atividades que compõem a Engenharia de Domínio.

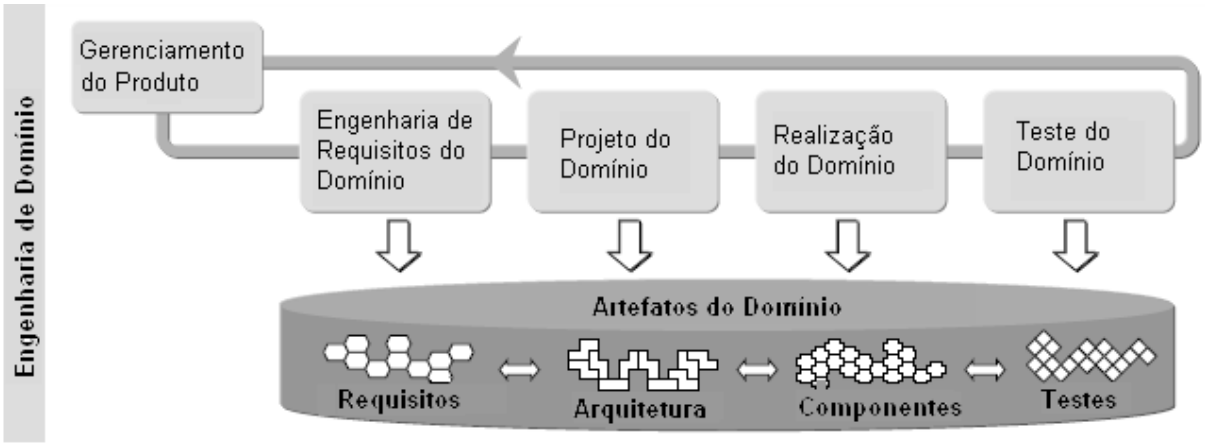


Figura 3.2 - Atividades da Engenharia de Domínio
Fonte Adaptada: (POHL, BÖCKLE, LINDEN, 2005)

A descrição de cada atividade está apresentada a seguir (POHL, BÖCKLE, LINDEN, 2005):

- **Gerenciamento do Produto:** define o que pertence ou não ao escopo da linha de produtos de software. Sua entrada consiste dos objetivos da linha de produtos e sua saída é um roteiro que determina as características comuns e variabilidade dos futuros produtos.

- **Engenharia de Requisitos do Domínio:** identifica o escopo do domínio juntamente com o *core assets*.

- **Projeto do Domínio:** desenvolve uma arquitetura de referência da linha de produtos de software e define um plano de produção. A arquitetura de referência fornece uma estrutura comum e de alto nível para todas as aplicações da linha de produtos de software.

- **Realização do Domínio:** implementa os *core assets*.

- **Teste do Domínio:** valida e verifica os *core assets* através de testes.

Os artefatos resultantes da Engenharia de Domínio são reusados para produzir novos sistemas durante a atividade de Engenharia de Aplicação.

3.2.2 Engenharia de Aplicação

A Engenharia de Aplicação é o processo em que sistemas, também chamados aplicações, são desenvolvidos com base nos *core assets* da linha de produtos de software produzidos pela Engenharia de Domínio.

A Figura 3.3 mostra as atividades que compõem a Engenharia de Aplicação.

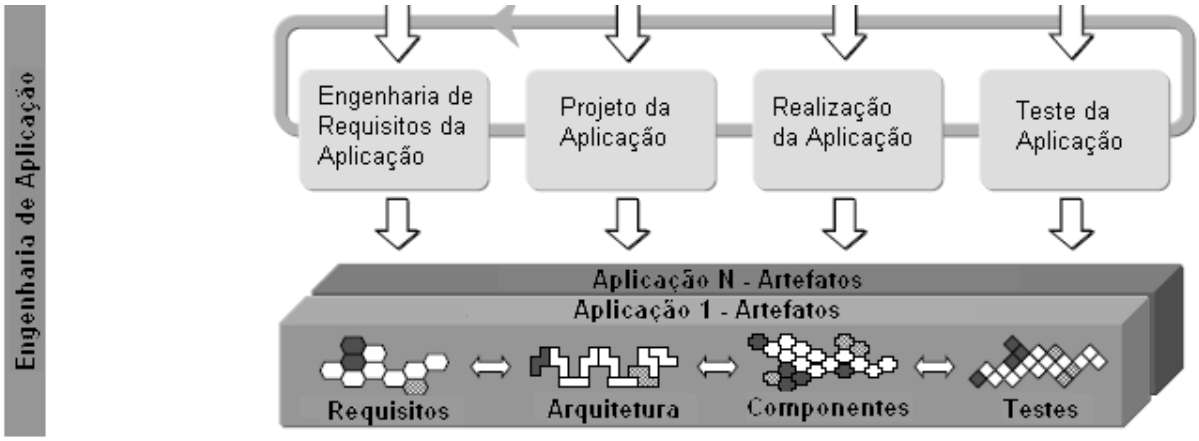


Figura 3.3 - Atividades da Engenharia de Aplicação
Fonte Adaptada: (POHL, BÖCKLE, LINDEN, 2005)

A descrição resumida destas atividades está apresentada a seguir (POHL, BÖCKLE, LINDEN, 2005):

- **Engenharia de Requisitos da Aplicação**: corresponde às atividades referentes à especificação dos requisitos da aplicação. É feita a consistência entre os requisitos especificados na análise do domínio e nesta atividade.

- **Projeto da Aplicação**: corresponde às atividades para produzir uma arquitetura de aplicação, utilizando a arquitetura de referência desenvolvida no projeto do domínio, incorporando as adaptações específicas da aplicação.
- **Realização da Aplicação**: cria a aplicação, utilizando a arquitetura desenvolvida no projeto de aplicação e os artefatos criados na Engenharia de Domínio, os componentes podem ser construídos como variantes de componentes criados no projeto de domínio.
- **Teste da Aplicação**: valida e verifica a aplicação de acordo com sua especificação.

3.2.3 Gerenciamento

Esta atividade tem responsabilidade de coordenar e supervisionar o desenvolvimento da linha de produtos de software e consistem de (NORTHROP, CLEMENTS, 2007):

- **Gerenciamento Organizacional**: determina uma estratégia de produção, através da criação um plano de produção que descreve as necessidades da organização e a estratégia para atender estas necessidades. Auxilia nas atividades técnicas e na interação entre as atividades essenciais da Engenharia de Domínio e da Engenharia de Aplicação, garantindo também que estas operações sejam documentadas.
- **Gerenciamento Técnico**: inspeciona as atividades relacionadas à Engenharia de Domínio e de Aplicação, garantindo que os artefatos dos *core assets* e os produtos desenvolvidos estejam de acordo com as atividades requeridas. Também decide o método de produção e fornece elementos do gerenciamento de projeto do plano de produção.

3.3 Principais Abordagens para Linha de Produtos de Software

Esta seção destina-se a apresentar as abordagens e os métodos de desenvolvimento de linha de produtos de software do SEI (NORTHROP, CLEMENTS, 2007) e do Gomaa (2005), que foram selecionadas como referência por serem consideradas mais apropriadas ao objetivo deste trabalho.

3.3.1 *Framework for Software Product Line Practice* - FSPLP

O *Framework for Software Product Line Practice* (FSPLP) é uma iniciativa do SEI e é um documento baseado em *web* cujo objetivo é auxiliar a comunidade de software no desenvolvimento de linhas de produtos de software (NORTHROP, CLEMENTS, 2007). Este documento define as Áreas das Práticas (*Practice Áreas*) para realizar as atividades essenciais de linha de produtos de software que correspondem ao desenvolvimento de *core assets*, desenvolvimento de produtos e gerenciamento. Uma Área da Prática é uma coleção de atividades que uma organização deve conhecer para realizar o trabalho essencial de linha de produtos de software. As Áreas das Práticas facilitam a execução das atividades essenciais porque subdividem estas atividades em conjunto de atividades menores e mais controladas (NORTHROP, CLEMENTS, 2007).

As Áreas das Práticas são classificadas em três categorias:

(1) Área das Práticas de Engenharia de Software: é constituída pelas atividades necessárias para aplicar a tecnologia apropriada para criação e evolução do *core assets* e dos produtos. Estas atividades são:

- **Definição de Arquitetura:** descreve as atividades necessárias para a definição da arquitetura de software.
- **Avaliação da Arquitetura:** descreve as atividades que são desenvolvidas para avaliar a arquitetura definida.

- **Desenvolvimento dos Componentes:** corresponde às atividades que produzem os componentes da arquitetura de software.
- **Recuperação de Ativos Existentes:** refere-se às atividades de reabilitação de parte de um sistema antigo para ser empregado em um novo sistema.
- **Engenharia de Requisitos:** realiza as técnicas de elicitação, análise, especificação, verificação e gerenciamento de requisitos.
- **Integração de Sistema de Software:** combina os componentes de software testados individualmente formando um sistema integrado.
- **Teste:** corresponde à identificação de falhas para serem reparadas e à verificação se o software tem o desempenho de acordo com sua especificação.
- **Entendimento do Domínio:** identifica áreas, problemas e soluções de domínio para a construção de produtos, assim como, comunica estas informações aos envolvidos.
- **Utilização de Software Avaliado Externamente:** consiste em adquirir, integrar, testar e gerenciar um software externo, como por exemplo, um software de prateleira (c*ommercial off-the-shelf* - COTS).

(2) Área das Práticas de Gerenciamento Técnico: é constituída pelas atividades necessárias para gerenciar a criação e evolução do *core assets* e dos produtos. Estas atividades são:

- **Gerenciamento de Configuração:** estabelece e mantém a integridade dos produtos do projeto de software durante todo o ciclo de vida de software do projeto.
- **Análise de Serviço, Compra e Realização:** consiste em tomada de decisões baseadas em fatores estratégicos como, por exemplo, custo,

planejamento, disponibilidade de equipe, qualidade esperada e ajustes propostos.

- **Medidas e Rastreio:** fornece suporte ao rastreio do projeto e guia as decisões de gerenciamento tomadas. Para determinar se os objetivos estão sendo alcançados, o gerenciador deve possuir informações que revelem como os esforços estão sendo aplicados.
- **Disciplina do Processo:** refere-se à capacidade que a organização possui para definir, seguir e melhorar processos.
- **Escopo:** limita o sistema ou um conjunto de sistemas através de comportamentos e aspectos que pertencem ou não a este(s) sistema(s).
- **Planejamento Técnico:** envolve planejamento do projeto e fornece base para outras funções de gerenciamento como rastreio e controle.
- **Gerenciamento de Risco Técnico**: fornece processo, método, ferramentas e infra-estrutura de recursos e responsabilidades organizacionais para identificar riscos e implementar as ações para tratá-los.
- **Suporte de Ferramentas:** aplica ferramentas familiares e fornece suporte para, concorrentemente, criar, manter e usar múltiplas versões de artefatos das linhas de produtos de software.

(3) Área das Práticas de Gerenciamento Organizacional: é constituída pelas atividades ou itens necessários para coordenar os esforços para o desenvolvimento da linha de produtos de software como um todo. São eles:

- **Caso de Negócio:** ferramenta que auxilia na tomada de decisões através de prognósticos de como estas decisões afetam a organização.

- **Gerenciamento da Interface com Cliente:** garante o treinamento apropriado daqueles que tem a responsabilidade de interagir com o cliente.

- **Estratégia de Aquisição:** consiste de um processo para aquisição de produtos e serviços através de contrato. Esta atividade aplica-se às organizações que preferem adquirir os produtos ao invés de desenvolvê-los.

- **Lançamento e Institucionalização:** auxiliam as mudanças organizacionais, preparando as organizações para adotar novas tecnologias ou a seguir novos caminhos para os negócios.

- **Análise de Mercado:** consiste da análise e pesquisa de fatores externos que determinam o sucesso do produto no mercado.

- **Operações:** reúne as estratégias de gerenciamento, políticas, processos de negócio e planos de trabalhos em práticas coerentes e unificadas para fazer com que a organização atinja seu objetivo.

- **Capital:** estabelece como o desenvolvimento da linha de produtos de software é financiado. Os recursos financeiros são necessários para preparar a organização para a abordagem da linha de produtos de software e envolvem treinamento, diferentes processos de desenvolvimento e diferentes práticas de gerenciamento.

- **Planejamento Organizacional:** consiste de um plano para a adoção da linha de produtos de software e um plano de financiamento para o desenvolvimento e manutenção da base do *core assets.*

- **Gerenciamento de Risco Organizacional:** é similar ao gerenciamento de risco técnico da Área das Práticas de Gerenciamento Técnico, porém, em nível estratégico.

- **Estruturação da Organização:** define como a organização forma os grupos para conduzir as responsabilidades inerentes no desenvolvimento da linha de produtos de software.

- **Tecnologia de Previsões:** ajuda a realizar um planejamento de mercado estratégico. Identifica as tendências e indica as previsões relevantes do mercado, diminui riscos relacionados a inovações, fornece base para planejamento e direciona investimentos em pesquisas e áreas de desenvolvimento.

- **Treinamento:** fornece habilidades e conhecimentos necessários para desenvolver o gerenciamento e os papéis técnicos da linha de produtos de software.

A Figura 3.4 mostra o relacionamento entre as três categorias apresentadas.

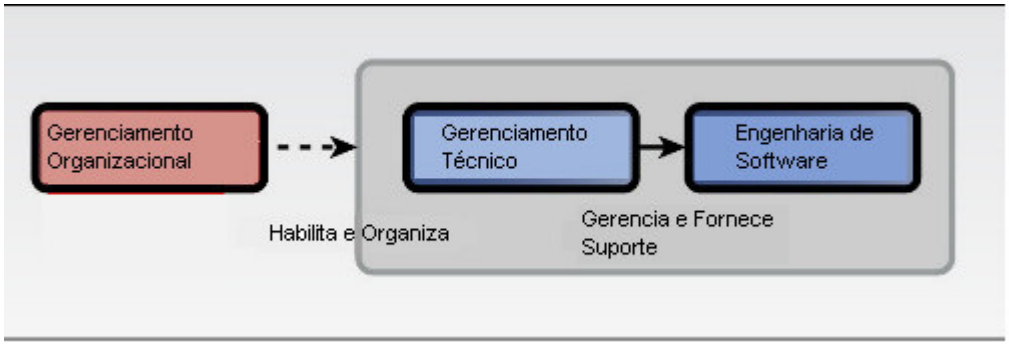


Figura 3.4 - Relacionamento entre as categorias das Áreas das Práticas
Fonte Adaptada: (NORTHROP, CLEMENTS, 2007)

3.3.2 *Evolutionary Software Product Line Engineering Process* – ESPLEP

Gomaa (2005) apresenta o *Evolutionary Software Product Line Engineering Process* - ESPLEP (Processo de Engenharia de Linha de Produtos de Software Evolucionário), com a representação dos modelos baseada em UML. Este processo é constituído por duas atividades principais: Engenharia de Linha de Produtos de Software e Engenharia de Aplicação de Software.

A Engenharia de Linha de Produtos de Software consiste do desenvolvimento de modelo de casos de uso de linha de produtos, modelo de análise de linha de produtos, arquitetura de linha de produtos e componentes reusáveis. Testes são executados nos componentes e nas configurações de aplicações da linha de

produtos de software. Os artefatos produzidos durante este processo são armazenados no repositório de linha de produtos de software.

Durante a Engenharia de Aplicação, é desenvolvida uma aplicação individual, considerada um membro da linha de produtos de software. Esta atividade consiste de definição dos requisitos da aplicação, e da derivação de modelo de casos de uso da aplicação, modelo de análise da aplicação e arquitetura da aplicação de software, a partir dos respectivos modelos da Engenharia de Linha de Produtos de Software. Com a arquitetura da aplicação e com os componentes apropriados, produzidos durante a atividade da Engenharia de Linha de Produtos de Software e armazenados no repositório, a aplicação executável é implantada. A Figura 3.5 apresenta esquematicamente o ESPLEP.

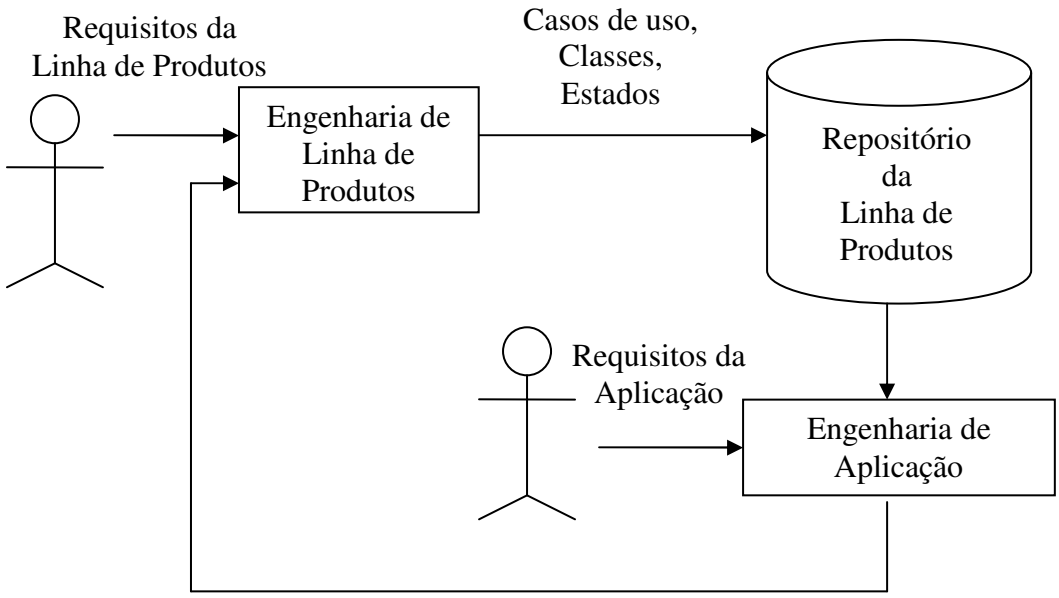


Figura 3.5 - Processo de Engenharia de Linha de Produtos de Software Evolucionário
Fonte Adaptada: GOMAA, 2005

Tem se, a seguir, a descrição mais detalhada destas atividades.

3.3.2.1 As Fases da Engenharia de Linha de Produtos de Software

Os requisitos da linha de produtos de software são entradas para esta atividade e os artefatos produzidos são armazenados no repositório da linha de produtos de software para serem utilizados como entradas da atividade de Engenharia de Aplicação.

As fases que compõem esta atividade são: (1) modelagem de requisitos, (2) modelagem de análise, (3) modelagem de projeto, (4) implementação incremental de componentes e (5) testes. A descrição destas fases é feita a seguir:

- **1. Modelagem de Requisitos:** Nesta fase são elaborados os modelos de casos de uso e de *features.* O modelo de casos de uso define os requisitos funcionais da linha de produtos de software através de atores e casos de uso. Entretanto, este modelo é estendido para representar os elementos comuns e a variabilidade na linha de produtos de software, através da identificação de casos de uso de núcleo (*kernel*), opcionais e alternativos. Um modelo de *features* também é desenvolvido, sendo representado em UML (GRISS, FAVARO, D´ALESSANDRO,1998). A modelagem de requisitos para linha de produtos de software é dividida em três atividades principais:

    - **Definição do escopo da linha de produtos**: são definidos, em alto nível, a funcionalidade, os elementos comuns e variabilidade e um número provável de membros da linha de produtos de software.

    - **Modelagem de caso de uso**: os requisitos funcionais das linhas de produtos de software são especificados através de casos de uso e atores. A funcionalidade comum da linha de produtos de software é definida pelos casos de uso núcleo. A variabilidade da linha de produtos de software é definida a partir de casos de uso opcionais e alternativos, e pela identificação de pontos de variação dentro dos casos de uso, que é o local no caso de uso onde a variação pode ser ocorrer (JACOBSON, GRISS, JOHNSON apud GOMAA, 2005). Os casos de uso núcleo são aqueles requeridos em todas as aplicações de uma linha de produtos de software. Os casos de uso opcionais são

aqueles requeridos por algumas aplicações, não todas. Os casos de uso alternativos são diferentes versões de um caso de uso e são requeridos em diferentes aplicações da linha de produtos de software; por exemplo: para uma aplicação de vendas, pode-se ter um caso de uso alternativo Emitir Conta para pagamentos realizados no local da venda; e um caso de uso alternativo Emitir Fatura para pagamentos de vendas remotas. Os casos de uso, classificados desta forma, são representados no diagrama através de estereótipos.

- **Modelagem de *feature***: As *features* representam as características comuns e a variabilidade da linha de produtos de software. As *features* podem ser identificadas a partir dos casos de uso.

- **2. Modelagem da Análise:** Nesta fase são desenvolvidos os modelos estáticos, dinâmicos e de máquinas de estados da linha de produtos de software. As atividades da modelagem de análise são:

  - **Modelagem estática**: O modelo estático define o relacionamento estrutural entre as classes do domínio, através do diagrama de classes. Este modelo contém a representação das características comuns e variabilidade da linha de produtos de software categorizando as classes através de estereótipos.

  - **Modelagem dinâmica**: O modelo dinâmico é desenvolvido para realizar os casos de uso; identifica os objetos que participam em cada caso de uso e as interações entre estes. São utilizados diagramas de interação, que podem ser diagramas de comunicação ou de seqüência.

  - **Modelagem de máquina de estados finitos**: Os objetos que são dependentes de estados são definidos por meio de máquinas de estados.

  - **Análise de dependência entre *feature* e classe**: esta atividade verifica se todas as *features* foram representadas pelas classes.

- **3. Modelagem de Projeto:** O modelo de análise (que enfatiza o domínio do problema) é mapeado para o modelo de projeto (que enfatiza o domínio da solução). É realizado projeto baseado em *patterns* arquiteturais e a arquitetura de software baseada em componentes da linha de produtos de software é elaborada.

- **4. Implementação Incremental de Componentes:** Nesta fase, são implementados incrementos, representados por subconjuntos selecionados de casos de uso de linha de produtos de software. A implementação inicia-se com os casos de uso núcleo, seguido pelos opcionais e alternativos.

- **5. Testes:** Nesta fase, são realizados os testes de integração e testes funcionais da linha de produtos de software. Durante os testes de integração, os componentes de cada incremento são testados em conjunto. Testes funcionais são realizados para cada caso de uso.

3.3.2.2 As Fases da Engenharia de Aplicação

Para o desenvolvimento da aplicação, são utilizados os artefatos desenvolvidos durante o processo de Engenharia de Linha de Produtos de Software, armazenados no repositório. As fases do processo de Engenharia de Aplicação são:

- **Modelagem de Requisitos da Aplicação:** O modelo de requisitos da aplicação é elaborado: seus requisitos funcionais são definidos em termos de atores e casos de uso. Os requisitos da aplicação auxiliam a determinar as *features* da linha de produtos de software que devem ser incorporadas na aplicação. Uma aplicação típica é constituída de todas as *features* relevantes a esta aplicação.

- **Modelagem de Análise da Aplicação:** São desenvolvidos os modelos estáticos e dinâmicos da aplicação. Somente as classes e relacionamentos relevantes para este membro da linha de produtos de software são selecionados.

- **Modelagem de Projeto da Aplicação:** Os componentes são selecionados e a arquitetura de software da aplicação é derivada da arquitetura da linha de produtos de software.

- **Implementação Incremental da Aplicação:** É adotada uma abordagem incremental para a implementação da aplicação. Cada incremento corresponde a um subconjunto de casos de uso da aplicação.

- **Teste da Aplicação:** São realizados os testes da aplicação através de testes de integração e testes funcionais do sistema. Nos testes de integração, é feita a integração de cada incremento. Durante os testes funcionais, o sistema é testado confrontando os requisitos funcionais.

3.4 Outras Abordagens Relevantes

Além das abordagens FSPLP (NORTHROP, CLEMENTS, 2007) e ESPLEP (GOMAA, 2005), foram consideradas outras abordagens que apresentaram métodos, técnicas e mecanismos interessantes e contribuíram para o entendimento dos conceitos e também para o desenvolvimento deste trabalho. Inicialmente, apresenta-se o FODA (KANG apud NORTHROP, CLEMENTS, 2007), um método para análise de domínio muito referenciado nas abordagens de linha de produtos de software. Em seguida, são descritas resumidamente as abordagens de: Synthesis (SPC, 1993), PuLSE (Bayer et al., 1999), FAST (HARSU, 2002) e KobrA (ATKINSON, BAYER, MUTHIG, 2000).

3.4.1 FODA

Feature-Oriented Domain Analysis (FODA) é um método de análise de domínio, desenvolvido pelo SEI (KANG apud NORTHROP, CLEMENTS, 2007), que é conhecido pela introdução de modelo de *features*. Além disso, o método FODA contribuiu de forma significativa para aumentar a popularidade da análise dirigida a

modelos na engenharia de software, pois propagou a utilização de visões complementares de um dado domínio, conduzindo a obtenção de informações mais completas sobre o mesmo (GRISS, FAVARO, D´ALESSANDRO,1998).

O processo do método FODA é constituído por duas fases (CZARNECKI, EISENECKER, 2005):

1. Análise de contexto: o propósito é definir os limites de domínio a ser analisado. Nesta fase, o escopo do domínio é definido.

2. Modelagem de domínio: o propósito é produzir um modelo de domínio. Nesta fase, os principais atributos comuns e a variabilidade decorrente das aplicações são identificados e modelados. Esta fase envolve os seguintes passos:

- **Análise de informação**: captura o conhecimento do domínio, na forma de entidades de domínio e relacionamento entre eles.
- **Análise de *feature***: captura o entendimento dos usuários finais das capacidades gerais da aplicação no domínio.
- **Análise operacional**: identifica os atributos comuns e diferenças entre controle e fluxo de dados dentro da aplicação do domínio.

Um modelo de *features* do FODA consiste de quatro elementos chaves, que são (CZARNECKI, EISENECKER, 2005):

- **Diagramas de *features***: representam decomposição hierárquica de *features* com a indicação se a *feature* é mandatória, alternativa ou opcional.
- **Definições de *features***: descreve todas as *features* com informação se cada *feature* está relacionada ao tempo de compilação, tempo de ativação ou tempo de execução.
- **Regras de composição para *features***: aponta as combinações de *features* válidas e não válidas.

- ***Rationales* para *features***: indicam as razões para escolha ou não de cada *feature*.

3.4.2 Synthesis

Synthesis é um método para construção de sistemas de software com base em instâncias de família de sistemas que possuem descrições similares. Foi desenvolvido pela *Software Productivity Consortium* em 1991 (SPC, 1993).

As principais características do processo Synthesis são (SPC, 1993):

- Formalização de um domínio com uma família de sistemas que compartilham características comuns, mas variam de forma bem definida;
- Construção de sistemas reduzidos para atender requisitos e decisões de engenharia representando as características de variação do domínio;
- Reuso de artefatos de software através de adaptações de componentes para satisfazer os requisitos e decisões de engenharia;
- Análise baseada em modelos de sistemas para entender as implicações de decisões de construção de sistemas e avaliar as possíveis alternativas.

Similarmente a outros métodos de linha de produtos de software, também é dividido em dois processos: Engenharia de Domínio e Engenharia de Aplicação. Além disso, Synthesis se baseia em quatro conceitos principais: família de produtos, processos iterativos, especificações completas e precisas das propriedades de um produto e reuso baseado em abstrações (GIMENES, TRAVASSOS, 2002).

A Figura 3.6 ilustra o processo de Synthesis.

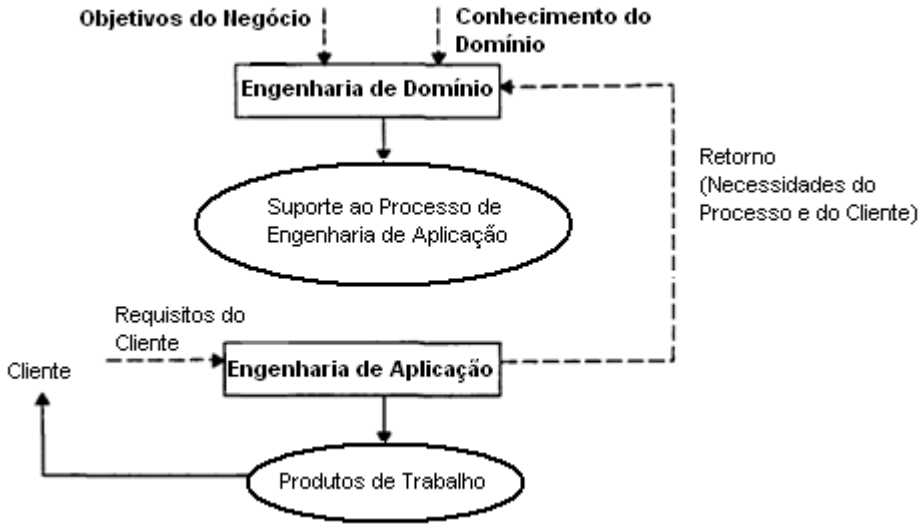


Figura 3.6 - O processo Synthesis
Fonte Adaptada: (SPC, 1993)

Tanto o processo de Engenharia de Domínio quanto o de Engenharia de Aplicação são iterativos, já que ambos devem fornecer suporte às necessidades envolvidas. A Engenharia de Domínio deve ser iterativa porque tanto as mudanças de mercado quanto os requisitos de clientes conduzem a evolução de produtos e necessidades da Engenharia de Aplicação. A Engenharia de Aplicação é iterativa para acomodar as alterações de requisitos de clientes (SPC, 1993).

3.4.3 PuLSE

PuLSE (*Product Line Software Engineering*) define uma metodologia para construção e utilização de linha de produtos de software também com base nas atividades de Engenharia de Domínio e Engenharia de Aplicação. Esta abordagem foi desenvolvida em 1999 pelo *Fraunhofer Institute for Experimental Software Engineering* – IESE (Bayer et al., 1999).

A metodologia PuLSE habilita o conceito de implantação de linha de produtos de software dentro de uma grande variedade de contextos empresariais. Isto é possível através do foco em produto através das fases apresentadas por esta metodologia, da customização de seus componentes, de uma escala de maturidade para

evolução empresarial estruturada e das adaptações para desenvolvimento de produtos principais (Bayer et al., 1999).

De acordo com a Figura 3.7, PuLSE é baseada em três elementos principais: fase de implantação, componentes técnicos e componentes de suporte (Bayer et al., 1999).

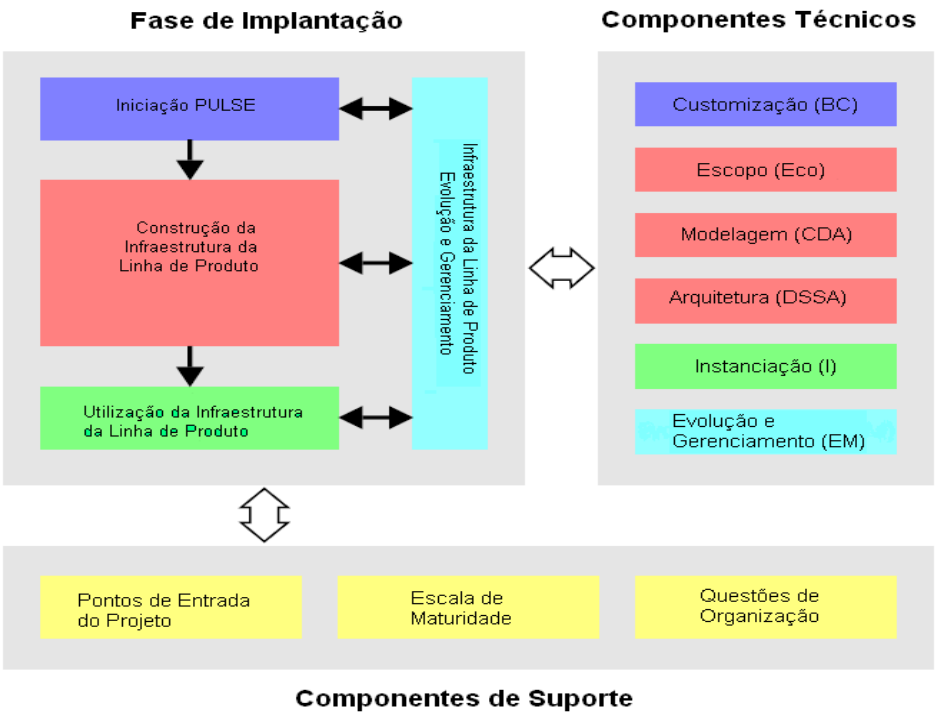


Figura 3.7 - Visão geral da metodologia PuLSE

A Fase de Implantação é constituída pelas atividades de iniciação, construção da infra-estrutura da linha de produtos de software, sua utilização, evolução e gerenciamento desta infra-estrutura.

Os Componentes Técnicos fornecem conhecimento para operacionalizar o desenvolvimento da linha de produtos de software. Os componentes técnicos são relacionados com: customização, definição do escopo, modelagem, desenvolvimento da arquitetura, instanciação e evolução e gerenciamento.

Os Componentes de Suporte são pacotes de informações, que auxiliam na melhor adaptação, evolução e implantação da linha de produtos de software. Estão relacionados com os componentes de suporte: escala de maturidade, itens de

organização e os pontos de entradas de projeto. Os pontos de entradas de projeto descrevem as situações padrões onde PuLSE pode ser aplicado.

3.4.4 FAST

A abordagem FAST (*Family-Oriented Abstraction, Specification and Translation*) foi introduzida por David Weiss no início dos anos de 1990 e mais tarde desenvolvida pelo *Lucent Technologies Bell Laboratories* em 1999 (HARSU, 2002).

FAST é uma alternativa ao desenvolvimento tradicional de software, podendo ser aplicada em qualquer contexto que possua múltiplas versões de produtos que compartilham conjuntos de atributos comuns, tais como comportamentos comuns, interfaces comuns e códigos comuns (MATINLASSI, 2004).

O principal objetivo de FAST é fazer com que o processo de engenharia de software se torne mais eficiente, através da redução de esforços, custo de produção e diminuição do tempo de chegada ao mercado (*time-to-market*) (MATINLASSI, 2004), (HARSU, 2002).

A abordagem FAST define um processo completo para a engenharia de linha de produtos de software com atividades e artefatos.

A Figura 3.8 ilustra o processo de FAST.

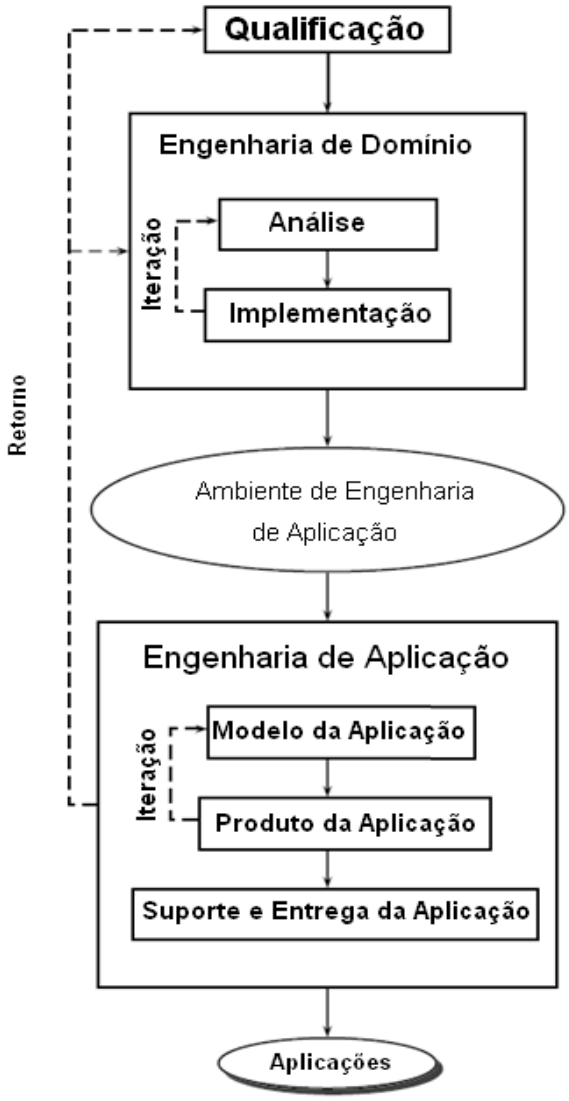


Figura 3.8 - Processo FAST
Fonte Adaptada: (WEISS, LAI apud HARSU, 2002)

A qualificação de domínio identifica as famílias adequadas ao investimento, através de uma análise do ponto de vista econômico. Estima o número de membros das famílias de software e o custo para produzí-las.

A Engenharia de Domínio estuda como os produtos da mesma família compartilham a base de recursos comum e como eles se diferem entre si. Isto é realizado através da análise de domínio e da implementação de domínio. A análise de domínio produz um modelo de domínio que, no contexto FAST, significa uma especificação e implementação de conceitos para um ambiente denominado ambiente de engenharia de aplicação. A implementação do domínio desenvolve e refina este

ambiente, que satisfaz o modelo de domínio. Este ambiente também fornece suporte para a Engenharia de Aplicação.

A Engenharia de Aplicação utiliza o ambiente de engenharia de aplicação para produzir membros das famílias de produtos de forma eficiente. Os requisitos da aplicação são representados em um modelo de aplicação, que pode ser refinado várias vezes. De acordo com este modelo, são gerados os produtos da aplicação que são conjunto de código e documentação. Na fase de entrega e suporte da aplicação, o cliente avalia a aplicação recebida e se não estiver de acordo, os requisitos são refinados e todo o processo inicia-se novamente.

3.4.5 KobrA

O acrônimo KobrA originou-se do termo em alemão *Komponentenbasierte Anwendungsentwicklung* que significa desenvolvimento de aplicação baseado em componente (ATKINSON et al. apud MATINLASSI, 2004). O método KobrA denota uma abordagem de desenvolvimento de linha de produtos de software incremental baseada em componentes e foi criado pelo *Fraunhofer Institute for Experimental Software Engineering* – IESE, em 2000 (ATKINSON, BAYER, MUTHIG, 2000).

O método KobrA é uma composição de várias tecnologias avançadas de engenharia de software, incluindo desenvolvimento de linha de produtos de software, desenvolvimento de software baseado em componentes, *frameworks*, inspeção centrada em arquiteturas, modelagem de qualidade e modelagem de processo (ATKINSON, BAYER, MUTHIG, 2000). A Figura 3.9 ilustra as atividades envolvidas no método KobrA.

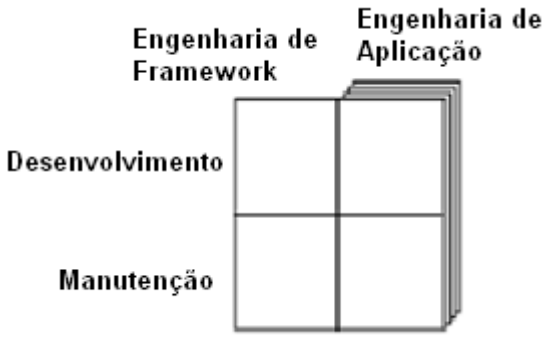


Figura 3.9 - Atividades de KobrA
Fonte Adaptada: (ATKINSON, BAYER, MUTHIG, 2000)

Como mostra a Figura 3.9, um dado *framework* pode ser instanciado múltiplas vezes para criar múltiplas aplicações (ATKINSON, BAYER, MUTHIG, 2000).

O propósito da atividade de Engenharia de *Framework* é desenvolver e posteriormente manter um *framework* genérico que envolve todas as variações de produtos que constituem a família, incluindo as informações entre as características comuns e variabilidade.

A finalidade da atividade de Engenharia de Aplicação é instanciar este *framework* para criar variantes particulares na família de produtos, cada uma de acordo com necessidades específicas de diferentes clientes. Estas variantes podem ainda ser mantidas posteriormente.

No método KobrA, um *framework* é uma representação estática de um conjunto de *Komponents* (abreviatura de componentes do método KobrA - *KobrA component*). A Figura 3.10 ilustra a especificação e realização de um *Komponent.*

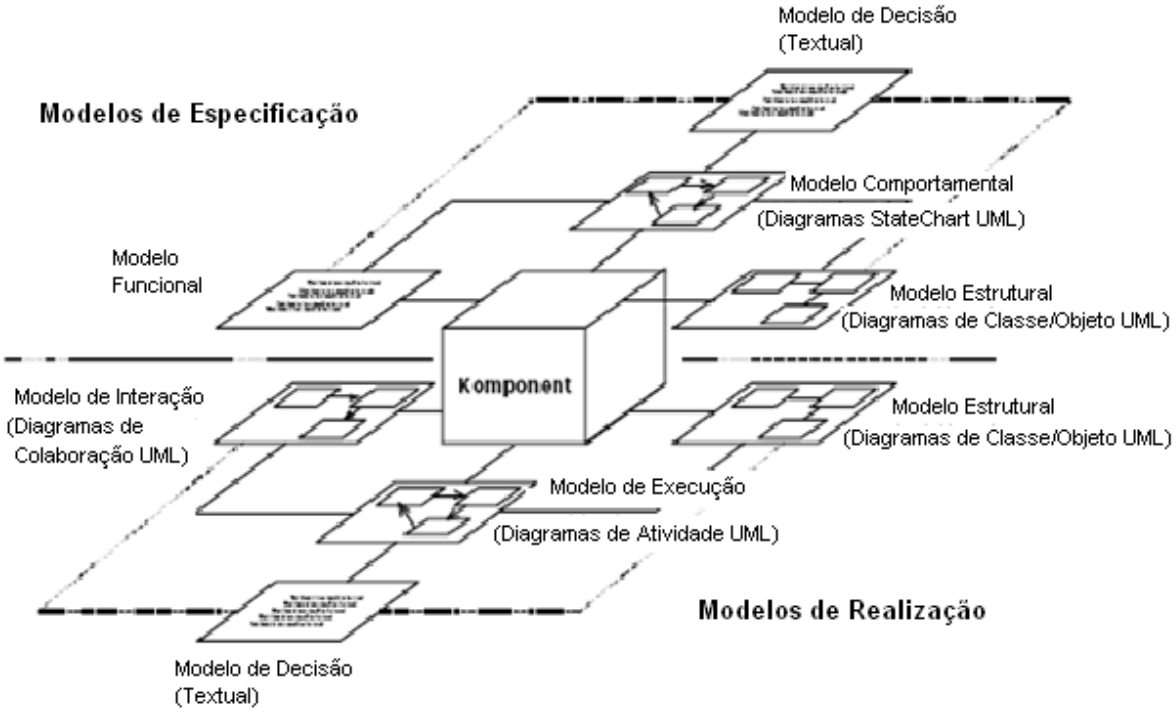


Figura 3.10 - Especificação e realização de *Komponent*
Fonte Adaptada: (ATKINSON, BAYER, MUTHIG, 2000)

Cada *Komponent* é descrito em dois níveis de abstração: o nível de especificação e o nível de realização. Os modelos do *Komponent* são representados por um conjunto de diagramas UML.

O nível de especificação é responsável por criar um conjunto de modelos que descreve as propriedades visíveis externamente e comportamentos de um *Komponent*. Este conjunto de modelos é formado por: modelo estrutural, modelo comportamental, modelo funcional e modelo de decisão. Os modelos estrutural, comportamental e funcional são modelos de especificação para um *Komponent;* o modelo de decisão contém informações de como os modelos são alterados para diferentes aplicações.

O nível de realização é responsável por criar um conjunto de modelos que descreve o projeto de um *Komponent*. Este conjunto de modelos é formado por: modelo de interação, modelo estrutural, modelo de atividade e modelo de decisão.

3.5 Benefícios e Custos da Linha de Produtos de Software

Os benefícios que podem ser obtidos ao se utilizar a abordagem da linha de produtos de software, ao invés de um desenvolvimento individual para cada sistema são (NORTHROP, CLEMENTS, 2007), (POHL, BÖCKLE, LINDEN, 2005):

- Redução de custos em produção em larga escala: um grande investimento é feito na especificação da arquitetura, que é o passo mais importante da fase de projeto. Esta arquitetura será reusada para a produção de novos produtos, assim como, serão reusadas outras decisões de projeto, código, e documentação. Quando estes artefatos são reusados na produção de diferentes tipos de aplicação, implica em uma redução de custo para cada aplicação.

- Diminuição do *time-to-market*: diferentemente do desenvolvimento de software convencional, o *time-to-market* não é constante. Este tempo deve ser mais alto inicialmente já que os artefatos comuns devem ser desenvolvidos; após esta fase, o *time-to-market* diminui consideravelmente, já que muitos artefatos podem ser reusados para cada novo produto.

- Aumento da qualidade do produto: cada novo produto leva vantagem da eliminação de defeitos através da produção anterior e, assim, uma melhora deve ocorrer para cada nova instanciação. Também, para cada linha de produtos de software há um plano de produção e um guia que especifica a abordagem da produção do produto correspondente. É um reuso planejado e não acidental.

- Diminuição dos riscos do produto: artefatos como planos de testes, casos de testes e testes são desenvolvidos e revisados para os componentes que serão reusados em muitos produtos, detectando as falhas e corrigindo-as.

- Aumento da satisfação do cliente: a linha de produtos de software possibilita alta qualidade nos produtos de software, prazos obedecidos para entrega e material de treinamento e documentação bem avaliados.

- Melhoria da utilização dos recursos humanos: devido à funcionalidade comum dos produtos e ao processo de produção, a experiência e produtividade aumentam. Além disso, recursos para treinamento para usar ferramentas, processos e componentes do sistema são gastos menos vezes.

Entretanto, como em qualquer abordagem de desenvolvimento de software, certos custos também devem ser considerados. No caso da linha de produtos de software, um exemplo de custo é o relacionado com a captura de requisitos para um grupo de sistemas. Esta tarefa exige uma análise complexa, pois se deve especificar os requisitos comuns e variáveis que sejam adequados para todos os sistemas.

Deve-se considerar também a dificuldade na definição de uma arquitetura que forneça suporte às diversas variações solicitadas pelas aplicações da linha de produtos de software. Profissionais experientes devem ser alocados para definir esta arquitetura.

Além disso, as definições de processo, ferramentas, procedimentos e testes devem ser extensíveis para acomodar as variações entre produtos, ou seja, estes artefatos precisam ser mais robustos para fornecer suporte a mais de um produto.

3.6 Considerações Finais

Este capítulo apresentou o resultado dos estudos realizados sobre linha de produtos de software. Dentre as diversas abordagens avaliadas, destacam-se FSPLP (NORTHROP, CLEMENTS, 2007) e ESPLEP (GOMAA, 2005).

A abordagem FSPLP foi selecionada como base para este trabalho porque é considerada referência para vários trabalhos de linha de produtos de software e vem evoluindo continuamente, através de resultados de estudos de instituições que desenvolvem linha de produtos de software e projetos realizados em colaborações diretas entre o SEI e seus clientes. O ESPLEP foi selecionado como base para este trabalho por ser uma abordagem que fornece uma visão bem detalhada do processo de linha de produtos de software.

Do FSPLP foram adotados a definição dos conceitos da linha de produtos de software e o modelo das atividades essenciais, que consiste de Engenharia de Domínio, Engenharia de Aplicação e Gerenciamento. Suas Área das Práticas forneceram informação importante para o entendimento do assunto e para a definição do processo GVLPS, proposto neste trabalho.

Gomaa (2005) por sua vez, apresenta um processo organizado em Engenharia de Domínio e de Aplicação, portanto consistente com o FSPLP, com a representação dos artefatos através de diagramas de UML. Descreve em detalhes a fases de análise e projeto, enfatizando o desenvolvimento e da arquitetura de linha de produtos de software. O autor mostra também como diagramas de casos de uso, diagramas estáticos, máquinas de estados e diagramas de interação dinâmica são utilizados na modelagem de linha de produtos de software. Assim, do ESPLEP foram adotadas as representações em UML para os artefatos do processo GVLPS.

As outras abordagens citadas também apresentam pontos interessantes como as diferentes fases descritas pelo método PuLSE e o desenvolvimento da aplicação baseado em componentes de KobrA. Além disso, todos esses métodos são constantemente citados em vários trabalhos (CZARNECKI, EISENECKER, 2005), (OLIVEIRA JUNIOR, 2005), (MATINLASSI, 2004), (TRIGAUX, HEYMANS, 2003), (GIMENES, TRAVASSOS, 2002) como métodos relevantes para o desenvolvimento de linha de produtos de software. Entretanto, estas abordagens não foram utilizadas como base deste trabalho porque a maioria delas também se baseia nas atividades de Engenharia de Domínio e Engenharia de Aplicação, assim como FSPLP e ESPLEP que detalham de forma bem abrangente estas atividades.

Assim, como estas outras abordagens foram pesquisadas, com o objetivo de selecionar o foco mais adequado para este trabalho, foram apresentadas resumidamente neste capítulo.

"Aproximem-se da beirada – disse-lhes.
Não podemos, temos medo – responderam.
Aproximem-se da beirada – repetiu.
Não podemos, cairemos – queixaram-se.
Aproximem-se da beirada – insistiu.
E se aproximaram. Ele os empurrou,
e eles voaram."

Guillaume Apollinaire

4 VARIABILIDADE EM LINHA DE PRODUTOS DE SOFTWARE

Nas últimas décadas, a necessidade por sistemas de software que apresentem maior capacidade de variação aumentou, devido à pressão do mercado por desenvolvimento econômico e em tempo mais rápido. Uma das soluções para esta demanda foi a filosofia de linha de produtos de software que utiliza a variabilidade para gerenciar as diferenças entre os produtos. Entretanto, uma das conseqüências desta abordagem é deslocar as decisões de projeto para etapas mais adiantadas do projeto dos sistemas.

A tarefa de introduzir variabilidade da linha de produtos de software não é trivial, pois vários fatores tais como, tamanho de elemento de software, instante para tomada de decisões de projeto e ambiente previsto de execução influenciam as escolhas nas decisões relacionadas com variabilidade (SVAHNBERG, VAN GURP, BOSCH, 2002).

Além disso, é importante considerar que a variabilidade não necessariamente é representada apenas na arquitetura ou no código fonte do sistema. Pode também ser representada como procedimentos durante o processo de desenvolvimento, fazendo uso de ferramentas externas ao sistema que está sendo construído (SVAHNBERG, VAN GURP, BOSCH, 2002).

Para que a abordagem de linha de produtos de software tenha os resultados esperados, é necessário que atividades tais como identificação da variabilidade, atribuição de restrições e implementação da variabilidade sejam definidas, realizadas e gerenciadas.

Neste capítulo são apresentados os conceitos sobre variabilidade, modelagem de *features*, processo de gerenciamento de variabilidade e mecanismos de variabilidade.

4.1 Conceitos sobre Variabilidade

Nesta seção são apresentados os conceitos relativos à variabilidade, ponto de variação, parte variável, variante, variação e mecanismo de variabilidade.

São encontradas, na literatura, várias definições sobre variabilidade que apresentam algumas diferenças entre si:

1. A variabilidade é a capacidade de um sistema ser alterado ou customizado. Melhorar a variabilidade em um sistema significa tornar mais fácil realizar certos tipos de mudanças (VAN GURP, BOSCH, SVAHNBERG, 2001).

2. A variabilidade é a habilidade dos artefatos de software variarem seu comportamento em algum ponto de seu ciclo de vida (SVAHNBERG, VAN GURP, BOSCH, 2002).

3. A variabilidade é a forma pela qual os membros de uma linha de produtos de software podem se diferenciar entre si (WEISS, CHI TAU apud OLIVEIRA JUNIOR, 2005).

4. A variabilidade é a habilidade de certos artefatos, pertencentes ao *core assets,* serem adaptados ou se adaptarem para serem utilizados em diferentes produtos da linha de produtos de software (BACHMANN, CLEMENTS, 2005).

Para este trabalho, adotou-se a definição apresentada por Van Gurp, Bosch e Svahnberg (2001), pois observou-se que é a definição mais adotada pelos trabalhos da área.

O ponto de variação também é definido de diferentes formas pelos autores:

1. O ponto de variação é o local do sistema em que os artefatos variáveis são relacionados com o restante do sistema (BACHMANN, CLEMENTS, 2005).

2. Segundo Gomaa (2005), é o local onde a mudança pode ocorrer em um artefato da linha de produtos de software, como por exemplo, em um caso de uso ou em uma classe.

3. O trabalho de Jacobson, Griss, Johnsson apud Czarnecki, Eisenecker, (2005) define o ponto de variação como um ou mais locais em que a variação irá ocorrer.

Adotou-se a definição de ponto de variação apresentada por Bachmann e Clements (2005) por considerá-la melhor apropriada para o contexto deste trabalho.

Bachmann e Clements (2005) citam que alguns autores preferem chamar ponto de variação como parte variável, porque, além da parte variável ser o local (um ponto) no *core asset* que é permitido variar, é também definida como um contêiner para artefatos usado para fazer adaptações em produtos, tais como, mecanismos de variabilidade, descrições de processo e variantes. Para este trabalho, adotou-se que a parte variável é composta do ponto de variação e suas variantes.

As variantes constituem um outro conceito relevante e podem ser definidas como as diferentes versões de um mesmo componente (GOMAA, WEBBER, 2004). São as possíveis instâncias em um ponto de variação. A forma como duas ou mais variantes se diferem entre si, é chamada de variação.

A variante é obtida através do mecanismo de variabilidade, também chamado mecanismo de variação. Um mecanismo de variabilidade é aquele responsável por realizar, ou seja, implementar a variabilidade (SVAHNBERG, VAN GURP, BOSCH, 2002); (BECKER, 2003); (BRAGANÇA, MACHADO, 2004b); (SCHNIEDERS, 2006a); (TRIGAUX, HEYMANS, 2003) e (GOMAA, WEBBER, 2004). Adotou-se a definição de Bachmann e Clements (2005) que explica melhor o mecanismo de variabilidade: é aquele que permite a criação e/ou seleção de variantes que estão de acordo com as restrições de uma parte variável de um *core asset.*

Juntando-se estes conceitos, a variabilidade pode ser representada por pontos de variação, cada um está associado a um conjunto de variantes, e cada variante

corresponde a uma alternativa projetada para realizar (instanciar) uma determinada variabilidade (TRIGAUX, HEYMANS, 2003).

Trabalhos como (CZARNECKI, EISENECKER, 2005), (GOMAA, 2005), (OLIVEIRA, et al., 2005), (VAN GURP, BOSCH, SVAHNBERG, 2001) e (BACHMANN, BASS, 2001) classificam as variantes e apresentam informação na sua representação. Com base nestes trabalhos, as variantes associadas a um ponto de variação podem ser classificadas em:

- Mandatória: quando uma variante é obrigatória;
- Opcional: quando uma variante pode ser necessária ou não;
- Alternativa Inclusiva: quando se deve escolher uma ou mais variantes;
- Alternativa Exclusiva: quando somente uma das variantes é necessária;
- Alternativa Mutuamente Inclusiva: quando duas ou mais variantes são sempre necessárias juntas.

A Figura 4.1 apresenta a definição da variabilidade na linha de produtos de software, baseada no trabalho de Bachmann e Clements (2005).

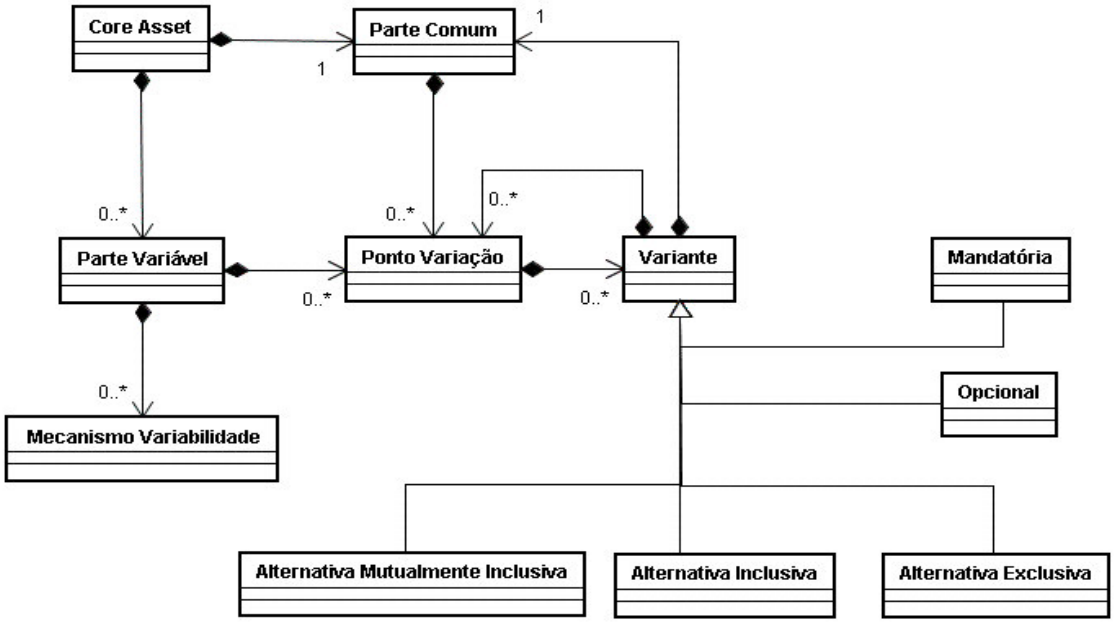


Figura 4.1 - Variabilidade na linha de produtos de software
Fonte Adaptada: (BACHMANN, CLEMENTS 2005)

A Figura 4.1 mostra que um *core asset* é composto por uma parte comum e pode ser constituído por zero ou mais partes variáveis. Se um *core asset* não possuir nenhuma parte variável, ele pode ser utilizado da forma como se apresenta; se o *core asset* possuir uma ou mais partes variáveis, alguns ajustes podem ser necessários. Se o *core asset* não possuir nenhuma parte comum, tudo deverá ser especificamente implementado para um determinado produto, o que contradiz a filosofia da linha de produtos de software.

A parte variável pode ser composta de zero ou muitos pontos de variação e um ponto de variação pode ser constituído por zero ou muitas variantes. As variantes podem ser mandatória, opcional, alternativa inclusiva, alternativa exclusiva ou mutuamente inclusiva.

Toda parte variável pode ser composta por zero ou mais mecanismos de variabilidade. A parte variável utiliza o mecanismo de variabilidade para criar ou selecionar as variantes de seus pontos de variação, ou seja, implementar a variabilidade.

A variante também pode ser composta de uma parte comum e de zero ou mais pontos de variação. Por exemplo, um *core assets* de um plano de projeto contém um ponto de variação que representa a dependência com o sistema operacional; possui como variantes duas listas de tarefas, cada uma para diferentes sistemas operacionais. Cada lista de tarefa variante possui uma parte comum, mas também uma parte variante que depende da interface a ser utilizada.

Por outro lado, a parte comum também pode apresentar pequenas variações representadas por pontos de variações.

Segundo Bachmann e Bass (2001) as variações podem ser categorizadas de acordo com a situação em que elas ocorrem:

- **Variação em função**: quando uma determinada função pode existir em um produto e não existir em outros.

- **Variação em dados**: quando uma estrutura de dados pode variar de um produto para outro, o que muitas vezes pode ocorrer em conseqüência da variação em função.

- **Variação em fluxo de controle**: quando uma interação padrão pode variar de um produto para o outro; por exemplo, em um mecanismo de comunicação entre componentes, um determinado conjunto de componentes pode ter uma comunicação com sistema de recuperação de erros.

- **Variação em tecnologia**: quando a plataforma (sistema operacional, hardware, *middleware*, interface, linguagem de programação) pode variar.

- **Variação em metas de qualidade**: quando uma meta importante de qualidade pode variar de produto para produto, dependendo da necessidade de cada um.

- **Variação em ambiente**: quando a forma de interação de um produto com o ambiente pode variar. Por exemplo, um determinado *middleware* pode ser chamado em Java ou C++; dependendo da linguagem escolhida acarretará em um mecanismo de chamada diferente.

4.2 Modelagem de *Features*

A modelagem de *features* é uma contribuição, originária da área da engenharia de domínio para a engenharia de software; que modela as propriedades comuns e variáveis de conceitos e suas interdependências organizando-as em um modelo coerente. Os modelos de *features* produzidos têm como objetivo fornecer uma representação abstrata, concisa e explícita da variabilidade presente no software (CZARNECKI, EISENECKER, 2005).

Segundo Czarnecki, Eisenecker (2005, p.38) existem duas definições para *feature* na literatura de Engenharia de Domínio:

(1) “Uma característica do sistema visível ao usuário final.”

(2) “Uma característica distinguível de um conceito que é relevante para os participantes deste conceito.”

Segundo Van Gurp, Bosch, Svahnberg (2001), uma *feature* pode ser definida como uma abstração de um requisito do sistema e a construção de um conjunto de *features* é o primeiro passo para interpretar e organizar os requisitos.

As *features* podem ser classificadas em (VAN GURP, BOSCH, SVAHNBERG, 2001), (GRISS, FAVARO, D´ALESSANDRO,1998):

- **mandatórias** - são as características essenciais, que identificam um produto;
- **opcionais** - são aquelas características que podem ser ou não necessárias ao sistema;
- **variantes** - correspondem a formas alternativas de configurar uma *feature* mandatória ou opcional;
- **externas** - são as oferecidas por uma plataforma alvo do sistema, não fazem parte diretamente do sistema, porém o utilizam e dependem deste sistema;
- do tipo **pontos de variação** - são aquelas *features* que podem atuar com ponto de variação. A *feature* do tipo ponto de variação pode ser mandatória ou opcional.

Outros autores apresentam outras classificações, em que as *features* podem ser (GOMAA, 2005), (ANASTASOPOULOS, GACEK, 2001):

- **parametrizadas**: são aquelas que possuem um parâmetro cujo valor precisa ser definido em tempo de configuração do sistema;
- **pré-requisito:** quando uma *feature* depende de uma outra *feature*; a *feature* da qual esta depende é chamada de *feature* pré-requisito.

*Feature*s variantes podem ainda ser classificadas em:

- **Inclusiva:** quando uma ou mais *features* podem ser selecionadas;
- **mutuamente inclusiva**: quando duas *features* são sempre necessárias juntas;

- **mutuamente exclusiva**: para uma *feature* ser incluída outra(s) *feature*(*s*) específica(s) não pode(m) ser incluída(s).

No contexto de linha de produtos de software, de acordo com o trabalho apresentado em (GOMAA, SHIN, 2007) e (ANTKIEWICZ, CZARNECKI, 2004), a modelagem de *features* é uma técnica que captura e representa as características comuns e a variabilidade dos membros de uma linha de produtos de software.

O modelo de *features* é importante neste contexto, porque facilita as distinções entre as características comuns e variabilidade, ajudando os desenvolvedores a identificar em que partes o sistema apresenta variabilidade (SVAHNBERG, VAN GURP, BOSCH, 2002), (ANTKIEWICZ, CZARNECKI, 2004), (GOMAA, SHIN, 2007). Como a especificação de requisitos é o único pré-requisito para construir um modelo de *features,* a identificação da variabilidade pode ser feita no início do processo de desenvolvimento da linha de produtos de software.

Alguns trabalhos como (GOMAA, 2005), (GRISS, FAVARO, D´ALESSANDRO,1998) citam que casos de uso e *features* podem ser usados de forma complementar para identificar a variabilidade. Isto porque, a variabilidade também pode ser capturada por pontos de variação identificados na descrição dos casos de uso.

Existem várias representações, na literatura, para modelo de *features*, e elas apresentam características semelhantes (CZARNECKI, EISENECKER, 2005), (GOMAA, 2005) e (VAN GURP, BOSCH, SVAHNBERG, 2001), (GRISS, FAVARO, D´ALESSANDRO,1998).

A seguir, apresenta-se a representação de Griss, Favaro, D´Alessandro (1998), pois mostra detalhes importantes de um modelo de *features*. Um exemplo deste modelo está apresentado na Figura 4.2:

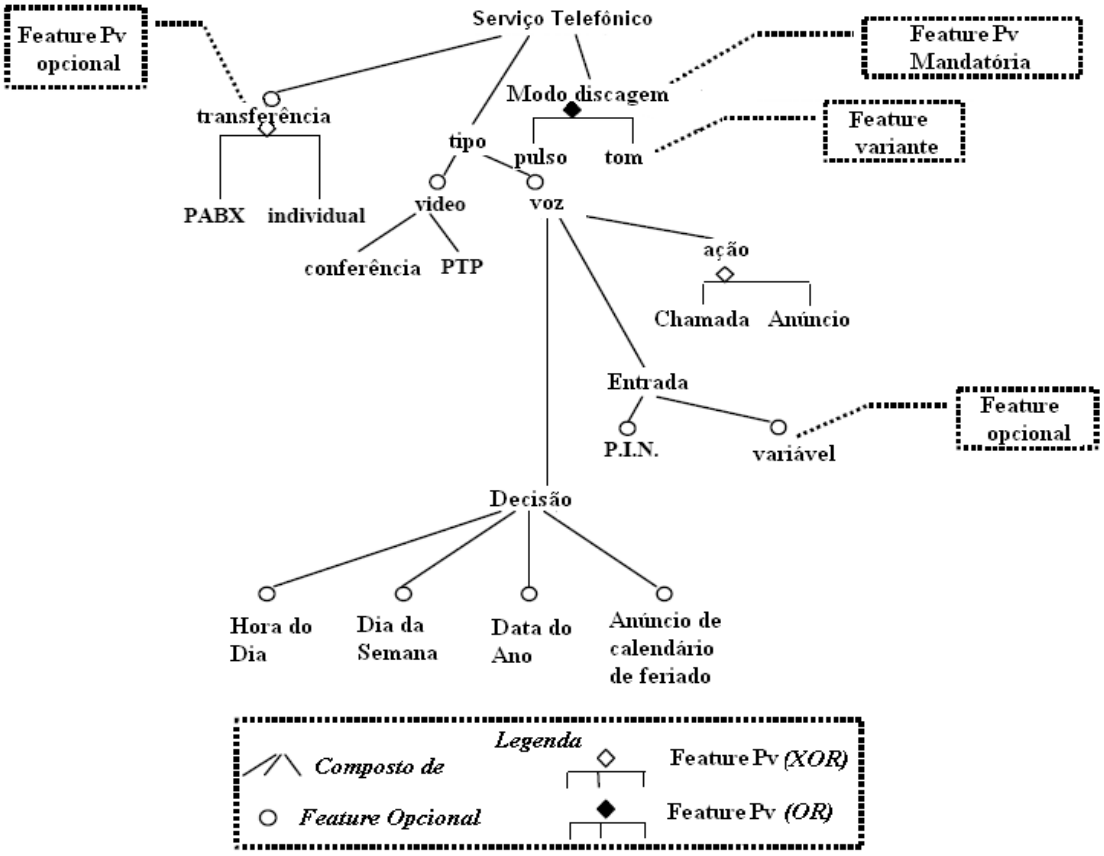


Figura 4.2 - Modelo de *features*
Fonte Adaptada: (GRISS, FAVARO, D´ALESSANDRO,1998)

De acordo com a legenda da Figura 4.2, o relacionamento Composto de significa que a *feature* pode ser composta de várias sub-*features.* Este relacionamento é representado por uma linha cheia entre a *features* e as sub-*features*. No exemplo, a *feature* serviço telefônico é composta pelas sub-*features*: transferência, tipo e modo de discagem.

Uma *feature* mandatória é representada pelo nome, por exemplo, tipo da *feature* serviço telefônico. Uma *feature* opcional é representada com um círculo acima do seu nome, por exemplo, as *features* vídeo e voz de tipo de serviço telefônico. Observa-se que quando uma *feature* mandatória é composta exclusivamente de *features* opcionais, ao menos uma das *features* opcionais deve ser selecionada. É o caso da *feature* tipo de serviço telefônico.

Nesta representação, as *features* podem também ter o papel de ponto de variação ou de variante.

A *feature* que descreve um ponto de variação é representada através do seu nome com um losango abaixo dele. O losango aberto representa uma *feature* mutuamente exclusiva (XOR) e o fechado, uma *feature* inclusiva (OR). No exemplo, a *feature* transferência é um ponto de variação opcional mutuamente exclusiva. A *feature* modo de discagem é do tipo ponto de variação mandatória inclusiva.

A *feature* variante é conectada em uma *feature* ponto de variação e não possui nenhum símbolo junto do seu nome. No exemplo, a *feature* transferência é um ponto de variação, tendo como variantes as *features* PABX e individual.

Buhne, Halmans e Pohl (2003) citam que o tipo das dependências entre as *features* também pode ser representado nos modelos de *features*. A Tabela 4.1 mostra os tipos de dependência mais relevantes nas modelagens de *features*.

Tabela 4.1 - Tipos de dependência usadas na modelagem de *features*

Tipo de Dependência	**Descrição**
Implementada por	A dependência implementada por é usada para descrever que uma *feature* é necessária para implementar outra.
Generalização/ especialização	A dependência generalização/especialização é usada para generalizar ou especializar *features*
Requer	A dependência requer é usada para descrever se uma *feature* precisa de outra.
(Mútua) Exclusiva	A dependência mutuamente exclusiva é usada quando uma *feature* entra em conflito com outra

Fonte Adaptada: (BUHNE, HALMANS, POHL, 2003)

Antkiewicz e Czarnecki (2004) apresentam uma ferramenta chamada *FeaturePlugin,* para apoiar a modelagem de *features*, a qual consiste de um *plug-in* para a plataforma de desenvolvimento Eclipse (ECLIPSE, 2009). Eclipse é um conjunto de projetos abertos que iniciou com um ambiente de desenvolvimento IDE (*Integrated Development Environment*) Java, mas atualmente fornece suporte para muitas outras atividades como modelagem, compiladores para várias linguagens, etc.

A Figura 4.3 mostra um exemplo de modelo de *features* construído através desta ferramenta.

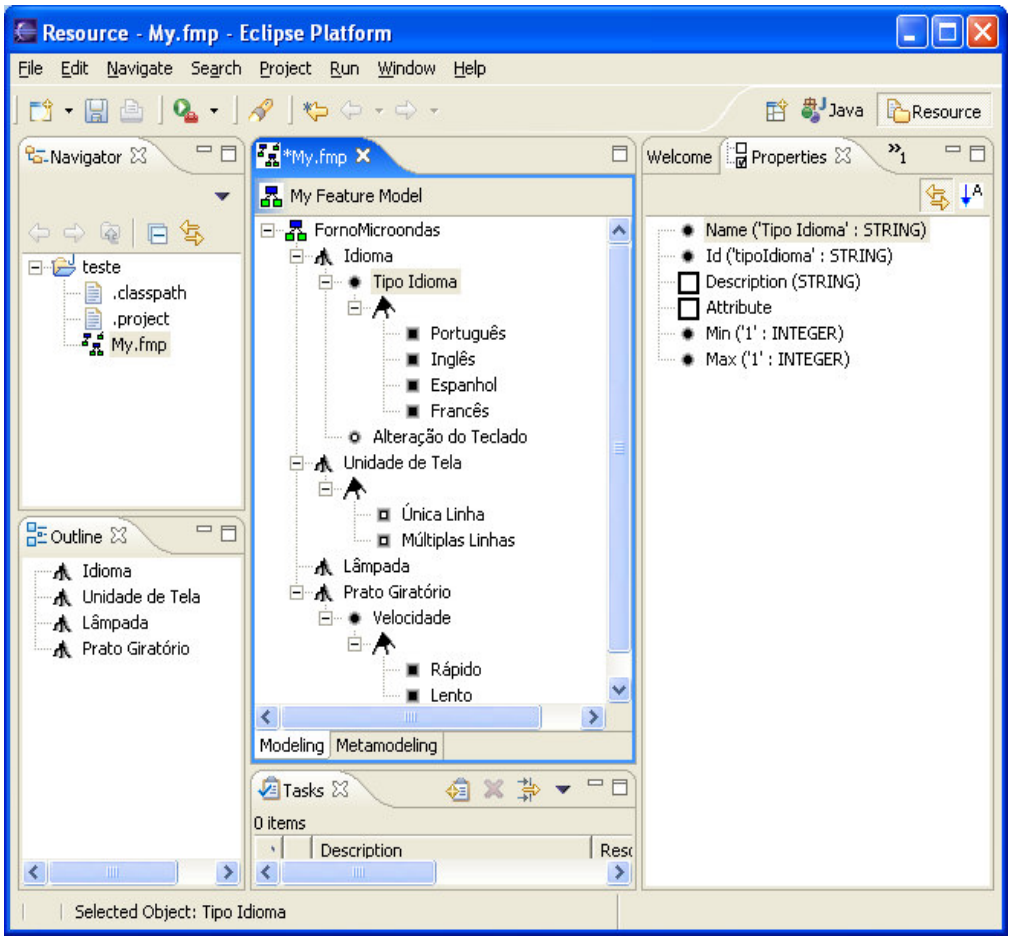


Figura 4.3 - *FeaturePlugin* na plataforma Eclipse

Na ferramenta *FeaturePlugin*, as *features* também são organizadas em hierarquias e permite modelar *features* mandatórias, opcionais e alternativas, e representar sub-*features.*

Além disso, a ferramenta implementa a modelagem de *feature* baseada em cardinalidade, que representa em seus modelos, o relacionamento entre *features* e sub-*features* (CZARNECKI, HELSEN, EISENECKER, 2005). Dessa forma, esta ferramenta é utilizada também na fase de especificação da variabilidade auxiliando na introdução da variante e na determinação da cardinalidade.

Existem, entretanto, alguns autores que apresentam a modelagem da variabilidade utilizando outros recursos, com base nos mecanismos de implementação de variabilidade. É o caso do trabalho de Gomaa e Webber (2004), que modela a variabilidade através de parametrização, informação escondida, herança e modelo de ponto de variação (*Variation Point Model*), utilizando modelos UML para suas representações. Keepence e Mannion (1999) também utilizam como mecanismo de variabilidade, padrões de projetos para modelar a variabilidade em seus produtos.

Para este trabalho, selecionou-se a representação de modelo de *features* apresentado por Gomaa (2005) pelo fato deste modelo utilizar estereótipos, o que facilita a distinção entre tipos de *features.* Neste modelo, a classificação das *features* é equivalente a utilizada no modelo apresentado pelo trabalho de Griss, Favaro, D´Alessandro (1998).

No modelo de Gomaa, *features* são representadas por caixas retangulares e cada *feature* possui um estereótipo que indica seu tipo. *Features* mandatórias são interligadas com *features* opcionais ou do tipo ponto de variação através de relacionamentos de associação, as *features* variantes são interligadas a *features* do tipo ponto de variação através de relacionamentos de agregação. Um exemplo deste modelo está apresentado na Figura 4.4:

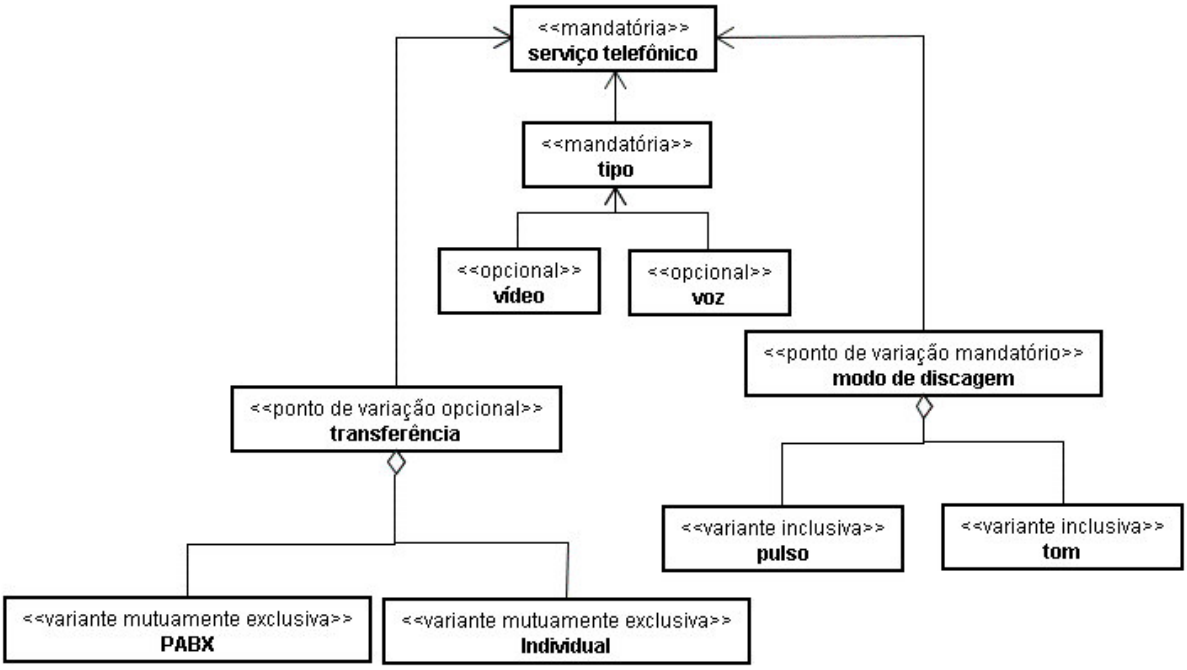


Figura 4.4 - Modelo de *features* com estereótipos

4.3 Processo de Gerenciamento da Variabilidade

O gerenciamento da variabilidade é uma atividade que tem como objetivo identificar, projetar, implementar e rastrear a flexibilidade na linha de produtos de software (VOELTER, GROHER, 2007).

Como já citado no início deste capítulo, este processo não é considerado uma tarefa trivial já que muitos fatores influenciam em como as decisões serão tomadas para realizar este gerenciamento, pois suas atividades permeiam os demais processos de linhas de produtos de software (KIM et al., 2006), (GOMAA, SHIN, 2007).

Muitas vezes, a variabilidade não tem localização precisa e, freqüentemente, causa impacto em muitos artefatos e em vários estágios do ciclo de vida do desenvolvimento da linha de produtos de software. Este fato pode ser observado, principalmente se a variabilidade atingir o desempenho, demanda de recursos ou interoperabilidade do sistema (BECKER, 2003).

Uma variação pode interferir na outra, já que variantes podem requerer ou excluir outras variantes, resultando em uma interdependência entre elas. Esta interdependência interfere fortemente na consistência e eficiência do gerenciamento de variabilidade porque aumenta a dificuldade da solução e deve ser considerada durante todo o ciclo de vida das variantes (BECKER, 2003).

As alterações ou as adaptações a serem gerenciadas, podem afetar a qualidade e o comportamento do sistema. Assim, o sucesso da linha de produtos de software depende de um bom gerenciamento das alterações tanto no desenvolvimento da linha de produtos de software quanto na derivação do produto.

Por esses motivos, o gerenciamento da variabilidade é considerado uma questão chave e desafiadora na linha de produtos de software (GOMAA, SHIN, 2007).

De uma forma geral, o gerenciamento da variabilidade pode ser realizado através das atividades apresentadas por (VAN GURP, BOSCH, SVAHNBERG, 2001), como mostra a Figura 4.5.

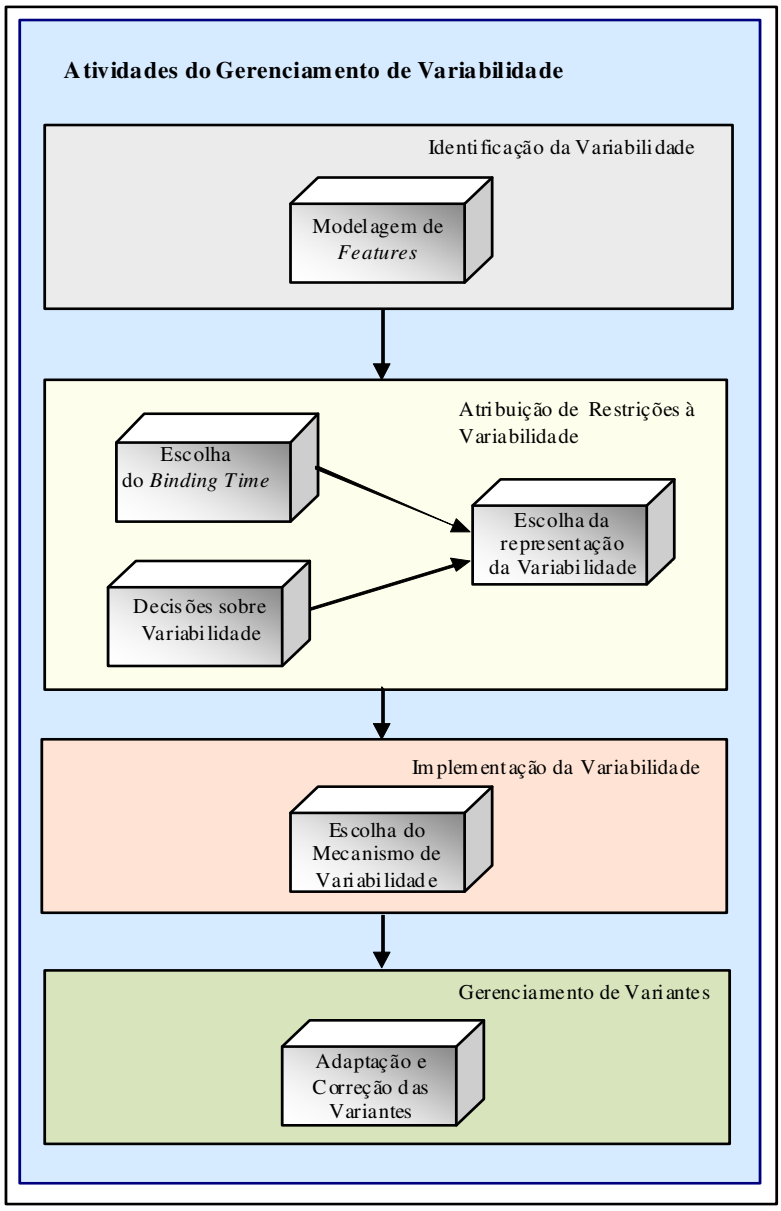


Figura 4.5 - Atividades do Gerenciamento de Variabilidade
Fonte Adaptada: (VAN GURP, BOSCH, SVAHNBERG, 2001)

- **Identificação da variabilidade** - consiste em identificar os pontos de variação, ou seja, onde a variabilidade é necessária. O modelo de *features* é utilizado na atividade de identificação. Através deste modelo, é possível indicar se a *feature* é mandatória ou opcional, os elementos que representam pontos de variação e as variantes associadas a estes pontos.

- **Atribuição de restrições à variabilidade** - após identificar o ponto de variação, é necessário tomar decisões para incluir a flexibilidade necessária para contemplar as necessidades atuais e futuras da linha de produtos de software, com custos aceitáveis:

    - decidir quando a *feature* variante deve ser introduzida, se durante projeto e implementação da linha de produtos de software e/ou produto de software;

    - definir como e quando as variantes devem ser adicionadas ao sistema;

    - decidir quando um ponto de variação deve ser relacionado a uma determinada variante;

    - escolher uma representação para a realização do ponto de variação. A forma como as novas variantes são adicionadas é uma informação relevante para esta decisão.

- **Implementação da variabilidade** - consiste em selecionar uma técnica de realização da variabilidade, ou seja, escolher um mecanismo de variabilidade. Este mecanismo deve ser selecionado de acordo com as necessidades do projeto da linha de produtos de software e deve ser capaz de responder da melhor forma possível às restrições de variabilidade apresentadas no passo anterior. Existem várias técnicas que podem ser utilizadas como mecanismo de variabilidade.

- **Gerenciamento de variantes** - envolve o controle da manutenção, ou seja, da adaptação ou da correção das variantes. Esta atividade inclui o controle para adicionar ou remover variantes, pois, como os requisitos podem mudar, variantes podem ser adicionadas ou removidas da linha de produtos de software.

4.4 Mecanismos de Variabilidade

Existem, na literatura, vários mecanismos para implementar a variabilidade em linha de produtos de software (SCHNIEDERS, 2006a), (SCHNIEDERS, 2006b), (GOMAA, WEBBER, 2004), (BRAGANÇA, MACHADO, 2004b), (JACOBOSON apud TRIAGUX, HEYMANS, 2003), (SVAHNBERG, VAN GURP, BOSH 2002), (ANASTASOPOULOS, GACEK, 2001), (SVAHNBERG, BOSCH, 2000), (KEEPENCE, MANNION, 1999).

A Tabela 4.2 relaciona os trabalhos mais relevantes e os mecanismos de variabilidade por eles apresentados.

Os mecanismos de variabilidade mais citados nestes trabalhos são:

- **Bibliotecas Estáticas e Dinâmicas:** as bibliotecas estáticas contêm um conjunto de funções externas que podem ser ligadas à aplicação depois da compilação. A forma como as funções são chamadas (protótipo) é conhecida pelo código compilado e, portanto, não precisa ser alterada. Entretanto, a implementação pode mudar de acordo com a seleção de diferentes bibliotecas e isso gera a variabilidade. As bibliotecas dinâmicas são carregadas em aplicações em tempo de execução.

- **Compilação Condicional:** a compilação condicional utiliza diretivas que habilitam o controle sobre partes do código que deverão ser incluídos ou excluídos durante a compilação do programa. A funcionalidade desejada é selecionada a partir de definição apropriada dos símbolos condicionais.

- **Configuração:** é usada para selecionar componentes do repositório e conectá-los para incluí-los no produto. É uma alternativa em que todas as variantes estão presentes em todos os pontos de variação da linha de produtos de software.

Tabela 4.2 - Principais mecanismos de variabilidade e autores

Autores / Mecanismos	Bachmann, Clements (2005)	Anastasopoulos, Gacek (2001)	Svahnberg, Van Gurp, Bosch (2002) Svanhnberg, Bosch (2000)	Bragança, Machado (2004b)	Schnieders (2006a) Schnieders (2006b)	Gomaa, Webber (2004)	Keepence, Mannion (1999)	Jacoboson Apud Trigaux, Heymans (2003)
Bibliotecas estáticas/dinâmicas		X	X	X				
Compilação Condicional	X	X	X					
Configuração	X		X					X
Encapsulamento/ Informação Escondida					X	X		
Extensão			X		X			X
Geração	X		X					X
Herança	X	X	X	X	X	X		X
Padrão de Projeto		X	X		X		X	
Parametrização	X	X	X	X	X	X		X
Programação Orientada a Aspectos	X	X	X					
Reflexão		X		X				
Templates	X		X					X

- **Encapsulamento/Informação Escondida:** é um conceito básico do desenvolvimento orientado a objetos. Neste mecanismo, a informação que possivelmente pode ser alterada está escondida dentro do objeto. Desta forma, várias versões do mesmo componente podem ser construídas com a mesma interface e, assim, a variante está escondida dentro de cada versão do componente (GOMAA, 2005), (SCHNIEDERS, 2006a).

- **Extensão e Pontos de Extensão:** usado quando partes de um componente pode ser estendido com comportamento adicional, selecionado de um conjunto de possíveis variantes (SCHNIEDERS, 2006a), (SCHNIEDERS, 2006b), (SVAHNBERG, BOSCH, 2000). Geralmente, estas possíveis variantes são modeladas independentemente e o usuário pode selecionar ou adicionar nova variante ao conjunto destas variantes. Harsu (2001) utiliza a técnica de extensão para o padrão de projeto *Strategy* de Gamma et al.(1995).

- **Geração:** neste mecanismo, um sistema denominado gerador traduz uma especificação, escrita em uma linguagem específica, para um componente em nível de código fonte que fará parte do produto ou aplicação. Por exemplo: uma interface gráfica pode ser gerada a partir de uma especificação gráfica ou textual (HARSU, 2001).

- **Herança:** neste mecanismo, uma subclasse é derivada de uma superclasse, através da introdução de novos atributos e novas operações na classe derivada. Assim, uma classe derivada, a partir de sua superclasse, é especializada para um contexto específico. No ESPLEP (GOMAA, 2005), a herança é usada para modelar diferentes subclasses de variantes de uma classe abstrata, as quais são utilizadas por diferentes membros da linha de produtos de software. Apesar desta abordagem permitir ao desenvolvedor de *core assets* fornecer um grande número de variantes baseado na superclasse, o número de variantes ainda é limitado, já que o usuário deve selecionar a variante a partir de um conjunto de escolhas pré-definidas (GOMAA, WEBBER, 2004).

- **Padrão de Projeto:** os padrões de projeto podem ser explorados no contexto da linha de produtos de software já que, além de permitir a modelagem de variações no domínio, muitos deles identificam características do sistema que podem variar e fornecem soluções reusáveis para gerenciar estas variações. Por exemplo, o padrão *Builder* pode ser usado para carregar um código variante em tempo de execução. Trabalhos como os de Keepence e Mannion (1999) e Van Gurp, Bosh, Svahnberg (2001) detalham e utilizam de padrões de projetos para modelar a variabilidade de linhas de produtos de software.

- **Parametrização:** é um mecanismo em que os atributos de um componente do *core assets* devem ser preenchidos. Assim, o comportamento do componente é determinado em função dos valores atribuídos aos parâmetros, que podem ser lidos, por exemplo, de um arquivo de configuração ou ser introduzidos pelo usuário. Desta forma, a idéia do mecanismo de parametrização é representar software reusável como uma biblioteca de componentes parametrizados. A parametrização pode aumentar o reuso na linha de produtos de software; entretanto, a manutenção de um código concentrado em definições de parâmetros, assim como, a seleção entre funcionalidade alternativa são consideradas tarefas complicadas e sujeitas a erros (ANASTASOPOULOS, GACEK, 2001), (GOMAA, WEBBER, 2004), já que o usuário deve se manter atento nas constantes decisões a serem tomadas e nas alterações a serem realizadas.

- **Programação Orientada a Aspectos:** é um estilo de programação inicialmente proposto por pesquisadores da Xerox PARC (CZARNECKI, EISENECKER, 2005), que realiza a separação e organização de responsabilidades espalhadas do sistema em módulos distintos. Esta separação de responsabilidades habilita alteração, inserção ou remoção de comportamentos do sistema em tempo de compilação, sem que isto afete as demais partes do sistema. A manutenção e o entendimento se tornam mais eficientes, já que um aspecto pertence a um único módulo distinto. Pode-se implementar a variabilidade em linhas de produtos de software a partir destas alterações no comportamento do sistema.

- **Reflexão**: é a habilidade de um programa manipular a si próprio, alterar sua estrutura e seu comportamento (LIESENBERG, STEHLING, 1999), (ANASTASOPOULOS, GACEK, 2001). Estes sistemas, denominados de reflexivos, incorporam estruturas que representam a si próprio; são aptos a examinar e alterar suas próprias propriedades, e a controlar sua implementação em tempo de execução. Desta forma, através deste mecanismo, é possível que o sistema obtenha informações sobre seu tipo de objeto, seus atributos, métodos, construtores e suas estruturas de herança. É possível ainda, dinamicamente, carregar uma nova classe, criar novas instâncias e chamar métodos (LEAL, 2003).

- ***Templates*:** neste mecanismo, componentes são configurados de forma específica para aplicação, um corpo genérico é preenchido com partes de um produto específico. A desvantagem deste mecanismo é que parâmetros do código específico do produto devem ser fornecidos ao sistema. Além disso, extensões com comportamentos adicionais não são possíveis neste mecanismo, já que se deve seguir um modelo genérico pré-estabelecido.

4.5 Exemplos da Utilização dos Mecanismos de Variabilidade

Em complemento ao conjunto de mecanismos de variabilidade citados por diversos autores e apresentados na seção anterior, apresenta-se nesta seção, de forma mais detalhada, o conjunto de mecanismos de variabilidade descrito no trabalho Svahnberg, Van Gurp e Bosh (2002). A diversidade de mecanismos de variabilidade apresentada possui uma organização sistemática de suas descrições, o que facilita a compreensão da utilização destes mecanismos. A descrição de cada mecanismo de variabilidade está organizada através dos tópicos: objetivo, motivação e solução.

Os mecanismos a serem apresentados são:

1. Reorganização Arquitetural;
2. Componente Arquitetural Variante;

3. Componente Arquitetural Opcional;
4. Substituição Binária - Diretivas de Ligação;
5. Arquitetura Centrada em Infra-Estrutura;
6. Especialização de Componente Variante;
7. Especialização de Componente Opcional;
8. Especialização de Componente Variante em Tempo de Execução;
9. Implementação de Componente Variante;
10. Condição Através de Constante;
11. Condição Através de Variável;
12. Superposição de Fragmento de Código.

4.5.1 Reorganização Arquitetural

Objetivo: Este mecanismo implementa a variabilidade através da reorganização da arquitetura da linha de produtos de software para arquiteturas específicas dos produtos.

Motivação: Embora os membros de uma linha de produtos de software compartilhem mesmos conceitos, o fluxo de controle e o fluxo de dados entre estes conceitos podem não ser os mesmos para todos os membros. Portanto, a arquitetura da linha de produtos de software é reorganizada para se gerar uma arquitetura concreta de um produto. Isto implica principalmente em mudanças no fluxo de controle, ou seja, na ordem em que os componentes são conectados entre si, e também, em como estes componentes são conectados entre si, ou seja, a interface requerida e fornecida dos componentes pode variar de produto para produto.

Solução: Neste mecanismo, os componentes são representados como subsistemas controlados por ferramentas de gerenciamento de configuração ou através de

Linguagem de Descrição de Arquitetura (*Architecture Description Language* - ADL). As variantes a serem incluídas no sistema são determinadas através das ferramentas de configuração. Através deste mecanismo é possível adicionar novas variações durante o projeto arquitetural, onde a arquitetura da linha de produtos de software é usada como um *template* para criar uma arquitetura específica de um produto.

4.5.2 Componente Arquitetural Variante

Objetivo: Este mecanismo fornece suporte para prever vários componentes arquiteturais com diferentes interfaces para representar a mesma entidade conceitual.

Motivação: Em alguns casos, um componente arquitetural pode ser substituído por outro com uma interface diferente, e representar um domínio diferente. Esta necessidade não deve afetar o restante da arquitetura. Por exemplo, em uma linha de produtos de software, alguns produtos trabalham com disco rígido, outros com *scanners*. O componente *scanner* substitui o componente disco rígido sem afetar o restante da arquitetura.

Solução: O mecanismo deve prever a existência de todos os possíveis componentes. A seleção do componente a ser usado e o instante de uso é delegada às ferramentas de gerenciamento de configuração. É possível adicionar novas variantes durante o projeto arquitetural, quando novos componentes podem ser adicionados e também durante o projeto detalhado, quando os componentes são projetados de forma mais concreta.

4.5.3 Componente Arquitetural Opcional

Objetivo: Este mecanismo fornece suporte para a existência de componente cuja presença é opcional no sistema.

Motivação: Alguns componentes arquiteturais podem estar presentes em alguns produtos e ausentes em outros. Isto significa que, em uma dada configuração de produto, certos componentes podem ter uma interação com determinados componentes arquiteturais e, em outras configurações, estes mesmos componentes não necessitam desta interação. Por exemplo, um servidor de armazenamento pode opcionalmente ser equipado com uma cache de disco rígido. Em uma configuração de produto, certos componentes precisam interagir com esta cache, enquanto que em outra configuração, a cache não é necessária.

Solução: Uma das soluções consiste em criar um componente nulo, ou seja, que possui uma interface que responde com valores fictícios, definidos de tal forma que os componentes saibam ignorar. O esforço, espaço e processamento consumidos para se ter uma técnica com valores fictícios são as desvantagem deste mecanismo.

4.5.4 Substituição Binária – Diretivas de Ligação

Objetivo: Este mecanismo fornece um sistema com implementações alternativas utilizando bibliotecas básicas de sistema.

Motivação: Em alguns casos, para fornecer suporte a uma nova plataforma, deve-se substituir uma biblioteca de sistema. Isto ocorre, por exemplo, quando se compila um sistema para diferentes dialetos do UNIX. Este mecanismo implementa a variabilidade através da biblioteca de sistema ou por uma biblioteca distribuída junto com o sistema. Por exemplo, um determinado jogo pode ser distribuído com bibliotecas diferentes para poder funcionar com o sistema Windows (tal como o X-windows), dispositivo Gráfico OpenGL ou usar um dispositivo gráfico padrão SVGA.

Solução: A solução consiste em representar as variantes como arquivos de bibliotecas independentes (*stand-alone*) e instruir o *linker* com um arquivo para ligar o sistema. A ligação pode ser feita durante compilação ou no tempo de execução do sistema. Este é um mecanismo de implementação de variabilidade bem desenvolvido e, portanto, não apresenta maiores conseqüências em sua utilização.

4.5.5 Arquitetura Centrada em Infra-Estrutura

Objetivo: Este mecanismo transforma as conexões entre componentes em uma entidade de classe primária.

Motivação: Parte do problema ao conectar os componentes, e em particular quando os componentes podem variar, é que o conhecimento das conexões está freqüentemente codificado nas interfaces dos componentes. A reorganização da arquitetura ou a substituição de um componente na arquitetura pode ser facilitada se a conexão for uma entidade explícita no sistema, onde as modificações possam ser implementadas.

Solução: A solução consiste em converter conexões em entidades de classes primárias, ou seja, em conectores. Desta forma, os componentes não ficam mais conectados a outros componentes, mas aos conectores, que passam a fazer parte da infra-estrutura. Esta infra-estrutura é então responsável por estabelecer a conexão entre a interface requerida de um componente com a interface provida de outros componentes. Esta infra-estrutura pode ser um padrão existente como CORBA (OMG, 2008), ou pode ser um padrão particularmente desenvolvido para o sistema. Esta solução não remove a necessidade de interfaces bem definidas ou problemas com ajustes de componentes para se trabalhar em diferentes ambientes operacionais, mas facilita o gerenciamento das conexões. Dependendo da infra-estrutura selecionada, o mecanismo é aberto para adicionar novas variantes durante o projeto arquitetural ou em tempo de execução. Assim, este mecanismo pode render uma arquitetura dinâmica que, em muitos casos é similar ao padrão de projeto *Adapter* (GAMMA, 1995).

4.5.6 Especialização de Componente Variante

Objetivo: O objetivo deste mecanismo é ajustar a implementação do componente à arquitetura do produto.

Motivação: Alguns mecanismos de implementação de variabilidade, aplicados em nível de projeto arquitetural, necessitam de suporte em estágios posteriores. Em particular, mecanismos onde as interfaces fornecidas variam, necessitam de suporte também para a interface requerida. Nestes casos, parte da implementação do componente, que está relacionada com a interface e representa uma variante precisa ser substituída. Este mecanismo pode ser usado também para adaptar um componente às necessidades particulares de um produto. Por exemplo, um sistema de armazenamento pode ser entregue com um cache tradicional ou um cache de disco rígido. O componente de sistema de arquivo deve saber qual dos sistemas de armazenamento está presente, já que a chamada para ambos os tipos é quase idêntica. Assim, o componente de sistema de arquivo é ajustado usando o mecanismo de implementação de variabilidade para trabalhar com o tipo de cache presente no sistema.

Solução: A solução consiste em separar partes de interfaces em classes distintas que podem decidir o melhor caminho para interagir com outros componentes. Uma ferramenta de gerenciamento de configuração pode decidir quais classes e variantes do componente devem ser incluídas na arquitetura do produto. As variantes disponíveis são incluídas durante o projeto detalhado, quando as classes de interface são projetadas.

4.5.7 Especialização de Componente Opcional

Objetivo: O objetivo deste mecanismo é incluir ou excluir partes do comportamento de uma implementação de um componente.

Motivação: A implementação de um componente particular pode ser customizada de várias formas, através da adição ou remoção das partes de seu comportamento. Por exemplo, dependendo do tamanho da tela de uma aplicação para um computador portátil, pode-se optar por não incluir certos recursos, e no caso, quando estes recursos interagem com outros, esta interação também precisa ser excluída da execução do código.

Solução: A solução consiste em separar um comportamento opcional em uma classe separada e criar classes nulas que podem agir como um *placeholder* (classe que pode ser substituída) quando o comportamento for excluído. Uma ferramenta de gerência de configuração pode ser utilizada para decidir qual destas classes deve ser incluída no sistema. Alternativamente, pode-se identificar o comportamento opcional com *flags* de tempo de compilação. A dificuldade em separar comportamentos opcionais em classes distintas deve ser considerada.

4.5.8 Especialização de Componente Variante em Tempo de Execução

Objetivo: O objetivo deste mecanismo é fornecer suporte à existência e à seleção entre várias especializações dentro da implementação do componente.

Motivação: É requerido da implementação de um componente que este se adapte ao ambiente em que está sendo executado, ou seja, a qualquer momento na execução do sistema, a implementação do componente é capaz de satisfazer os requisitos do usuário e do resto do sistema. Isto implica que a implementação do componente é equipada com um número de execuções alternativas e é capaz de, em tempo de execução, selecionar uma delas. Um exemplo pode ser um computador portátil, como um *handheld*, que trabalha com conexões de comunicação com diferentes larguras de banda, como um telefone celular. Este computador necessita, por exemplo, de diferentes estratégias para que o sistema operacional recupere informações através da conexão estabelecida. Desta forma, as variantes precisam ser representadas como estratégias dentro dos componentes relacionados.

Solução: Uma das soluções consiste em aplicar dois padrões de projeto, *Strategy* e *Template Method* (GAMMA, 1995). Comportamento alternativo é colocado em classes separadas e mecanismos são introduzidos para selecionar, em tempo de execução, a classe adequada. O mecanismo é aberto para novas variações durante o projeto detalhado, desde que classes e conceitos orientados a objetos estejam em foco durante esta fase. A conseqüência deste mecanismo é que um código comum pode ser duplicado em estratégias diferentes, quando poderia ser reusado.

4.5.9 Implementação de Componente Variante

Objetivo: O objetivo deste mecanismo é fornecer suporte a várias implementações concorrentes e co-existentes de um componente arquitetural.

Motivação: Um componente arquitetural tipicamente pertence a algum domínio ou sub-domínio. Estes domínios podem ser implementados usando alguns padrões e um sistema deve tipicamente fornecer suporte a estes diferentes padrões simultaneamente. Por exemplo, um servidor de disco rígido pode fornecer suporte à vários padrões de sistemas de arquivos de rede, e é capaz de escolher entre eles em tempo de execução. A arquitetura deve fornecer suporte a estas diferentes implementações de componente e outros componentes no sistema deve ser capaz de dinamicamente determinar para qual implementação de componente os dados ou mensagens devem ser enviados.

Solução: A solução consiste em implementar várias versões de componentes, aderentes à mesma interface. Existem certos padrões de projeto que facilitam este processo: *Strategy, Broker, Abstract Factory* e *Builder* (GAMMA, 1995). A conseqüência do uso deste mecanismo é que o sistema deve fornecer suporte a várias versões de um domínio simultaneamente e estes domínios diferentes podem conduzir aos módulos de códigos similares, que poderiam ser reusados, mas foram projetados de formas distintas.

4.5.10 Condição Através de Constante

Objetivo: O objetivo é fornecer suporte a várias formas para desempenhar uma operação, das quais somente uma será usada em um dado sistema.

Motivação: Basicamente esta é uma versão refinada da Especialização de Componente Variante, onde a variante não é grande suficiente para ser considerada como classe. Pode-se usar este mecanismo por razões de desempenho e para ajudar o compilador remover um código que não é usado. Por exemplo, existem diferentes tipos de cache em diferentes servidores de armazenamento, podendo ser um cache de disco rígido ou um cache tradicional, o componente de sistema de arquivo deve chamar corretamente um destes tipos de cache presentes no sistema. No caso em que a variante se conecta com outra variante ou componente, é também uma forma de se conseguir que o código passe pelo compilador; já que o método, ao chamar uma classe inexistente, poderia interromper o processo de compilação.

Solução: Pode-se utilizar dois tipos de declarações condicionais. Uma forma é a diretiva de pré-processador como ifdefs em C++ ou o tradicional “if” em linguagem de programação. Outro caminho é utilizar *constructs templates* de C++. Este mecanismo é aplicado durante a implementação do componente e é ativado durante a compilação do sistema, quando parâmetros em tempo de compilação são usados. A utilização de ifdefs ou outra diretiva de pré-processador pode ser um risco, já que o número de caminhos de execução tende a aumentar quando isto é utilizado, dificultando a manutenção e correção de erros.

4.5.11 Condição Através de Variável

Objetivo: O objetivo deste mecanismo é fornecer suporte a várias formas para executar uma operação, das quais somente uma será usada em dado momento, mas permite que a escolha seja feita durante a execução.

Motivação: Algumas vezes, a variabilidade provida pelo mecanismo de condição através de constante precisa ser estendida em tempo de execução. Como as constantes são avaliadas durante a compilação, esta alteração não pode ocorrer e, por causa disto, uma variável deve ser utilizada no lugar de uma constante.

Solução: A solução consiste em substituir a constante usada no mecanismo condição através de constante por uma variável e fornecer funcionalidade para alterar esta variável. Este mecanismo não pode ser usado em qualquer diretiva de compilação. Pode ser aplicado durante a implementação, quando novas variantes podem ser adicionadas, entretanto, durante a compilação, a adição de variantes não pode mais ocorrer. É um mecanismo considerado muito flexível, entretanto, da mesma forma que o mecanismo de condição através de constante, se os pontos de variação para uma particular *feature* variante estiverem espalhados pelo código, sua manutenção pode se tornar difícil.

4.5.12 Superposição de Fragmento de Código

Objetivo: Este mecanismo introduz novas considerações no sistema, sem afetar diretamente o código fonte.

Motivação: Devido ao fato de um componente poder ser utilizado em vários produtos, não é desejável introduzir considerações específicas de produto no componente. Entretanto, em algumas vezes, pode ser necessário introduzir estas especificações para habilitar o uso do componente.

Solução: A solução é desenvolver o software para funcionar genericamente, e então sobrepor restrições de produtos específicos no estágio em que o trabalho com o código fonte estiver terminado. Existem algumas técnicas para isso, como é o caso de Programação Orientada a Aspectos, onde diferentes interesses são identificados e colocados no código fonte antes de ser compilado e um comportamento adicional é incorporado no comportamento existente. A conseqüência deste mecanismo é que interesses diferentes são separados de uma funcionalidade principal. Isto pode

tornar mais difícil o entendimento do funcionamento final do código final, já que os caminhos de execução não são óbvios.

4.6 Considerações Finais

Este capítulo introduziu o conceito de variabilidade e definiu os termos mais importantes relacionados a este assunto.

O capítulo mostrou também, as formas mais relevantes de representação da variabilidade. Para este trabalho, foi selecionado o modelo de *features* apresentado por Gomma (2005) porque apresenta uma melhor representação para realizar a distinção entre *features*, com a utilização de estereótipos.

Foram também apresentadas as atividades relacionadas com o gerenciamento de variabilidade da linha de produtos de software, segundo Van Gurp, Bosch, Svahnberg (2001) e Svahnberg, Bosch (2000) e observou-se que este processo é considerado um dos mais importantes e desafiadores no contexto de linha de produtos de software. Os passos apresentados neste trabalho foram utilizados como base na definição do processo GVLPS.

Observou-se ainda que a implementação da variabilidade é uma das atividades do gerenciamento da variabilidade e pode ser realizada através de vários mecanismos.

Foram apresentados dois conjuntos de mecanismos de variabilidade. No primeiro conjunto foram listados mecanismos citados em vários trabalhos. No segundo conjunto foram descritos, com mais detalhes, os mecanismos apresentados por Svahnberg, Van Gurp e Bosh (2002).

Os diversos mecanismos de variabilidade apresentados mostraram que a maioria dos mecanismos citados proporciona vantagens e também desvantagens, principalmente no que diz respeito a dificuldades de entendimento do sistema e de manutenção, desempenho, esforços de implementação, consumo de processamento, duplicação de código, entre outras.

Dentre os mecanismos apresentados, destaca-se no primeiro conjunto, o baseado em reflexão citado por Anastasopoulos e Gacek (2001) e no segundo conjunto, o mecanismo denominado especialização de componente variante em tempo de execução apresentado por Svahnberg, Van Gurp e Bosh (2002). Estes autores apresentam nestes mecanismos idéias de reflexão e adaptação em tempo de execução e comentam que podem ser técnicas interessantes para serem utilizadas na implementação da variabilidade.

Apesar destes mecanismos serem citados, não foram encontrados, na literatura, maiores detalhes sobre sua implementação.

Para o processo GVLPS foi selecionado um mecanismo de variabilidade baseado em modelos de objetos adaptáveis e em reflexão, onde sua principal vantagem é a flexibilidade apresentada para criação e seleção de suas variantes, o que facilita, desta forma, realizar alterações e reuso do sistema.

"Três regras básicas:
1 – No caos está a simplicidade.
2 – No conflito está a harmonia.
3 – No meio da dificuldade está a oportunidade."

Albert Einstein

5 REFLEXÃO E MODELOS DE OBJETOS ADAPTÁVEIS

A variabilidade de uma linha de produtos de software pode ser implementada através da reflexão. A técnica de reflexão possibilita ao sistema responder questões sobre si próprio e principalmente, atuar sobre si mesmo.
Uma arquitetura de modelo de objetos adaptáveis é um tipo particular de arquitetura reflexiva que abrange sistema orientado a objetos e que pode ser estendida para incluir novos elementos. Este capítulo introduz o conceito de reflexão e detalha a arquitetura de modelos de objetos adaptáveis.

5.1 Conceito de Reflexão

Atualmente, muitos sistemas necessitam ser dinâmicos e configuráveis para que possam rapidamente ser alterados para se adaptarem a novas regras de negócio. Cada vez mais, usuários necessitam de mudanças em seus sistemas, com o mínimo de mudança no código.

Esta flexibilidade pode ser possível através de arquiteturas reflexivas ou meta-arquiteturas, que se baseiam no conceito de reflexão e podem se adaptar dinamicamente em tempo de execução, de acordo com um novo requisito do usuário.

A reflexão é a habilidade de um programa manipular como dados algo que representa o estado do programa durante sua própria execução (GABRIEL, BOBROW, WHITE, 1993). É também definida como a atividade realizada por sistemas que atuam sobre si próprios, alterando sua estrutura e seu comportamento, e muitas vezes, é chamada de computação reflexiva (LIESENBERG, STEHLING, 1999).

O conceito de reflexão surgiu primeiramente na área de inteligência artificial e se propagou para várias outras áreas da computação como programação lógica e orientação a objetos (CAZZOLA, 1998).

Um sistema reflexivo é aquele que contém a arquitetura reflexiva (MAES, 1987).

Uma arquitetura reflexiva é aquela que possibilita que uma linguagem de programação reconheça a reflexão como um conceito fundamental e, assim, forneça ferramentas para manipular a computação reflexiva explicitamente. Isto significa que o interpretador de tal linguagem tem que fornecer ao sistema, acesso aos dados que representam as características do sistema por si próprios. Para isso, deve-se incluir o código que determina como estes dados podem ser manipulados (MAES, 1987).

A linguagem de programação que fornece suporte a mecanismos de reflexão permite que programas examinem e alterem suas próprias propriedades, controlando sua implementação em tempo de execução. Através deste mecanismo, os objetos podem ter informações sobre seu tipo, seus atributos, seus métodos, seus construtores e suas estruturas de herança. Além disso, é possível ainda, ao carregar uma nova classe, a criação de novas instâncias e chamadas de métodos, em tempo de execução, mesmo sem possuir detalhes desta classe em tempo de compilação (LEAL, 2003).

A arquitetura reflexiva é composta por dois níveis: nível base e meta-nível (CZARNECKI, EISENECKER, 2005), (CAZZOLA, 1998). O nível base inclui a aplicação lógica e descreve a computação que é esperada que o sistema faça. Os objetos deste nível são chamados de entidade-base. O meta-nível fornece informações sobre propriedades selecionadas do sistema e faz com que o software tenha conhecimento sobre si próprio. Os objetos que trabalham neste nível são chamados de meta-entidade.

Leal (2003) apresenta um estudo que compara o mecanismo de reflexão de três linguagens de programação orientadas a objetos: Smalltalk, Java e C++.

A linguagem Smalltalk apresenta um mecanismo de reflexão poderoso e flexível porque possui um modelo de objeto puro, ou seja, é uma linguagem completamente orientada a objetos. No caso de um modelo de objeto completamente reflexivo,

qualquer objeto pode, em tempo de execução, reconhecer suas estruturas e seu comportamento.

A linguagem Java não é considerada tão poderosa em reflexidade quanto à linguagem Smalltalk devido a certas limitações, como por exemplo, o fato de existirem tipos básicos que não são objetos. Entretanto, mesmo assim, seu mecanismo de reflexão, em conjunto com suas outras potencialidades e sua extensa biblioteca, permite construir um grande número de aplicações que aceita manutenção e atualização dinâmica. Esta capacidade faz com que a linguagem Java seja muito utilizada para aplicações reflexivas.

A linguagem C++, por outro lado, não apresenta suporte para reflexão. Possui apenas um mecanismo, denominado *Run-Time Type Identification* (RTTI) que permite, de forma limitada, algumas operações em tempo de execução, como conhecer o tipo de objeto, conhecer o nome do tipo e compará-lo, etc.

Apesar da programação reflexiva ser de difícil entendimento, este paradigma é considerado um caminho interessante para implementar variabilidade em linha de produtos de software, já que a funcionalidade básica do sistema pode ser refletida, ou seja adaptada e, assim, manipulada de acordo com uma determinada configuração (ANASTASOPOULOS, GACEK, 2001). Além disso, as principais vantagens de se utilizar um sistema reflexivo são aumento da flexibilidade, possibilidade de extensão e reutilização de código (LEAL, 2003), características também desejadas na abordagem de linha de produtos de software.

Bragança e Machado (2004a) (2004b) também sugerem a possibilidade de utilizar a reflexão para trabalhar com a variabilidade em tempo de execução, já que esta técnica permite o carregamento dinâmico de tipos, a instanciação dinâmica de objetos e a execução dinâmica de métodos.

5.2 Modelos de Objetos Adaptáveis

O propósito de se usar os modelos de objetos adaptáveis é obter um modelo mais genérico possível, com um conjunto de padrões de projeto que permite facilmente

incluir novos tipos de objetos, alterar objetos existentes e alterar o relacionamento entre os objetos.

Os modelos de objetos adaptáveis, também chamados de modelos de objetos ativos (FOOTE, YODER, 1998) e modelos de objetos dinâmicos (RIEHLE, TILMAN, JOHNSON, 2000) são aqueles que representam classes, atributos, relacionamentos e comportamentos como um metadado (YODER, 2007).

Metadados são dados que descrevem outros dados, ao invés de características do próprio domínio de aplicação (FOOTE, YODER, 1998). Por exemplo, em uma biblioteca, os metadados são informações como título, autor, data de publicação e descrição do conteúdo, ou seja, as informações que descrevem os dados de um determinado item contido na biblioteca. Os diversos itens que compõe a biblioteca são as entidades que são, por exemplo, livros, alunos, empréstimos. A entidade é um elemento abstrato que encapsula informações e pode representar uma classe, uma estrutura de dados, etc. Desta forma, pode-se dizer que metadado descreve uma entidade.

O metadado é alterado para refletir as mudanças desejadas, alterando assim, o comportamento do sistema. Neste contexto, maior flexibilidade pode ser conseguida se certos aspectos do sistema, como as regras de negócios, forem movidos para uma base de dados para que possam ser facilmente alterados. O modelo resultante permite introduzir novos produtos sem programação e realizar mudanças nos modelos de negócio em tempo de execução. Desta forma, o modelo de objeto adaptável guarda seu metadado em uma base de dados e o interpreta; quando esta informação descritiva é alterada, de forma controlada, estas mudanças são imediatamente refletidas no sistema (YODER, BALAGUER, JOHNSON, 2001).

Os sistemas de modelos de objetos adaptáveis diferem de sistemas orientados a objetos. Neste último, normalmente existem classes que descrevem os diferentes tipos de entidades de negócio juntamente com atributos e métodos associados a estas classes. Desta forma, as regras de negócio estão modeladas através das classes e uma alteração nestas regras causa uma alteração do código, causando a necessidade de gerar uma nova versão da aplicação. No sistema de modelos de objetos adaptáveis as entidades de negócio não são modeladas como classes e sim

por descrições (metadados) que são interpretados em tempo real. Assim, ao se alterar as regras de negócio, as descrições serão alteradas e imediatamente refletidas na aplicação (WELICKI, YODER, WIRFSBROCK, 2007). Desta forma, na arquitetura orientada a objetos, as entidades de negócio são representadas por classes, e na arquitetura de modelos de objetos adaptáveis, as entidades de negócio são representadas por metadados.

A característica principal em um sistema de modelo de objetos adaptáveis é a possibilidade de definição e de as regras de integridade de um modelo de domínio serem configuradas por especialistas do domínio para uma nova aplicação. A grande maioria de sistemas que utiliza modelos de objetos adaptáveis gerencia produtos de mesma natureza e é estendido para adicionar novos produtos. Estes sistemas são chamados de Arquitetura de Produto Definida pelo Usuário, ou *User Defined Produtct Architecture* (RIEHLE, TILMAN, JOHNSON, 2000).

Os modelos de objetos adaptáveis podem ser utilizados, por exemplo, nos seguintes casos (RIEHLE, TILMAN, JOHNSON 2000), (JOHNSON, WOOLF 1997):

- Existe a necessidade de criar novos tipos de objetos com diferentes propriedades em tempo de execução;
- As classes requerem um grande número de subclasses e ou a variedade total de subclasses que podem ser requerida é desconhecida;
- Existe a necessidade de construir aplicações que permitam que os usuários finais configurem muitos tipos de objetos;
- A aplicação requer freqüentes mudanças e rápido desenvolvimento.

Geralmente, a arquitetura de modelos de objetos adaptáveis é composta por vários padrões de projeto (YODER, JOHNSON, 2002). Os mais comumente usados são:

- *TypeObject*: fornece um caminho para definir novas entidades dos sistemas dinamicamente;
- *Property*: implementa os atributos destas novas entidades;

- *TypeSquare*: habilita novos tipos de atributos com a junção dos padrões *TypeObject* e do *Property;*
- *Accountability*: separa atributos e relacionamentos;
- *Strategy*: define o comportamento dos tipos de entidades;
- *RuleObject*: permite a construção de estratégias configuráveis.

Neste tipo de arquitetura, são ainda utilizados, em conjuntos com estes padrões citados, outros padrões como:

- *Composite*: constrói objetos em estruturas de árvore dinâmicas;
- *Interpreter* e *Builder*: são geralmente utilizados para construir estruturas a partir de um meta-modelo ou interpretar resultados.

Os padrões de projeto que compõe a arquitetura de modelos de objetos adaptáveis são detalhados nas seguintes seções.

5.2.1 Padrão *TypeObject*

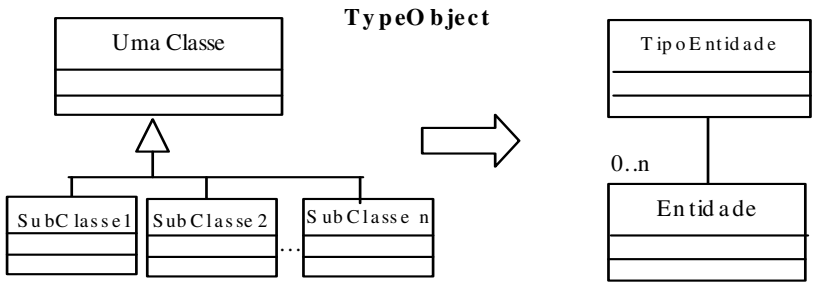
Geralmente, os sistemas orientados a objetos utilizam uma classe separada para cada tipo de objeto. Assim, para introduzir um novo tipo de objeto é necessário desenvolver uma nova classe e, conseqüentemente, implementar seu código correspondente. Um problema enfrentado geralmente por desenvolvedores ocorre quando se tem uma classe da qual se deseja criar um número desconhecido de subclasses (YODER, BALAGUER, JOHNSON, 2001).

O padrão *TypeObject* separa uma entidade de um tipo de entidade, fazendo com que um número desconhecido de subclasses se tornem simples instâncias de uma classe genérica. O *TypeObject* permite que novas classes sejam criadas em tempo de execução por meio da instanciação da classe genérica. Estas classes novas são chamadas *TypeObject* (JOHNSON, WOOF, 1997), (YODER, BALAGUER, JOHNSON, 2001).

A Figura 5.1 apresenta um exemplo onde, uma dada hierarquia é representada com a classe Tipo de Entidade e suas instâncias são representadas através das classes Entidade (YODER, BALAGUER, JOHNSON, 2001).

Observa-se na Figura 5.1 que a estrutura de herança classe Sensor com suas subclasses Sensor111, Sensor222 e Sensor333 passa a ser representada pela estrutura de classes Entidade e Tipo de Entidade.

Esta substituição é possível quando o comportamento entre subclasses é similar ou quando existe a possibilidade de dividi-las em objetos separados. Observa-se também nesta figura que a diferença principal entre as subclasses são seus atributos (YODER, JOHNSON, 2002).



Ex.: Padrão *TypeObject* para Sensor e Tipo de Sensores

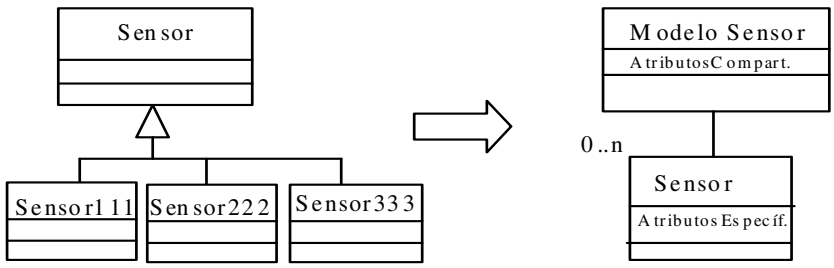


Figura 5.1 - Padrão *TypeObject*
Fonte Adaptada (YODER, 2007)

5.2.2 Padrão *Property*

Geralmente os atributos de um objeto são implementados através de uma variável de instância, sendo que estas variáveis são definidas em cada subclasse. Com a aplicação do padrão *TypeObject*, todos os objetos de tipos diferentes são pertencentes à mesma classe e a grande dificuldade é como variar os atributos desta classe.

A solução é utilizar o padrão *Property*, em que uma variável de instância englobe uma coleção de atributos, ao invés de cada atributo ser uma diferente variável de instância. Isto pode ser conseguido através de um dicionário, um vetor ou um *hashtable*. Cada atributo é associado com uma única chave que é utilizada para acessar, alterar e remover atributos em tempo de execução (MANOLESCU, JOHNSON, 1999). A Figura 5.2 ilustra um exemplo do padrão *Property*. Percebe-se que cada propriedade da classe Atuador passa a ser representada por uma classe propriedade.

Desta forma, através da aplicação do padrão *Property,* é possível variar os atributos das novas classes geradas e, assim, cada vez que um novo atributo é requerido, uma nova classe é criada.

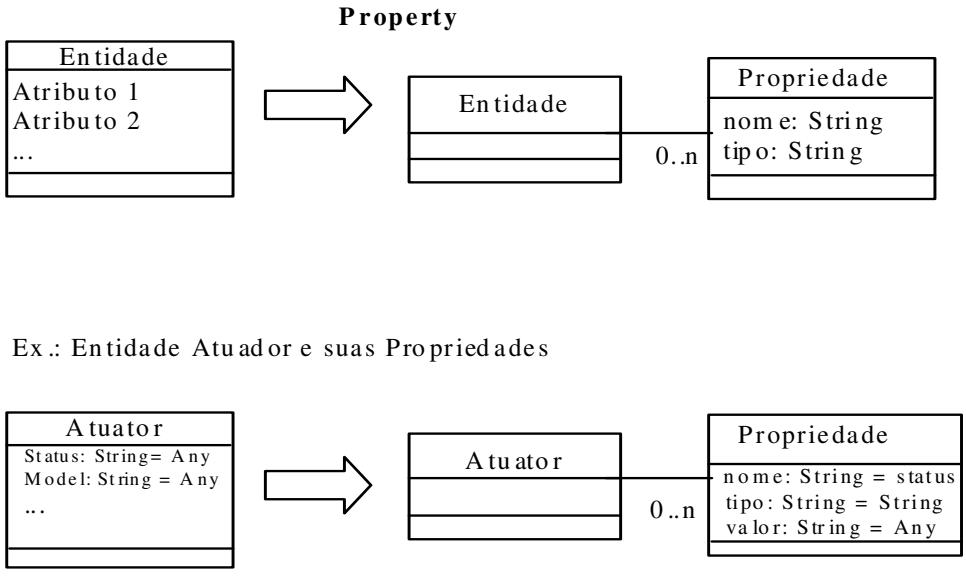


Figura 5.2 - Padrão *Property*
Fonte Adaptada (YODER, 2007)

5.2.3 Padrão *TypeSquare*

Geralmente, a arquitetura de modelos de objetos adaptáveis contém os padrões *TypeObject* e *Property* utilizados em conjunto. Este padrão é chamado *Type Square*.

No padrão *TypeSquare*, o padrão *TypeObject* é aplicado duas vezes, uma antes de se aplicar o padrão *Property* e outra após a aplicação deste padrão. O padrão *TypeObject* divide o sistema em Entidade e Tipos de Entidade e, estas entidades possuem atributos que podem ser definidos através do padrão *Property*. Cada propriedade tem um tipo, chamado TipoPropriedade e cada Tipo de Entidade pode então especificar os tipos de propriedades para suas entidades (YODER, BALAGUER, JOHNSON, 2001). A Figura 5.3 representa a arquitetura *TypeSquare*.

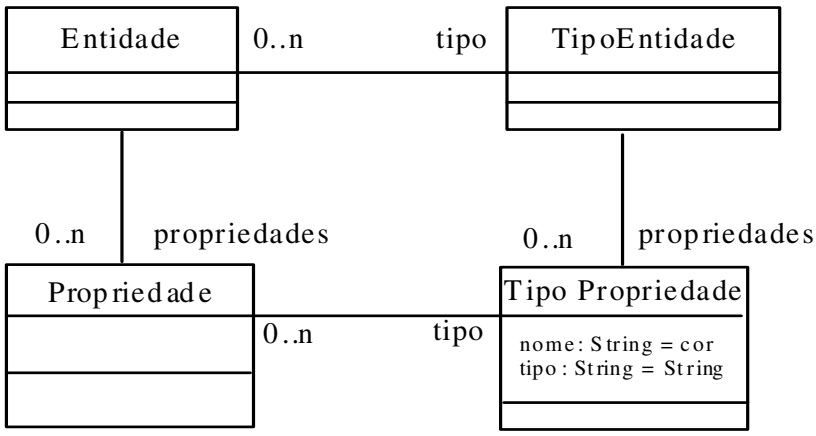


Figura 5.3 - Padrão *TypeSquare*
Fonte Adaptada (YODER, 2007)

Com este padrão é possível conhecer o nome da propriedade e se o valor de determinada propriedade é um número, um dado ou uma *string*, etc.

5.2.4 Padrão *Strategy*

Para sistemas orientados a objetos, as regras de negócios podem ser representadas de várias formas. Algumas regras definem os tipos de entidades nos sistemas, junto com seus atributos; outras regras podem definir os subtipos através de subclasses; outras regras definem restrições básicas tais como a cardinalidade de relacionamentos e se um determinado atributo é requerido ou não (YODER, JOHNSON, 2002).

Entretanto, existem regras de negócios que são mais complexas e que precisam ser definidas de formas diferentes. Um exemplo é o caso das regras de negócio que são relacionadas ao comportamento de um objeto. Para este tipo de regras de negócio, os modelos de objetos adaptáveis trabalham com estratégias que são funções básicas necessárias para novos tipos de entidades.

O padrão *Strategy* é aquele que define uma interface genérica para uma família de algoritmos para que clientes possam trabalhar com qualquer um deles. O comportamento do objeto é definido por uma ou mais estratégias. Por isso, é fácil alterar este comportamento, bastando alterar a estratégia existente ou criar uma nova estratégia (YODER, JOHNSON, 2002).

O padrão *Strategy* define uma série de algoritmos onde cada um deles é encapsulado em um objeto, tornando-os intercambiáveis. O padrão S*trategy* faz com que o algoritmo varie independente dos clientes que o utiliza (GAMMA et al.,1995).

A Figura 5.4 apresenta a estrutura do padrão *Strategy*.

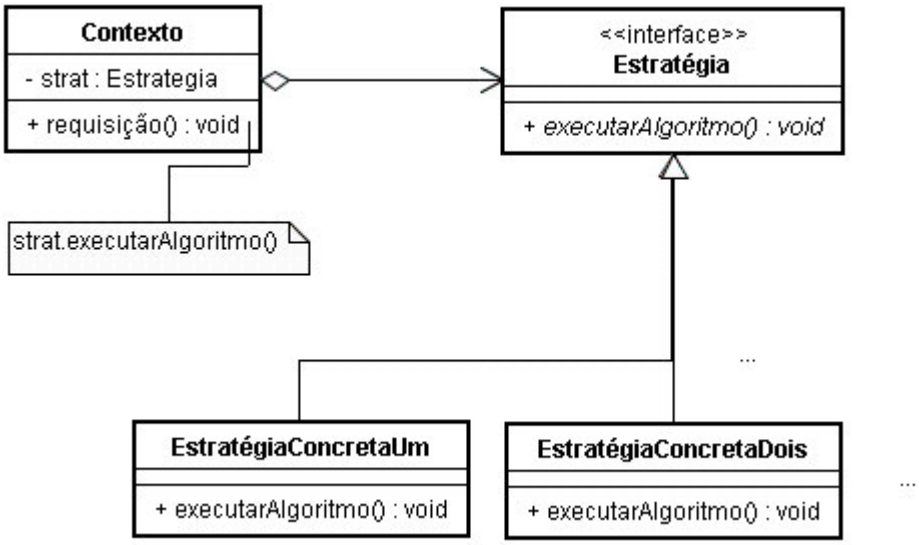


Figura 5.4 - Padrão *Strategy*
Fonte Adaptada (GAMMA et al., 1995)

A estrutura do padrão *Strategy* é composta por uma interface Estratégia, uma classe Contexto e classes EstrategiaConcreta. A interface Estratégia declara uma interface comum para todos os algoritmos relacionados. Esta interface é utilizada pela classe Contexto para chamar o algoritmo definido pelo objeto EstratégiaConcreta.

O objeto EstratégiaConcreta implementa o algoritmo utilizando a interface Estratégia. A classe Contexto é configurada com um objeto EstratégiaConcreta e mantém a referência para o objeto Estratégia. A classe Contexto também define uma interface que permite a classe Estratégia acessar seus dados (GAMMA et al.,1995).

5.2.5 Padrão *Accountability*

O padrão *Accountability* é utilizado para definir relacionamentos entre classes e definir regras que administram estes relacionamentos.

Atributos são propriedades que geralmente se referem aos dados primitivos como números, *strings* ou datas. Entidades possuem uma associação simples (*one-way association*) com seus respectivos atributos. Relacionamentos são propriedades que

se referem às outras entidades e possuem associação binária (*two-way association*) (YODER, JOHNSON, 2002). Esta distinção, que já faz parte da clássica modelagem entidade-relacionamento e atualizada nas notações modernas da orientação a objetos, também faz parte da arquitetura de modelos de objetos adaptáveis. Assim, esta distinção leva a divisão das propriedades em duas subclasses, uma de atributos e outra de relacionamentos.

Uma forma de separar atributos de relacionamentos é utilizar o padrão *Property* duas vezes, uma para atributos e outra para associação. Um caminho alternativo para realizar esta separação é criando duas subclasses de *Property*, uma subclasse para atributos e outra para associação, em que esta última deverá conhecer sua cardinalidade e é chamada de *Accountability* (YODER, BALAGUER, JOHNSON, 2001).

Como mostra a Figura 5.5, grupos de um domínio podem ter múltiplos relacionamentos entre eles e o sistema precisa conhecer estes relacionamentos e outras regras relevantes de negócio para manipulá-los. Desta forma, a aplicação precisa mostrar os relacionamentos entre entidades. Cada relacionamento atribui papéis aos seus participantes, sendo que estes papéis podem mudar e o sistema precisa saber quais são os papéis correspondentes de seus participantes e quais estão disponíveis (YODER, JOHNSON, 2003).

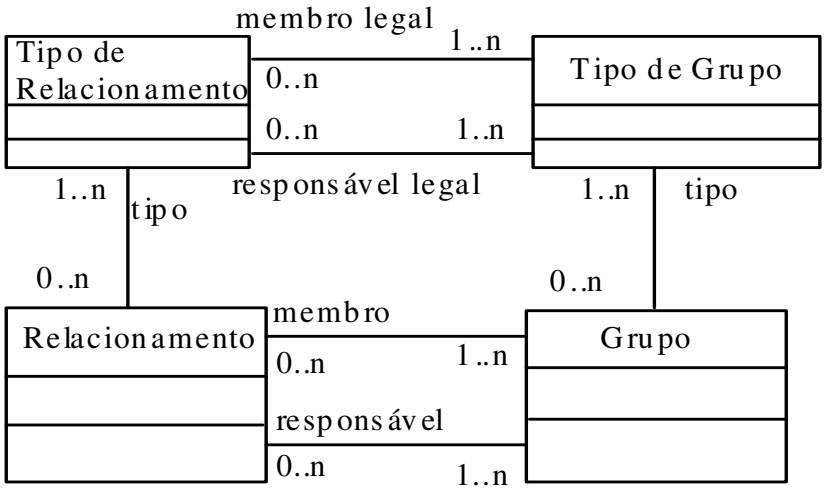


Figura 5.5 - Padrão *Accountability*
Fonte Adaptada (YODER, 2007)

Cada instância de um Relacionamento representa um relacionamento entre dois grupos, o Tipo de Relacionamento indica a natureza do relacionamento, o que permite manipular qualquer número de relacionamentos entre grupos. Instâncias de Tipo de Relacionamento são criadas para cada tipo de relacionamento necessário. Cada Tipo de Relacionamento deve conhecer os grupos, que possuem relacionamentos daqueles tipos, associados a ele.

5.2.6 Padrão *Composite*

O objetivo do padrão *Composite* é compor objetos em estruturas de árvore para representar hierarquias todo-parte. O padrão *Composite* permite clientes tratar da mesma forma, tanto objetos individuais quanto objetos compostos (GAMMA et al.,1995).

A característica principal do padrão *Composite* é possuir uma classe abstrata que representa os objetos primitivos e compostos. Assim é possível tratar grupos de objetos e objetos individuais através de uma única interface, o que minimiza a complexidade da hierarquia.

A Figura 5.6 apresenta a estrutura do padrão *Composite*.

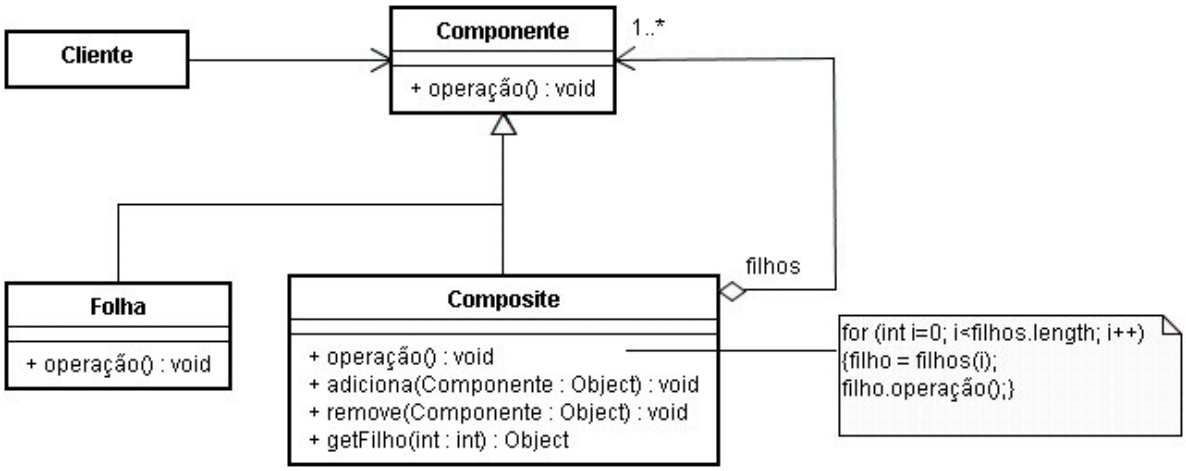


Figura 5.6 - Padrão *Composite*
Fonte Adaptada (GAMMA et al., 1995)

A estrutura do padrão *Composite* é constituída pelos objetos Componente, Folha, *Composite* e Cliente.

O objeto Componente declara a interface para os objetos que fazem parte da classe *composite* e implementa um comportamento padrão para a interface comum, de forma apropriada, para todas as classes. Assim, esta classe representa qualquer tipo de objetos na classe *composite*. O objeto Componente declara uma interface para acessar e gerenciar seus componentes filhos. Este objeto também é responsável por definir e implementar, se necessário, uma interface para acessar um objeto pai de um componente em um estrutura recursiva.

Os objetos Folhas são aqueles que não possuem objetos filhos e definem o comportamento de objetos primitivos na classe *composite*. O objeto *Composite* define o comportamento para objetos que tem objetos filhos e guardam os objetos filhos. Esta classe é responsável por implementar operações como inclusão ou remoção relacionadas aos objetos filhos na interface Componente. O objeto cliente manipula objetos na classe *composite* através da interface Componente.

O padrão *Composite* torna mais fácil o acréscimo de novos tipos de componentes, já que os clientes não precisam ser alterados para tratar novas subclasses de Componente ou *Composite*.

O Cliente utiliza a classe interface Componente para interagir com os objetos na classe *composite*. Se o receptor for um objeto Folha, a requisição é realizada diretamente, entretanto, se o receptor for um objeto *Composite*, então a requisição é passada para seus componentes dos filhos, realizando operações adicionais antes ou depois da requisição ser completada (GAMMA et al.,1995).

5.2.7 Padrão *RuleObject*

Muitas vezes, a necessidade de utilização de regras de negócio mais poderosas faz com que as estratégias evoluam e se tornem mais complexas. Estas estratégias mais complexas são chamadas de *RuleObjects.* O objetivo do padrão *RuleObject* é fazer com que o projeto e implementação de processos de negócios

computadorizados sejam extensíveis e adaptáveis, sem causar muitas mudanças, já que as regras que governam estes processos são intercambiáveis (ARSANJANI, 2001), (CARDOSO, 2005).

Através do padrão *RuleObject* é possível obter o conceito de estratégias configuráveis, já que, novas regras podem ser criadas a partir de regras já existentes (THOMÉ, 2004), (CARDOSO, 2005). A Figura 5.7 apresenta a estrutura do padrão *RuleObject.*

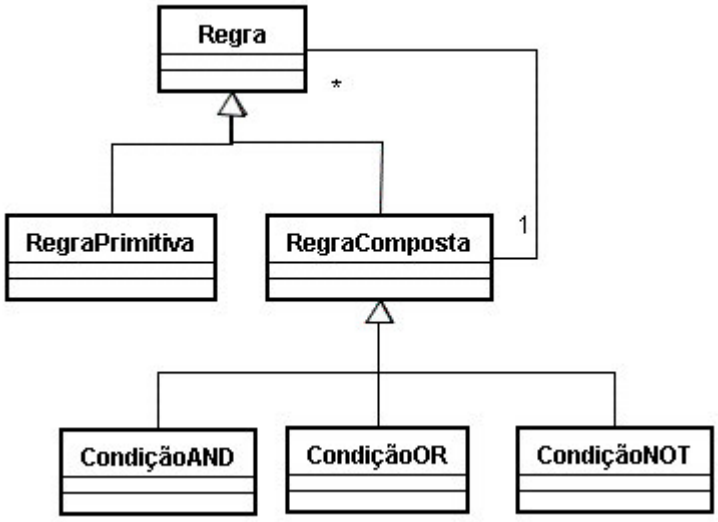


Figura 5.7 - Padrão *RuleObject*
Fonte Adaptada (YODER, 2007)

As regras podem ser primitivas ou uma combinação de regras através da utilização do padrão *Composite*. As regras que representam predicados são compostas por conjunções e disjunções; as regras que representam valores numéricos são compostas por regras de adição e subtração; as regras que representam conjuntos são compostas por regras de união e intersecção (YODER, JOHNSON, 2002).

Uma arquitetura freqüentemente encontrada em sistemas adaptáveis é mostrada na Figura 5.8 que representa um diagrama criado através da adição do padrão *RuleObject,* para a representação do comportamento, com o padrão *TypeSquare* .

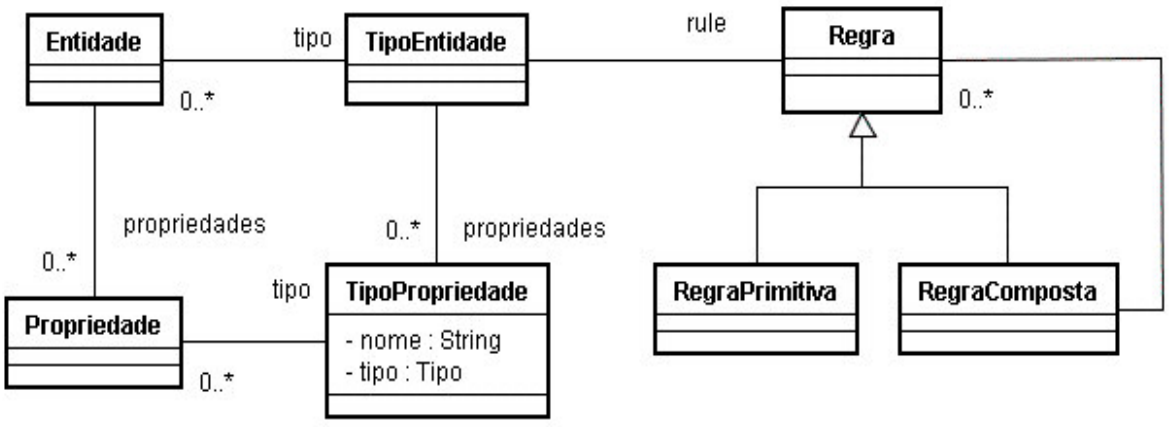


Figura 5.8 - *TypeSquare* com regras
Fonte Adaptada (YODER, JOHNSON, 2002)

Se as regras de negócio descrevem um *workflow*, uma arquitetura *Micro-Workflow* como a descrita em Manolescu (2000), pode ser usada. A arquitetura *Micro-Workflow* descreve classes que representam estrutura *workflow* como uma combinação de regras tais como repetição, condicional, seqüencial, bifurcação e regras primitivas. Estas regras podem ser construídas em tempo de execução para representar um processo workflow particular (YODER, JOHNSON, 2002).

Segundo Yoder e Johnson (2002) as regras também podem se construídas a partir de sistemas *table-driven* ou por uma abordagem orientada à gramática. Nos sistemas *table-driven*, as regras são capturadas como elementos de modelos lógicos e implementadas como tabelas de base de dados, *triggers* e *object actions* (PERKINS, 2000). A abordagem orientada à gramática, também conhecida como Projeto de Objeto Orientado à Gramática (*Grammar-oriented Object Design* - GOOD), é um método de identificação e mapeamento de sub-sistemas reusáveis em modelos de negócio para arquiteturas de software baseadas componentes. Outras informações sobre este método para a construção de regras pode ser encontrados em (ARSANJANI, ALPIGINI, 2001).

5.2.8 Padrão *Interpreter*

O padrão *Interpreter* define uma representação e um interpretador para sua gramática, ou seja, define como representar e interpretar sentenças na linguagem. A Figura 5.9 ilustra a estrutura do padrão *Interpreter*.

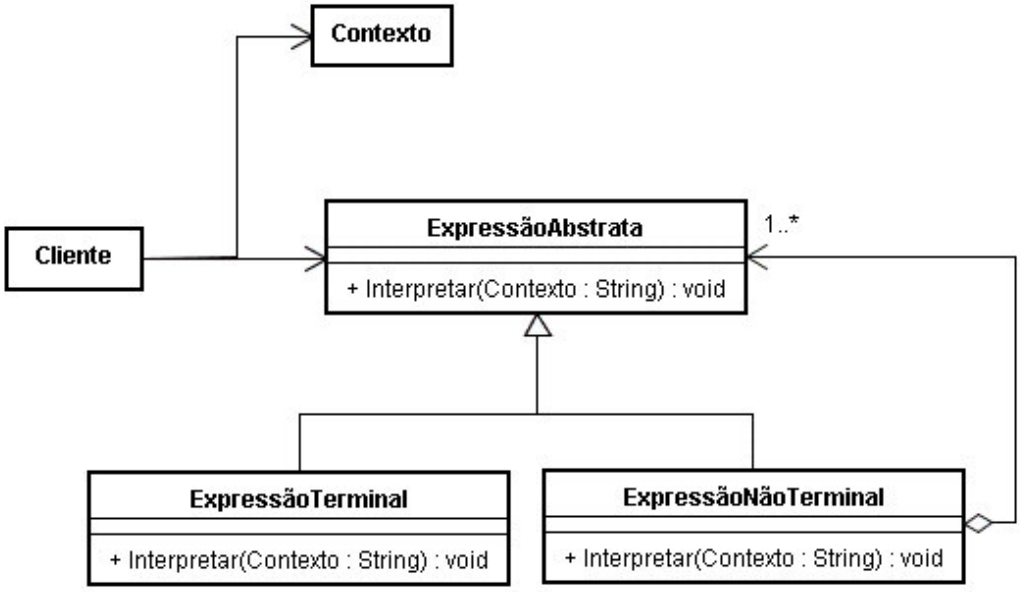


Figura 5.9 - Padrão *Interpreter*
Fonte Adaptada (GAMMA et al., 1995)

Os participantes desta estrutura são as classes ExpressãoAbstrata. ExpressãoTerminal, ExpressãoNãoTerminal, Contexto e Cliente (GAMMA et al.,1995).

O objeto ExpressãoAbstratra declara uma operação Interpretar que é comum para todos os nós da árvore. A ExpressãoTerminal implementa uma operação Interpretar associada com símbolos terminais na gramática; portanto, para cada símbolo terminal na sentença, é requerido um objeto Expressão Terminal.

Uma classe do tipo ExpressãoNãoTerminal é requerida para toda regra *R :: R1R2...Rn* na gramática, onde para cada símbolos R1 até Rn é necessário manter uma instância do tipo ExpressãoAbstrata para implementar uma operação Interpretar para os símbolos terminais da gramática. Esta operação chama a si mesma recursivamente nas variáveis representando R1 até Rn.

A classe Contexto contém informações que são globais ao interpretador. A classe Cliente constrói uma árvore de sintaxe abstrata representando uma sentença particular na linguagem que a gramática define. A árvore de sintaxe abstrata é construída a partir das instâncias das classes ExpressãoNãoTerminal e ExpressãoTeminal. A classe Cliente ainda possui a responsabilidade de chamar a operação Interpretar.

Desta forma, o Cliente constrói uma sentença na forma de uma árvore de sintaxe abstrata com instâncias de ExpressãoNãoTerminal e ExpressãoTerminal, inicia o objeto Contexto e chama a operação Interpretar. A operação Interpretar de cada ExpressãoTerminal define o caso base na recursão. A operação Interpretar em cada nó usa o objeto Contexto para armazenar e acessar o estado do interpretador (GAMMA et al., 1995).

5.2.9 Padrão *Builder*

O padrão *Builder* separa a construção de um objeto complexo de sua representação para que o mesmo processo de construção possa criar diferentes representações. O *Builder* permite que uma classe se preocupe apenas com a construção do objeto. A estrutura do padrão *Builder* pode ser observada na Figura 5.10.

Os participantes do padrão *Builder* são as classes Construtor (*Builder*), ConstrutorConcreto, Diretor e Produto.

O Construtor possui a função de especificar a interface abstrata para criar partes de um objeto Produto. O objeto ConstrutorConcreto constrói e monta partes do Produto pela implementação da interface Construtor. Este objeto também é responsável por definir e rastrear a representação por ele criada e fornecer uma interface para recuperar o produto.

A classe Diretor constrói o objeto utilizando a interface Construtor. O Produto representa o objeto complexo que será construído. A classe ConstrutorConcreto constrói a representação interna do Produto e define o processo pelo qual o mesmo deve ser montado (GAMMA et al.,1995).

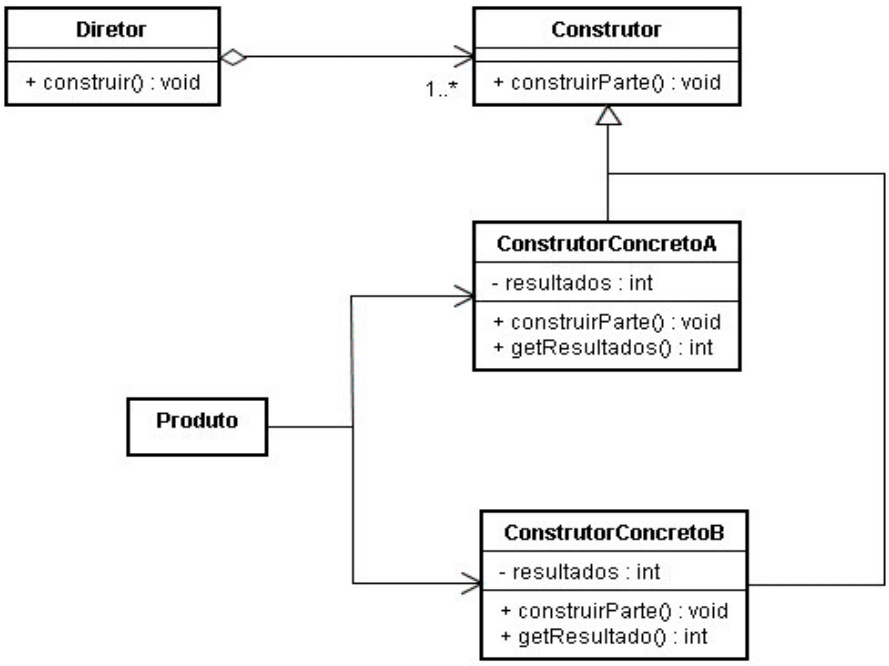


Figura 5.10 - Padrão *Builder*
Fonte Adaptada (GAMMA et al., 1995)

Desta forma, quando o cliente necessita de um Produto passa as informações necessárias para a classe Diretor. Utilizando estas informações, a classe Diretor ordena a criação do Produto através do Construtor, que é habilitado a construir qualquer Produto através de uma interface uniforme. Assim, a classe Diretor seleciona o Construtor e envia as informações necessárias à construção do Produto. O Produto construído pode ser acessado diretamente do Construtor.

5.3 Construção da Arquitetura de Modelos de Objetos Adaptáveis

A arquitetura de modelos de objetos adaptáveis é geralmente constituída pela aplicação de um ou mais padrões de projetos apresentados. A Figura 5.11 apresenta uma arquitetura construída através dos padrões de modelos de objetos adaptáveis mais utilizados.

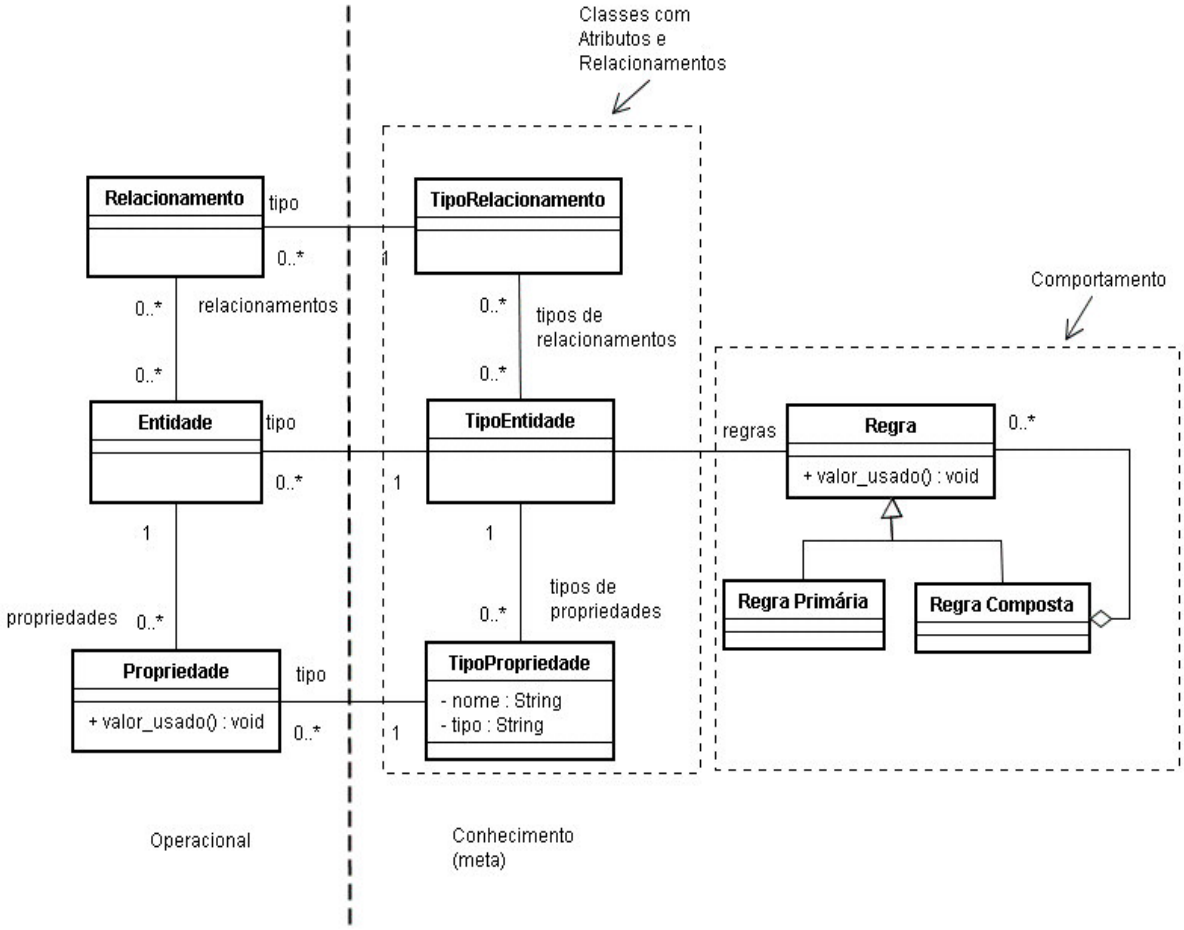


Figura 5.11 - Arquitetura de modelos de objetos adaptáveis
Fonte Adaptada (YODER, JOHNSON, 2003)

A arquitetura de modelos de objetos adaptáveis apresentada na Figura 5.11 é dividida em dois níveis: o Operacional e o Conhecimento. O nível Operacional é composto por classes Propriedade, Entidade e Relacionamento. Estas classes guardam informações relacionadas ao domínio do problema. O nível de Conhecimento possui classes como tipo de relacionamento, tipo de entidade e tipo de propriedade e as classes destinadas ao comportamento do sistema. Estas classes são responsáveis por guardarem informações das classes do nível operacional.

Observa-se também que na arquitetura de modelos de objetos adaptáveis, apresentada na Figura 5.11, foram aplicados o padrão *TypeSquare* para especificar as entidades, tipos de entidades, propriedades e tipos de propriedades; o padrão

Accountability para especificar relacionamentos e tipos de relacionamentos; e o padrão *RuleObject* para especificar as regras de negócio.

Entretanto, neste trabalho, para utilizar os modelos de objetos adaptáveis no mecanismo de variabilidade do processo GVLPS, definiu-se a arquitetura apresentada na Figura 5.12.

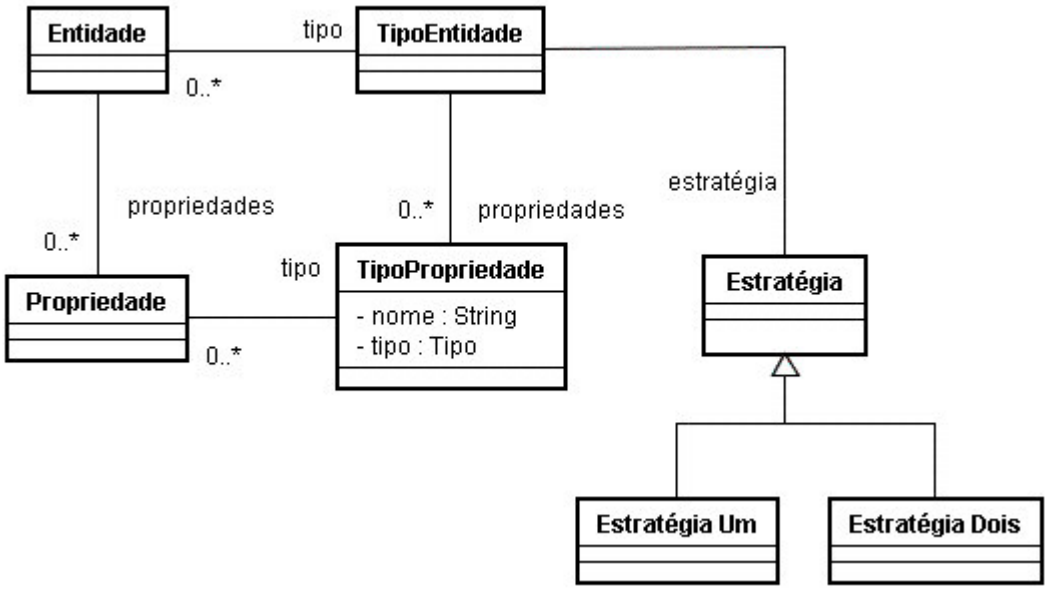


Figura 5.12 - Arquitetura de modelos de objetos adaptáveis para o mecanismo de variabilidade do processo GVLPS
Fonte Adaptada (YODER, 2007)

Esta arquitetura é constituída pelos padrões de projetos *TypeObject, Property, TypeSquare* e *Strategy* e se mostra suficiente para o objetivo de variabilidade desejado, porque através destes padrões já é possível variar a entidade, o tipo de entidade, suas propriedades, tipos de propriedades e seu comportamento.

Assim, para se implementar o mecanismo de variabilidade baseado nesta arquitetura, os padrões *TypeObject, Property, TypeSquare* e *Strategy* devem ser aplicados no modelo que representa a variabilidade no processo GVLPS.

A aplicação de um padrão de projeto em um modelo significa transformar este modelo de acordo com o padrão utilizado. O processo de transformar um modelo usando um padrão de projeto é denominado *pattern based refactoring* e é realizado da seguinte forma (FRANCE et al., 2003):

- Identificar os elementos do modelo fonte que irão participar da solução apresentada pelo padrão de projeto. Modificá-los de forma a se comportarem com o padrão de projeto especificado;

- Adicionar novos elementos ao modelo de acordo com as necessidades a fim de que se possa realizar o comportamento especificado no padrão de projeto selecionado.

Através da arquitetura apresentada na Figura 5.12 é possível criar novas entidades e variar o tipo de entidade, suas propriedades, seus tipos de propriedades e seu comportamento. Por isso, com um mecanismo de variabilidade baseado nestes padrões é possível permitir a variação do produto de forma mais flexível.

5.4 Considerações Finais

A reflexão é considerada uma técnica promissora e emergente mesmo para sistemas complexos (RUIZ et. al., 2003).

Apesar de ser uma técnica de difícil entendimento, o maior benefício oferecido pela reflexão é a flexibilidade, já que as arquiteturas reflexivas podem dinamicamente se adaptar a novos requisitos em tempo de execução.

A idéia de se utilizar padrões de projeto para modelar variabilidade em linha de produtos de software não é nova, pois autores como Keepence e Mannion (1999) e van Gurp, Bosch e Svahnberg (2001) já utilizaram padrões de projetos em seus trabalhos, criando técnicas que auxiliaram desenvolvedores reconhecer mais facilmente onde a variabilidade é necessária. O trabalho apresentado em (KEEPENCE, MANNION, 1999) utiliza padrões de projetos para modelar a variabilidade na linha de produtos de software na fase de projeto. Já o trabalho descrito em (VAN GURP, BOSCH, SVAHNBERG, 2001) cita padrões de variabilidade para fazer a classificação de diferentes técnicas de realização de variabilidade.

Os modelos de objetos adaptáveis são considerados uma alternativa interessante para o projeto orientado a objetos, entretanto, como toda arquitetura, possui vantagens e desvantagens.

A utilização de modelos de objetos adaptáveis pode ser vantajosa principalmente quando o sistema é continuamente alterado e os usuários devem ser capazes de estendê-lo. Neste caso, a opção de se utilizar modelos de objetos adaptáveis pode ser muito adequada, já que esta arquitetura oferece facilidade para adaptar o sistema aos novos requisitos de negócio (YODER, 2007).

Outra vantagem proporcionada pelos modelos de objetos adaptáveis é a redução do número de classes, o que facilita a alteração do sistema, tornando-o mais fácil de entendê-lo e mantê-lo (JOHNSON, 1998).

A flexibilidade também é um fator a ser considerado já que, quando ocorrer mudanças no sistema, o mesmo não requer nova compilação. Em conseqüência da diminuição do tempo de desenvolvimento, outra vantagem a ser considerada é a redução do *time to market.*

Desta forma, facilidades para alteração, extensão e flexibilidade são as vantagens fornecidas pelos objetos adaptáveis que levou este trabalho a propor um mecanismo de variabilidade para o processo GVLPS baseado nesta técnica.

Entretanto, os modelos de objetos adaptáveis são considerados difíceis para construir porque são constituídos de dois sistemas de objetos: (1) o interpretador escrito em linguagem de programação orientada a objeto e (2) o modelo de objetos do domínio de negócio, que é interpretado em tempo de execução, podendo ser alterado de forma imediata e controlada no sistema que o interpreta (JOHNSON, 1998). Esta dificuldade exige profissionais experientes para entendê-los e desenvolvê-los. Desta forma, as instituições devem ser cautelosas ao selecionar esta técnica, planejando um treinamento apropriado e documentação adequada.

Além disso, os modelos de objetos adaptáveis necessitam de uma estrutura para armazenar, construir e interpretar seus dados, assim como mão de obra para operar esta estrutura.

Como o sistema baseado em modelos de objetos adaptáveis é um interpretador, ele apresenta um baixo desempenho. A contínua evolução tecnológica associada às técnicas de otimização do sistema podem apresentar melhorias com relação a este baixo desempenho.

"Deus envia o vento,
mas é o homem que deve içar as velas."

Santo Agostinho

6 PROCESSO PARA GERENCIAMENTO DE VARIABILIDADE DE LINHA DE PRODUTOS DE SOFTWARE - GVLPS

Este capítulo apresenta o processo para Gerenciamento de Variabilidade de Linha de Produtos de Software - GVLPS, definido neste trabalho.

O propósito deste processo é facilitar a análise, a seleção e a implementação das soluções reutilizáveis de software para a derivação de novos produtos para linha de produtos de software, que serão considerados membros da mesma família. Para isso, o processo fornece os passos para a identificação e projeto das variantes durante a atividade de Engenharia de Domínio, assim como, para a utilização de um mecanismo para implementar estas variantes dentro de diferentes aplicações na atividade de Engenharia da Aplicação.

O processo GVLPS pretende estabelecer uma abordagem sistemática para a estruturação de linha de produtos de software buscando, com isso, o aumento do número de possíveis produtos derivados da linha de produtos de software e a melhoria da qualidade dos artefatos produzidos.

O diferencial deste processo, em relação aos demais trabalhos encontrados na literatura (VAN GURP, BOSCH, SVAHNBERG, 2001), (OLIVEIRA et al., 2005), (KIM et al., 2006), é o mecanismo de variabilidade sugerido, o qual é baseado em modelos de objetos adaptáveis e em reflexão. A partir deste mecanismo, pretende-se realizar as alterações no sistema com maior flexibilidade, facilitando a reutilização dos seus artefatos.

6.1 Contexto do Processo para Gerenciamento de Variabilidade de Linha de Produtos de Software

O processo GVPLS foi definido com base nos conceitos de duas abordagens de linha de produtos de software: ESPLEP - *Evolutionary Software Product Line Engineering Process*, baseada em UML (GOMAA, 2005) e FSPLP - *Framework for Software Product Line Practice* (NORTHROP, CLEMENTS, 2007), ambas descritas no Capítulo 3.

A Figura 6.1 mostra o relacionamento do processo GVLPS com as atividades da linha de produtos de software.

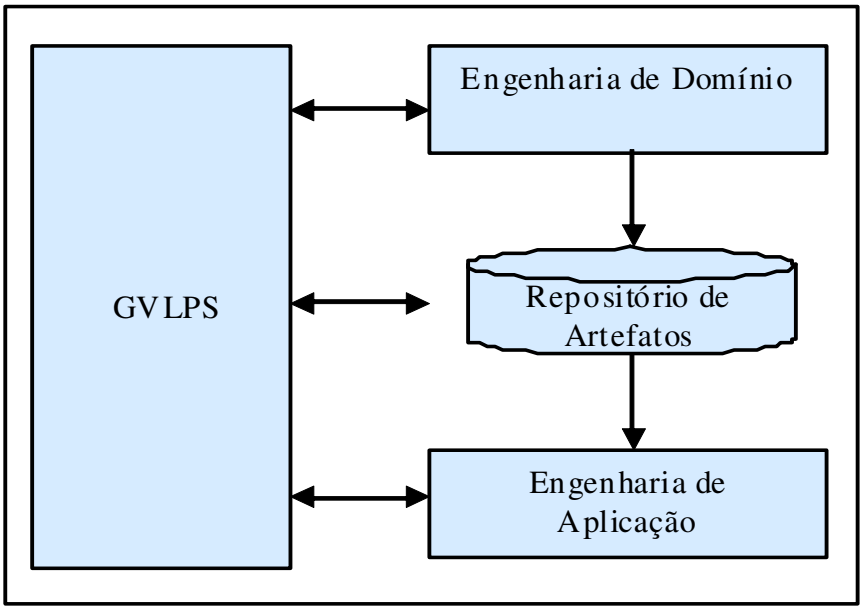


Figura 6.1 - Contexto do GVLPS

Este diagrama foi elaborado com base na Figura 3.1 (Atividades Principais para Linha de Produtos de Software) e na Figura 3.5 (Processo de Engenharia de Linha de Produtos de Software Evolucionário), as quais ilustram os processos das abordagens utilizadas como referência.

O processo GVLPS pode ser considerado como uma sub-atividade da atividade de Gerenciamento Técnico da abordagem FSPLP, a qual está totalmente interligada e

interage com as atividades de Engenharia de Domínio e de Engenharia de Aplicação, apoiando a criação e a e evolução do *core assets* e dos produtos (NORTHROP, CLEMENTS, 2007). O ESPLEP (GOMAA, 2005), por seu lado, é compatível com o FSPLP e fornece uma visão mais detalhada do processo de linha de produtos de software.

Durante a Engenharia de Domínio, decisões iniciais sobre variabilidade - como, por exemplo, quais requisitos são comuns ou variáveis - devem ser tomadas e descritas nos artefatos adequados. As decisões sobre a variabilidade, mais relacionadas com implementação - como, por exemplo, qual a técnica de realização da variabilidade escolhida para implementar as variantes - são tomadas durante a Engenharia de Aplicação, quando um produto é construído. O processo GVLPS utiliza artefatos do repositório produzidos durante a Engenharia de Domínio para gerar as variantes durante a Engenharia de Aplicação.

Além de FSPLP e ESPLEP, o processo GVLPS também se baseia nos passos do método de gerenciamento de variabilidade recomendados por van Gurp, Bosch, Svahnberg (2001), apresentado na Seção 4.3 deste trabalho.

Este método foi adotado como base deste trabalho pelo fato de ser um dos primeiros trabalhos a apresentar uma proposta de atividades para gerenciamento de variabilidade. O trabalho foi citado por vários artigos da área (BECKER et al., 2002), (SCHNIEDERS, 2006a), (OLIVEIRA et al., 2005) e (BÜHNE, HALMANS, POHL, 2003) e originou a proposta de gerenciamento de variabilidade de Oliveira Junior (2005).

O processo GVLPS também se fundamenta nos trabalhos de Gomaa (2005), Griss, Favaro, d'Alessando (1998) e Czarnecki, Eisenecker (2005) que apresentam modelo de *features* para a definição da variabilidade.

6.2 Descrição do Processo para Gerenciamento de Variabilidade de Linha de Produtos de Software

Esta seção apresenta a descrição em alto nível do processo GVLPS. Suas atividades são apresentadas na Figura 6.2 e consistem de Identificação, Especificação e Implementação de Variabilidade.

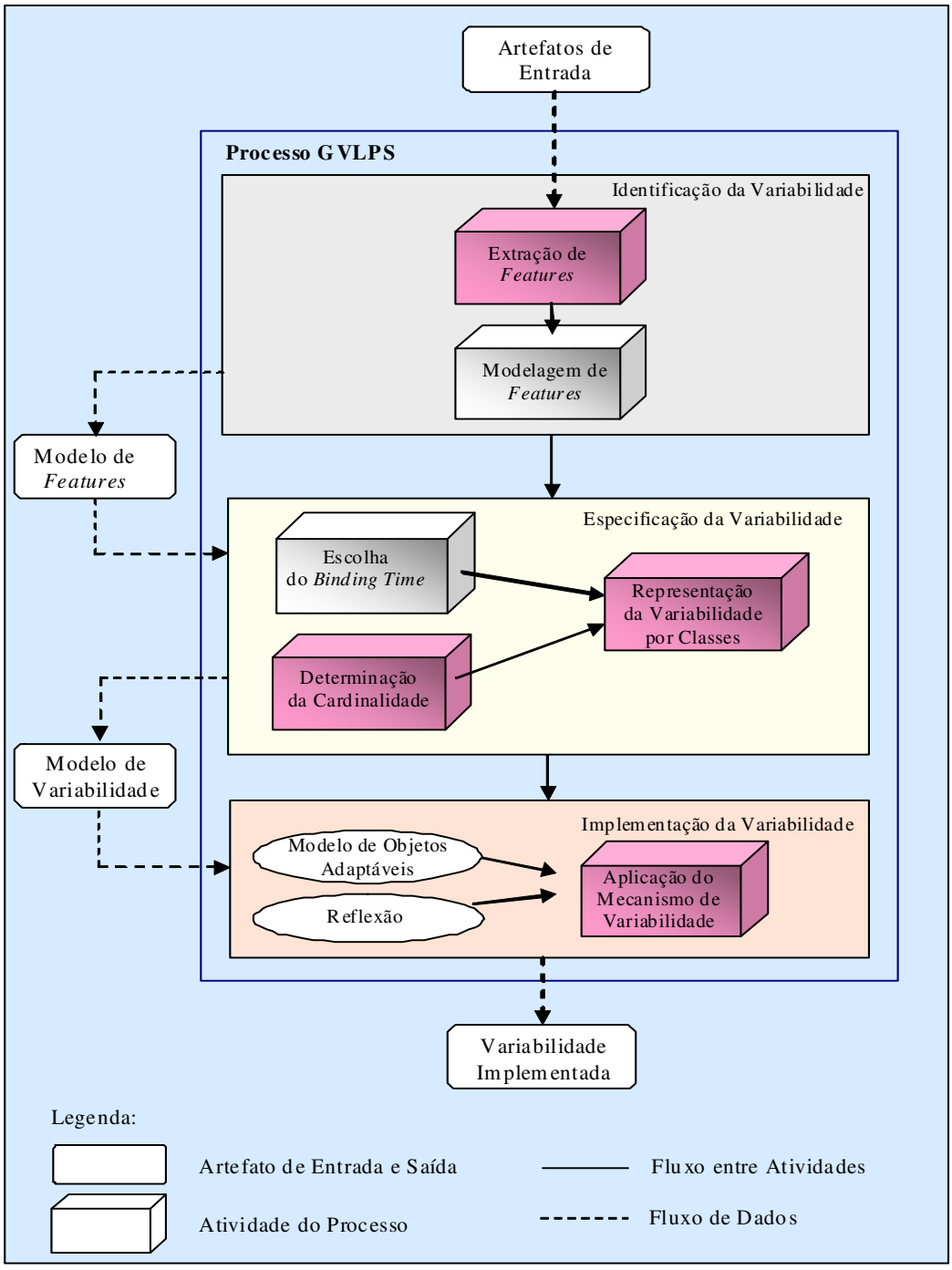


Figura 6.2 - Processo GVLPS

As entradas do processo GVLPS são geradas pelas atividades de Análise de Domínio e consistem de Descrição Textual do Problema, Extração dos Requisitos e Definição dos Casos de Uso.

O processo GVLPS inicia-se com a atividade de Identificação da Variabilidade que consiste da Extração de *Features* e Modelagem de *Features*. A atividade de Extração de *Features* utiliza o modelo de casos de uso de linha de produtos de software para identificar as variantes e a atividade de Modelagem de *Features* representa estas variantes através de um modelo de *features*.

Após a Identificação da Variabilidade, é feita a Especificação da Variabilidade, descrevendo-a com mais detalhes, para que seja mais facilmente implementada por algum mecanismo de variabilidade. Para isso, algumas decisões são tomadas; são exemplos de decisões, a determinação da cardinalidade dos relacionamentos entre as *features* e os momentos em que as variantes são selecionadas (tempo de resolução ou *binding time*). A variabilidade, inicialmente representada por um modelo de *features*, é convertida em um diagrama de classes que expressa mais detalhes sobre a variabilidade, antes de iniciar sua implementação.

Finalmente, para a Implementação da Variabilidade, o GVLPS adotou a aplicação de um mecanismo de variabilidade baseado em modelos de objetos adaptáveis e em reflexão, devido às facilidades de extensão, alteração e flexibilidade que estas técnicas proporcionam.

Para a definição do processo GVLPS, as atividades do método de gerenciamento de variabilidade proposto por van Gurp, Bosch e Svahnberg (2001), apresentado na Seção 4.3, foram reavaliadas e detalhadas. A Tabela 6.1 resume o resultado da comparação entre as atividades do processo de gerenciamento de variabilidade de van Gurp, Bosch e Svahnberg (2001) e do processo GVLP. O símbolo ✓ indica a existência da atividade no processo.

Tabela 6.1 - Comparação entre o processo de van Gurp, Bosch e Svahnberg (2001) e o processo GVLPS

Atividades	**Processo de (VAN GURP, BOSCH, SVAHNBERG, 2001)**	**Processo GVLPS**
Identificação da Variabilidade	✓	✓
Extração de *Features* a partir de modelo de casos de uso de linha de produtos de software		✓
Modelagem de *Features*	✓	✓
Decisões sobre a Variabilidade	Atribuição de Restrições à Variabilidade	Especificação de Variabilidade
Escolha do *Binding Time*	✓	✓
Determinação da Cardinalidade		✓
Representação da Variabilidade por Classes		✓
Aplicação do Mecanismo de Variabilidade		✓
Gerenciamento de Variantes	✓	✓

A atividade Atribuição de Restrições à Variabilidade foi denominada de Especificação de Variabilidade no processo GVLPS, pois detalha a variabilidade através das atividades de determinação da sua cardinalidade e de sua representação a partir dos diagramas de classes. As atividades que são mantidas no processo GVLPS são a Identificação da Variabilidade, a Modelagem de *Features* e a Escolha do *Binding Time.* Também são incorporadas novas atividades no processo GVLPS: a Extração de *Features* a partir de casos de uso para a Identificação da Variabilidade, a Determinação da Cardinalidade, a Representação da Variabilidade por Classes e a Aplicação do Mecanismo de Variabilidade.

Além disso, o processo GVLPS detalha como as atividades são realizadas, assim como, quais os artefatos de entradas e saídas de cada atividade. A atividade de Gerenciamento de Variantes, apesar de não estar explícita na Figura 6.2, é realizada através do mecanismo de variabilidade do processo GVLPS, pois este mecanismo permite a manutenção, correção e adaptações das variantes.

A atividade de Aplicação do Mecanismo de Variabilidade é o maior diferencial do processo GVLPS, em relação ao processo de van Gurp, Bosch e Svahnberg (2001).

Esta atividade foi detalhada, através do mecanismo de variabilidade baseado em modelos de objetos adaptáveis e em reflexão, enquanto que van Gurp, Bosch e Svahnberg (2001) apenas listam opções de mecanismos possíveis de serem aplicados.

6.3 Artefatos de Entrada

As atividades apresentadas na Figura 6.3 - Descrição Textual do Problema, Extração dos Requisitos e Definição dos Casos de Uso - fazem parte da análise do domínio. Apesar de não fazerem parte do processo GVLPS, são apresentadas porque são responsáveis pelo desenvolvimento dos artefatos que constituem o *core asset,* os quais são utilizados como entrada para processo GVLPS.

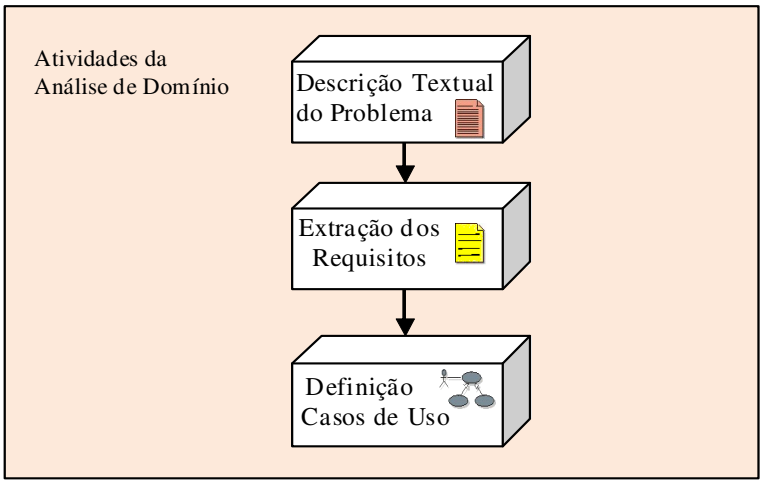


Figura 6.3 - Atividades para obtenção dos artefatos de entrada

A primeira atividade a ser realizada é a Descrição Textual do Problema, cujo texto resultante permite a Extração dos Requisitos do Sistema. O processo de Extração dos Requisitos do Sistema não é apresentado neste trabalho, pois o detalhamento desta atividade não faz parte do escopo desta tese.

A partir do conjunto de requisitos extraídos, é realizada a atividade de Definição dos Casos de Uso, cujo resultado é o modelo de casos de uso de linha de produtos de

software, constituído por diagramas e descrição dos casos de uso. O modelo de caso de uso de linha de produtos de software é o principal artefato de entrada do GVLPS e é utilizado para identificar as possíveis variantes.

Este artefato deve ser elaborado com base nas recomendações de modelagem UML para a linha de produtos de software, apresentado na abordagem ESPLEP (GOMAA, 2005), descrita na Seção 3.3.2 deste trabalho. Esta modelagem define os casos de uso com foco em linha de produtos de software e não em um único sistema.
Para o processo GVLPS, os casos de uso são classificados em comum, opcional e alternativo e esta informação é representada no diagrama através de estereótipos. Desta forma, um caso de uso pode ser:

- comum - é aquele utilizado em todas as aplicações de uma linha de produtos de software;
- opcional - é aquele utilizado em algumas aplicações, não em todas;
- alternativo - é aquele que possui implementações alternativas.

Para exemplificar a utilização da classificação de casos de uso, é apresentado na Figura 6.4, o diagrama de casos de uso de linha de produtos de software para o software de uma câmera digital para fotografias.

Observa-se que o caso de uso Tirar Foto é do tipo comum, ou seja, representa as características comuns que todas as câmeras desta linha de produtos de software devem possuir. Apesar disso, seu detalhamento pode apresentar variabilidade, ou seja, variações nos detalhes de sua descrição. Por exemplo, para se tirar foto pode-se precisar de um mecanismo de *zoom* com uma determinada capacidade. Esta capacidade de *zoom* pode variar de uma máquina para outra, por exemplo, 1x até 3x, 1x até 4x ou 1x até 5x. Desta forma, a capacidade de *zoom* é a variação que o caso de uso Tirar Foto apresenta e é capturada através do ponto de variação *zoom*, sendo que a capacidade 1x a 3x, 1x a 4x e 1x a 5x são as variantes que pertencem a este ponto de variação.

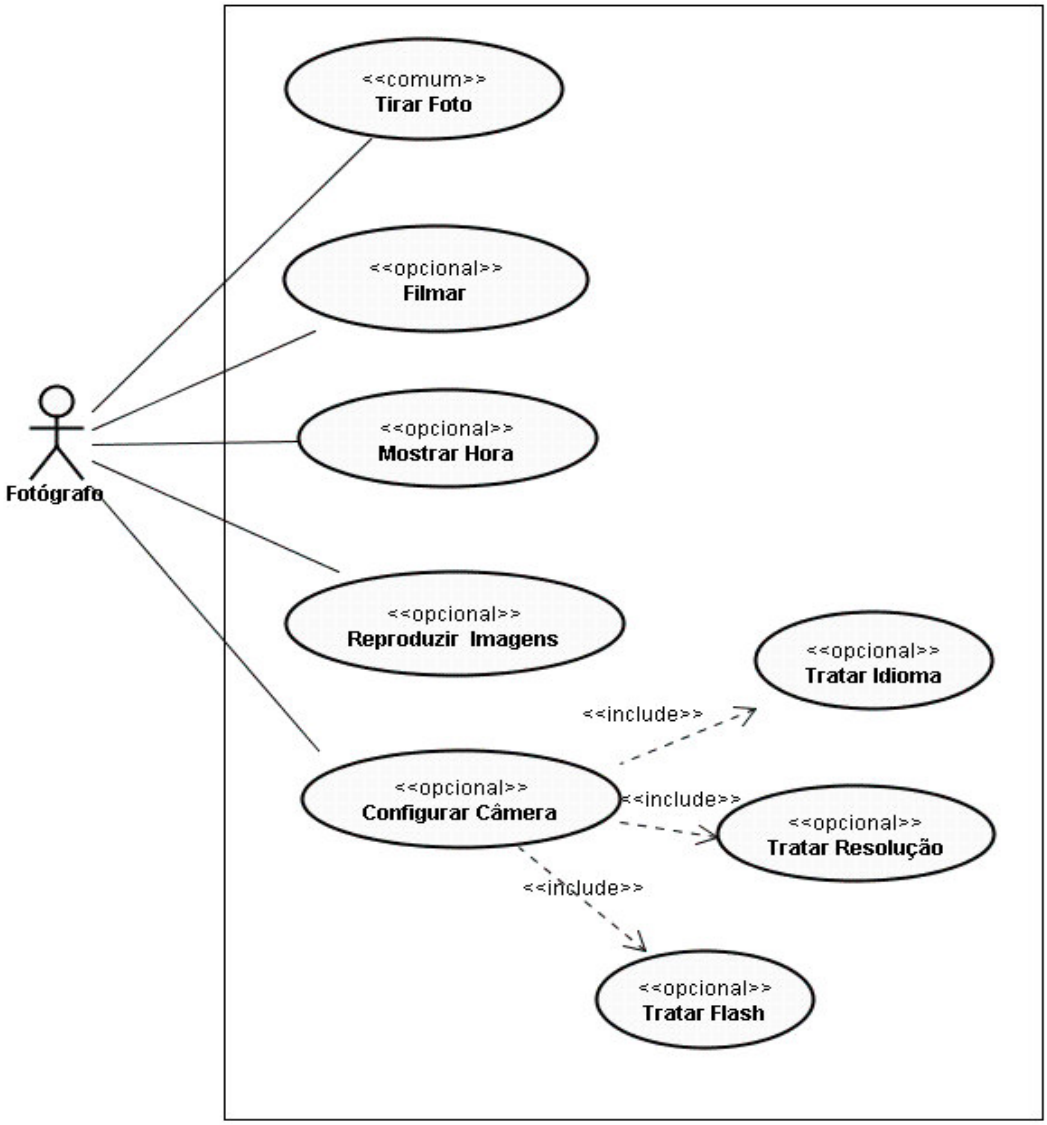


Figura 6.4 - Diagrama de casos de uso de linha de produtos de software para câmera digital

As variações nesta linha de produtos de software são expressas também através dos casos de uso do tipo opcionais: Filmar, Mostrar Hora, Reproduzir Imagens e Configurar Câmera.

O caso de uso opcional Configurar Câmera incorpora os casos de uso: Tratar Idioma, Tratar Resolução e Tratar *Flash* e, por isso, existem os relacionamentos de inclusão (*include*) entre eles.

O caso de uso Tratar Idioma representa características de variação de idioma que a câmera digital pode ter, por exemplo, a câmera pode apresentar suas instruções em

inglês, francês, português, ou espanhol. O caso de uso Tratar *Flash* representa características de variação para o acionamento de *flash* da câmera digital, que pode ser, por exemplo, acionar *Flash* Manual (para uma câmera simples) ou *Flash* Automático (para uma câmera sofisticada). O caso de uso Tratar Resolução representa características de variação do dispositivo de resolução da imagem da câmera digital, que pode ser, por exemplo, de 5, 6 ou 8 mega *pixel*.

A variabilidade é capturada tanto pelos pontos de variação dos casos de uso comuns quanto pelos casos de uso opcionais e alternativos. Assim, casos de uso do tipo comum representam características comuns e variabilidade, enquanto os casos de uso do tipo opcionais e alternativos representam exclusivamente variabilidade em uma linha de produtos de software.

Para que o modelo de casos de uso de linha de produtos de software possa dar apoio à identificação de variabilidade, a descrição de casos de uso deve ser feita do ponto de vista de linha de produtos de software, detalhando pontos de variação e variantes. O padrão de descrição adotado no processo GVLPS está apresentado no Anexo A.

Apresenta-se, a seguir, a descrição do caso de uso Tirar Foto.

1) Nome do Caso de Uso: Tirar Foto.
2) Categoria de Reuso: comum.
3) Sumário: A câmera realiza operações de *zoom* e foco, solicitadas pelo fotógrafo e tira foto.
4) Ator: Fotógrafo.
5) Evento Iniciador: Ativação do modo de fotografia da câmera.
6) Pré-Condição: câmera ligada.
7) Descrição:

   Passo 1: Fotógrafo ativa modo fotografia da câmera.

   Passo 2: A câmera seleciona dispositivo para tirar fotografia.

Passo 3: Fotógrafo seleciona capacidade de *zoom*.

Passo 4: **Ponto de Variação 1** - tratamento de *zoom*.

Passo 5: Fotógrafo focaliza imagem.

Passo 6: A câmera aciona dispositivo de foco.

Passo 7: Fotógrafo aciona o botão para tirar foto.

Passo 8: A câmera aciona o dispositivo para tirar foto.

Passo 9: **Ponto de Variação 2** - tratamento de memória interna.

8) Pontos de Variação:

1. Tratamento de *zoom* (passo 4):

A câmera aciona dispositivo de *zoom* cuja a capacidade pode ser de 1x a 3x, 1x a 4x ou 1x a 5x.

Variantes: 1x a 3x, 1x a 4x ou 1x a 5X.

2. Tratamento de memória interna (passo 9):

A câmera aciona dispositivo de memória interna que pode ser de tamanhos: 1G, 2G ou 3G.

Variantes: 1G, 2G e 3G.

9) Pós-Condição: Fotografia armazenada na memória e câmera desativa dispositivo de foco.

10) Extensão: não tem.

11) Inclusão: não tem.

6.4 Identificação da Variabilidade

A Identificação da Variabilidade é uma atividade que define as variações e os locais onde elas ocorrem na representação de linha de produtos de software (SVAHNBERG, VAN GURP, BOSCH, 2002). O GVLPS optou por utilizar o modelo de *features* como recurso para a identificação da variabilidade, com base em

(NORTHROP, CLEMENTS, 2007), (GOMAA, 2005), (SVAHNBERG, VAN GURP, BOSCH, 2002), (ANTKIEWICZ, CZARNECKI, 2004), (BECKER et al., 2002), (CAPILLA, DUEÑAS, 2001) e (KRUEGER, 2002 apud SVAHNBERG, GURP, BOSCH, 2002). Através de um modelo de *features* é possível organizar o resultado da análise das características comuns e variabilidade para um dado domínio (GRISS, FAVARO e D'ALESSANDRO, 1998).

Esta atividade recebe o modelo de casos de uso de linha de produtos de software, como artefato de entrada e, a partir dele, gera o modelo de *features*. É constituída pelas atividades de Extração de *Features* e de Modelagem de *Features* como mostra a Figura 6.5.

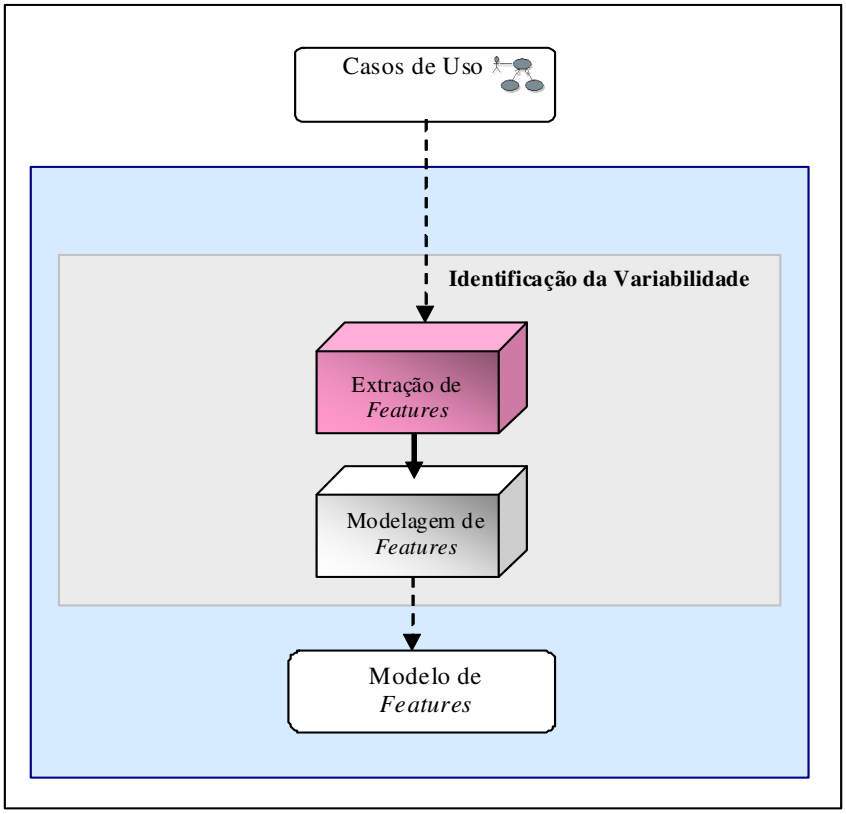


Figura 6.5 - Atividades da Identificação da Variabilidade.

6.4.1 Extração de *Features*

No processo GVLPS, a Extração de *Features* é realizada a partir do modelo de casos de uso de linha de produtos de software, como na proposta de Gomaa (2005) e Griss, Favaro, d´Alessandro (1998). Neste caso, a descrição dos casos de uso explicita os pontos de variação e as variantes, além de detalhar a funcionalidade do sistema, e permitir complementar a atividade de Extração de *Features*.

Na UML, as classes são definidas a partir de substantivos encontrados no vocabulário do sistema que está sendo modelado (BOOCH, RUMBAUGH, JACOBSON, 2006). No processo GVLPS, a Extração de *Features* a partir dos casos de uso é realizada de forma análoga à identificação de classes na UML, encontrando-se os elementos (substantivos) necessários para realizar a funcionalidade dos casos de uso. Estes elementos encontrados são candidatos a se tornarem *features*.

Para classificar as *features*, o GVLPS optou por se basear nas classificações de van Gurp, Bosch, Svahnberg (2001), Griss, Favaro, D´Alessandro (1998) e Gomaa (2005), Anastasopoulos, Gacek, (2001), apresentadas na Seção 4.2 deste trabalho. Os tipos de *features* selecionados, considerados suficientes para descrever a variação em um dado domínio, são:

- Mandatórias;
- Opcionais;
- Pontos de Variação Mandatória;
- Pontos de Variação Opcional;
- Variante Inclusiva;
- Variante Mutuamente Inclusiva;
- Variante Mutuamente Exclusiva.

As *features* mandatórias representam as características essenciais da linha de produtos de software, e são identificadas através de casos de uso comuns. Estes casos de uso, também permitem identificar parte da variabilidade, geralmente através de pontos de variações nas suas descrições. As *features* opcionais, do tipo ponto de variação e variantes, representam também a variabilidade na linha de produtos de software e são identificadas através de casos de uso opcionais e alternativos.

Desta forma, para realizar a Extração de *Features*, foi definido o seguinte procedimento no processo GVLPS:

1) *Passo 1- Identificar features mandatórias:*

    a. Listar casos de uso comuns. Estes casos de uso são os considerados fundamentais, ou seja, aqueles que são sempre requeridos pelo sistema em questão. Para câmera digital, é a funcionalidade que toda câmera deve possuir. Exemplo: o caso de uso comum para a câmera digital é Tirar Foto.

    b. Gerar *features* candidatas a *features* mandatórias, através dos casos de uso comuns encontrados. Para isso, é necessário listar os elementos que certamente devem existir para implementar tal funcionalidade. Estes elementos são também identificados a partir da descrição detalhada dos casos de uso. Exemplo: os elementos que devem existir para implementar a funcionalidade do caso de uso Tirar Foto são: câmera digital, *zoom* e memória interna.

2) *Passo 2- Identificar features opcionais:*

    a. Listar casos de uso opcionais. Estes casos de uso são os considerados não fundamentais para o propósito principal do sistema. Esta funcionalidade, geralmente é representada por uma seqüência de interações isoladas da funcionalidade principal do sistema. Exemplo: os casos de uso que implementam funcionalidade não fundamental ao

propósito principal do sistema (tirar foto) são: Mostrar Hora, Filmar, Reproduzir Imagem, Configurar Câmera.

b. Gerar *features* candidatas a *features* opcionais, através dos casos de uso opcionais. Para isso, é necessário listar os elementos que realizam a funcionalidade não fundamental para o propósito principal do sistema. Estes elementos são também identificados a partir da descrição detalhada dos casos de uso. Exemplo:

- Para o caso de uso Mostrar Hora – relógio;
- Para o caso de uso Filmar – dispositivo de filmagem, áudio.
- Para o caso de uso Reproduzir imagem - dispositivo de filmagem, áudio.
- Para o caso de uso Configurar Câmera: idioma, dispositivo de seleção de resolução e *flash*.

3) *Passo 3 - Identificar features do tipo ponto de variação:*

a. Através do(s) caso(s) de uso comum(ns): Listar os elementos, encontrados no Passo1, que implementam a funcionalidade fundamental e que podem apresentar variações. Estes elementos são candidatos a *features* do tipo ponto de variação mandatório. Exemplo: *zoom* e memória interna.

b. Através do(s) caso(s) de uso opcional(is): Listar os elementos, encontrados no Passo 2, que implementam a funcionalidade opcional e que podem apresentar variações. Estes elementos são candidatos a *features* do tipo ponto de variação opcional. Exemplo: idioma, dispositivo de seleção de resolução e *flash*.

4) *Passo 4 - Identificar features variantes*

Listar as possibilidade de variações para os elementos listados como ponto de variação no passo 3. Exemplo:

- Dispositivo de *Zoom* pode ser de capacidade de: 1x a 3x, 1x a 4x ou 1x a 5x.
- Memória Interna pode ser de tamanho: 1G, 2G ou 3G.
- Dispositivo de Idioma pode ser de: português, inglês, espanhol ou francês.
- Dispositivo de Seleção da Resolução pode ser: 5Mpixel, 6Mpixel ou 8Mpixel.
- Dispositivo para tratar *flash*: pode ser automático ou manual.

5) *Passo 5 - Realizar classificação das features encontradas:*

Classificar as *features* variantes como: inclusiva, mutuamente exclusiva ou mutuamente inclusiva.

Aplicando-se o procedimento de Extração de *Features* a partir do diagrama de casos de uso de linha de produtos de software para câmara digital, apresentado na Figura 6.4, foi elaborada a Tabela 6.2 que relaciona os casos de uso da câmara digital às *features* identificadas.

O relacionamento entre casos de usos e *features* é uma associação muitos para muitos (GOMAA, 2005). Um caso de uso pode gerar muitas *features*, como por exemplo, o caso de uso Tirar Foto que gera várias *features*; por outro lado, uma mesma *feature* pode ser extraída de mais de um caso de uso, como é o caso da *feature* Dispositivo de Filmagem que foi extraída dos casos de uso Filmar e Reproduzir Imagem.

Tabela 6.2 - *Features* geradas de casos de uso

Casos de Uso	**Classificação do Casos de Uso**	***Features***	**Pontos de Variação**	**Classificação da *Features***
Tirar Foto	comum	câmera digital		mandatória
Tirar Foto	comum	1x a 3x	*zoom* (mandatória)	mutuamente exclusiva
Tirar Foto	comum	1x a 4x	*zoom* (mandatória)	mutuamente exclusiva
Tirar Foto	comum	1x a 5x	*zoom* (mandatória)	mutuamente exclusiva
Tirar Foto	comum	1G	memória interna (mandatória)	mutuamente exclusiva
Tirar Foto	comum	2G	memória interna (mandatória)	mutuamente exclusiva
Tirar Foto	comum	3G	memória interna (mandatória)	mutuamente exclusiva
Tratar Idioma	opcional	espanhol	idioma (opcional)	inclusiva
Tratar Idioma	opcional	português	idioma (opcional)	inclusiva
Tratar Idioma	opcional	francês	idioma (opcional)	inclusiva
Tratar Idioma	opcional	inglês	idioma (opcional)	inclusiva
Tratar Resolução	opcional	5Mpixel	dispositivo seleção resolução (opcional)	mutuamente exclusiva
Tratar Resolução	opcional	6Mpixel	dispositivo seleção resolução (opcional)	mutuamente exclusiva
Tratar Resolução	opcional	8Mpixel	dispositivo seleção resolução (opcional)	mutuamente exclusiva
Tratar Flash	opcional	manual	flash (opcional)	mutuamente exclusiva
Tratar Flash	opcional	automático	flash (opcional)	mutuamente exclusiva
Reproduzir Imagens	opcional	dispositivo de filmagem		opcional
Reproduzir Imagens	opcional	áudio		opcional
Mostrar Hora	opcional	relógio		opcional
Filmar	opcional	dispositivo de filmagem		opcional

6.4.2 Modelagem de *Features*

A atividade de Modelagem de *Features* consiste em relacionar as *features* extraídas na atividade anterior para construir o modelo de *features.*

Como exemplo da modelagem, apresenta-se na Figura 6.6 o modelo de *features* da linha de produtos de software para a câmera digital. Este modelo foi construído utilizando a representação de Gomaa (2005) apresentada na Figura 4.4 (Modelo de *Feature*s com Estereótipos) na Seção 4.2 deste trabalho.

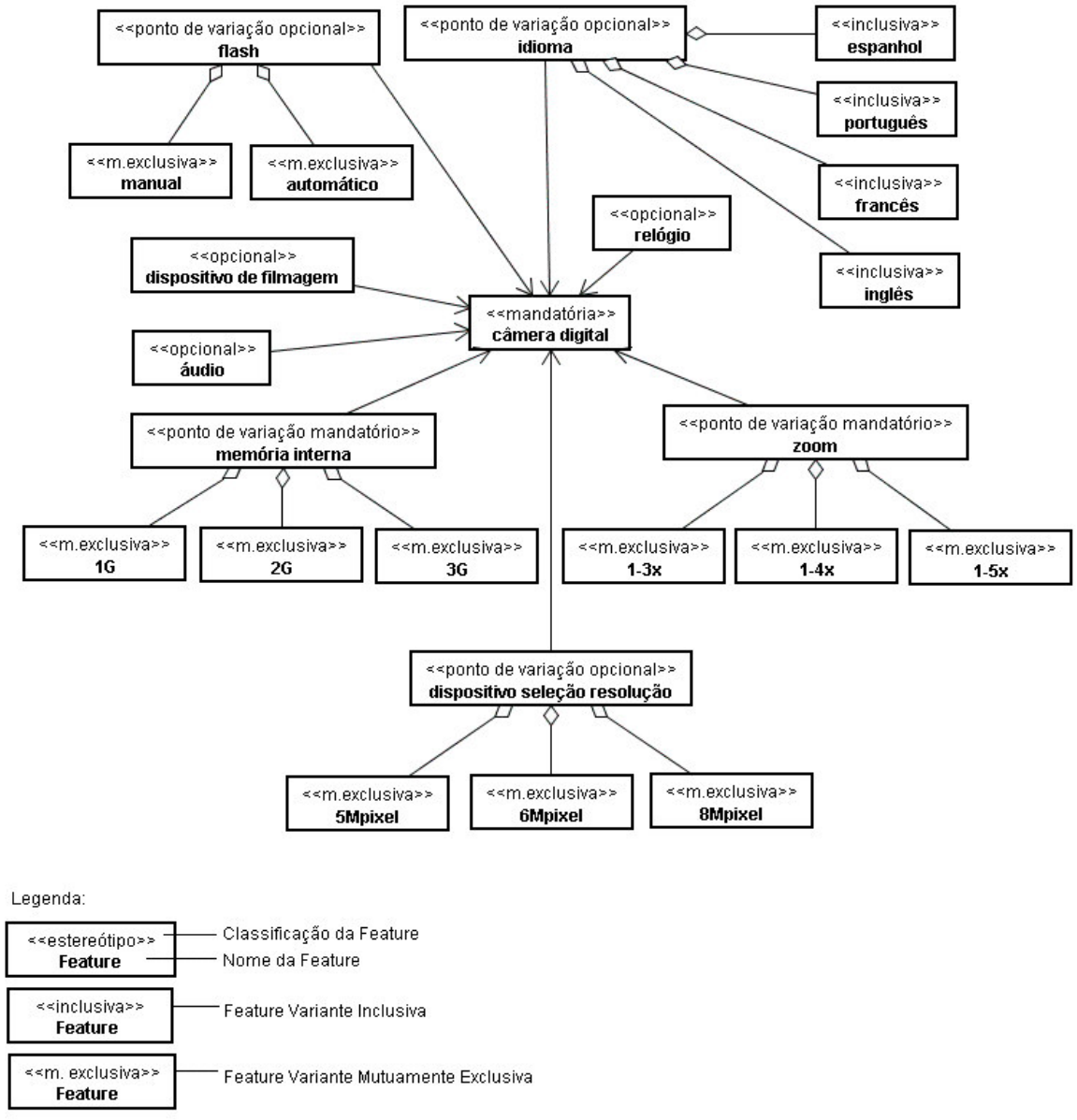


Figura 6.6 - Modelo de *features* para a linha de produtos de software da câmera digital

A *feature* mandatória câmera digital está conectada com as *features* que representam sua variabilidade:

- *features* opcionais que podem existir ou não, dependendo do tipo da câmara: relógio, áudio e dispositivo de filmagem;
- *features* do tipo ponto de variação opcional que representa a variação de uma característica opcional: idioma, dispositivo de seleção de resolução e *flash;*
- *features* ponto de variação mandatória que representa a variação de uma característica obrigatória: *zoom*, memória interna.

6.5 Especificação da Variabilidade

A atividade de Especificação da Variabilidade consiste em ajustar as variantes às necessidades atuais e futuras do sistema e assim proporcionar maior flexibilidade à linha de produtos de software. Para alcançar este objetivo, devem ser realizadas três atividades: Determinação da Cardinalidade, Representação da Variabilidade por Classes e Escolha do *Binding Time*, como mostra a Figura 6.7.

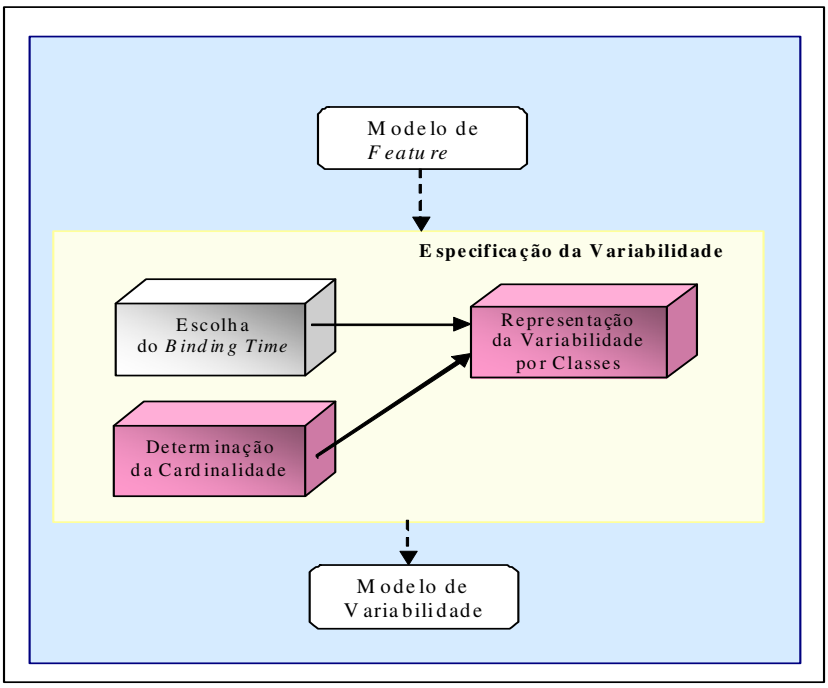


Figura 6.7 - Atividades da Especificação da Variabilidade

6.5.1 Determinação da Cardinalidade

A Determinação da Cardinalidade consiste em definir o número de variantes que podem ser escolhidas para um determinado ponto de variação (ANTKIEWICZ, CZARNECKI, 2004). A cardinalidade do relacionamento entre estas *features* pode ser:

- [1] - pode ser escolhida uma variante;
- [*] - podem ser escolhidas muitas variantes;
- [0..1] - corresponde a uma variante opcional, ou seja, pode existir ou não;
- [0..*] - a variante é opcional e, se existir, podem ser escolhidas muitas variantes;
- [1..*] - podem ser escolhidas 1 ou muitas variantes;
- [n..m] - podem ser escolhidas no mínimo n variantes e no máximo m variantes.

A cardinalidade é apresentada no diagrama de classes que representa a variabilidade, indicando assim, as possibilidades quantitativas das variações. A Figura 6.8 ilustra um exemplo de utilização da cardinalidade em um diagrama de classes.

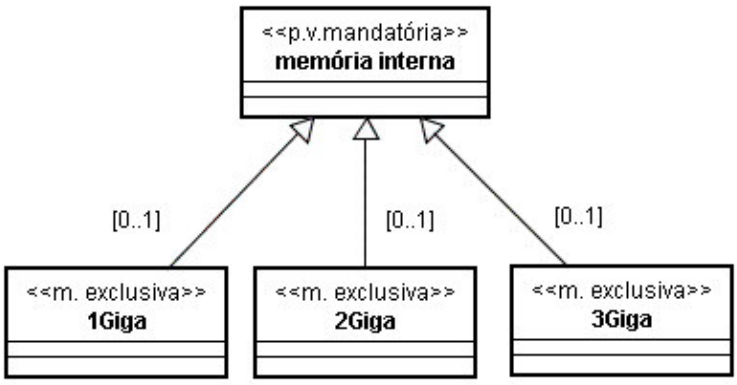


Figura 6.8 - Exemplo de cardinalidade das variantes

A cardinalidade, na Figura 6.8, indica que podem ser escolhidas zero ou uma variante de cada tipo de memória para o ponto de variação memória interna. Entretanto, deve-se também obedecer ao tipo de classificação das classes, indicadas por seus estereótipos. No caso, apenas uma variante de um tipo de memória pode ser escolhida.

6.5.2 Representação da Variabilidade por Classes

A atividade de Representação da Variabilidade por Classes consiste em utilizar classes para representar as *features,* convertendo o modelo de *features* em uma representação mais adequada para a Engenharia de Aplicação.

As variantes são representadas no modelo de *features*, entretanto, esta representação é ainda distante de modelagem convencional de sistemas de software. Por isso, o modelo de classes é adotado por diversos autores para representar a variabilidade, pois possui uma notação com recursos como herança, multiplicidade e os atributos que expressam detalhes das características comuns e da variabilidade (GOMAA, 2005), (POHL, BOCKLE, LINDEN, 2005).

Desta forma, o diagrama de classes de uma linha de produtos de software deve conter as classes necessárias para representar as *features* que realizam a funcionalidade determinada pelos casos de uso. Além disso, o diagrama de classes de linha de produtos de software deve mostrar classes correspondentes aos tipos de *features* definidos na Seção 6.4.1. Os tipos de *feature* são representados no diagrama através de estereótipos.

A Tabela 6.3 mostra as classes necessárias para representar as *features* do exemplo da câmera digital.

Tabela 6.3 - Dependência entre *features* e classes

Nome da *Feature*	**Nome da Classe**
Câmera digital	câmera digital dispositivo de imagem memória interna dispositivo de saída dispositivo de configuração
flash	dispositivo de *flash*
idioma	dispositivo de configuração
relógio	relógio
dispositivo de filmagem	dispositivo de filmagem
áudio	dispositivo de configuração dispositivo de filmagem
zoom	dispositivo de imagem
memória interna	memória interna
dispositivo seleção de resolução	dispositivo de configuração

A Figura 6.9 apresenta um diagrama de classes de linha de produtos de software para câmera digital.

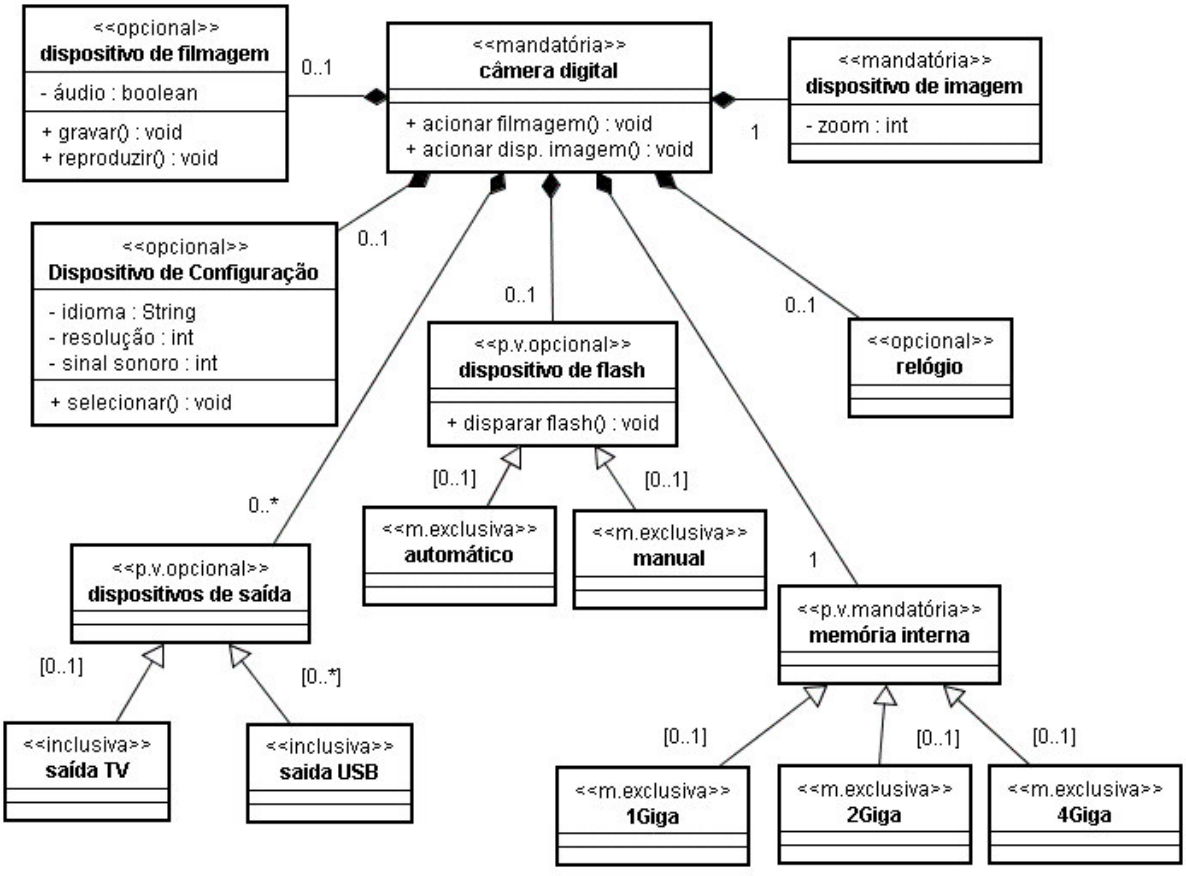


Figura 6.9 - Diagrama de classes de linha de produtos de software para câmera digital

O diagrama de classes de linha de produtos de software da câmera digital mostrado na Figura 6.9, foi elaborado a partir do modelo de *features* da Figura 6.6.

Este diagrama apresenta classes que representam elementos da câmera, ou seja, os elementos que representam o domínio do problema. Observa-se que uma câmera digital deve possuir um dispositivo de imagem e memória interna, mas pode possuir ou não um dispositivo de filmagem, relógio, dispositivo de configuração e dispositivo de *flash*. Além disso, a câmera pode ter zero ou muitos dispositivos de saída. Observa-se também que certas *features* podem ser transformadas em atributos porque são elementos que correspondem a uma propriedade de uma classe, como é o caso, por exemplo, da *feature* idioma, transformada em propriedade da classe dispositivo de configuração.

O diagrama também apresenta as cardinalidades das classes que representam as *features* variantes, indicando o número de variantes que podem ser escolhidas para cada ponto de variação.

6.5.3 Escolha do *Binding Time*

A atividade de Escolha do *Binding Time*, também chamada de determinação do tempo de resolução, define em que momento as variantes associadas a um ponto de variação devem ser incorporadas ao sistema para fazer parte do produto a ser produzido (SVAHNBERG, VAN GURP, BOSCH, 2002), (BRAGANÇA, MACHADO, 2004a), (FRITSCH, LEHN, STROHM, 2002).

As decisões de *binding time* podem ser tomadas em vários momentos do desenvolvimento do sistema ou mesmo durante a execução do sistema.

O *binding time* pode ocorrer nos seguintes momentos:

- Derivação da arquitetura;
- Compilação;
- Ligação (*linking*);

- Tempo de Execução.

A Tabela 6.4 apresenta o impacto da escolha do *binding time* em relação às características de flexibilidade, desempenho e linhas de código de um sistema.

Tabela 6.4 - Impacto do *Binding Time*

	Flexibilidade	Desempenho	Linhas de Código
Implementação	-	+	+
Compilação	+	+	+
Ligação	+	+	+
Tempo de Execução	++	- -	-

Fonte Adaptada (VOELTER, 2006)

Os símbolos “+” e “-” significam, respectivamente, a melhora ou não de características como flexibilidade, desempenho e aumento ou diminuição do número de linhas de código.

Quando o *binding time* ocorre no momento da implementação, ou seja, as variantes são incorporadas ao sistema durante a implementação, a flexibilidade diminui, pois a seleção de variantes ocorre apenas com alteração do código. Neste caso, o desempenho melhora, porque a variante selecionada é implementada diretamente no código, evitando que o sistema realize ou possibilite ao usuário realizar qualquer tipo de escolha posterior.

O *binding time* pode ocorrer durante a compilação ou ligação, através de diretivas do pré-processador ou do editor de ligações, por exemplo, ligando objetos ou bibliotecas diversas. Neste caso, a flexibilidade e o desempenho melhoram porque podem ser selecionadas partes do código e bibliotecas para serem incluídas durante a compilação do programa.

Quando o *binding time* ocorre em tempo de execução, a flexibilidade do sistema aumenta e o desempenho é prejudicado devido às características das tecnologias utilizadas para incluir a variabilidade em tempo de execução.

Quanto ao número de linhas de código, este diminui significantemente no caso do *binding time* ocorrer em tempo de execução. Isto ocorre devido ao fato das tecnologias utilizadas para realizar a variabilidade em tempo de execução permitirem reduzir o número de classes a serem implementadas e, conseqüentemente, o número de linhas de código.

Do ponto de vista do gerenciamento de variabilidade, uma das necessidades existentes é o aumento da flexibilidade na seleção e implementação das variações, ou seja, deseja-se realizar mais facilmente alterações nos sistemas e viabilizar, assim, a reutilização de artefatos de software.

A busca pela flexibilidade levou o processo GVLPS adotar o mecanismo de variabilidade baseado em modelos de objetos adaptáveis e em reflexão, que fornece meios para seleção das variantes e habilita a criação de objetos e acionamentos de métodos dinamicamente em tempo de execução.

Por isso, o *binding time* do processo GVLPS deve ocorrer em tempo de execução, devido ao mecanismo de variabilidade utilizado.

6.6 Implementação da Variabilidade

A Implementação da Variabilidade consiste em selecionar e aplicar algum mecanismo de variabilidade para a linha de produtos de software.

No Capítulo 4 deste trabalho foram apresentados vários mecanismos de variabilidade, que possuem características diferentes, e podem ser selecionados em função do tipo e das necessidades do sistema. Cada mecanismo tem vantagens e desvantagens e pode ser mais ou menos apropriado dependendo da natureza da linha de produtos de software considerada (GOMAA, WEBBER, 2004). Estes autores citam, por exemplo, que a técnica de parametrização limita a variabilidade para a alteração de uma funcionalidade, pois o código é condicionado a utilização de parâmetros pré-definidos.

Ainda segundo estes autores, a técnica da herança permite, ao desenvolvedor do *core assets,* fornecer um número limitado de variantes baseado na classe pai e o usuário deve selecionar uma variante a partir de um conjunto de escolhas pré-definidas na fase de projeto do sistema.

Para este trabalho, selecionou-se o mecanismo de variabilidade baseado nos padrões de modelos de objetos adaptáveis e em reflexão porque, além de oferecer facilidade para adaptar o sistema aos novos requisitos que possam surgir, as técnicas de modelos adaptáveis e reflexão são consideradas apropriadas principalmente em casos em que o sistema necessita ser continuamente alterado ou estendido (YODER, 2007).

Através dos padrões de modelos de objetos adaptáveis é possível criar um número maior de variantes do que, por exemplo, a técnica de herança, porque com o auxílio de um repositório existente no mecanismo de variabilidade, as variantes podem ser facilmente configuradas, sem a necessidade de alterações do projeto do sistema. Este mecanismo trabalha com um conjunto de padrões de projeto que permite facilmente incluir novos tipos de objetos e alterar objetos existentes através de sua representação por metadados e por isso possibilita o sistema se adaptar a novos requisitos.

Para aumentar a flexibilidade do mecanismo proposto, técnicas de computação reflexiva também são empregadas, através de mecanismos fornecidos pela plataforma de desenvolvimento selecionada. O intuito de se utilizar estes mecanismos é fazer com que todas as alterações permitidas pelo emprego dos padrões de modelos de objetos adaptáveis sejam realizadas dinamicamente em tempo de execução.

O fato da criação e seleção das variantes ocorrerem em tempo de execução, sem necessidade de nova programação e conseqüentemente sem precisar de nova compilação, reduz o tempo da inclusão dos novos requisitos, além de facilitar a operação.

Assim, o mecanismo de variabilidade baseado em modelos de objetos adaptáveis e em reflexão permite habilitar a criação e seleção de novas variantes para um

determinado produto de uma forma mais flexível que a apresentada por outras técnicas.

Esta idéia vai ao encontro com as necessidades da linha de produtos de software, onde o sistema deve estar sempre habilitado para permitir a criação de novos produtos com a introdução de diferentes variações.

Com este mecanismo de variabilidade, o processo GVLPS pretende auxiliar aos projetos que necessitam constantemente de alterações e não desejam alterar seu código.

A Figura 6.10 mostra a atividade do processo GVLPS que faz parte da Implementação da Variabilidade juntamente com as técnicas utilizadas. Tem-se, como resultado desta atividade, o sistema implementado com as variantes selecionadas através dos modelos de objetos adaptáveis e reflexão.

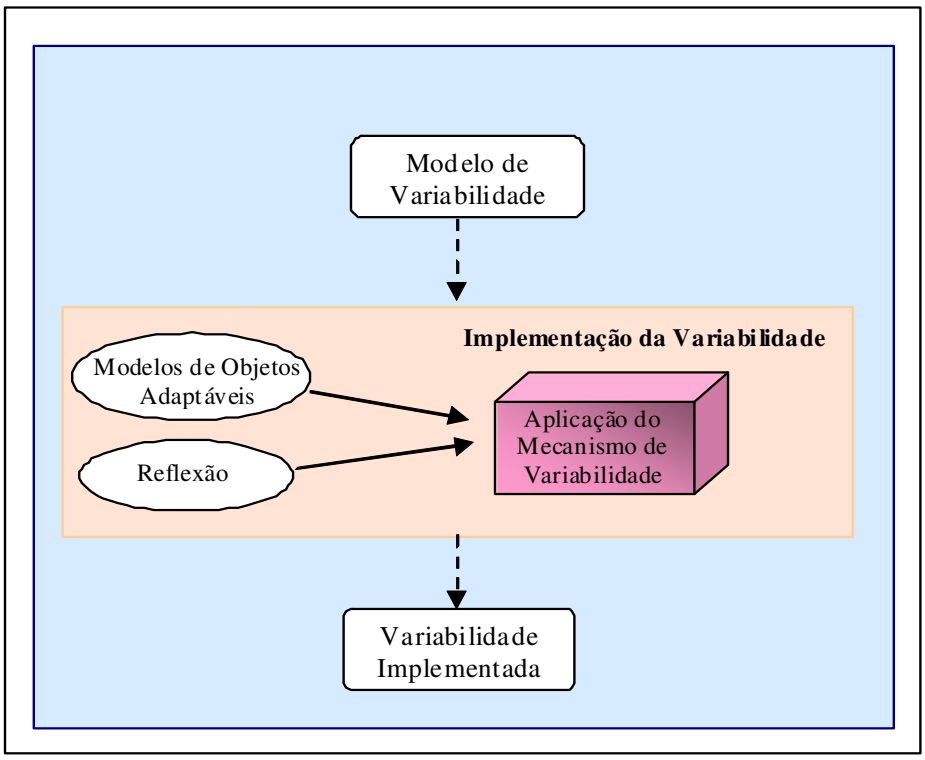


Figura 6.10 - Atividade da Implementação da Variabilidade.

A variabilidade é incorporada ao sistema em tempo de execução e, desta forma, os objetos são instanciados dinamicamente, os tipos são carregados dinamicamente, e os métodos, executados dinamicamente.

O propósito de se utilizar os padrões de modelos de objetos adaptáveis é obter um modelo tão genérico quanto possível, já que este conjunto de padrões permite a inclusão de novos tipos de objetos, alterações de suas propriedades e acionamento de seus métodos. Para tanto, como mostrado na Seção 5.3, os seguintes padrões de projeto de modelos de objetos adaptáveis são utilizados: *TypeObject*, *Property*, *TypeSquare* e *Strategy*.

A Figura 6.11 ilustra a arquitetura do mecanismo de variabilidade do processo GVLPS.

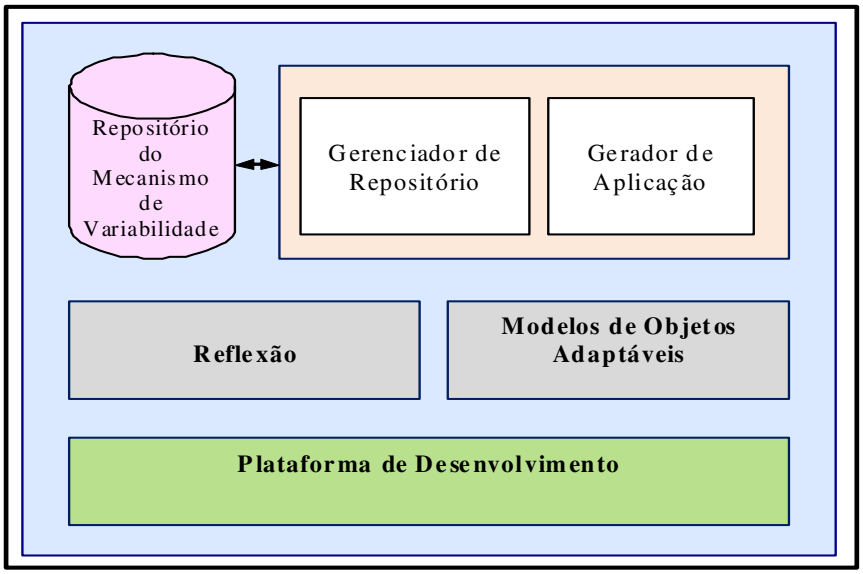


Figura 6.11 - Arquitetura do mecanismo de variabilidade do processo GVLPS

O sistema de Gerenciador de Repositório é responsável por possibilitar ao usuário configurar as informações relacionadas aos pontos de variação e variantes e armazená-las no repositório do mecanismo de variabilidade. Este repositório é um artefato que pertence ao repositório da linha de produtos de software.

Através do sistema Gerenciador de Repositório é possível cadastrar as variantes representadas por objetos correspondentes e definir seus tipos, além de cadastrar

as possibilidades de propriedades e métodos das classes. O repositório do mecanismo de variabilidade é responsável por armazenar as informações das variantes e disponibilizá-las para o sistema Gerador de Aplicação.

O sistema Gerador de Aplicação é responsável por possibilitar ao usuário selecionar e criar as novas variantes que farão parte da aplicação, assim como definir suas propriedades e acionar suas funcionalidades.

Assim, o usuário seleciona do repositório do mecanismo de variabilidade, as variantes e os objetos correspondentes são criados dinamicamente; tipos apropriados, propriedades e métodos, que foram cadastrados anteriormente na configuração são carregados. A partir daí seus métodos já estão configurados e aptos a serem acionados dinamicamente.

Também estão representadas na Figura 6.11, as técnicas de modelos de objetos adaptáveis e reflexão, utilizadas para o desenvolvimento do mecanismo de variabilidade, e a Plataforma de Desenvolvimento que se refere à plataforma Java utilizada.

Assim, o mecanismo de variabilidade baseado em modelos de objetos adaptáveis e em reflexão permite manipular a funcionalidade do sistema de acordo com a configuração desejada. Esta característica e, a flexibilidade na criação e seleção de variantes fazem com que o software seja mais facilmente adaptável a um novo sistema, melhorando assim sua característica de reuso.

6.7 Considerações Finais

Este capítulo definiu o GVLPS, um processo de gerenciamento de variabilidade de linha de produtos de software constituído por três atividades: Identificação da Variabilidade, Especificação da Variabilidade e Implementação da Variabilidade.

Considera-se o emprego de um processo de gerenciamento de variabilidade importante para a linha de produtos de software, pois aumenta a possibilidade de mais produtos serem derivados desta linha, desde que o processo de gerenciamento

apresente artefatos e atividades bem delineados para a produção dos artefatos a serem utilizados.

Apesar de ser baseado no processo de van Gurp, Bosch, Svahnberg (2001), o processo GVLPS incorpora novas atividades como a Extração de *Features*, Determinação de Cardinalidade, Representação de Variabilidade por Classes e Aplicação do Mecanismo de Variabilidade. Além disso, o processo GVLPS detalha como as atividades são realizadas, e também os artefatos de entradas e saídas de cada uma delas.

Na geração dos artefatos de entrada do processo GVLPS, adotou-se a representação de casos de uso de linha de produtos de software de Gomaa (2005), contudo, foi necessário criar um novo padrão de descrição de casos de uso para detalhar os pontos de variação e variantes.

Também foi necessário criar, para a atividade de Extração de *Features,* um procedimento para identificar e classificar as *features* a partir do modelo de casos de uso. Na bibliografia pesquisada, foi encontrado um procedimento mais complexo, e que não atendia as necessidades do processo GVLPS (GRISS, FAVARO, D´ALESSANDRO, 1998).

Para a atividade de Implementação da Variabilidade, foi selecionado um mecanismo de variabilidade baseado em modelos de objetos adaptáveis e em reflexão. A reflexão é tida como uma técnica atraente para implementar a variabilidade em linha de produtos de software (ANASTASOPOULOS, GACEK, 2001), pelo seu potencial de criação de objetos e acionamento de métodos dinamicamente em tempo de execução. Esta particularidade, em conjunto com as vantagens oferecidas pelos modelos de objetos adaptáveis, facilita o reuso e promove maior flexibilidade à variação de produto, características que são desejadas na abordagem de linha de produtos de software.

O mecanismo proposto não tem a pretensão de solucionar todos os problemas apresentados por outras técnicas, mas sim, oferecer uma opção de mecanismo de variabilidade, proporcionando mais flexibilidade ao sistema, a partir de um modelo que permita mais facilmente realizar as alterações no mesmo.

"Só há uma coisa que faz
com que um sonho seja impossível:
o medo do fracasso."

Paulo Coelho

7 APLICAÇÃO DO PROCESSO GVLPS AO SOFTWARE DO LSB

Este capítulo tem como objetivo apresentar a aplicação do processo GVLPS ao software dos Lançadores de Satélite Brasileiros (LSB). O envolvimento da autora no projeto de veículos lançadores de satélites do IAE permitiu identificar que esta área tem um potencial para utilizar uma linha de produtos de software. Devido à natureza deste programa, cujos projetos são complexos e de longa duração, percebeu-se que um processo de desenvolvimento de software que permita aumentar a flexibilidade para alteração e o reuso do software, evitaria despender esforços repetitivos para desenvolver software para veículos da mesma família.

Como as informações do VLS são sigilosas, criou-se o LSB, um veículo fictício, para ser explorado na avaliação do processo GVLPS. Suas características foram baseadas no VLS, cuja descrição está apresentada no Capítulo 2 deste trabalho.

Inicialmente é apresentada uma visão geral da estratégia para a aplicação do processo GVLPS ao software do LSB. Em seguida, são apresentados a descrição textual e o modelo de casos de uso de linha de produtos de software, que são os artefatos de entrada para o processo GVLPS. Logo após, é descrita e comentada a aplicação das atividades do processo ao software do LSB, com os respectivos artefatos de saída que consistem de diagramas, modelos e exemplos necessários para o entendimento do GVLPS. As atividades do GVLPS são Identificação, Especificação e Implementação da Variabilidade.

Para atividade de Implementação da Variabilidade, são detalhadas as estratégias para a construção do mecanismo de variabilidade, que consistem em aplicar os modelos de objetos adaptáveis no modelo do software do LSB e projetar as classes que apresentam variação para configurar o repositório do mecanismo de variabilidade com metadados relacionados à estas classes. É apresentado também,

o funcionamento do sistema Gerador de Aplicação, responsável pela criação das variantes.

7.1 Estratégia para a Aplicação do Processo GVLPS ao Software do LSB

A estratégia para a aplicação do processo GVLPS ao software do LSB é apresentada para um melhor entendimento da organização das atividades entres os participantes do processo. Para esta aplicação foram levantados dois perfis: (1) gerente de linha de produtos – que possui uma visão geral das diversas aplicações que podem ser geradas, assim como, dos artefatos que consituem o *core assets* e das possíveis variações a serem criadas e (2) projetista do LSB – que possui uma visão mais específica da aplicação, deve saber interpretar determinada configuração e especificar detalhes de variantes. A estratégia para a aplicação do processo GVLPS ao software do LSB pode ser resumida da seguinte forma:

- Atividades do gerente de linha de produtos de software do LSB:
    - 1) Através das atividades de Identificação e Especificação da Variabilidade do processo GVLPS, obtém o diagrama de classes de linha de produtos de software que representa a variabilidade do software do LSB;
    - 2) Na atividade de Implementação da Variabilidade, aplica-se os padrões de modelos de objetos adaptáveis no diagrama de classes obtido no item 1 e obtém o diagrama de classes genérico do LSB;
    - 3) A partir do diagrama de classes genérico do LSB, configura o repositório do mecanismo de variabilidade que é alimentado com os dados, relacionados às variantes do LSB, os quais permitem a implementação das variantes;
    - 4) Cadastra os dados que permitem a implementação das variantes;

- 5) Faz a manutenção do repositório do mecanismo de variabilidade, inserindo, alterando e excluindo as variantes.

- Atividades do projetista do LSB:
  - 1) Obtém a configuração do LSB a ser gerado;
  - 2) Verifica se as variantes do LSB estão cadastradas no repositório do mecanismo de variabilidade; caso forem necessárias novas variantes, aciona o gerente de linha de produtos de software do LSB para cadastrar estas variantes;
  - 3) Seleciona as variantes para o sistema LSB no repositório do mecanismo de variabilidade;
  - 4) Implementa a variabilidade do LSB, através da criação de variantes;
  - 5) Avalia o software gerado do LSB, através do acionamento dos métodos das variantes criadas.

7.2 Artefatos de Entrada

São apresentados, como artefatos de entrada para o processo GVLPS, a descrição textual do LSB e o modelo de casos de uso de linha de produtos de software, conforme apresentado na Seção 6.3.

7.2.1 Descrição Textual dos LSB

Foi elaborada a descrição textual do LSB, com base no VLS, mas descaracterizando as informações que possam ter caráter confidencial. As principais características do LSB estão apresentadas a seguir:

- O LSB é um veículo lançador convencional, constituído por quatro estágios, com o objetivo de colocar em órbitas de baixa altitude, entre 200 e 1000Km, Satélites de Coleta de Dados assim como de Sensoriamento Remoto, com massas entre 100 e 200Kg, lançados a partir do Centro de Lançamento.

- O sistema de controle do LSB é constituído basicamente por sensores, controlador (constituído de computador de bordo e software) e atuadores.

- O conjunto de sensores é constituído por: sensores de pressão, de posição e de velocidade.

- As atividades a serem realizadas para controle de vôo são: pilotagem, guiagem, apontamento e basculamento.

- O conjunto de atuadores é constituído por: atuadores de tubeiras móveis, sistema de rolamento, sistema de gás frio.

- O LSB segue uma trajetória pré-estabelecida e calcula correções da trajetória no decorrer do vôo, em função dos dados obtidos através dos sensores e de cálculos do controle.

- Para seguir a trajetória corretamente (incluindo as correções de trajetória), o LSB aciona seus atuadores.

- O LSB tem as fases de preparação do veículo e do vôo.

- Na fase de preparação, os equipamentos e o software são configurados e ativados.

- A fase de vôo do veículo é constituída pelas seguintes etapas: primeiro estágio, primeiro-segundo estágio (etapa em que o segundo estágio foi ignitado, porém ainda não houve a separação do primeiro estágio), segundo estágio, vôo não controlado 1 (sem atuação da tubeira do terceiro estágio), terceiro estágio, vôo não controlado 2 (sem atuação do sistema de rolamento), basculamento e fim do controle.

- O veículo segue uma seqüência pré-estabelecida de eventos, que correspondem ao acionamento de tarefas adequadas nos instantes especificados.

- Estes eventos são: ativação de tubeiras, ativação de rolamento, separação do estágio 1, ignição e separação do estágio 2, ignição e separação do estágio 3, separação da coifa de proteção do satélite, ativação do sistema de gás frio, separação da baia de controle, início e fim do algoritmo de apontamento, ativação do *spin-up* e separação da baia de equipamentos (onde se encontra o computador de bordo), finalizando assim o controle do vôo.

- O veículo envia dados para um sistema de telemedidas.

- Dependendo da missão, algumas características podem ser alteradas, no caso, foram consideradas as seguintes variações:

    a. Número e tipos de sensores;

    b. Número e tipos de atuadores;

    c. Sequência de eventos a ser seguida;

    d. Trajetória a ser seguida;

    e. Tipos de sistemas de simulação: utilizando sensores simulados, reais ou ambos;

    f. Tipos de dados de telemedidas e a forma de transmiti-los;

    g. Segurança - envio redundante de sinais para ativação dos atuadores.

7.2.2 Modelo de Casos de Uso de Linha de Produtos de Software

A partir da descrição textual, foram extraídos os requisitos funcionais do software do LSB, os quais foram utilizados para a definição e modelagem dos casos de uso de linha de produtos de software para o LSB.

Durante o vôo, os LSB executam uma seqüência de eventos pré-estabelecidos; os acionamentos destes eventos são controlados pelo relógio do computador, seguindo a programação pré-estabelecida. Desta forma, considerou-se o ator tempo, no modelo de casos de uso de linha de produtos de software, o qual aciona os casos de uso que correspondem ao tratamento dos eventos da seqüência.

Os casos de uso de linha de produtos de software do LSB foram definidos, considerando-se as características apresentadas na descrição textual do problema apresentada na Seção 7.2.1 e forma preparados de acordo com os padrões de diagrama e descrição de casos de uso apresentados na Seção 6.3. Os casos de uso identificados e suas descrições sucintas estão apresentadas a seguir.

Controlar Vôo: O software do LSB controla o vôo através de acionamento de atuadores. Estes acionamentos são resultados dos cálculos que o software realiza a partir de dados obtidos de leituras dos sensores. Assim, o LSB segue sua trajetória, corrigindo quando necessário, e dispara seus eventos nos instantes pré-estabelecidos.

Enviar Telemedidas: O software do LSB envia os dados dos sensores de pressão, posição e velocidade para estação de solo. Estes dados são formatados em blocos de dados.

Simular Vôo: O software do LSB possui recursos para realizar simulações de vôo.

Ler Sensores: O software do LSB lê, durante o vôo, os sensores de pressão, posição e velocidade.

Acionar Pré-Testes: O software do LSB possui um modo de testes que o habilita realizar testes de equipamentos do veículo na fase anterior ao vôo.

Tratar Trajetória: O software do LSB calcula correções da trajetória no decorrer do vôo, através das leituras dos sensores, ou segue trajetória nominal.

Acionar Atuadores: Após realizar os cálculos de controle, o software do LSB aciona os atuadores para seguir e/ou corrigir a trajetória. Os atuadores podem ser tubeiras móveis ou atuadores de gás frio.

Acionar Sistema de Segurança: O software do LSB possui um sistema de segurança responsável por enviar um sinal redundante aos sistemas de gás frio, controle de rolamento e circuito de segurança de atuação (pirotécnicos). Assim, o sistema de segurança garante que os atuadores serão acionados no tempo pré-estabelecido.

Estes casos de uso foram classificados de acordo com os tipos comum, alternativo e opcional. A Figura 7.1 apresenta o diagrama de casos de uso de linha de produtos de software com a informação dos tipos dos casos de uso.

Figura 7.1 - Modelo de caso de uso de linha de produtos de software do veículo LSB

Os casos de uso Controlar Vôo, Enviar Telemedidas, Ler Sensores, Acionar Atuadores e Tratar Trajetória foram considerados do tipo comum, pois constituem a funcionalidade fundamental que todo software de um veículo lançador de satélites deve possuir.

Observa-se que os casos de uso comuns Ler Sensores, Tratar Trajetória e Acionar Atuadores estão incorporados no caso de uso Controlar Vôo, ou seja, fazem parte da sequência do caso de uso Controlar Vôo.

Os casos de uso Simular Vôo, Acionar Pré-teste e Ativar Sistema de Segurança Redundante foram considerados do tipo opcional, que podem existir ou não em um membro da família de LSB.

Como citado na Seção 6.3, a descrição dos casos de uso deve contemplar os dados que foram incluídos na definição das características da linha de produtos de software, detalhando os pontos de variação e as variantes. A descrição do caso de uso é feita usando o padrão apresentado no anexo A.

Apresenta-se, a seguir, a descrição detalhada do caso de uso Controlar Vôo.

1) Nome do Caso de Uso: Controlar Vôo.

2) Categoria de Reuso: comum.

3) Sumário: O software do LSB segue uma trajetória com base na trajetória nominal; para isso, dispara os eventos programados de acordo com as leituras dos sensores e cálculos do controle e aciona seus atuadores.

4) Ator: Tempo.

5) Evento Iniciador: identificação do instante para início da leitura dos dados dos sensores.

6) Pré-Condição: veículo em vôo.

7) Descrição:

Passo 1: O sistema lê dados dos sensores;

Passo 2: O sistema determina os tempos das ocorrências dos próximos eventos, através do programação de eventos. **Ponto de Variação 1:** tratamento da sequência de eventos;

Passo 3: O sistema ativa o sistema de atuadores, em função dos instantes de ocorrência de eventos;

Passo 4: O sistema calcula a posição e a velocidade do LSB, com base nos dados obtidos dos sensores;

Passo 5: O sistema, calcula a correção da trajetória do LSB em relação à trajetória nominal.

Passo 6: O sistema envia dados para o sistema de acionamento de atuadores.

8) Ponto de Variação:

Tratamento da sequência de eventos (Passo2): O sistema dispara sua sequência de eventos pré-estabelecida que, dependendo do veículo, pode ser A ou B. Variantes: seqüência de eventos A ou sequência de eventos B - Alternativa exclusiva.

9) Pós-Condição: vôo do veículo controlado, trajetória seguida e sequência de eventos disparadas.

10) Extensão: Não tem.

11) Inclusão: Usa: Ler Sensores, Acionar Atuadores, Tratar Trajetória.

7.3 Identificação da Variabilidade

As atividades relacionadas à atividade de Identificação da Variabilidade são a Extração de *Features* e a Modelagem de *Features*. As próximas seções ilustram a aplicação destas atividades.

7.3.1 Extração de *Features*

Nesta seção é apresentada a identificação de *features* a partir dos casos de uso de linha de produtos de software do LSB, através dos seguintes passos: identificar *features* mandatórias, identificar *features* opcionais, identificar *features* do tipo ponto de variação, identificar *features* variantes e suas classificações (Seção 6.4.1).
São apresentados, a seguir, a aplicação dos passos da Extração de *Features* aos casos de uso de linha de produtos de software do LSB.

1) *Passo 1- Identificar features mandatórias:*

    a. Listar casos de uso comuns - os casos de uso comum do software do LSB são: Controlar Vôo, Ler Sensores, Acionar Atuadores, Tratar Trajetória e Enviar Telemedidas.

    b. Gerar *features* candidatas a *features* mandatórias - A Tabela 7.1 mostra os elementos que devem existir para implementar a funcionalidade de cada caso de uso comum listado. Estes elementos são também identificados a partir da descrição detalhada dos casos de uso e são candidatos a *features* mandatórias.

Tabela 7.1 - Casos de uso comuns e seus elementos

Casos de Uso Comuns	**Elementos**
Controlar Vôo	Controle
Controlar Vôo	Sequência de Eventos
Ler Sensor	Sensor
Acionar Atuadores	Atuador
Enviar Telemedidas	Telemedida
Tratar Trajetória	Trajetória

Na descrição textual do LSB, também são encontrados os outros elementos relacionados ao controle de vôo do veículo como: pilotagem, guiagem, apontamento, basculamento, e etc. Estes elementos foram representados pelo elemento Controle no escopo deste trabalho, por se referirem aos algoritmos de controle de vôo.

2) *Passo 2- Identificar features opcionais:*

   a. Listar casos de uso opcionais - Os casos de uso opcionais do software do LSB são: Simular Vôo, Acionar Pré-Teste e Ativar Sistema de Segurança Redundante.

   b. Gerar *features* candidatas a *features* opcionais - A Tabela 7.2 mostra os elementos necessários para implementar a funcionalidade dos casos de uso opcionais. Estes elementos são também identificados a partir da descrição detalhada dos casos de uso e são candidatos a *features* opcionais.

Tabela 7.2 - Casos de uso opcionais e seus elementos

Casos de Uso Opcionais	**Elementos**
Simular Vôo	Simulação
Acionar Pré-Testes	Pré-Testes
Ativar Sistema de Segurança Redundante	Sistema de Segurança

3) *Passo 3 - Identificar features do tipo ponto de variação:*

   a. Através do(s) caso(s) de uso comum(ns) - Os elementos que implementam a funcionalidade fundamental do software do LSB pode apresentar as variações mostradas na Tabela 7.3. Estes elementos são candidatos a *features* do tipo ponto de variação mandatória.

   b. Através do(s) caso(s) de uso opcional(is) - Os elementos que implementam a funcionalidade opcional pode apresentar as variações mostradas na Tabela 7.4. Estes elementos são candidatos a *features* do tipo ponto de variação opcional.

Tabela 7.3 - *Feature* ponto de variação em casos de uso comum

Casos de Uso Comum	***Features* Ponto de Variação Mandatória**
Controlar Vôo	Sequência de Eventos
Ler Sensor	Sensor
Acionar Atuadores	Atuador
Enviar Telemedida	Telemedida
Tratar Trajetória	Trajetória

Tabela 7.4 - *Feature* ponto de variação em casos de uso opcionais

Casos de Uso Opcionais	***Features* Ponto de Variação Opcional**
Simular Vôo	Simulação
Ativar Sistema Segurança Redundante	Sistema Segurança

4) Passo 4 - Identificar features variantes

As possibilidades de variações para os elementos listados como ponto de variação encontrados no Passo 3, a partir dos casos de uso comum e opcionais, são mostradas na Tabela 7.5 e 7.6 respectivamente. Estes elementos são candidatos a *features* variantes.

Tabela 7.5 - Variantes de ponto de variação em casos de uso comuns e alternativos

Casos de Uso Comuns	***Features* Ponto de Variação**	***Features* Variante**
Controlar Vôo	Sequência de Eventos	Sequência A, Sequência B
Ler Sensores	Sensor	Sensor Pressão, Sensor Velocidade e Sensor Posição
Acionar Atuadores	Atuadores	Atuador Tubeiras e Atuadores à Gás
Enviar Telemedida	Telemetria	Bloco Medidas 1 e Bloco Medidas 2 [1]
Tratar Trajetória	Trajetória	Trajetória Nominal A Trajetória Nominal B

[1] Bloco de Medidas 1 e 2 são denominações para o bloco de informações que são enviados ao solo pelo Sistema de Telemetria do LSB.

Tabela 7.6 - Variantes de ponto de variação em casos de uso opcionais

Casos de uso Opcionais	***Features* Ponto de Variação**	***Features Variante***
Simular Vôo	Simulação	Simulação com Sensores Simulados, Simulação com Sensores Reais, Simulação Híbrida
Ativar Sistema de Segurança	Sistema de Segurança	Sistema de Segurança de Rolamento, Sistema de Segurança de Gás Frio e Sistema de Segurança Pirotécnico

5) *Passo 5 - Realizar a classificação das features encontradas:*

Classificar as *features* variantes em: inclusivas, mutuamente exclusivas e mutuamente inclusivas (Seção 4.2). A Tabela 7.7 mostra a classificação para *features* variantes.

Tabela 7.7 - Classificação das variantes

Variantes	**Categoria**
Sequência A, Sequência B	Mutuamente exclusiva
Sensor Pressão, Sensor Velocidade, Sensor Posição	Inclusiva
Atuadores Tubeiras, Atuadores à Gás	Inclusiva
Trajetória Nominal A e Trajetória Nominal B	Mutuamente exclusiva
Bloco Medidas 1 e Bloco Medidas 2	Mutuamente inclusiva
Simulação com Sensores Simulados, Simulação com Sensores Reais, Simulação Híbrida	Inclusiva
Sistema de Segurança de Rolamento, Sistema de Segurança de Gás Frio e Sistema de Segurança Pirotécnico	Inclusiva

Com as informações das descrições dos casos de uso e das Tabelas 7.1 a 7.7, foi elaborada a Tabela 7.8 que relaciona os casos de uso e *features*.

Tabela 7.8 - *Features* geradas de casos de uso

Casos de Uso	**Classificação do Casos de Uso**	***Features***	**Pontos de Variação**	**Classificação das *Features***
Controlar Vôo	comum	controle		mandatória
Controlar Vôo	comum	sequência A	sequência de eventos (mandatória)	mutuamente exclusiva
Controlar Vôo	comum	sequência B	sequência de eventos (mandatória)	mutuamente exclusiva
Tratar Trajetória	comum	trajetória nominal A	Trajetória (mandatória)	mutuamente exclusiva
Tratar Trajetória	comum	trajetória nominal B	Trajetória (mandatória)	mutuamente exclusiva
Acionar Pré-Testes	opcional	pré-testes		opcional
Ler Sensores	comum	sensor pressão	sensor (mandatória)	inclusiva
Ler Sensores	comum	sensor velocidade	sensor (mandatória)	inclusiva
Ler Sensores	comum	sensor posição	sensor (mandatória)	inclusiva
Acionar Atuadores	comum	tubeiras	atuador (mandatória)	inclusiva
Acionar Atuadores	comum	atuadores à gás	atuador (mandatória)	inclusiva
Enviar Telemedidas	comum	bloco medidas 1	telemedidas (mandatória)	mutuamente inclusiva
Enviar Telemedidas	comum	bloco medidas 2	telemedidas (mandatória)	mutuamente inclusiva
Simular Vôo	opcional	sensores simulados	simulação (opcional)	inclusiva
Simular Vôo	opcional	sensores reais	simulação (opcional)	inclusiva
Simular Vôo	opcional	simulação híbrida	simulação (opcional)	inclusiva
Acionar Sistema de Segurança Redundante	opcional	sistema de segurança rolamento	Sistema de Segurança (opcional)	inclusiva
Acionar Sistema de Segurança Redundante	opcional	sistema de segurança gás frio	Sistema de Segurança (opcional)	inclusiva
Acionar Sistema de Segurança Redundante	opcional	sistema de segurança pirotécnico	Sistema de Segurança (opcional)	inclusiva

7.3.2 Modelagem de *Features*

Nesta atividade foi elaborado o modelo de *features* para linha de produtos de software do LSB, usando as *features* extraídas a partir de seus casos de uso, apresentadas na Tabela 7.8. O resultado está apresentado na Figura 7.2.

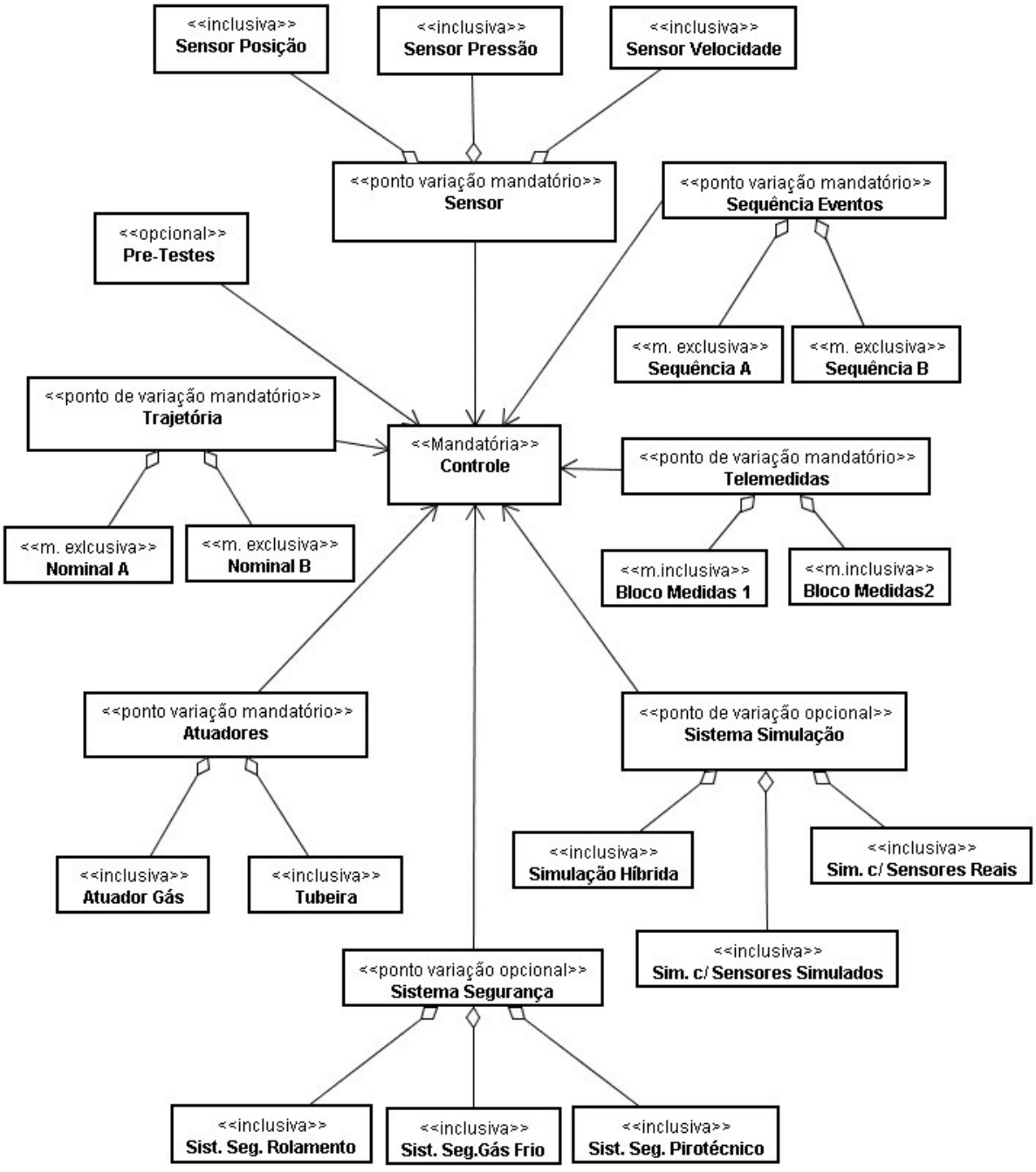


Figura 7.2 - Modelo de *features* para a linha de produtos de software do LSB

Observa-se que o modelo de *features* contém elementos necessários para realizar a funcionalidade comum e a variabilidade da linha de produtos de software do LSB. A funcionalidade comum é realizada através da *feature* mandatória Controle, das *features* do tipo ponto de variação mandatória Sensores, Atuadores, Telemedidas, Sequência de Eventos e Trajetória e das *features* variantes representadas nestes pontos de variação. A funcionalidade opcional é realizada através da *feature* opcional Pré-Testes, assim como, as *features* do tipo ponto de variação opcional Sistema de Simulação e Sistema de Segurança, e das *features* variantes representadas nestes pontos de variação.

7.4 Especificação da Variabilidade

As atividades que compõem a atividade de Especificação da Variabilidade são: a Determinação da Cardinalidade, Escolha do *Binding Time* e a Representação da Variabilidade.
As próximas Seções 7.4.1 a 7.4.3 mostram a aplicação destas atividades.

7.4.1 Determinação da Cardinalidade

A Determinação da Cardinalidade define o número de variantes que podem ser escolhidas em um determinado ponto de variação. Estes dados são obtidos a partir da descrição textual do sistema da Seção 7.2.1 e do modelo de *features* apresentado da Seção 7.3.2. A Tabela 7.9 apresenta as cardinalidades das variantes do LSB.

Tabela 7.9 - Cardinalidade das variantes do LSB

Pontos de Variação	Variantes	Cardinalidades
Sensor	Sensor Pressão	[1..*]
Sensor	Sensor Velocidade	[1..*]
Sensor	Sensor Posição	[1..*]
Atuador	Atuadores de Tubeiras	[1..*]
Atuador	Atuadores à Gás	[1..*]
Sist. de Segurança	Sistema de Segurança de Rolamento	[0..*]
Sist. de Segurança	Sistema de Segurança de Gás Frio	[0..*]
Sist. de Segurança	Sistema de Segurança Pirotécnico	[0..*]
Sist. de Simulação	Simulação Híbrida	[0..*]
Sist. de Simulação	Simulação com Sensores Reais	[0..*]
Sist. de Simulação	Simulação com Sensores Simulados	[0..*]
Telemedidas	Bloco Medidas1	[1..*]
Telemedidas	Bloco Medidas2	[1..*]
Trajetória	Nominal A	[0..1]
Trajetória	Nominal B	[0..1]
Seqüência de Eventos	Seqüência A	[0..1]
Seqüência de Eventos	Seqüência B	[0..1]

Observa-se que deve existir pelo menos um sensor para cada tipo de sensores e pelo menos um atuador para cada tipo de atuadores. Podem existir sistemas de segurança e simulações, e se existirem podem ser escolhidos mais de um para cada tipo. Deve existir pelo menos um bloco de telemedidas para cada tipo. Podem existir zero ou uma trajetória e seqüência de eventos para cada tipo.

7.4.2 Representação da Variabilidade por Classes

Nesta atividade, o modelo de *features*, elaborado na atividade de Identificação da Variabilidade e apresentada na Figura 7.2, foi transformado em diagrama de classes

da linha de produtos de software. Esta transformação consistiu da identificação de classes que representem as *features* e que realizem as funcionalidades definidas pelos casos de uso.

A Tabela 7.10 mostra as classes necessárias para representar as *features* da linha de produtos do LSB.

Tabela 7.10 - Dependência entre *features* e classes

Nome da *Feature*	**Nome da Classe**
Controle	Controle
Sensor	Sensor
Atuador	Atuador
Sistema de Segurança	Sistema de Segurança
Trajetória	Trajetória
Sequência de Eventos	Sequência de Eventos
Pré-Testes	Sistema Pré-testes
Telemedidas	Sistema Telemetria
Sistema Simulação	Sistema Simulação

O diagrama de classes resultante está apresentado na Figura 7.3.

7.4.3 Escolha do *Binding Time*

No processo GVLPS aplicado ao software do LSB, o *Binding Time* ocorre em tempo de execução. O mecanismo de variabilidade do GVLPS, baseado em modelos de objetos adaptáveis e em reflexão definem esta característica e as variantes, que serão implementadas como objetos e criados dinamicamente em tempo de execução.

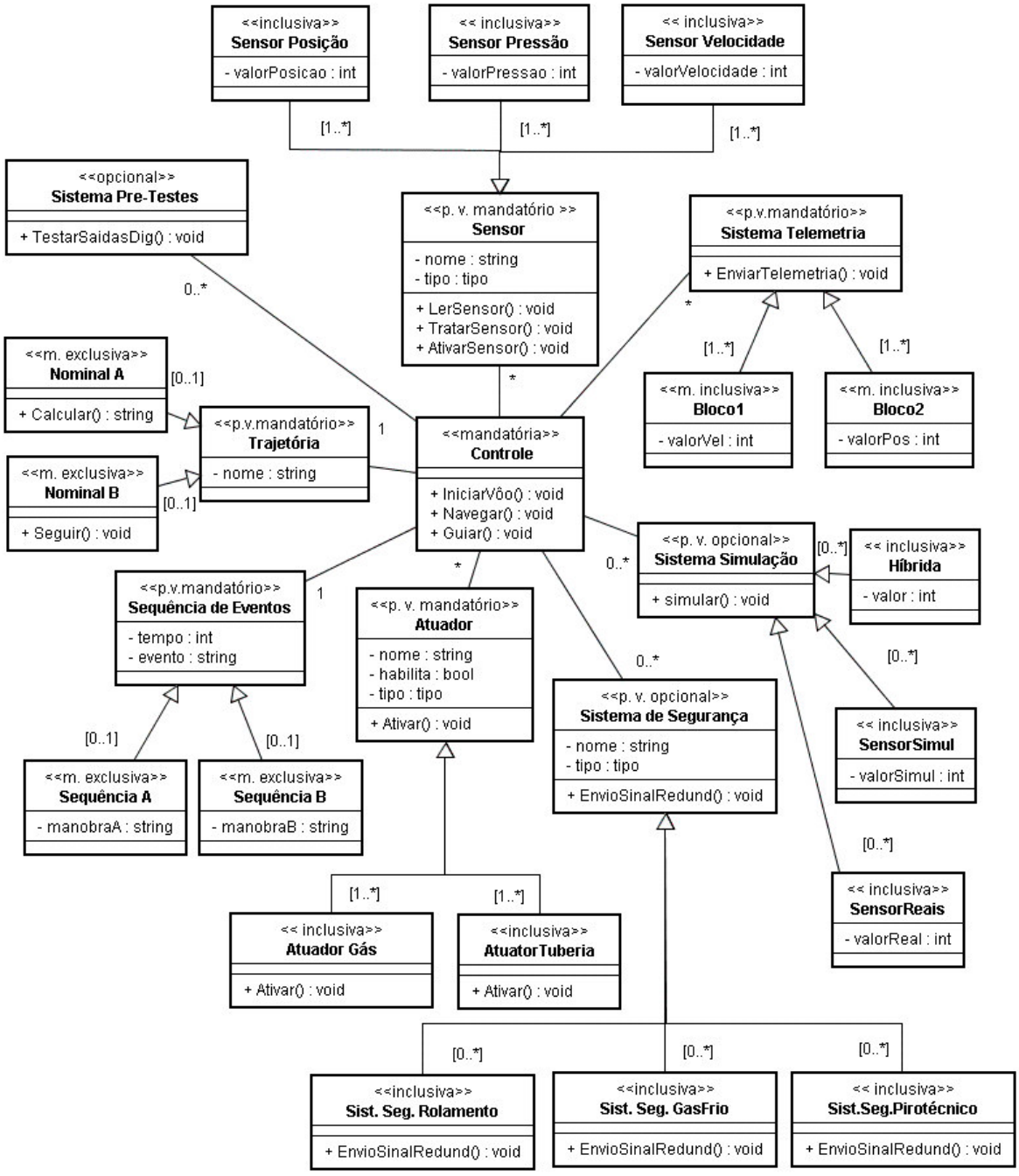


Figura 7.3 - Diagrama de classes de linha de produtos de software do LSB

7.5 Implementação da Variabilidade

Esta seção apresenta o mecanismo que implementa a variabilidade no processo GVLPS aplicado ao software do LSB. O mecanismo de variabilidade é baseado nas

técnicas de modelos de objetos adaptáveis e em reflexão, apresentadas no Capítulo 5. Sua arquitetura foi apresentada na Seção 6.6 desta tese.

A implementação do mecanismo de variabilidade inicia-se com a aplicação dos padrões de projetos dos modelos de objetos adaptáveis no diagrama de classes de linha de produtos de software do LSB apresentado na Figura 7.3.

A aplicação dos padrões no processo GVLPS é exemplificada nos pontos de variação Sensor, Atuadores e Sistema de Segurança, que estão destacados na Figura 7.4.

Esta figura é o diagrama de classes da Figura 7.3 parcial, mostrando apenas as variantes que serão implementadas, e com uma disposição das classes para permitir melhor visualização após a aplicação dos padrões de modelos de objetos adaptáveis.

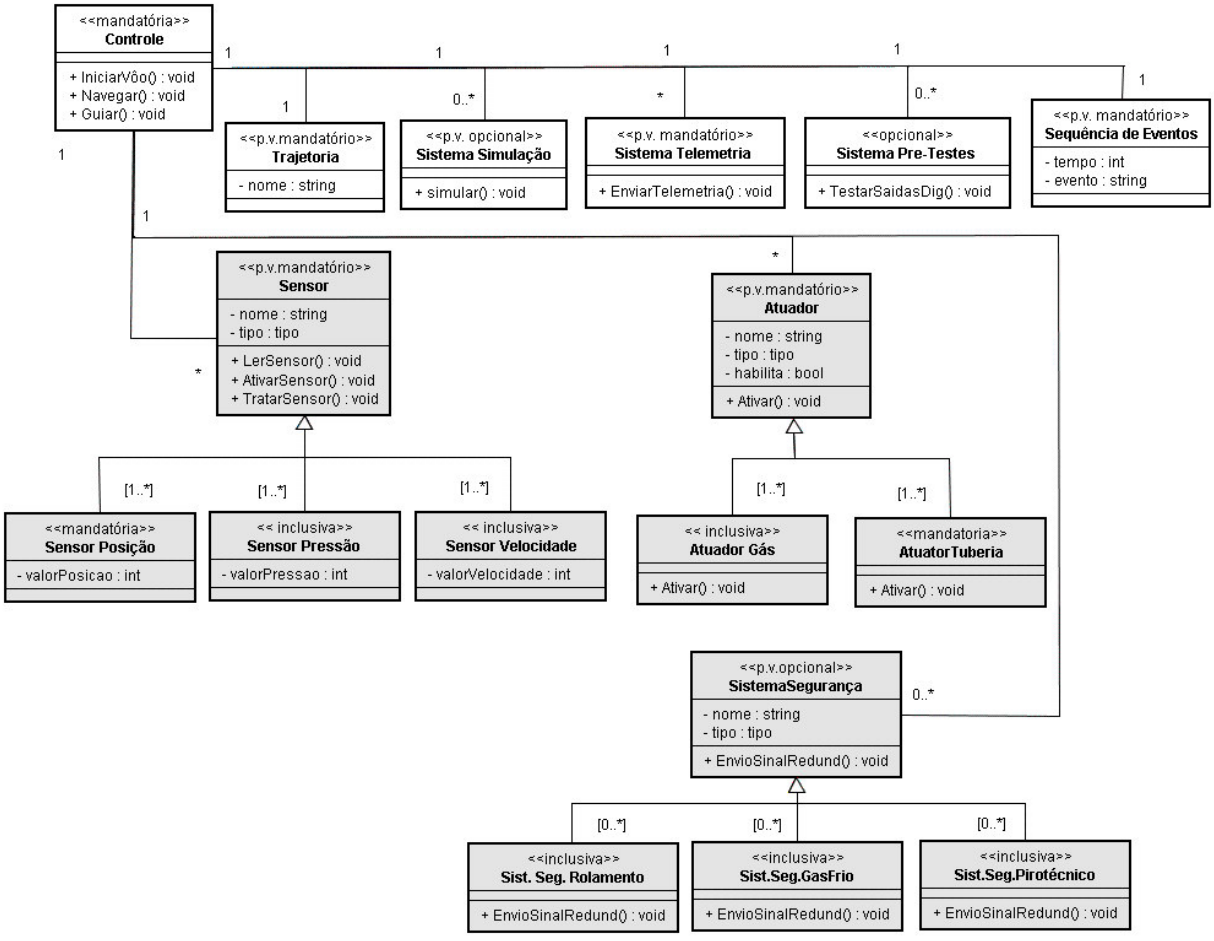


Figura 7.4 - Diagrama de classes parcial de linha de produtos de software do LSB

Com a aplicação dos modelos de objetos adaptáveis é possível generalizar o modelo de classes tal que seja possível criar novas instâncias de tipos de sensores, de atuadores e de sistema de segurança, que correspondem a novas variantes. Para isso, serão utilizados os padrões de modelos de objetos adaptáveis: *TypeObject, Property, TypeSquare* e *Strategy*, que foram apresentados nas Seções 5.2.1 a 5.2.4 deste trabalho.

7.5.1 Aplicação do Padrão *TypeObject*

O padrão *TypeObject* separa uma entidade de um tipo de entidade, fazendo com que um número desconhecido de subclasses sejam instâncias de uma classe genérica. O padrão *TypeObject* foi usado, no caso, para generalizar os tipos de Sensores, Atuadores e Sistema de Segurança. Após a aplicação do padrão *TypeObject*, o diagrama de classes de linha de produtos de software do LSB da Figura 7.4 passa a ser representado como mostra a Figura 7.5.

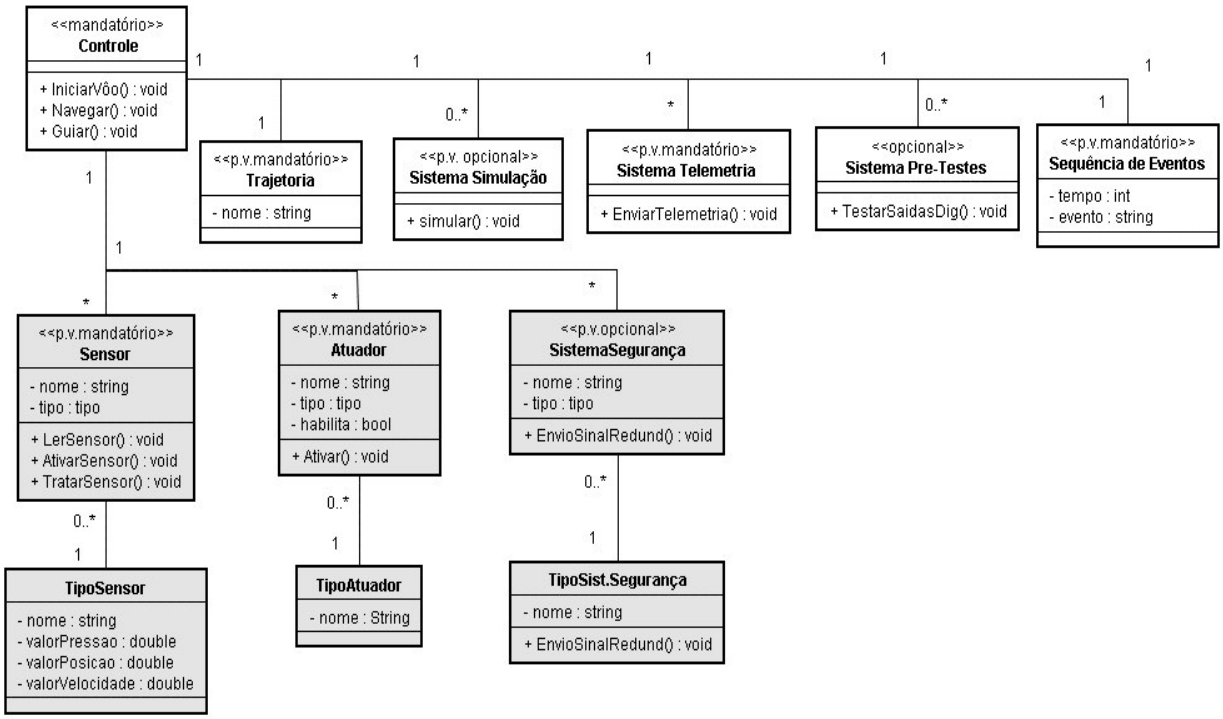


Figura 7.5 - Diagrama de classes do LSB após a aplicação do padrão *TypeObject*

Observa-se que, após a aplicação deste padrão, as classes TipoSensor, TipoAtuador e TipoSist.Segurança substituíram as classes que representavam variantes, que possuíam o relacionamento de herança com os pontos de variação Sensores, Atuadores e SistemaSegurança. O objetivo é permitir a criação de vários tipos de sensores, vários tipos de atuadores e vários tipos de sistema de segurança, implementando, assim, a variabilidade para a linha de produtos de software do LSB.

Os atributos e métodos comuns a todos os sensores, atuadores e sistema de segurança se mantêm, respectivamente, nas classes Sensores, Atuadores e SistemaSegurança; os atributos específicos de cada tipo passam para as classes TipoSensor, TipoAtuador e TipoSist.Segurança. Por exemplo, um sensor de pressão tem um atributo específico valorPressão, um sensor de Posição tem o atributo específico valorPosição e o sensor de velocidade, um atributo específico valorVelocidade.

7.5.2 Aplicação do Padrão *Property*

A aplicação do padrão *TypeObject* na parte de sensores, na seção anterior, permitiu que vários tipos de sensores fossem representados através da classe genérica TipoSensor. Entretanto, estes tipos de sensores podem ter atributos diferentes uns dos outros e estes atributos devem ser representados. O mesmo ocorre para os tipos de atuadores e tipos de Sistema de Segurança, que são representados por suas classes genéricas TipoAtuador e TipoSist.Segurança respectivamente.

A representação de tipos com atributos diferentes foi possível através da aplicação do padrão *Property*, que cria uma nova classe Propriedade para representar as propriedades para cada classe entidade, como mostra a Figura 7.6.

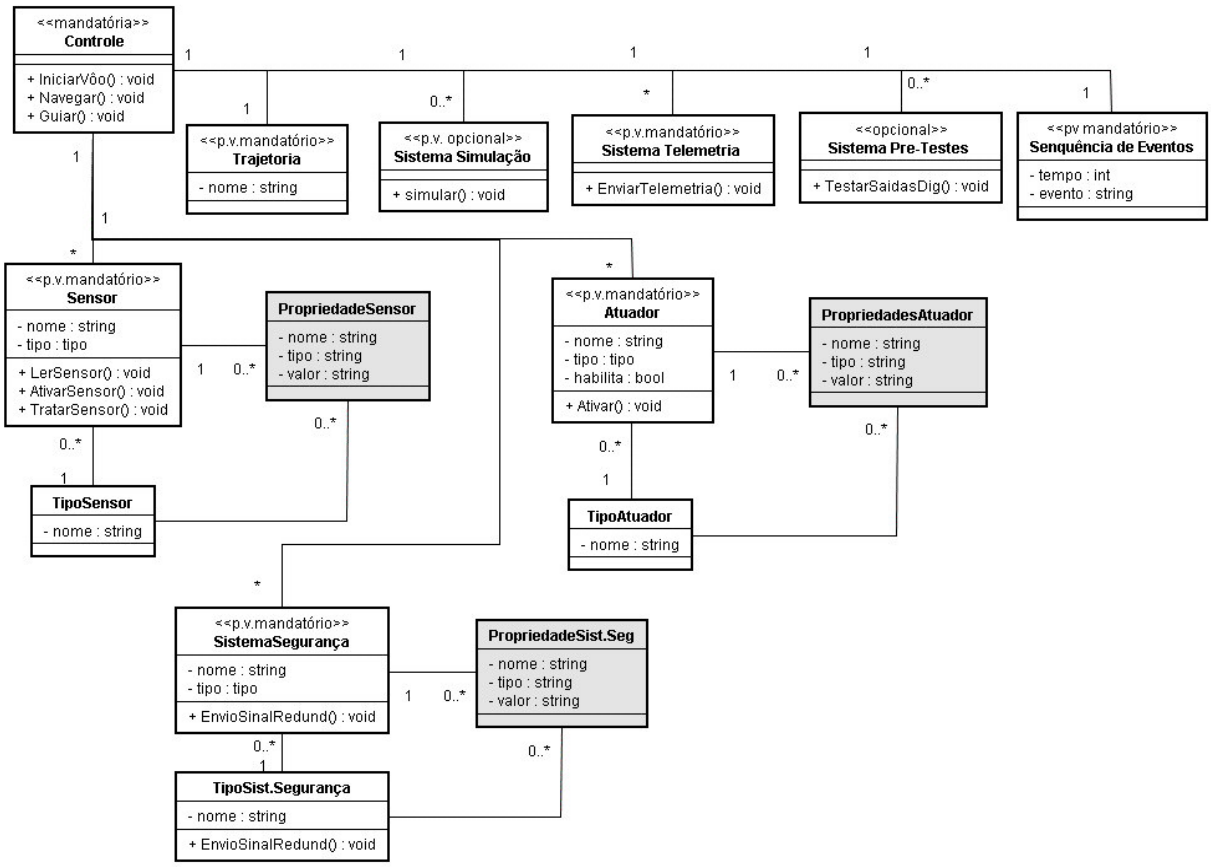


Figura 7.6 - Diagrama de classes do LSB após a aplicação do padrão *Property*

Observa-se que após a aplicação do padrão *Property,* os atributos específicos que pertenciam a classes TipoSensor, TipoAtuador e TipoSist.Segurança são removidos destas classes. Estes atributos passam a ser representados, respectivamente, pelas classes PropriedadeSensor, PropriedadeAtuador e PropriedadeSis.Seg. Os atributos valorPosição, valorVelocidade e valorPressão foram removidos na classe TipoSensor.

7.5.3 Aplicação do Padrão *TypeSquare*

Após a aplicação do padrão *Property*, todos os atributos estão representados pela classe Propriedade; entretanto, ainda não é possível identificar quais atributos pertencem as quais tipos de entidade. Por exemplo, no caso de Sensores, não é

possível identificar quais atributos pertencem a um determinado tipo de sensor. São, então, incluídas as classes TipoPropriedade para permitir esta identificação. O resultado da aplicação do padrão *TypeSquare* nas classes do LSB é apresentada na Figura 7.7.

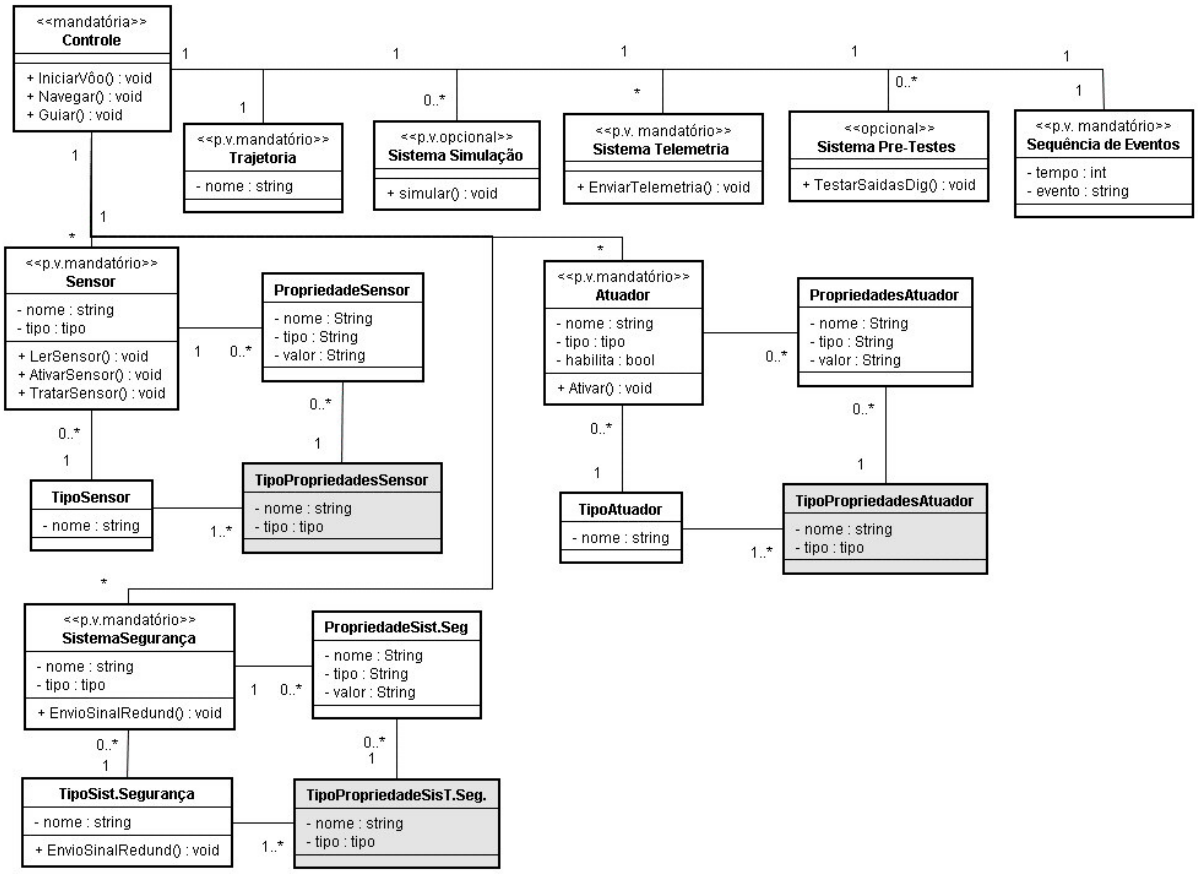


Figura 7.7 - Diagrama de classes do LSB após a aplicação do padrão *TypeSquare*

Observa-se que, após a aplicação do padrão *TypeSquare*, cada classe TipoEntidade é associada com uma classe TipoPropriedade adicionada, possibilitando a definição de atributos e tipos de atributos para cada tipo de sensor, atuador ou sistema de segurança.

Desta forma, a classe Sensor possui atributos representados pela classe PropriedadeSensor. Cada atributo representado por esta classe tem um tipo, especificado pela classe TipoPropriedadeSensores. A classe TipoSensor especifica os tipos das propriedades para suas entidades.

7.5.4 Aplicação do Padrão *Strategy*

Para representar o comportamento das entidades, foi aplicado o padrão *Strategy*. Este padrão permite trabalhar com uma interface genérica para um conjunto de métodos, facilitando a chamada para cada um deles.

A Figura 7.8 apresenta o diagrama de classes do LSB após a aplicação do padrão *Strategy*.

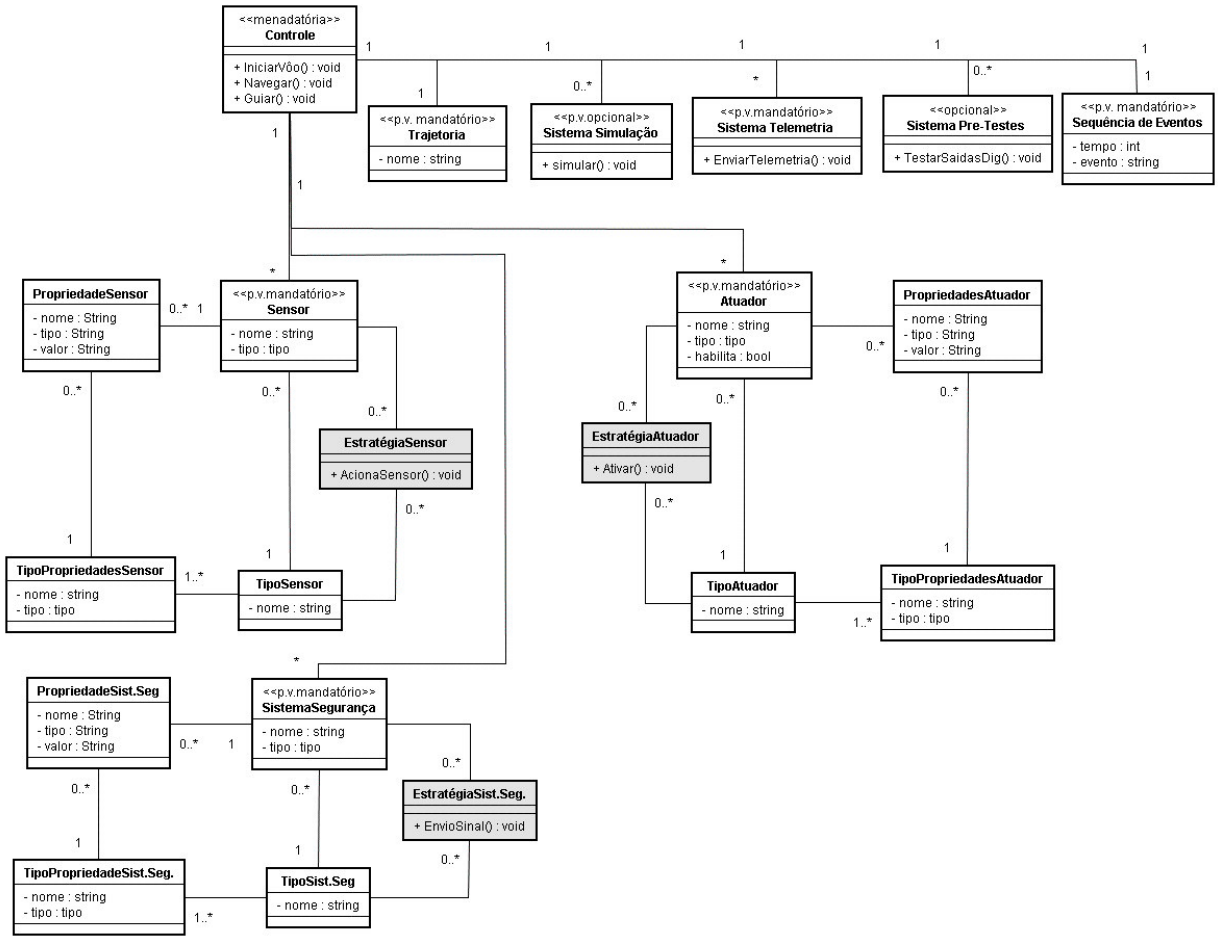


Figura 7.8 - Diagrama de classes do LSB após a aplicação do padrão *Strategy*

Observa-se que, após a aplicação do padrão *Strategy*, os métodos pertencentes a classes Entidades são retiradas delas e passam a ser representados por uma classe Estratégia. Desta forma, os métodos das classes Sensor, Atuador e

SistemaSegurança passam a pertencer a classes EstratégiaSensor, EstratégiaAtuador e EstratégiaSist.Seg. respectivamente.

7.5.5 Projeto do Mecanismo de Variabilidade

Após aplicação dos padrões de modelos de objetos adaptáveis, o modelo torna-se genérico o suficiente para servir como base para a criação de outras variantes. Este modelo de classes pode ser visto como um modelo genérico de classes de linha de produtos de software do LSB porque os tipos de sensores, de atuadores e de sistema de segurança não estão explicitados diretamente no modelo resultante.

Os modelos de objetos adaptáveis, ao serem implementados, tem seu metadado armazenado em repositório para interpretá-lo (Seção 5.2). Este metadado corresponde às informações relacionadas às classes, tipos de classes, atributos, tipos de atributos, e métodos. Assim, é possível configurar aplicações com diferentes tipos de sensores e atuadores e manipular classes correspondentes com seus respectivos atributos e métodos.

Desta forma, a partir do diagrama de classes mostrado na Figura 7.8, é possível criar novas instâncias (variantes) de uma determinada classe, não havendo a limitação de números de classes como as encontradas em alguns mecanismos de variabilidade apresentados.

A partir do diagrama de classes apresentado na Figura 7.8, foi feita a implementação do mecanismo de variabilidade para o software do LSB, que consiste de forma sucinta, em projetar e implementar as classes que apresentam variação e configurar o repositório do mecanismo de variabilidade para armazenar os objetos, ou seja, as variantes criadas pelo mecanismo de variabilidade.

Inicialmente, são projetadas e implementadas as classes: Sensor, Atuador, e Sistema de Segurança e as respectivas classes de seus tipos. O repositório do mecanismo de variabilidade deve ser utilizado para armazenar os possíveis objetos (instâncias) a serem criadas pelo mecanismo de variabilidade. Para a manipulação

de dados, foi implementado o sistema Gerenciador de Repositório, como citado na Seção 6.6.

Para esclarecer a organização dos dados no repositório do mecanismo de variabilidade, apresenta-se um exemplo considerando cinco objetos cadastrados para a classe Sensor, conforme mostra a Tabela 7.11.

Tabela 7.11 - Objetos da classe Sensor

ID	Sensor
01	S1
02	S2
03	S3
04	S4
05	S5

Vários sensores podem fazer parte de um veículo. Na Tabela 7.12 está apresentada a configuração de dois veículos: o LSB1 com sensores S1, S2, S3, S4 e o LSB2 com S1, S3, S4 e S5.

Tabela 7.12 - Objetos Veículos e Sensores relacionados

Veículo	Sensor
LSB1	S1
LSB1	S2
LSB1	S3
LSB1	S4
LSB2	S1
LSB2	S3
LSB2	S4
LSB2	S5

Os objetos sensores podem ser de vários tipos. Foram considerados os sensores dos tipos velocidade, posição e pressão que são cadastrados conforme apresentado na Tabela 7.13.

Tabela 7.13 - Objetos da classe TipoSensor

ID	TipoSensor
01	velocidade
02	posicao
03	pressao

A classe Sensor possui um atributo comum Tipo. Para cada objeto da classe Sensor instanciado, é definido um objeto da classe TipoSensor ao atributo Tipo; em outras palavras, o atributo Tipo contém o tipo de sensor.

No exemplo, foi considerado que S1 e S4 são sensores de velocidade, S2 e S5 são sensores de posição e S3 é sensor de pressão. O repositório do mecanismo de variabilidade deve armazenar o relacionamento entre objetos da classe Sensor e objetos da classe TipoSensor, como mostra a Tabela 7.14.

Tabela 7.14 - Relacionamento das classes Sensor e TipoSensor

Sensor	TipoSensor
S1	velocidade
S2	posicao
S3	pressao
S4	velocidade
S5	posição

Os objetos da classe TipoSensor podem ter atributos diferentes. Para viabilizar estas diferenças, os objetos da classe TipoSensor são associados à classe TipoPropriedade, como mostra a Tabela 7.15.

O TipoSensor velocidade está relacionado com valor_velocidade e nome_velocidade do TipoPropriedade, assim como, o TipoSensor posição e pressão estão relacionados respectivamente com o nome_posição, valor_posição e nome_pressao e valor_pressão do TipoPropriedade.

Tabela 7.15 - Relacionamento das classes TipoSensor e TipoPropriedade

TipoSensor	TipoPropriedade
velocidade	valor_velocidade
velocidade	nome_velocidade
posicao	nome_posicao
posicao	valor_posicao
pressao	nome_pressao
pressao	valor_pressao

Finalmente, para que os métodos possam ser chamados dinamicamente, estes métodos devem ser conhecidos. Assim, utilizando-se o padrão *Strategy*, cria-se três classes de estratégias concretas: Ler Sensor, TestarSensor e AtivarSensor que são implementadas a partir de uma interface Estratégia. Estas classes são armazenadas previamente no repositório do mecanismo de variabilidade. A Tabela 7.16 apresenta as estratégias concretas que podem ser associados à classe Sensor.

Tabela 7.16 - Métodos da classe Sensor

NomeClasse	NomeMetodo
sensor	LerSensor
sensor	TestarSensor
sensor	AtivarSensor

7.5.6 Funcionamento do Sistema Gerador de Aplicação

O sistema Gerador de Aplicação tem o objetivo de realizar a seleção e criação das variantes, a partir do repositório, no mecanismo de variabilidade do processo GVLPS. O funcionamento deste sistema é ilustrado através da criação de variantes do ponto de variação Sensor.

Para isso, para cada sensor, são realizadas as seguintes atividades:

- Selecionar o tipo de sensor que se deseja criar;

- Acionar a criação do sensor;
- Este acionamento especifica o sensor com o tipo de sensor selecionado, carrega os tipos de propriedades, propriedades e métodos cadastrados para este sensor;
- O acionamento dos métodos pertencentes à classe Sensor se torna disponível.

A Figura 7.9 ilustra estas atividades através de um diagrama de casos de uso das atividades relacionadas à criação do objeto Sensor.

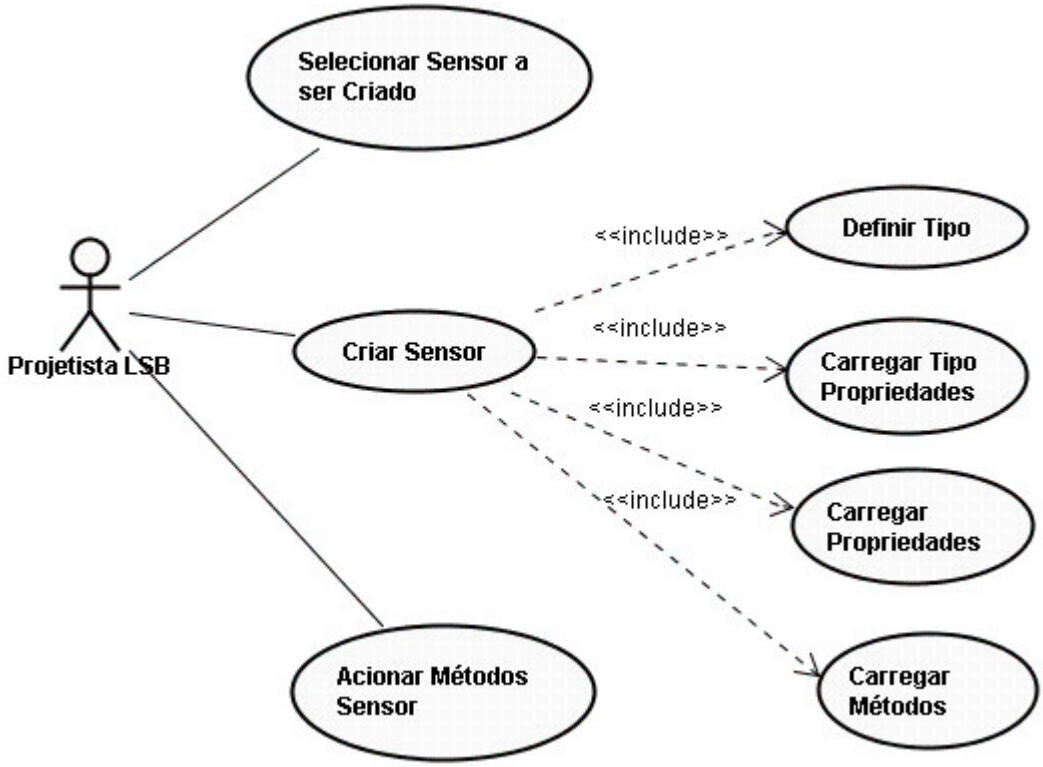


Figura 7.9 - Casos de uso para criar Sensor

A dinâmica da criação das variantes relacionadas ao ponto de variação Sensor pode ser melhor explicada através de um diagrama de seqüência. A Figura 7.10 apresenta o diagrama de seqüência para o Caso de Uso Criar Sensor. Para criar os sensores, o sistema deve estar iniciado, ou seja, o conteúdo do repositório do mecanismo de variabilidade deve estar disponível para o projetista.

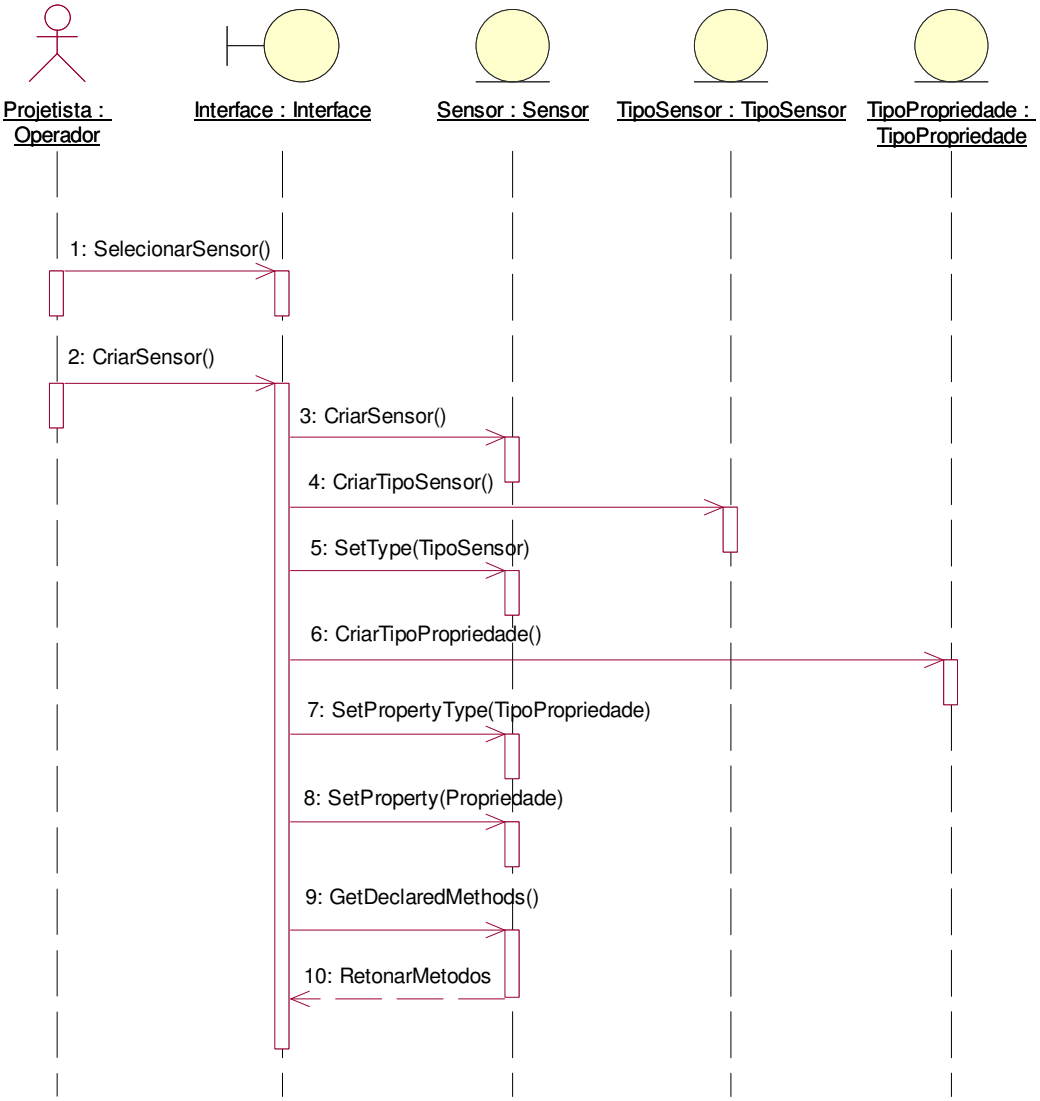


Figura 7.10 - Diagrama de seqüência – Criar Sensor

Conforme mostra a Figura 7.10, ao iniciar, o sistema se encarrega de buscar os sensores, tipos de sensores e tipos de propriedades já previamente definidos através do sistema Gerenciador de Repositório. O Projetista do LSB seleciona, entre uma lista de sensores e tipos de sensores carregados, o tipo de sensor deseja criar. A partir disso, o projetista pode acionar a criação deste objeto. Os objetos Sensor e o TipoSensor são criados; a classe Sensor deve relacionar o objeto TipoSensor ao objeto Sensor criado, e para isso aciona o método setType.

Após este acionamento, é criado o objeto TipoPropriedade e acionado o método setPropertyType para interligar este objeto criado ao objeto TipoSensor, já atribuído ao objeto Sensor.

Na seqüência, o objeto Propriedade é criado como uma *hashtable* na classe Sensor e é acionado o método setProperty para interligar os objetos Propriedade e TipoPropriedade.

Os métodos relativos à classe Sensor são carregados, através do método getDeclaredMethods para ficarem disponível ao acionamento do operador.

O procedimento detalhado através do diagrama de seqüência apresentado na Figura 7.10 habilita a possibilidade de criação de vários sensores de vários tipos, com várias propriedades também de diferentes tipos. O mesmo procedimento de criação se aplica aos atuadores e sistema de segurança do software do LSB.

A seqüência de operações apresentadas através do diagrama de seqüência da Figura 7.10 é válida para todos os pontos de variação encontrados através do processo GVLPS.

7.6 Considerações Finais

Apresentou-se neste capítulo, a aplicação do processo GVLPS ao software do LSB.

Através da aplicação do processo GVLPS ao software do LSB foi possível exemplificar como são realizadas cada atividade do processo, assim como, apresentar como os artefatos de cada atividade do processo são gerados.

As estratégias para a construção do mecanismo de variabilidade também foram apresentadas. Detalhou-se a aplicação dos padrões de projetos de modelos de objetos adaptáveis para se obter um modelo genérico do LSB. O objetivo em se construir um modelo genérico do LSB é habilitar a criação das variantes.

Foi exemplificado como o repositório do mecanismo de variabilidade deve ser povoado para auxiliar a criação das variantes e finalmente, mostrou-se a dinâmica da criação destas variantes através deste mecanismo.

Ressalta-se que para a utilização dos modelos de objetos adaptáveis é necessário um estudo prévio dos padrões de projeto que os constituem, pois, encontrou-se certa dificuldade no entendimento da aplicação destes padrões. Para concretizar este entendimento, além da consulta à literatura, foram realizadas várias tentativas de modelagem e implementação dos padrões de projetos relacionados.

Apesar da aplicação do processo GVLPS no software do LSB, o processo pode ser adotado, de forma geral, por qualquer outra área.

"Não passe pela vida.
Cresça através dela."

Eric Butterworth

8 PROTÓTIPO DO MECANISMO DE VARIABILIDADE DO GVLPS

Este capítulo apresenta o desenvolvimento do protótipo do mecanismo de variabilidade do processo GVLPS apresentado no Capítulo 6. Este protótipo é constituído pelos sistemas Gerenciador de Repositório e Gerador de Aplicações, bem como, por um repositório que armazena as informações relacionadas às variantes da aplicação a ser gerada.

O protótipo do mecanismo de variabilidade do processo GVLPS foi desenvolvido para ilustrar a utilização dos padrões de modelos de objetos adaptáveis e reflexão. Tanto o entendimento quanto a implementação dos modelos de objetos adaptáveis e mecanismos de reflexão foram considerados trabalhosos e, portanto neste capítulo, são detalhados com mais profundidade para permitir maior compreensão destas técnicas.

A apresentação do protótipo do mecanismo de variabilidade do processo GVLPS é feita através dos seguintes tópicos:

- Ambiente de Implementação: consiste da apresentação da plataforma de desenvolvimento e linguagem utilizada, assim como o aplicativo para base de dados selecionado.

- Implementação dos padrões de modelos de objetos adaptáveis: consiste da apresentação dos trechos de código que implementam os padrões *TypeObject, Property, TypeSquare* e *Strategy*.

- Implementação do mecanismo de reflexão: consiste da apresentação do código reflexivo, através do mecanismo de reflexão da linguagem de programação selecionada para o desenvolvimento do protótipo.

- Descrição dos sistemas que constituem o protótipo:

- Gerenciador de Repositório - responsável pela preparação do repositório do mecanismo de variabilidade com as informações relacionadas às variantes, seus tipos, propriedades e métodos, que serão utilizadas pelo Gerador de Aplicação.

- Gerador de Aplicação - responsável pela seleção e criação das variantes do sistema e que permite também acionar o sistema para execução.

8.1 Ambiente de Implementação

Para o desenvolvimento do protótipo do mecanismo de variabilidade do processo GVLPS foi utilizada a linguagem de programação Java, desenvolvida pela Sun Microsystems (SUN, 2008). Um dos principais motivos, que levaram à escolha da linguagem Java, foi o suporte que ela oferece à programação reflexiva através de seus mecanismos de reflexão fornecidos pela *Reflection API* (*Application Programming Interface*), que é composta pelas classes do pacote *java.lang.reflect.* Este pacote fornece classes e interfaces para a obtenção de informações reflexivas sobre classes e objetos (SUN, 2008). Este suporte foi considerado suficiente para o desenvolvimento deste protótipo porque atende as necessidades de adaptação que o sistema visa.

Além disso, existe uma utilização da linguagem Java em modelos de objetos adaptáveis em um projeto bem sucedido de aplicação espacial, que consiste do projeto de uma arquitetura distribuída, configurável e adaptável para um software de controle de satélites (THOMÉ, 2004).

Para a implementação do protótipo foi utilizado um ambiente de desenvolvimento integrado Java, denominado JBuilder, da *Embarcadero Technologies* (EMBARCADERO, 2008) que apresenta suporte à linguagem Java.

Para implementar o sistema Gerenciador de Repositório foi utilizado o aplicativo Microsoft Access 2007. Este aplicativo foi selecionado, pois era suficiente para o

porte do protótipo, entretanto, poder-se-ia utilizar qualquer outro gerenciador de banco de dados para fazer parte do protótipo descrito.

8.2 Implementação dos Padrões de Modelos de Objetos Adaptáveis

O mecanismo de variabilidade, implementado através de objetos adaptáveis e reflexão, possibilita, a partir da seleção prévia pelo usuário, a criação de novos objetos, o estabelecimento de seus tipos, a atribuição de suas propriedades de acordo com o objeto criado e ainda, o acionamento de métodos apresentados por estes objetos criados, em tempo de execução.

Portanto, para projeto de software de um novo veículo, realiza-se uma nova seleção de objetos, no caso sensores, atuadores e sistema de segurança, armazenados no repositório. Os objetos correspondentes são instanciados, através do fornecimento de suas opções de tipos, propriedades e métodos a serem acionados, desde que as informações sobre os objetos, seus tipos, propriedades e métodos estejam definidas no repositório.

Desta forma, inicialmente, foram criadas classes do tipo ponto de variação como Sensor, Atuador e Sistema de Segurança. Estas classes foram sendo, então, gradualmente modificadas, inserindo-se novos atributos e métodos para que assim fossem transformadas através dos padrões de modelos de objetos adaptáveis.

Nesta seção apresentam-se detalhes da implementação dos padrões de modelos de objetos adaptáveis utilizados para o desenvolvimento do protótipo.

8.2.1 Implementação do Padrão *TypeObject*

Conforme citado na Seção 5.2.1, o objetivo deste padrão é criar várias subclasses como instâncias de uma classe genérica. No caso do protótipo, foi utilizado para criar vários objetos sensores, atuadores e sistemas de segurança diferentes de tipos

diferentes. Apresenta-se a implementação deste padrão a partir das classes Sensor e TipoSensor, como ilustra a Figura 8.1 a seguir.

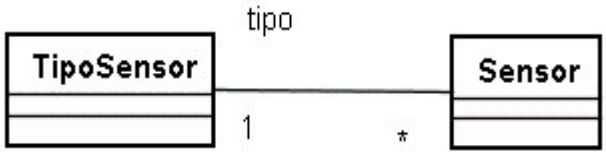


Figura 8.1 - Padrão *TypeObject* para Sensor e TipoSensor

Desta forma, gera-se uma classe tipo de entidade do padrão *TypeObject* que foi chamada TipoSensor com atributos como, por exemplo, atributo name e seus métodos de acesso. A Figura 8.2 mostra a classe TipoSensor criada.

```
public class TipoSensor
{
  protected String name;
  ...

  //----------Atribui o nome do tipo de sensor-----------------
  public void setName(String nome)
  {
    name = nome;
  }

  //---------Recupera nome do tipo de Sensor-------------------
  public String getName()
  {
    return name;
  }
  ...

}
```

Figura 8.2 - Classe TipoSensor

Após isso, cria-se a classe entidade do padrão *TypeObject*, que no caso é a classe Sensor. A classe Sensor possui um atributo type do tipo da classe tipo de entidade, ou seja, do tipo da classe TipoSensor criado anteriormente e mostrado na Figura 8.2.

Na classe Sensor, foram criados métodos de acesso ao atributo type, onde o parâmetro esperado pelo método setType é do tipo TipoSensor, assim como o valor de retorno do método getType(), como mostra o trecho de código na Figura 8.3:

```
public class Sensor
  {
    protected String name;
    protected TipoSensor type;
    ...

    public String getName()
    {
      return name;
    }
    //-------------------------------------------------------------
    public void setName(String nome)
    {
      name = nome;
    }
    //-------------Atribui o tipo de sensor----------------------
    public void setType(TipoSensor Type)
    {
      type = Type ;
    }
    //-------------Retorna tipo de sensor------------------------
    public TipoSensor getType()
    {
      return type;
    }
```

Figura 8.3 - Classe Sensor

Para a execução do padrão *TypeObject* basta instanciar objetos da classe Sensor e TipoSensor e atribuir, a um determinado sensor, um tipo escolhido. Isto pode ser feito como mostra o código apresentado na Figura 8.4.

```
  void btnsensores_actionPerformed(ActionEvent e)
    {
      Sensor s = new Sensor();
      TipoSensor ts = new TipoSensor();
      ...

      ts.setName("velocidade");
      s.seType(ts);
       ...

    }
```

Figura 8.4 - Criação do objeto Sensor

Desta forma, utilizando-se o padrão *TypeObject* é possível instanciar vários objetos Sensores variando-se seu tipo, sem ter que criar novas diferentes classes para cada tipo a ser instanciado.

Entretanto, apenas com a utilização do padrão *TypeObject,* os sensores instanciados terão sempre os mesmos atributos, já que serão instanciados a partir da classe genérica Sensor. Como o objetivo é obter sensores de diversos tipos, com atributos diferentes, aplicam-se os padrões *Property* e *TypeSquare*, como mostram as seções a seguir.

8.2.2 Implementação do Padrão *Property*

A classe TipoSensor implementada possui o atributo name que pode ser comum a todos os diferentes sensores que devem ser instanciados. Entretanto, diferentes tipos de sensores geralmente possuem atributos diferentes, mais adequados para as medidas físicas às quais estão associados. Por exemplo, um sensor A do tipo velocidade necessita de um atributo *status* que o sensor B do tipo posição não precisa.

Para isso, uma alternativa é criar uma lista de propriedades para cada tipo de sensor, que pode ser implementada através de uma lista, vetor ou *hashtable*. No caso do protótipo, foi selecionada a implementação através de *hashtable* que possibilita armazenar objetos de diferentes tipos e recuperá-los, através de suas chaves.

Para a implementação do padrão *Property*, apresentado na Seção 5.2.2, primeiramente foi criada uma classe TipoPropriedade com seus atributos comuns e métodos de acesso, como mostra o código na Figura 8.5.

```
public class TipoPropriedade
 {
   protected String name;
   ...

   //----------Atribui o nome do tipo de Propriedade----------
   public void setName(String nome)
   {
     name = nome;
   }
   //-------------Retorna o nome do Tipo de Propriedade--------
   public String getName()
   {
     return name;
   }
   ...

 }
```

Figura 8.5 - Classe TipoPropriedade

Para guardar e manipular os diversos tipos de propriedades, uma *hashtable*, denominada propertyTypes foi inserida na classe TipoSensor, juntamente com seus métodos de acesso. Os atributos são inseridos na *hashtable* sempre na forma de pares de informações, no caso, nome_atributo e valor, onde nome_atributo é a chave através da qual os valores são recuperados. Na Figura 8.6 é possível observar a *hashtable* inserida na classe TipoSensor.

```
public class TipoSensor
{
  protected String name;
  protected Hashtable propertyTypes = new Hashtable();
  ...

  //--------Atribui o nome do tipo de sensor-------------------
  public void setName(String nome)
  {
    name = nome;
  }

  //--------Recupera nome do tipo de Sensor--------------------
  public String getName()
  {
    return name;
  }



  //-------- Recupera valor hashtable--------------
```

```
    public TipoPropriedade getPropertyType (String name)
    {
      return  (TipoPropriedade) propertyTypes.get(name);
    }

    //----------- Preenche hashtable ----------------------
    public void setPropertyType( String name, Object value)
    {
      propertyTypes.put(name,value);
    }
    ...

}
```

Figura 8.6 - *Hashtable* na classe TipoPropriedade

O método setPropertyType é responsável por preencher a *hashtable* com as informações name e value. Observa-se que o valor inserido na *hashtable* deve ser do tipo *Object* porque estes valores são tipos de propriedades, ou seja, são objetos da classe TipoPropriedade. Isto também pode ser observado através do tipo de retorno TipoPropriedade do método getPropertyType. O método getPropertyType é responsável por recuperar os valores da *hashtable* propertyTypes através da chave name.

Desta forma, os tipos de sensores podem possuir vários atributos diferentes, como ilustra a Figura 8.7.

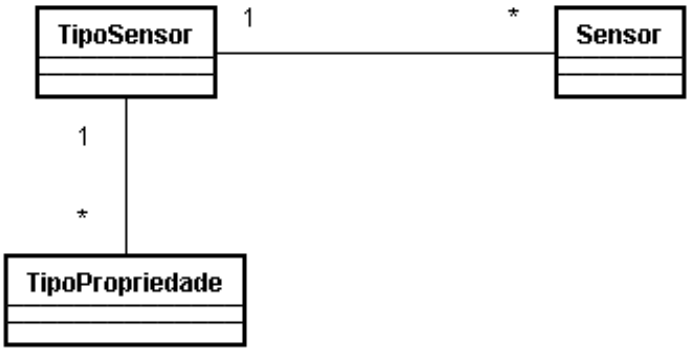


Figura 8.7 - Padrão *Property* para propriedades da classe TipoSensor

Da mesma forma que a classe TipoSensor, a classe Sensor também tem variações de propriedades. Assim, a classe Sensor também deve ser acrescida com uma

hashtable para guardar as propriedades e métodos que manipulem estas propriedades.

O código, na Figura 8.8, mostra a inclusão da *hashtable* na classe Sensor, assim como seus métodos de acesso.

```
public class Sensor
{
  protected String name;
  protected TipoSensor type;
  protected Hashtable properties = new Hashtable();

  ...

  //-----------------Atribui o tipo de sensor-----------------
  public void setType(TipoSensor Type)
  {
      type = Type ;
  }

  //-----------------Retorna tipo de sensor--------------------
  public TipoSensor getType()
  {
    return type;
  }

  //----------Atribui valores à lista de propriedade----------
  protected void setProperty (String name, Object value)
  {
    properties.put(name,value);
  }

  ...

}
```

Figura 8.8 - *Hashtable* na classe Sensor

A Figura 8.9 ilustra a estrutura resultante da aplicação dos padrões *TypeObject* e *Property* no protótipo apresentado.

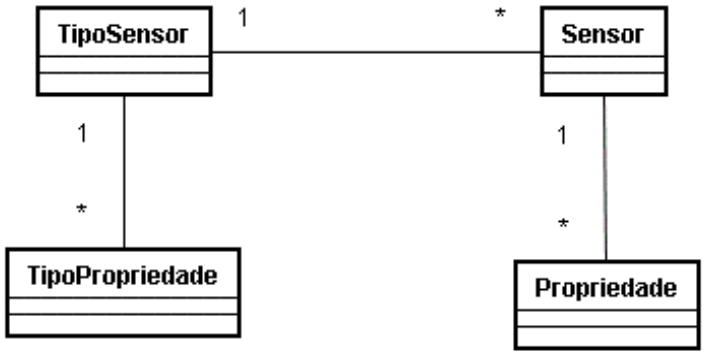


Figura 8.9 - *TypeObject* e *Property* nas classes Sensor e TipoSensor

8.2.3 Implementação do Padrão *TypeSquare*

O padrão *TypeSquare*, como explicado na Seção 5.2.3, é a utilização do *TypeObject* e *Property* em conjunto, sendo o *TypeObject* aplicado duas vezes, uma antes da aplicação do *Property*, e outra depois.

A implementação do padrão *TypeSquare* foi iniciada nas seções 8.2.1 e 8.2.2, com a aplicação do padrão *TypeObject* e do padrão *Property*, respectivamente. Então, para finalizar a implementação do *TypeSquare,* é feita a segunda aplicação do *TypeObject,* que interliga as classes TipoPropriedade e Propriedade, como é mostrado na Figura 8.10. e pelo código apresentado, na Figura 8.11.

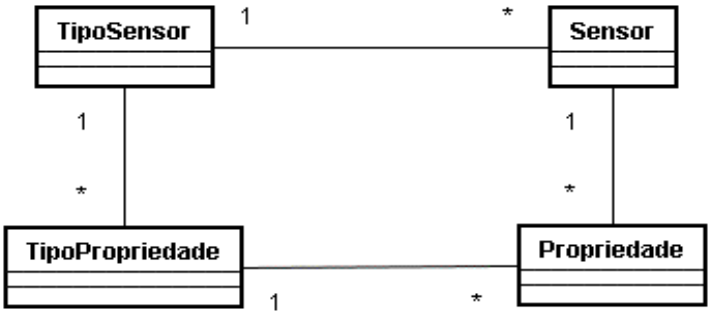


Figura 8.10 - *TypeSquare* nas classes Sensor e TipoSensor

```
void btnSensor_actionPerformed(ActionEvent e)
{
 try
 {

   Sensor s = new Sensor();
   TipoSensor velocidade = new TipoSensor ();
   TipoPropriedade valor_sensorVel = new TipoPropriedade ();
   TipoPropriedade valor_sensorVel_ret = new TipoPropriedade ();


   //---------- Preenche a hashtable de TipoPropriedade ----------
   //--- da classe TipoSensor com objeto TipoPropriedade----------

   s.type.setPropertyType(valor_sensorVel.getName(),
                                                  valor_sensorVel);

   ...

   //---------- Recupera da hashtable de TipoPropriedade ---------
   //--- da classe TipoSensor o objeto TipoPropriedade----------


   valor_sensorVel_ret=
               s.type.getPropertyType(valor_sensorVel.getName());


   //---recupera o nome do tipo da Propriedade--------------------
   //----------- armazenada na hashtable de TipoSensor ----------

   String nome_propriedade = valor_sensorVel_ret.getName();


   //-preenche a hashtable de Propriedades da classe Sensor-------
   //-----com o nome do tipo de propriedade adquirido acima-------

   s.setProperty(nome_propriedade, new Double (5.0));


  //--------------recupera a propriedade armazenada---------------
   s.getProperty(nome_propriedade);


 }

 catch (Exception erro)
 {
   System.out.println("Erro btnSensor_actionPerformed: " +
                                                 erro.getMessage());
 }

}
```

Figura 8.11 - Conclusão do *Padrão TypeSquare*

No código da Figura 8.11, observa-se que o conteúdo da string nome_propriedade, inserida como propriedade para o objeto Sensor, é retornado diretamente a partir da lista de tipos de propriedades do objeto TipoSensor através do método getPropertyType. Em outras palavras, o atributo nome_propriedade, inserido como propriedade da classe Sensor, recebe o nome do objeto valor_sensorVel, que é um tipo de propriedade da classe TipoSensor.

Desta forma, estabelece-se uma conexão entre estas propriedades. Por exemplo, para o sensor A do tipo Velocidade, o tipo de propriedade valor_sensorVel terá o valor de 5.0.

8.3 Implementação do Mecanismo de Reflexão

A aplicação do mecanismo de reflexão possibilita a criação de objetos e a chamada de seus métodos de forma dinâmica, em tempo de execução, o que pode melhorar a flexibilidade e a reusabilidade oferecidas pelos modelos de objetos adaptáveis (Seção 5.1).

Para a construção do protótipo, os mecanismos de reflexão foram utilizados para:

- obter a classe desejada a partir do repositório;
- obter os métodos disponíveis da classe obtida;
- acionar os métodos da classe obtida, sendo necessário para isso:
    - encontrar o método desejado;
    - preparar o parâmetro que deve ser passado na chamada ao método;
    - obter uma nova instância da classe carregada;
    - chamar o método desejado.

O primeiro passo é obter um objeto *java.lang.Class* para a classe desejada. Um caminho de se obter um objeto Class é mostrado no código apresentado na Figura 8.12.

```
void btnAdTipoSensor_actionPerformed(ActionEvent e)
{
   Class Sensor;
   String nome_classe = "vls_aom.Sensor";
   try
   {

    //instancia object sensor
     Sensor = Class.forName(nome_classe);
   ...
   }
}
```

Figura 8.12 - Obtenção do objeto *Class*

O método forName retorna o objeto do tipo *Class*, associado a uma classe ou interface de acordo com o nome passado através do parâmetro, carregando a classe especificada. No caso do código apresentado na Figura 8.12, deseja-se obter a classe Sensor pertencente ao pacote vls.aom. Nesse pacote estão agrupadas todas as classes relacionadas com a geração de variantes do software do LSB.

Observa-se que o parâmetro é uma variável do tipo *String*, que contém o nome da classe que se deseja obter. Desta forma é possível obter qualquer classe do repositório de uma forma genérica.

Pode-se ainda, recuperar, a partir do repositório, os métodos disponíveis de cada classe obtida, como mostra o código na Figura 8.13.

```
void btnAdTipoSensor_actionPerformed(ActionEvent e)
{
   Class Sensor;
   String nome_classe = "vls_aom.Sensor";
   try
   {

    //instancia object tipo_sensor
     Sensor = Class.forName(nome_classe);


    //lista todos os métodos declarados pela classe
    Method m[] = sensor[SelectedIndex()].getDeclaredMethods();
    for (int i = 0; i < m.length; i++ )
            System.out.println(m[i].toString());

   ...
   }
}
```

Figura 8.13 - Obtenção dos métodos da classe

O método getDeclaredMethods lista todos os métodos declarados na classe carregada.

Existe também uma série de métodos do pacote *java.lang.reflect* que obtêm informações sobre as classes e métodos manipulados, como por exemplo, os métodos getParameterTypes, getExceptionTypes, getReturnType, que retornam respectivamente o tipo de parâmetro, erro de exceção e tipo de retorno do método manipulado. Muitos exemplos que utilizam estes métodos foram implementados e testados como objeto de estudo e de entendimento dos mecanismos reflexivos apresentados pela linguagem Java, durante a construção do protótipo.

Depois de obter a classe Sensor e os seus métodos, estes métodos podem ser acionados.

O código da Figura 8.14 mostra como é realizada a chamada dinâmica de um método.

```
void CriaSensores()
 {
      //---Instancia Sensores de acordo com a lista de sensores
      String nome_classe = "vls_aom.Sensor";
      Method meth;
      String arglist[] = new String[1];
      Object objectSensor[] = new Object[10];
      Object retobj;

      try
      {
          //para manipular a classe Sensor
          sensor[SelectedIndex()]=
                                  Class.forName(nome_classe);

          //encontra o método setName
          meth= sensor[SelectedIndex()].
              getMethod("setName", new Class[] {String.class});

          // valor do parâmetro do método setName
          arglist[0]= this.lstSensores.getSelectedValue().
                                                  toString();

          //obtendo a instância do objeto Sensor
          objectSensor[SelectedIndex()]=
                     sensor[getSelectedIndex()].newInstance();

          //chamando o método setName
          retobj=    meth.invoke(objectSensor[SelectedIndex()],
arglist);

      }
      catch (Exception erro)
      {
     System.out.println("Erro    ao    criar    sensor:    "    +
erro.getMessage());
      }
 }
```

Figura 8.14 - Chamada dinâmica de método

O método getMethod é utilizado para encontrar o método desejado através de seu nome. Para isso, devem ser passados como parâmetro do método getMethod, o nome e o tipo dos parâmetros do método desejado. No código da Figura 8.14, nota-se que o nome do método desejado é setName, método da classe Sensor e que possui um parâmetro do tipo *String*.

Encontrado o método desejado, o valor do parâmetro, utilizado na chamada do método, é preparado. O *array* de string arglist[] contém o conteúdo que é passado como parâmetro, que no caso, é o valor selecionado em uma lista this.lstSensores.getSelectedValue.

A classe Sensor, é instanciada, chamando-se seu construtor e, para isso, utiliza-se o método newInstance. A nova instância é manipulada através o objeto denominado objectSensor.

Finalmente, o método invoke chama dinamicamente o método setName, para isso necessita da instância da classe Sensor, ou seja, o objeto objectSensor e dos valores a serem passados como parâmetro ao método através do *array* arglist[].

Na Figura 8.15 mostra-se a parte da implementação do padrão de modelos de objetos adaptáveis *TypeObject* utilizando a reflexão. O método setType da classe Sensor é chamado, passando como parâmetro um objeto do tipo TipoSensor.

```
void btnsensores_actionPerformed(ActionEvent e)
  {
    //---------------------TypeObject--------------------------
    TipoSensor arglist1[] = new TipoSensor[1];
    Class sensor[] = new Class[10];
    Object objectTipoSensor[] = new Object[10];
    int i = 0;
    try
    {
       …

      //chama setType -> TypeObject = completa tipo sensor com
      //TipoSensor
        Method meth = sensor[this.lstSensores.
        getSelectedIndex()].getMethod
               ("setType", new Class[] {TipoSensor.class});

       //método seType com um parametro do tipo de sensor
        arglist1[0]=(TipoSensor)
        objectTipoSensor[this.lstTiposSensores.
                                    getSelectedIndex()];
     // o parametro é o objeto tipo_de_sensor
         Object retobj= meth.invoke(objectSensor[this.
                lstSensores. getSelectedIndex()],arglist1);
 …
```

Figura 8.15 - *TypeObject* com reflexão

Variações entre número de parâmetros, tipos, valores de retorno podem ocorrer, como mostra o código do método RecuperarValor na Figura 8.16.

```
void RecuperarValor()
{
   try
   {
     //Instancia Propriedade
     propriedade = Class.forName("vls_aom.Propriedade");

     //tipos de argumentos
     Class[] argumentTypes = {Connection.class, String.class,
                                                 String.class};
     Method meth =
     propriedade.getMethod("getValorPropriedade",argumentTypes);

     objectPropriedade = propriedade.newInstance();


     Object[] arguments = {conex,
          this.lstTipoPropriedade.getSelectedValue().toString(),
                                                   NomeSensor};

     Object retobj = meth.invoke(objectPropriedade, arguments);

     txtValorPropriedade.setText(retobj.toString());
   }
   catch (Exception erro)
   {
     System.out.println("Ocorreu um erro ao recuperar a
                              Propriedade " +  erro.getMessage());
   }

}
```

Figura 8.16 - Método RecuperarValor

Como é necessário passar diferentes tipos como argumentos do método getMethod, deve-se criar um *array* do tipo Class[] argumentTypes e preenchê-lo com os 3 valores, um do tipo Connection (Connection.class) e dois do tipo String (String.class). O *array* do tipo Object[] arguments é completado com os valores dos tipos pertencentes ao argumentTypes: conex, this.lstTipoPropriedade.getSelectedValue().toString() e NomeSensor; os quais são passados como parâmetros do método invoke.

Um caminho para se chamar dinamicamente qualquer método de forma genérica pode ser observado no código apresentado na Figura 8.17.

```
public Object ChamaMetodo(String nomeMetodo, Class partypes[],
Object arglist[], Object objectSensor)
 {
   Object retobj = new Object();
   try
   {
       //instancia objeto criado
       Class  sensor = Class.forName("vls_aom.Sensor");

       //chama o metodo
       Method meth = sensor.getMethod(nomeMetodo, partypes);
      retobj = meth.invoke(objectSensor, arglist);
   }

   catch(Exception erro)
   {
           System.out.println("Erro ao chamar Acionar metodo:"
                              + nomeMetodo + erro.getMessage());
   }

   return retobj;
 }
```

Figura 8.17 – Chamada de métodos dinamicamente

O padrão *Strategy* também utilizado nos modelos de objetos adaptáveis, explicado na Seção 5.2.4, foi implementado com mecanismos reflexivos para a classe Sensor e utilizou o método ChamaMetodo, mostrado na Figura 8.17. O padrão *Strategy* é mostrado na Figura 8.18.

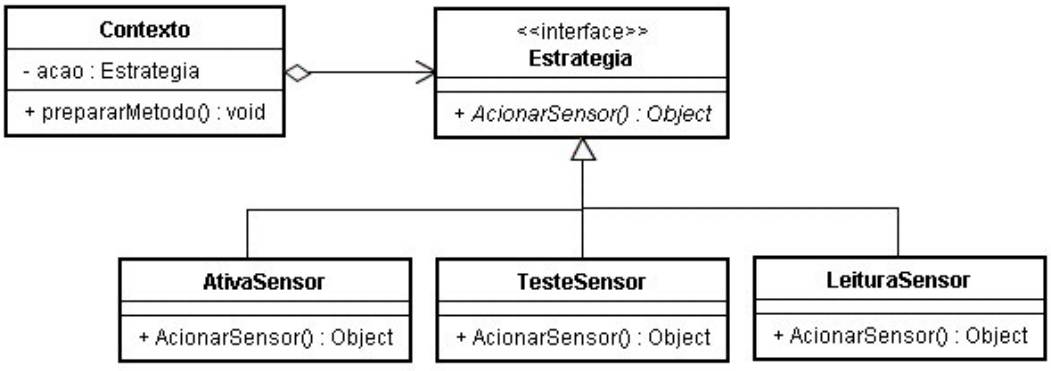


Figura 8.18 - Padrão *Strategy*

A interface Estratégia possui o método AcionarSensor, como mostrado na Figura 8.19, que é implementado em todas as classes estratégias concretas.

```
public interface Estrategia
{
   public Object AcionarSensor(String nomeMetodo,
                String nomeSensor, Object objectSensor);
}
```

Figura 8.19 - Interface Estratégia

A três estratégias concretas do padrão *Strategy* são as classes AtivaSensor, TesteSensor e LeituraSensor. Estas três classes possuem o método AcionarSensor que chama o método ChamaMetodo passando os parâmetros de acordo com ação que deverá ser executada.

A Estratégia Concreta AtivaSensor é mostrada na Figura 8.20.

```
public class AtivaSensor implements Estrategia
{
  public Object AcionarSensor(String nomeMetodo,
                      String nomeSensor, Object objectSensor)
   {
      Sensor s = new Sensor();

      Class[] partypes = {String.class};
      Object arglist[] = {nomeSensor};

      //chamada dinamica na classe sensor
      return (s.ChamaMetodo(nomeMetodo, partypes,
                                    arglist, objectSensor));

   }
}
```

Figura 8.20 - Estratégia Concreta AtivaSensor

A Estratégia Concreta TesteSensor é mostrada na Figura 8.21.

```
public class TesteSensor implements Estrategia
{
   public Object AcionarSensor(String nomeMetodo,
                       String nomeSensor, Object objectSensor)
   {
      Sensor s = new Sensor();

      Class[] partypes = {String.class, double.class};
      Object arglist[] = {nomeSensor, new Double(0.0000)};

      //chamada dinamica na classe sensor
      return (s.ChamaMetodo(nomeMetodo, partypes,
                                      arglist, objectSensor));
   }
}
```

Figura 8.21 - Estratégia Concreta TesteSensor

A estratégia Concreta LeituraSensor é mostrada na Figura 8.22.

```
public class LeituraSensor implements Estrategia
 {

   public Object AcionarSensor(String nomeMetodo,
                       String nomeSensor, Object objectSensor)
    {
       Sensor s = new Sensor();

       Class[] partypes = {String.class};
       Object arglist[] = {nomeSensor};

       //chamada dinamica na classe sensor
       return (s.ChamaMetodo(nomeMetodo, partypes,
                                      arglist, objectSensor));
    }

 }
```

Figura 8.22 - Estratégia Concreta LeituraSensor

A classe Contexto, que repassa a requisição do cliente para a Estratégia, possui o código mostrado na Figura 8.23.

```
void prepararMetodo(String nomeMetodo)
{
   // dependendo de como acao sera inicializada
   // chamará o metodo correto
   // AcionaSensor para leitura, teste ou ativacao.

   String nomeClasse = new String();
   Estrategia acao = null; //da interface Estrategia
   Object retobj = new Object();

   try
     {
         if (nomeMetodo.equals("LerSensor"))          {
              acao = new LeituraSensor();
              retobj = acao.AcionarSensor(nomeMetodo,
                                  nomeSensor, objectSensor);
         }
         else
         {
              if (nomeMetodo.equals("TestarSensor"))
              {
                  acao = new TesteSensor();
                  retobj = acao.AcionarSensor(nomeMetodo,
                                       nomeSensor,objectSensor);

              }
              else
              {
                  if (nomeMetodo.equals("AtivarSensor"))
                  {
                       acao = new AtivaSensor();
                       retobj = acao.AcionarSensor(nomeMetodo,
                                   nomeSensor, objectSensor);
                  }
               }
         }
     }
```

Figura 8.23 - Classe Contexto

Observa-se, na Figura 8.23, que o objeto acao do tipo interface Estratégia, pode ser iniciado como objeto de qualquer estratégia concreta LeituraSensor, TesteSensor ou AtivaSensor. Desta forma, dependendo da ação a ser realizada, aciona corretamente o método AcionarSensor da estratégia LeituraSensor, ou da estratégia TesteSensor ou ainda, da estratégia AtivaSensor.

8.4 Descrição dos Sistemas Gerenciador de Repositório e Gerador de Aplicação

Esta seção apresenta o sistema Gerenciador de Repositório e o sistema Gerador de Aplicação do protótipo do mecanismo de variabilidade do processo GVLPS.

O sistema Gerenciador de Repositório é responsável pelo cadastro e manipulação, no repositório do mecanismo de variabilidade, das variantes representadas por objetos correspondentes, de seus tipos, propriedades e métodos. Desta forma, este sistema configura todas as informações relacionadas aos pontos de variação e variantes do sistema e armazena-as no repositório.

O sistema Gerador de Aplicação permite ao usuário selecionar as variantes do repositório, que foram cadastradas através do sistema Gerenciador de Repositório, para que sejam instanciadas para fazer parte da aplicação. A partir deste sistema também é possível definir suas propriedades e acionar suas funcionalidades.

A Figura 8.24 o relacionamento entre os sistemas Gerenciador de Repositório e Gerador de Aplicação e o repositório.

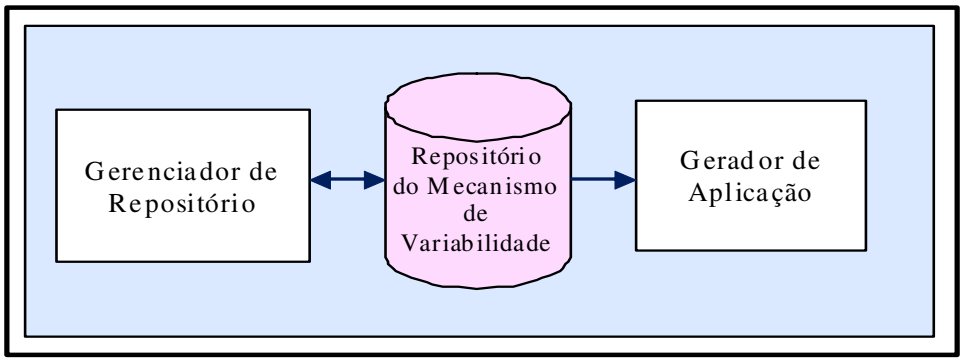


Figura 8.24 - Relacionamento entre os sistemas Gerenciador de Repositório e Gerador de Aplicação e o repositório do mecanismo de variabilidade

8.4.1 Sistema Gerenciador de Repositório

A Figura 8.25 mostra o diagrama de estados que representa como é feita a navegação entre as telas do sistema Gerenciador de Repositório. Cada estado está representado por uma tela do sistema e os eventos são os acionamentos de comandos.

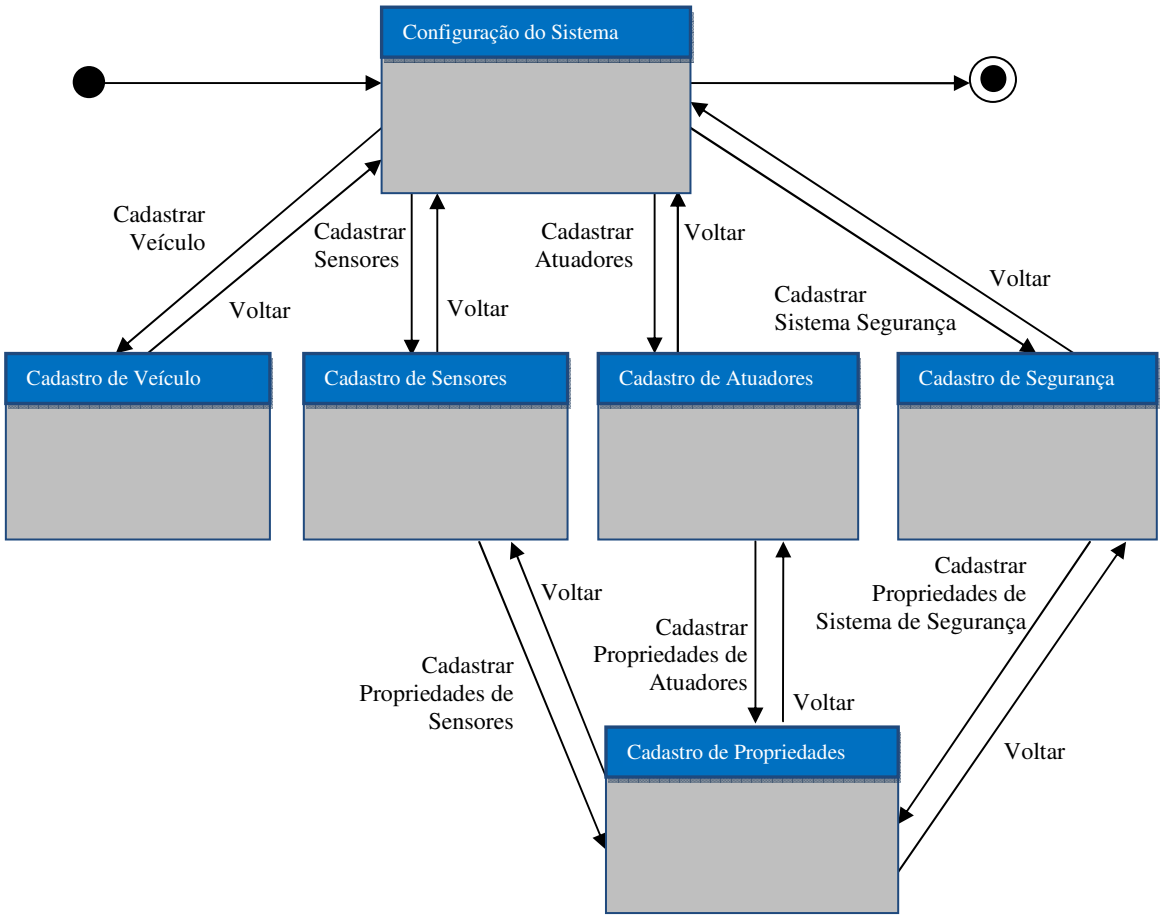


Figura 8.25 - Navegação de telas do sistema de Gerenciador de Repositório

Ao ser iniciado, o sistema de Gerenciador de Repositório mostra a tela principal Configuração do Sistema, apresentada na Figura 8.26.

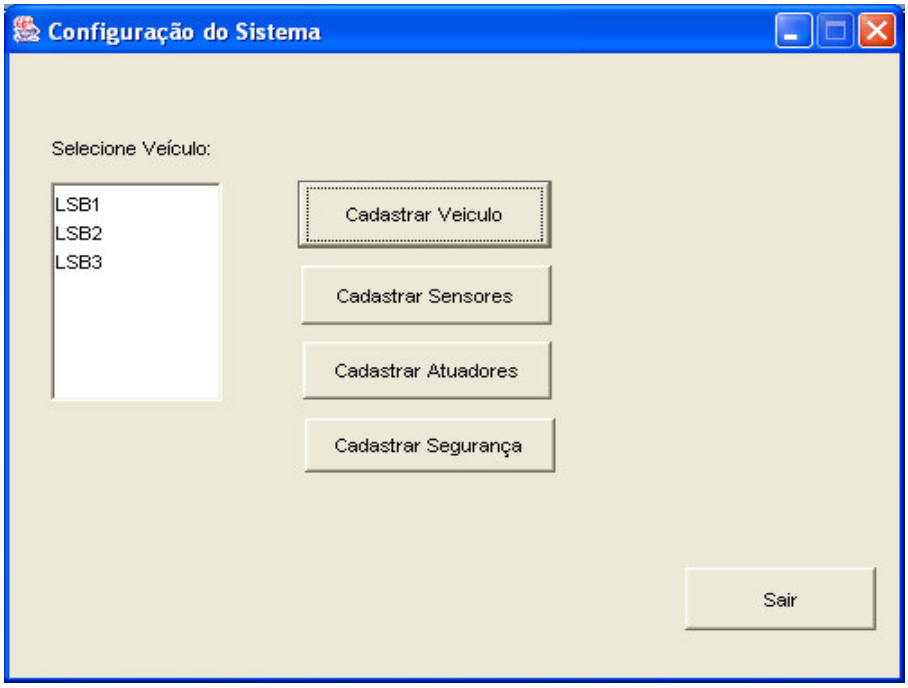


Figura 8.26 - Tela de Configuração do Sistema

A partir desta tela pode-se selecionar um veículo cadastrado ou cadastrar novos objetos referentes a veículos, sensores, atuadores ou sistema de segurança.

A Figura 8.27 apresenta a tela através da qual se pode cadastrar um novo veículo ou remover um veículo existente. Após informar o nome e o código do veículo a ser cadastrado ou removido no repositório, deve-se selecionar o botão Cadastrar ou Remover. Na Figura 8.27 observa-se a mensagem indicando sucesso para o cadastro do veículo LSB4.

A Figura 8.28, exibe a tela a partir da qual sensores e tipos de sensores podem ser cadastrados ou removidos de um veículo cadastrado. Através desta tela também é feita a associação do tipo de sensor para cada sensor. Este processo é idêntico para os elementos atuadores e sistema de segurança.

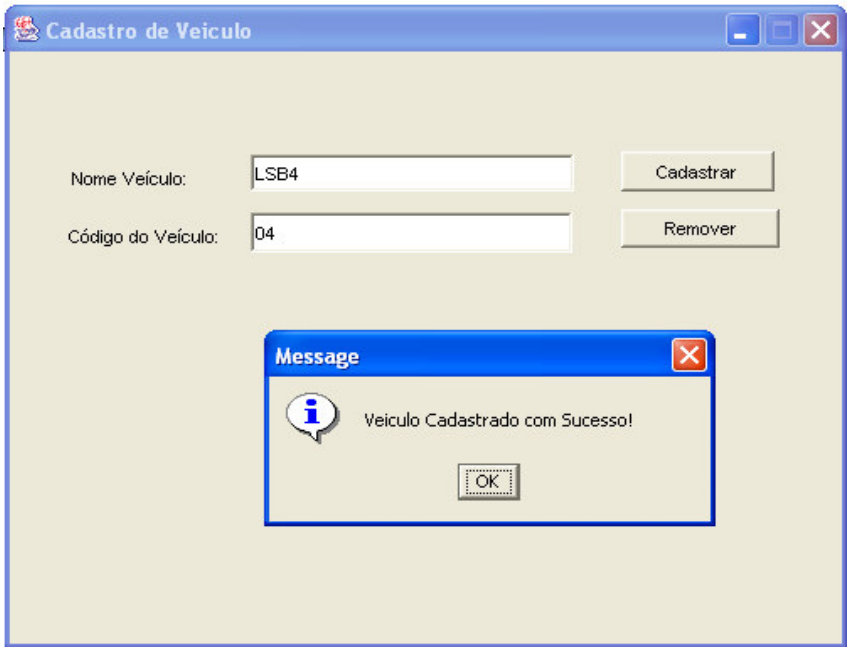


Figura 8.27 - Tela Cadastro de Veículo

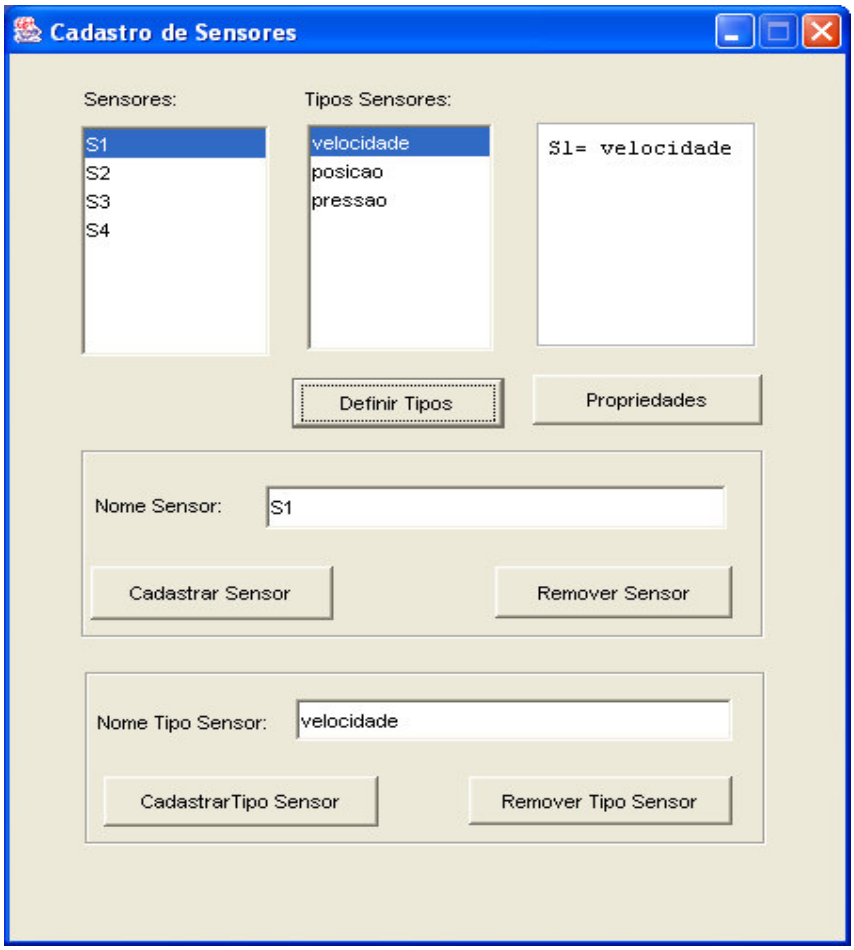


Figura 8.28 - Tela Cadastro de Sensores e Tipos de Sensores

A partir do acionamento do botão Propriedades, a tela Cadastro de Propriedade é exibida, como mostra a Figura 8.29. Através desta tela os tipos de propriedades podem ser cadastrados ou removidos, assim como, pode ser realizada a atribuição de valores para as propriedades existentes.

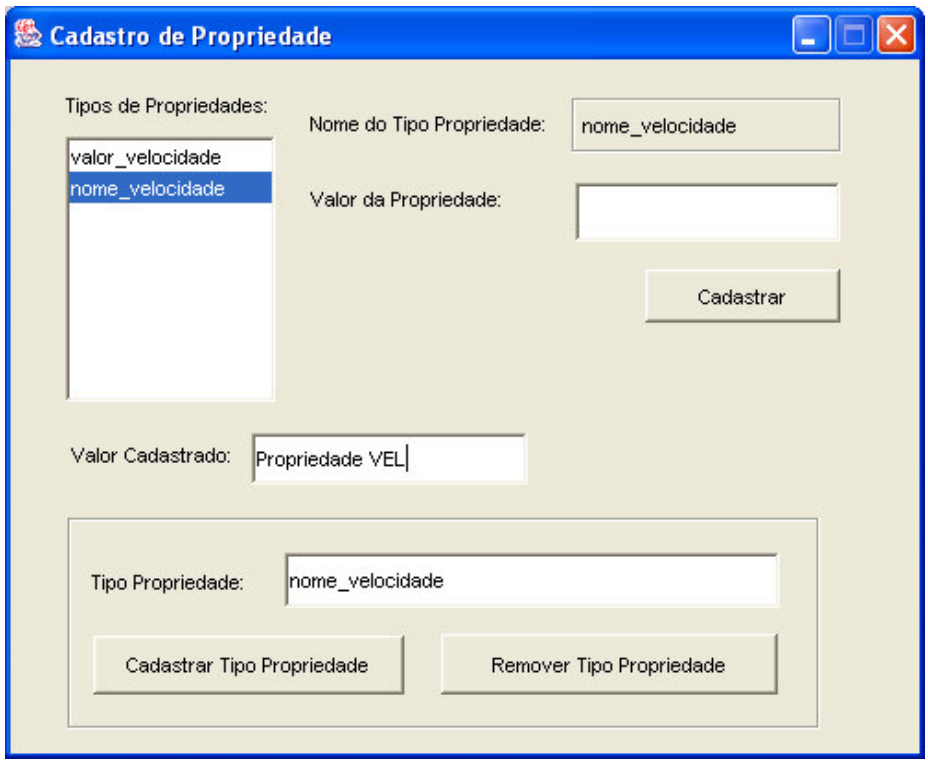


Figura 8.29 - Tela Propriedades

Repetindo-se estas operações, através do sistema Gerenciador de Repositório, para todos os elementos do veículo é possível configurar um veículo. Ainda, selecionando-se os elementos constituintes (sensores, atuadores e sistemas de segurança), pode-se fazer a manutenção dos componentes da linha de produtos de software do LSB.

Exemplos das tabelas geradas no repositório podem ser observados nas Figuras 8.30 a 8.32, que mostram respectivamente as tabelas Sensor, Sensor_TipoSensor e Propriedade.

A tabela da Figura 8.30 mostra os objetos sensores relacionados aos objetos veículos.

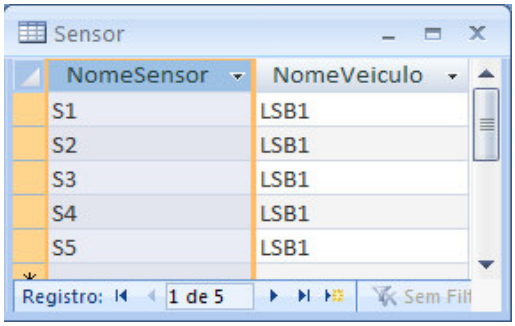


Figura 8.30 - Tabela Sensor

Na tabela da Figura 8.31 pode-se observar os objetos sensores relacionados aos objetos tipos de sensores.

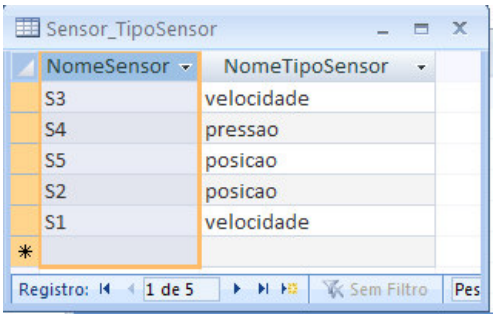


Figura 8.31 - Tabela Sensor_TipoSensor

A tabela da Figura 8.32 mostra os objetos sensores relacionados aos objetos tipos de propriedades e seus valores de propriedades respectivamente.

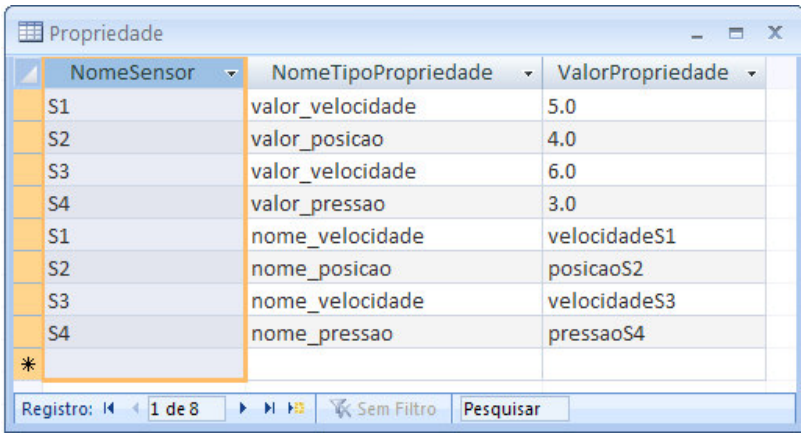


Figura 8.32 - Tabela Propriedade

8.4.2 Sistema Gerador de Aplicação

A Figura 8.33 mostra o diagrama de estados que representada como é feita a navegação entre as telas do sistema Gerador de Aplicação.

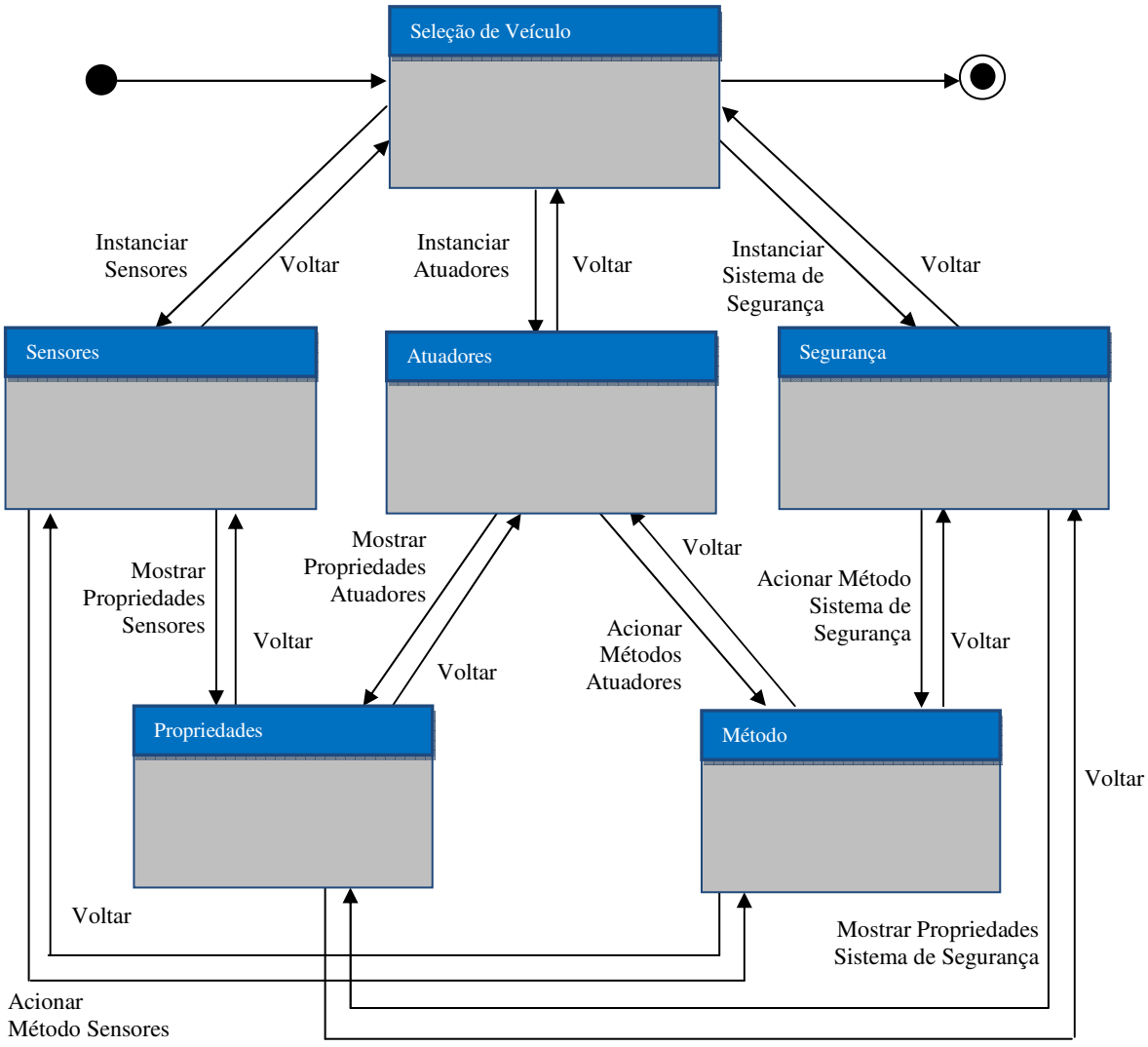


Figura 8.33 - Navegação de telas do sistema Gerador de Aplicação

Ao ser iniciado, o sistema Gerador de Aplicação mostra a tela principal Seleção de Veículo, apresentada na Figura 8.34.

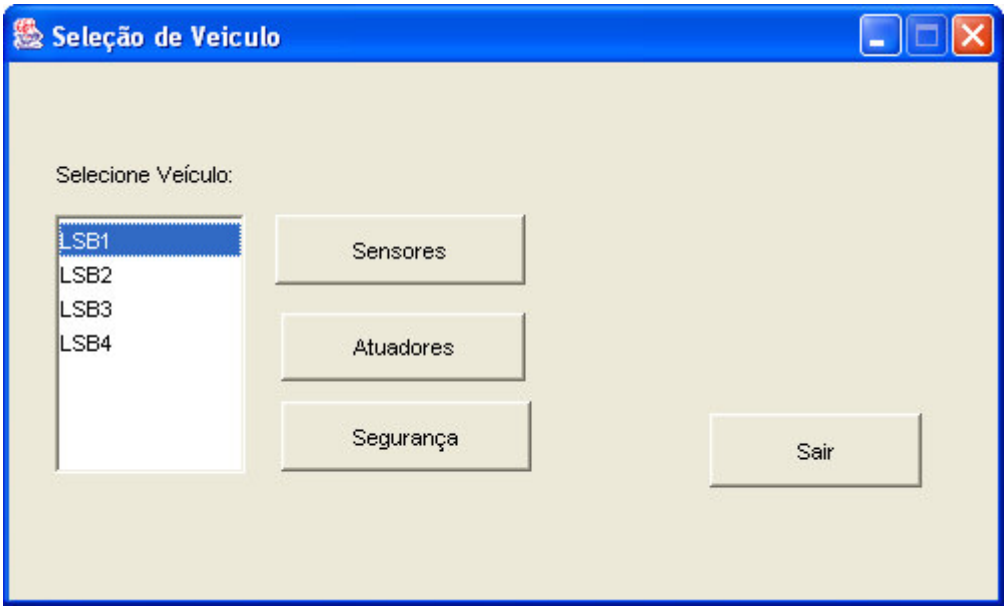


Figura 8.34 - Tela Seleção de Veículo

Após a seleção do veículo, o sistema disponibiliza os objetos sensores, atuadores e sistema de segurança cadastrados para o veículo selecionado, de acordo com a configuração previamente realizada através do sistema Gerenciador de Repositório. Pode-se, então selecionar os objetos correspondentes a sensores, atuadores e sistema de segurança que se deseja instanciar. A Figura 8.35 mostra a tela de opções para a criação de sensores.

Figura 8.35 - Tela Sensores

Selecionando-se o botão Instanciar Sensores, objetos sensores do tipo definidos no repositório são instanciados dinamicamente.

Observa-se a implementação da criação dos objetos sensores e tipos de sensores com os códigos a seguir na Figura 8.36 e 8.37.

```
void CriaSensores()
 {
   ...
   Class sensor[] = new Class[20];

   sensor[this.lstSensores.getSelectedIndex()] =
                                    Class.forName(nome_classe);
   ...
 }
```

Figura 8.36 - Criação de objetos Sensores

```
void CriaTipoSensores()
 {
    ...
    Class tipo_sensor[] = new Class[20];

      tipo_sensor[this.lstSensores.getSelectedIndex()] =
                              Class.forName("vls_aom.TipoSensor
                           ");
   ...
}
```

Figura 8.37 - Criação de objetos TipoSensores

O método setType é chamado dinamicamente para definir o tipo de sensor, fazendo parte da implementação do padrão *TypeObject* explicada na Seção 8.2.1. A Figura 8.38 mostra o código para se definir os tipos de sensores.

```
void DefineTipoSensor()
{

   Object objectTipoSensor[] = new Object[20];
   TipoSensor arglist1[] = new TipoSensor[1];
   ...

   try
   {
     Method meth = sensor[this.lstSensores.getSelectedIndex()]
      .getMethod("setType", new Class[] {TipoSensor.class});

     arglist1[0] = (TipoSensor) objectTipoSensor
                         [this.lstSensores.getSelectedIndex()];

     Object retobj = meth.invoke(objectSensor
            [this.lstSensores.getSelectedIndex()], arglist1);
    ...
   }
```

Figura 8.38 - Definição dos tipos de Sensores

Portanto, é possível selecionar qual sensor se deseja instanciar e de qual tipo ele será. As opções de sensores e tipos de sensores são disponibilizadas através do repositório.

A partir da Tela Sensores, mostrada na Figura 8.35 também é possível ter acesso às propriedades e aos métodos dos sensores instanciados.

A Figura 8.39 mostra as propriedades definidas no repositório para o objeto sensor instanciado e que também são criadas dinamicamente, através do padrão *Property,* explicado na Seção 8.2.2, e de acordo com o tipo de sensor instanciado.

O código da Figura 8.40 mostra como é realizada a criação dinâmica dos tipos de propriedades e a chamada do método setPropertyType para preenchimento do Tipo de Propriedades, utilizando o padrão *TypeSquare* mostrado na Seção 8.2.3.

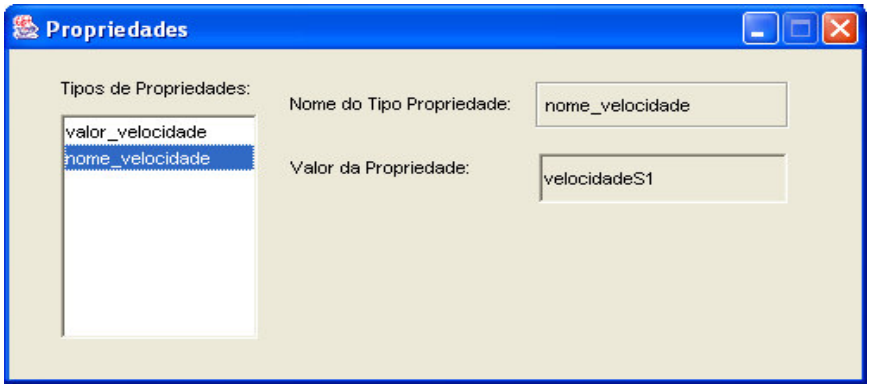


Figura 8.39 - Tela Propriedades

```
void CriarTipoPropriedade()
{
  String arglist[] = new String[1];
  String nome_TipoPropriedade[] = new
              String[lstTipoPropriedade.getModel().getSize()];
  try
  {
   //preenche vetor nome_TipoPropriedade a partir da lista
   //de tipos de propriedades
   for (int i=0; i < lstTipoPropriedade.getModel().getSize();
                                                        i++)
   {
     nome_TipoPropriedade[i] =   this.lstTipoPropriedade.
                       getModel().getElementAt(i).toString();
   }

   for (int i = 0; i < this.lstTipoPropriedade.getModel().
                                                  getSize(); i++)
   {

   //Instancia TipoPropriedade
    tipo_propriedade[i] = Class.forName
                                    ("vls_aom.TipoPropriedade");
    ...

    //Tipos de Propriedades para o tipo de sensor
    //escolhido (relaciona tipoSensor com tipoPropriedade)

    Class[] argumentTypes = {String.class, Object.class};
    meth = TipoSensor.getMethod("setPropertyType",
                                          argumentTypes );

    Object[] arguments = {nome_TipoPropriedade[i],
                                       objectTipoPropriedade[i]};
    retobj = meth.invoke(objectTipoSensor, arguments);
    ...
}
```

Figura 8.40 - Criação e preenchimento de tipos de propriedades

A Figura 8.41 mostra os possíveis métodos que podem ser acionados pelos objetos sensores instanciados e o acionamento dos mesmos. A implementação do padrão *Strategy* apresentada na seção anterior 8.3 foi aplicada na implementação da classe Metodo para a manipulação dos métodos selecionados para acionamento.

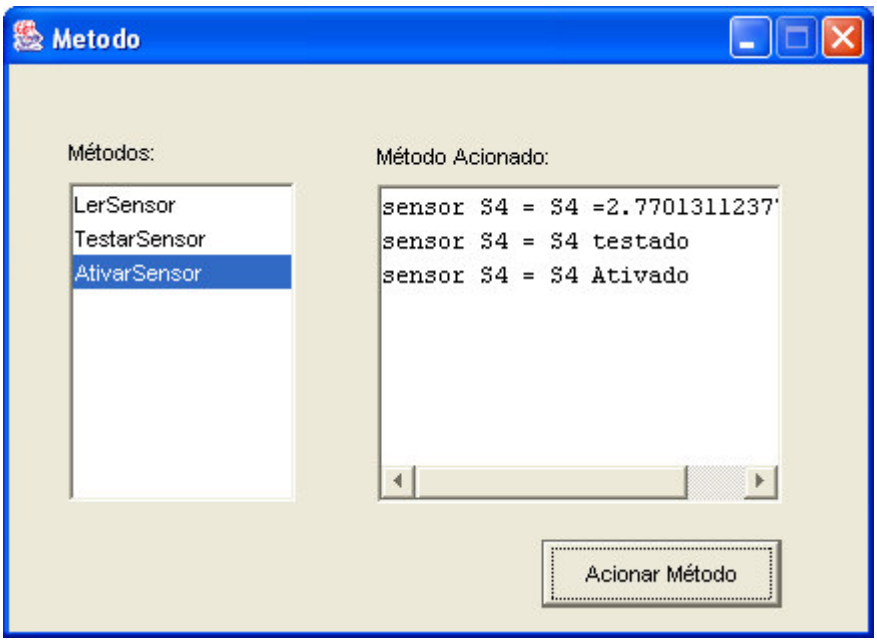


Figura 8.41 - Tela Método

8.5 Considerações Finais

Este capítulo apresentou a implementação das técnicas empregadas no desenvolvimento de um mecanismo de variabilidade proposto para o processo GVLPS, baseados em modelos de objetos adaptáveis e mecanismos de reflexão, aplicado ao veículo LSB.

A implementação dos modelos de objetos adaptáveis e dos mecanismos de reflexão são considerados trabalhosos e difíceis. Por isso, foi desenvolvido um protótipo para apresentar detalhes destas implementações, permitindo maior compreensão destas técnicas empregadas

Através deste protótipo foi possível mostrar:

- seleção e criação dinâmica dos objetos sensores do software LSB;

- criação e definição de forma dinâmica do tipo de sensor para o sensor criado, através do padrão *TypeObject*;

- criação e associação dinâmica das propriedades e tipos de propriedades para os objetos criados, através do padrão *Property* e *TypeSquare*;

- seleção e acionamento dinâmico dos métodos dos sensores criados, através do padrão *Strategy*.

Os padrões de modelos de objetos adaptáveis implementados com os mecanismos de reflexão fornecem maior flexibilidade na escolha da criação de objetos porque utilizam chamadas genéricas. Por outro lado, devido às características da arquitetura dos modelos de objetos adaptáveis e da linguagem reflexiva, estas técnicas apresentam um baixo desempenho. Entretanto, levando-se em consideração a crescente evolução da tecnologia, este fator não deve ser tido como um ponto desfavorável à sua utilização.

No caso do protótipo apresentado, apenas por questões de facilidades aos testes de execução, foram criadas interfaces para que usuários pudessem manipular e operar o veículo. No caso real, como o software do veículo é embarcado, a configuração do veículo a partir dos objetos específicos de sensores, atuadores e sistema de segurança deve ser realizada previamente. A criação de objetos e acionamentos de métodos seriam realizados em tempo de execução, sendo disparados obedecendo ao tempo estabelecido no projeto, sem a utilização de interfaces de usuário.

"Jamais um pessimista
descobriu os segredos das estrelas,
navegou por mares ignotos
ou abriu novos paraísos para o espírito."

Helen Keller

9 CONSIDERAÇÕES FINAIS

Este capítulo apresenta as contribuições do trabalho, incluindo os artigos publicados em conferências internacionais, as sugestões para trabalhos futuros, as conclusões e uma visão retrospectiva do desenvolvimento do trabalho.

9.1 Contribuições

A principal contribuição deste trabalho é a definição do processo GVLPS, que é constituído pelas atividades de identificação, especificação e implementação da variabilidade para linha de produtos de software. Esta definição consiste da descrição das atividades e dos seus artefatos de entrada e saída.

O processo GVLPS diferencia-se dos outros trabalhos já apresentados (VAN GURP, BOSCH, SVAHNBERG, 2001), (OLIVEIRA JUNIOR, 2005), porque define e detalha o mecanismo de variabilidade para a atividade de implementação da variabilidade. Esses trabalhos relacionam os possíveis mecanismos existentes, discutem as vantagens e as desvantagens, mas não detalham um mecanismo específico (SCHNIEDERS, 2006a), (SCHNIEDERS, 2006b), (GOMAA, WEBBER, 2004), (SVAHNBERG, BOSCH, 2000), (KEEPENCE, MANNION, 1999).

Além disso, o processo GVLP incorpora novas atividades em relação a estes processos de gerenciamento de variabilidade existentes, como a Extração de *Features* a partir de modelo de casos de uso de linha de produtos de software, a Determinação da Cardinalidade, a Representação da Variabilidade por Classes e a Aplicação do Mecanismo de Variabilidade.

O mecanismo de variabilidade detalhado neste trabalho se baseia nos modelos de objetos adaptáveis e na reflexão. Estas técnicas são citadas na literatura como uma solução alternativa e, apesar de ser recomendada por alguns autores (ANASTASOPOULOS, GACEK, 2001), (BRAGANÇA, MACHADO, 2004b), não foram encontrados, até o momento, trabalhos que as usassem na implementação de um mecanismo de variabilidade.

Desta forma, o mecanismo de variabilidade do processo GVLPS é também uma contribuição deste trabalho.

Este mecanismo de variabilidade proporciona maior flexibilidade na implementação das variantes e possibilidade para adaptar o sistema às novas necessidades que possam surgir. Assim, o mecanismo de variabilidade permite:

- acomodar possíveis alterações de requisitos, através da configuração das descrições (metadados) que modelam as regras de negócio. Isto significa que é possível continuamente alterar a configuração das descrições e adaptar o sistema para criar uma nova variante;

- criar novos tipos de objetos com diferentes propriedades em tempo de execução. Isto significa que é possível criar diferentes tipos de variantes que participam do produto, de forma dinâmica, em tempo de execução e de acordo com a seleção do usuário;

- construir aplicações que permitam aos usuários finais configurarem muitos tipos de objetos, permitindo, portanto, a derivação de novos produtos para linha de produtos de software;

- economia no desenvolvimento e no tempo da inclusão dos novos requisitos porque não existe a necessidade de programação e compilação para criar novas variantes, desde que estejam configuradas na base de dados.

Além disso, o mecanismo de variabilidade apresentado obedece à filosofia básica da linha de produtos de software, pois explora um reuso planejado e as similaridades de produtos relacionados.

O processo GVLPS foi avaliado através de sua aplicação no domínio de veículos lançadores de satélites. A partir do conhecimento do VLS, foi definido o veículo hipotético LSB que serviu de base para a execução das atividades do processo. Sobre uma parte do LSB, foram realizadas a identificação da variabilidade através de casos de uso de linha de produtos de software, a especificação da variabilidade através de *features* e a implementação por meio do mecanismo de variabilidade baseado em modelos de objetos adaptáveis e em reflexão.

O software do LSB foi selecionado para estudo de caso, pois apresenta configurações interessantes de variações para ilustrar a aplicação do processo GVLPS e do mecanismo de variabilidade. Além disso, possui a característica que exige freqüentes adaptações de modelos e artefatos para suas várias versões, por ser um projeto de longa duração.

Para a aplicação do processo GVLPS, foi implementado um protótipo do mecanismo de variabilidade, que consistiu do repositório para armazenar as variantes da linha de produtos de software do LSB, do sistema Gerenciador de Repositório e do sistema Gerador de Aplicação. Este protótipo mostrou a viabilidade de:

- configurar o repositório para a criação de novas variantes e tipos de variantes para o LSB;
- definir novos tipos e propriedades para estas variantes;
- criar novos tipos de variantes e propriedades em tempo de execução;
- criar novas configurações diferentes para outros veículos, pertencentes a mesma linha de produtos.

A aplicação do processo GVLPS ao software do LSB pode também ser considerada como uma contribuição da tese, pois foi um passo inicial da pesquisa na área de linha de produtos de software e de modelos de objetos adaptáveis para aplicação espacial.

Pode-se ponderar que a linguagem Java, adotada neste trabalho, e os seus mecanismos de reflexão não são recursos recomendados para aplicação críticas. Entretanto, com a rápida evolução da tecnologia, tanto em sistema operacionais

quanto plataformas de desenvolvimento, e a tendência crescente de utilização de técnicas flexíveis e adaptáveis em projetos espaciais, acredita-se que a abordagem apresentada possa ser considerada viável para software com características críticas em futuro próximo.

As conferências e congressos da área espacial têm confirmado, nos últimos anos, a tendência para inovação tecnológica, inclusive no que diz respeito à área de software. Em 2008, na chamada de trabalhos de uma das principais conferências internacionais na área espacial, *59th International Astronautical Congress* – IAC 2008 foram solicitadas idéias inovadoras relacionadas a sistemas adaptáveis, reconfiguráveis e modulares para veículos espaciais, que privilegiassem software de rápido projeto, desenvolvimento e integração. Muitas seções apresentaram foco em desenvolvimento tecnológico e inovações, geralmente com alto risco, mas com um potencial uso de técnicas avançadas, visando arquiteturas e software eficientes, flexíveis e confiáveis, inclusive para operações embarcadas autônomas (IAF, 2009).

Esta conferência ainda ressaltou que aspirações consideradas inviáveis há poucos anos já são realidades concretas atuais e, por isso, idéias devem ser estudadas hoje para que no futuro próximo se tornem realidade, apoiadas por tecnologia cada vez mais emergente. Nos dias atuais é preciso conceitualizar técnicas originais e inovadoras para sistemas espaciais para sólidas e potenciais aplicações no futuro.

Também neste contexto, destacam-se alguns trabalhos que foram desenvolvidos na área espacial utilizando a abordagem de linha de produtos de software (GIMENES, TRAVASSOS, 2002), (McCOMAS et al., 2000), (HABLI, KELLY, HOPKINS, 2007), modelos de objetos adaptáveis (CARDOSO et al., 2008) e reflexão (THOMÉ, 2004).

Novas tecnologias para sistemas espaciais também foram apresentadas na *10th International Conference on Space Operations* - SpaceOps 2008 (AIAA, 2008), e um trabalho com idéias sobre modelos de objetos adaptáveis e reflexão, que deram origem a esta tese, foi aceito, apresentado e publicado nos seus anais (BURGARELI, MELNIKOFF, FERREIRA, 2008a).

A *19th Australian Conference on Software Engineering* - AWSEC 2008 (CURTIN UNIVERSITY OF TECHNOLOGY, 2008), também lançou em 2008 uma sessão especial voltada para a área aeroespacial. Nesta conferência também foi aceito e

publicado um outro trabalho com idéias de reuso de software que contribuíram para a tese (BURGARELI, MELNIKOFF, FERREIRA, 2008b).

Um trabalho originado desta tese, no qual foram resumidos os principais assuntos aqui apresentados, foi selecionado pela *6th ACIS International Conference on Software Engineering, Research, Management and Application* - SERA 2008 (ACIS, 2008) e publicado como capítulo de livro pela Spring-Verlag (BURGARELI, MELNIKOFF, FERREIRA, 2008c).

Outros dois trabalhos, também originados desta tese, enfatizando o processo GVLPS, foram aceitos, um para ser publicado nos anais da conferência IEEE *Eastern European Regional Conference on the Engineering of Computer Based Systems* - ECBS-EERC 2009 (UNIVERSITY OF NOVI SAD, 2009), (BURGARELI, MELNIKOFF, FERREIRA, 2008d) e outro para ser apresentado na IEEE *Aerospace Conference* 2009 (IEEE, 2009), - conferência que também promove idéias inovadoras na área espacial.

9.2 Trabalhos Futuros

Conforme apresentado no Capítulo 2, existe o Programa Cruzeiro do Sul do Ministério da Aeronáutica, que prevê o projeto e a construção de uma família de lançadores de satélites. Dentro deste contexto, um possível trabalho futuro de porte a ser considerado é o desenvolvimento e a implantação de uma linha de produtos de software para lançadores de satélite. A estratégia traçada no programa favorece esta abordagem, pois os membros da família têm a sua complexidade aumentada a cada versão. Considerando também que o projeto de lançadores de satélite é complexo e de longa duração, a linha de produtos de software pode estabelecer uma filosofia de projeto com reuso organizado e benefícios como qualidade e economia, além de preservar a memória e o conhecimento sobre os projetos.

Neste contexto, o processo GVLPS poderá ser implantado de forma mais completa e passará a ser um recurso útil, pois auxilia e contribui em todas as fases, desde a

análise, a geração de artefatos até a implementação das variantes da linha de produtos de software.

Um outro ponto que a ser explorado no processo GVLPS é em relação à melhoria do mecanismo de variabilidade do processo GVLPS. O protótipo do mecanismo de variabilidade apresentado permite, por exemplo, acionar os métodos da classe Sensor para realizar leituras e testes do sensores. Assim, o sistema gerado pelo sistema Gerador de Aplicação do mecanismo de variabilidade do GVLPS permite realizar a avaliação funcional do software do LSB porque é possível observar seu comportamento através do acionamento de seus métodos. Desta forma, o sistema Gerador de Aplicação pode ser estendido para permitir visualizar outras características de comportamento do software LSB, como, por exemplo, acompanhar as mudanças de estados dos eventos ou a sequência de leituras de sensores e acionamento de atuadores, de acordo com a ocorrência dos eventos do LSB.

Neste trabalho foi desenvolvido um protótipo do mecanismo de variabilidade do processo GVLPS relacionado com a parte funcional do software do LSB. Como o software do LSB é crítico e de tempo real, para o projeto real dever-se-á levar em consideração os requisitos não funcionais, como a segurança e o desempenho, o que implica em realizar novas atividades de pesquisa para evoluir o processo GVLPS. As técnicas utilizadas pelo mecanismo de variabilidade do GVLPS aumentam a flexibilidade para implementação da variabilidade, mas interferem no desempenho e não são convencionais para a aplicação em sistemas de segurança. Desta forma, é necessário investir no sistema Gerador de Aplicação para que o sistema gerado possa ter maior desempenho. Ainda, podem-se buscar outros padrões de projetos para melhorar o mecanismo de variabilidade do processo GVLPS e considerar outras linguagens de programação que trabalhem com reflexão.

9.3 Conclusões e Retrospectiva

A abordagem de linha de produtos de software tem como meta, o reuso organizado para a geração de sistemas de uma determinada família. Para isso, as atividades de Engenharia de Domínio, Engenharia de Aplicação e Gerenciamento prevêem, respectivamente, a organização de sistemas em partes comuns e variáveis, a geração da aplicação e o funcionamento harmônico destas atividades. Por outro lado, o processo de reuso organizado fornece produção com economia, qualidade e desenvolvimento eficaz.

O sucesso da linha de produtos de software está diretamente relacionado ao gerenciamento de variabilidade, que viabiliza a derivação de mais produtos. Entretanto, o gerenciamento da variabilidade é considerado uma tarefa desafiadora devido à dificuldade apresentada para se estabelecer as inúmeras decisões a serem tomadas para identificar, especificar e implementar a variabilidade.

Desta forma, este trabalho buscou estabelecer um processo de gerenciamento de variabilidade que apresentasse atividades bem delineadas para a produção dos artefatos a serem utilizados na linha de produtos de software.

Por ser um tema recente, constatou-se, que não existe uma padronização dos termos relacionados à linha de produtos de software e, por isso, as definições dos termos feitas por diversos autores, muitas vezes, apresentam inconsistência. Isto gerou certa dificuldade na compreensão da literatura consultada, o que levou à necessidade de se estabelecer uma padronização destes termos para utilização neste trabalho.

Como existem diversas abordagens para linha de produtos de software, foi necessário analisá-las e, após selecionar duas, consideradas mais apropriadas ao objetivo do trabalho, foi necessário estudá-las com mais profundidade, para que pudesse se beneficiar das principais características de ambas no processo GVLPS.

O entendimento e a implementação das técnicas de reflexão e dos modelos de objetos adaptáveis constituíram um passo difícil no desenvolvimento deste trabalho. Para tanto, foi necessário primeiramente, estudar a limitada literatura da área e

realizar entrevistas com especialistas de engenharia de software experientes na área de reflexão e objetos adaptáveis. Para concluir estes estudos, foi necessário também a realização de vários pequenos experimentos práticos, envolvendo a implementação de código de alguns exemplos para o entendimento apropriado das técnicas.

Outra atividade que demandou tempo considerável foi o estudo da documentação e o entendimento do código fonte do VLS, que é considerado um software complexo. Como resultado deste estudo, vários diagramas UML foram criados para concretizar o entendimento do software.

O principal diferencial do processo GVLPS em relação aos trabalhos de literatura é o mecanismo de variabilidade baseado em modelos de objetos adaptáveis e em reflexão, pois a utilização destas técnicas permite que este mecanismo possibilite flexibilizar a adaptação dos artefatos para se obter um número mais satisfatório de variantes e permitir realizar mais facilmente as alterações no sistema, conduzindo assim a uma maior reusabilidade.

Observa-se que as técnicas de modelos de objetos adaptáveis e reflexão oferecem vantagens por permitirem adaptar o sistema a novos requisitos e causar a redução de classes. Entretanto, algumas desvantagens como dificuldades para o desenvolvimento e redução do desempenho devem ser consideradas.

É importante salientar, também, que o mecanismo de variabilidade do GVLPS não pretende solucionar os problemas apresentados por outras técnicas, mas proporcionar maior flexibilidade ao sistema, como uma alternativa para implementação da variabilidade.

Observa-se, através da análise dos artigos consultados (NORTHROP, CLEMENTS, 2007), (SCHNIEDERS, 2006a), (BECKER, 2003), (SVAHNBERG, BOSCH, 2000) que, de maneira geral, as instituições ou as organizações que desenvolvem software vêm reconhecendo a importância e apóiam a idéia de se reaproveitar o esforço empregado no desenvolvimento de sistemas para reduzir prazo, recursos, e aumentar a produtividade e qualidade dos sistemas desenvolvidos.

Este conceito também está sendo seguido no desenvolvimento de veículos lançadores de satélites brasileiros que é considerado um projeto ousado, cujo seu software é complexo e exige processos e métodos adequados ao seu desenvolvimento.

Por isso, a aplicação do processo GVLPS ao software do LSB teve a finalidade de mostrar a viabilidade de um desenvolvimento de software que permite aumentar o reuso, evitando esforços repetitivos para desenvolver novo software para diferentes veículos na mesma família.

Através do processo GVLPS também é possível fornecer aos desenvolvedores uma visão das possíveis variações que o software pode ter, sem esta visão, a evolução ou adaptação do software se torna mais difícil.

Além disso, com uma abordagem de desenvolvimento de software mais recente como a da linha de produtos de software, que se apóia em artefatos que podem ser desenvolvidos em tecnologias também mais modernas, é possível obter uma documentação mais clara, com modelos atualizados. Desta forma, estes artefatos podem ser compreendidos, permanecerem ativos por mais tempo e servirem de base para a modelagem de software de outros novos veículos.

Assim, através do processo GVLPS, espera-se melhorar o desenvolvimento de software de veículos lançadores, tornando as informações do projeto institucionalizadas e a documentação atualizada.

Entende-se que o trabalho apresentado também vai ao encontro com as perspectivas de novas versões de veículos lançadores, já que a abordagem de linha de produtos de software visa o reuso, e o gerenciamento de variabilidade flexibiliza este reuso através das atividades desenvolvidas e artefatos gerados por este processo.

Finalmente, com relação à pesquisa, pretende-se, com este trabalho, ampliar e diversificar o conjunto dos poucos trabalhos existentes no assunto e colaborar, de forma modesta, para o avanço de estudos na área de desenvolvimento de software.

REFERÊNCIAS BIBLIOGRÁFICAS

ACIS. Prague, Czech Republic. Apresenta Software Engineering Research, Management and Applications – SERA, 2008. Disponível em: <http://dsrg.mff.cuni.cz/sera/>. Acesso em: 24 abr. 2009.

AEB - Agência Espacial Brasileira. Brasília. Apresenta Programa Nacional de Atividades Espaciais, Conselho Superior da Agência Espacial Brasileira, 2006. Disponível em <http://www.aeb.gov.br/>. Acesso em: 12 dez. 2006.

AEB. **Programa Nacional de Atividades Espaciais 2005-2014.** Brasília: Agência Espacial Brasileira, 2005. Disponível em: http://www.aeb.gov.br/area/download/pnae_web.pdf . Acesso em: 27 abr.2009

AIAA American Institute of Aeronautics and Astronautics, Inc. Apresenta SpaceOps, 2008. Disponívem em: <http://www.aiaa.org/spaceops2008/>. Acesso em: 27 abr. 2009.

ANASTASOPOULOS, M.; GACEK, C. **Implementing product line variabilities**. In: SYMPOSIUM ON SOFTWARE REUSABILITY - SSR - PUTTING SOFTWARE REUSE IN CONTEXT, 1., 2001, Toronto, Ontario, Canada. Proceedings. New York: ACM, 2001, p.109-117. DOI= <http://doi.acm.org/10.1145/375212.375269>

ANDRADE, L. C. **Comparação entre os processos de pesenvolvimento de software RUP e CRC/WB+ em relação ao desenvolvimento de sistemas embarcados de tempo real.** 2000. Trabalho de Graduação - Instituto Tecnológico de Aeronáutica – ITA, São José dos Campos, 2000.

ANTKIEWICZ, M.; CZARNECKI, K. **FeaturePlugin: feature modeling plug-in for Eclipse**. In: OOPSLA WORKSHOP ON ECLIPSE TECHNOLOGY EXCHANGE, 4., 2004, Vancouver, British Columbia, Canada. Proceedings. New York: ACM, 2004, p. 67-72. DOI= <http://doi.acm.org/10.1145/1066129.1066143>

ARSANJANI, A. **Rule Object 2001: a pattern language for adaptive and scalable business rule construction.** In: CONFERENCE ON PATTERN LANGUAGES OF PROGRAMS - PLoP, 8., 2001, Monticello, Illinois, USA. Proceedings. 2001.

ARSANJANI, A., ALPIGINI., J. **Using grammar-oriented object design to seamlessly map business models to software architectures.** In: CONFERENCE ON MODELING AND SIMULATION - IASTED, 2001, Pittsburgh, PA. Proceedings. 2001.

ATKINSON, C.; BAYER, J.; MUTHIG, D. **Component-based product line development: the KobrA approach.** In: SOFTWARE PRODUCT LINE CONFERENCE– SPLC, 1.,2000, Denver, Colorado, 2000. Proceedings. 2000.

BACHMANN, F. ;BASS, L. **Managing variability in software architectures**. In SYMPOSIUM ON SOFTWARE REUSABILITY - SSR - PUTTING SOFTWARE REUSE IN CONTEXT, 1., 2001, Toronto, Ontario, Canada. Proceedings. New York: ACM, 2001,p.126-132. DOI= <http://doi.acm.org/10.1145/375212.375274>.

BACHMANN, F.; CLEMENTS, P. C. **Variability in software product lines.** Pittsburg: Software Engineering Institute -SEI, 2005. (Technical Report -CMU/SEI-2005-TR-012).

BAYER, J.; FLEGE, O.; KNAUBER, P.; LAQUA, R.; MUTHIG, D.; SCHMID, K.; WIDEN, T.; DEBAUD, J. 1999. **PuLSE: a methodology to develop software product lines**. In: SYMPOSIUM ON SOFTWARE REUSABILITY - SSR, 1999, Los Angeles, California, United States. Proceedings. New York: ACM, NY,1999, p. 122-131. DOI= <http://doi.acm.org/10.1145/303008.303063>

BECKER, M. **Towards a general model of variability in product families.** In: SOFTWARE VARIABILITY MANAGEMENT WORKSHOP, 2003, Portland, 2003, p.19-27.

BECKER, M., GEYER, L., GILBERT, A., BECKER, K. **Comprehensive variability modelling to facilitate efficient variability treatment.** In: INTERNATIONAL WORKSHOP ON SOFTWARE PRODUCT-FAMILY ENGINEERING, 4., 2002, Lecture Notes In Computer Science, vol. 2290. London: Springer-Verlag, 2002, p. 294-303.

BOOCH, G.; RUMBAUGH, J. JACOBSON, I. **UML guia do usuário**, São Paulo: Editora Campos, 2006.

BRAGANÇA, A.; MACHADO, R.J. **Engenharia de domínio no suporte ao aumento de flexibilidade nos sistemas de software**. In:ENCONTRO PARA A QUALIDADE NAS TECNOLOGIAS DA INFORMAÇÃO E COMUNICAÇÕES - QUATIC,5., 2004a, Porto, Portugal. Actas.Technical Session of Software Product Quality, 2004a, p.15-21. Edição IPQ, ISBN-972-763-069-3.

BRAGANÇA A.; MACHADO, R.J. **A methodological approach to domain engineering for software variability enhancement**. In: WORKSHOP ON METHOD ENGINEERING FOR OBJECT ORIENTED AND COMPONENT BASED DEVELOPMENT (within the 19th ACM SIGPLAN Conference on Object-Oriented

Programming, Systems, Languages, and Applications - OOPSLA'04), 2., 2004b, Vancouver, Canada. Proceedings. Sydney, Australia: COTAR Edition, 2004b, p. 39-50. [ISBN-0-9581915-3-0].

BÜHNE, S.; HALMANS, G.; POHL, K. **Modelling dependencies between variations point in use case diagrams.** In: WORKSHOP ON REQUIREMENTS ENGINEERING, 9.,2003. Proceedings. Klagenfurt; Velden: Foundation for Software Quality - REFSQ, 2003.

BURGARELI, L. A.; MELNIKOFF, S. S. S.; FERREIRA, M. G. V. **A software model reuse strategy for Brazilian Satellite Launcher.** In: AUSTRALIAN CONFERENCE ON SOFTWARE ENGINEERING - ASWEC, 19., 2008a, Perth, WA. Proceedings. Washington: Computer Society, IEEE, 2008a, p.627 – 632.

BURGARELI, L. A.; MELNIKOFF, S. S. S.; FERREIRA, M. G. V. **A software modeling approach based on MDA for the Brazilian Satellite Launcher.** In: INTERNATIONAL CONFERENCE ON SPACE OPERATIONS - SPACEOPS, 10., 2008b, Heidelberg, Germany. Proceedings. Reston, VA: AIAA, 2008b.

BURGARELI, L. A.; MELNIKOFF, S. S. S.; FERREIRA, M. G. V. **A variability management strategy for software product lines of Brazilian Satellite Launcher vehicles.** In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING RESEARCH, MANAGEMENT AND APPLICATIONS - SERA, 6.,2008c, Prague, Czech Republic. Berlin Heidelberg: Springer-Verlag, 2008c, p.1-14.

BURGARELI, L. A.; MELNIKOFF, S. S. S.; FERREIRA, M. G. V. **A variation mechanism based on adaptive object model for software product line of Brazilian Satellite Launcher.** In: IEEE EASTERN EUROPEAN REGIONAL CONFERENCE ON THE ENGINEERING OF COMPUTER BASED SYSTEMS - ECBS-EERC, 1., 2009d, Novi Sad, Serbia. Proceedings. Washington: Computer Society, IEEE, 2009d. No prelo.

CAPILLA, R.; DUEÑAS, J. C. **Modelling variability with features in distributed architectures**. In: INTERNATIONAL WORKSHOP ON SOFTWARE PRODUCT-FAMILY ENGINEERING, 4, 2001, Lecture Notes In Computer Science, vol. 2290. London: Springer-Verlag, 2001,p. 319-329.

CARDOSO, P. E. **Uma nova arquitetura para a representação das regras de negócio em modelos de objetos dinâmicos**. 2005. 133p. Dissertação (Mestrado) – Instituto Nacional de Pesquisas Espaciais – INPE, São José dos Campos, 2005.

CARDOSO, P. E.; BARRETO, J.P.; CARDOSO, L.S.;HOFFMANN, L.T. **Using design patterns, components and metadata to design the command and monitoring frameworks of the INPE´s satellite control system**. In: INTERNATIONAL CONFERENCE ON SPACE OPERATIONS - SPACEOPS, 10., 2008, Heidelberg, Germany. Proceedings. Reston, VA: AIAA, 2008.

CAZZOLA, W. **Evaluation of object-oriented reflective models.** In: WORKSHOP ON OBJECT-ORIENTED TECHNOLOGY, 1998. S. Demeyer and J. Bosch, Eds. Lecture Notes In Computer Science, vol. 1543. London: Springer-Verlag, 1998, p. 386-387.

COMANDO DA AERONÁUTICA. **Relatório da investigação do acidente ocorrido com o VLS-1 V03 em 22 de agosto de 2003**, Alcântara Maranhão: Ministério da Defesa, 2004. Disponível em <http://www.aeb.gov.br/area/PDF/VLS-1_V03_Relatorio_Final.pdf>. Acesso em: 27 abr. 2009.

CTA - CENTRO TÉCNICO AEROESPACIAL. São José dos Campos. Apresenta CTA e AEB apresentam programa Cruzeiro do Sul, 2006. Disponível em: <http://www.cta.br/noticias11.htm>. Acesso em: 15 nov. 2006.

CURTIN UNIVERSITY OF TECHNOLOGY OF AUSTRALIA. Perth, WA. Apresenta Australian Software Engineering Conference – ASWEC, 2008. Disponível em <http://www.aswec2008.curtin.edu.au/>. Acesso em: 21 abr. 2009.

CZARNECKI, K.; EISENECKER, U.W. **Generative programming: methods, techniques, and applications.** Canada: Addison -Wesley, 2005.

CZARNECKI, K.; HELSEN, S.; EISENECKER, U.W. Formalizing cardinality-based feature models and their specialization. **Software Process Improvement and Practice**, University of Waterloo, Canada, jan/mar, 2005,p.7-29. Disponível em: <http://swen.uwaterloo.ca/~kczarnec/spip05a.pdf>. Acesso em: 27 abr. 2009.

DEFENSE SYSTEM SOFTWARE DEVELOPMENT. **DOD-STD-2167A**. Department of Defense, Washington, U.S: Government Printing Office, 1988.

ECLIPSE. Ottawa, Ontario, Canada. Apresenta The Eclipse Foundation, 2009. Disponível em: <http://www.eclipse.org/>. Acesso em: 12 fev. 2009.

EMBARCADERO TECHNOLOGIES, INC. San Francisco, CA. Apresenta JBuilder, 2009. Disponível em: <http://www.codegear.com/products/jbuilder>. Acesso em: 25 nov. 2008.

FOOTE, B.; YODER, J. **Metadata and active object models**. Collected papers from the PLoP ’98 and EuroPLoP ’98 Conference , 1998.

FRANCE, R.; GHOSH, S.; SONG, E.; KIM, D. **A metamodeling approach to pattern-based model refactoring**, IEEE Computer Society, 2003, p. 52-58. Disponível em: < http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=01231152 > Acesso em: 27 abr. 2009.

FRITSCH, C.; LEHN, A.; STROHM, T.; **Evaluating variability implementation mechanisms.** In: INTERNATIONAL WORKSHOP ON PRODUCT LINE ENGINEERING, 2002. Proceedings. Seattle, USA, 2002.

GABRIEL, R.G.; BOBROW, D.G.; WHITE, J.L. **The shape of the design space**. In: OBJECT ORIENTED PROGRAMMING – The CLOS PERSPECTIVE. Cambrige, MA: The MIT Press,1993, pp.29-61.

GAMMA, E., HELM, R., JOHNSON, R.; VLISSIDES, J. **Design patterns: elements of reusable object-oriented software**, Boston: Addison-Wesley, 1995.

GIMENES, I. M. S.; TRAVASSOS, G. H. **O enfoque de linha de produto para desenvolvimento de software**. In: JORNADA DE ATUALIZAÇÃO EM INFORMÁTICA DA SBC, 22., 2002, Florianópolis. Anais... Florianópolis, 2002.

GRISS, M. L.; FAVARO, J.; ALESSANDRO, M. **Integrating feature modeling with the RSEB**. In: INTERNATIONAL CONFERENCE ON SOFTWARE REUSE – ICSR, 5., 1998, Canada. Proceedings. Washington: IEEE Computer Society, 1998.

GOMAA, H.; **A software design method for real-time systems**, Communications of the ACM, Vol 27, nº 9, 1984.

GOMAA, H.; **Designing software product lines with UML from use cases to pattern-based software architectures**. Boston, MA: Addison-Wesley, 2005.

GOMAA, H.; SHIN, M. E. **Automated software product line engineering and product derivation**. In: ANNUAL HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES, 40., 2007, Waikoloa, Big Island. Proceedings. Washington: IEEE, Computer Society, 2007.

GOMAA, H.; WEBBER, D. L. **Modeling adaptive and evolvable software product lines using the variation point model**. In: ANNUAL HAWAII INTERNATIONAL

CONFERENCE ON SYSTEM SCIENCES, 37., 2004, Waikoloa, Big Island. Proceedings. Washington: IEEE Computer Society, 2004.

HABLI, I.; KELLY, T.; HOPKINS, I. **Challenges of establishing a software product line for an aerospace engine monitoring system**. In: INTERNATIONAL SOFTWARE PRODUCT LINE CONFERENCE, 11., 2007, Kyoto, Japan. Proceedings. Washington: IEEE Computer Society, 2007, p.193-202.

HARSU, M. **A Survey of product-line architectures.** Tampere: Institute of Software Systems, University of Technology, March 2001.(Technical Report 23).

HARSU, M. **FAST product-line architecture process.** Tampere: Institute of Software Systems, University of Technology, January 2002. (Technical Report 29).

HATLEY D. J.; PIRBHAI, I. A. **Estratégias para especificação de sistema em tempo real**, São Paulo: McGraw-Hill, 1990.

IAE – Instituto de Aeronáutica e Espaço. São José dos Campos. Apresenta a Missão Espacial Brasileira, 2006a. Disponível em: <http://www.iae.cta.br/historico_espacial.htm>. Acesso em: 12 dez. 2006.

IAE – Instituto de Aeronáutica e Espaço. São José dos Campos. Apresenta veículos lançadores de satélites - VLS, 2006b. Disponível em: <http://www.iae.cta.br/FoguetesdeSondagem/VLS/vls_descricaotecnica.htm> Acesso em: 12 dez. 2006.

IAE – Instituto de Aeronáutica e Espaço. São José dos Campos. Apresenta áreas de pesquisa - perfil da missão do VLS1, 2007. Disponível em: <http://www.iae.cta.br/FoguetesdeSondagem/vls_areas_pesquisa.htm>. Acesso em: 15 fev. 2007.

IAF - International Astronautical Federation. Paris, France. Apresenta IAC International Astronautical Congress, 2009. Disponível em: <http://www.iafastro.org/index.php?id=738>. Acesso em: 24 abr. 2009.

IEEE. Big Sky, Montana. Apresenta Aerospace Conference, 2009. Disponível em: <http://www.aeroconf.org/>. Acesso em: 24 abr. 2009.

IEEE. **IEEE-Std-983**:Guide for Software Quality Assurance Planning. IEEE Software Engineering Standards Subcommittee, 1986.

JOHNSON, R.E. **Dynamic object model**. Urbana-Champaign: Department Of Computer Science, University of Illinois, 1998.

JOHNSON, R.; WOOLF, B. **Type object**. In: PATTERN LANGUAGES OF PROGRAM DESIGN 3, Robert Martin, Dirk Riehle, and Frank Buschmann ed., Addison-Wesley, 1997.

KEEPENCE, B.; MANNION, M. **Using patterns to model variability in product families**. IEEE Software, 1999. Los Alamitos, CA: IEEE Computer Society Press, 1999.

KIM, Y.; KIM, J.; SHIN, S.; BAIK, D. **Managing variability for software product-line**. In: SOFTWARE ENGINEERING RESEARCH, MANAGEMENT AND APPLICATIONS CONFERENCE – SERA, 4., 2006, Seattle, WA. Proceedings. Washington: IEEE Computer Society, 2006, p.74 - 81.

LEAL, P. R. **Mecanismos de reflexividade: um resumo e comparação em 3 linguagens orientadas a objectos, SmallTalk, Java e C++**. In: WORKSHOP DE DESENVOLVIMENTO DE SOFTWARE ORIENTADO POR OBJECTOS. Monte da Caparica: Faculdade de Ciências e tecnologia da Universidade Nova de Lisboa, jan. 2003.

LEMES, M.J.R. **Uma abordagem para concepção de planos de garantia da qualidade de software**. 1997. Dissertação (Mestrado) - Instituto Tecnológico de Aeronáutica – ITA, Engenharia Eletrônica e de Computação, São José dos Campos, SP, 1997.

LIESENBERG, H.K.E.; STEHLING, R.O. **Projeto de uma arquitetura de software reflexiva para a linguagem Xchart**. In: BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING - SBES, 8.,1999, Florianópolis - Santa Catarina - Brasil ,13 -15 out. 1999.

MAES, P. **Concepts and experiments in computational reflection**. In: OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES AND APPLICATIONS - OOPSLA, 1987. Orlando, Florida. Proceedings. New York: ACM Sigplan Notices, dez. 1987, p.147-155.

MANOLESCU, D.A. **Micro-Workflow: A workflow architecture supporting compositional object-oriented software development**. 2000. PhD thesis - University of Illinois. Champaign, Urbana, 2000. (Computer Science Technical Report UIUCDCS-R-2000-2186).

MANOLESCU, D. A., JOHNSON, R. E.; **Dynamic object model and adaptive workflow**. In: METADATA AND ACTIVE OBJECT-MODEL PATTERN MINING WORKSHOP- (within the Object-Oriented Programming Systems, Languages And Applications- Oopsla).Proceedings. Dever,1999.

MATINLASSI, M. **Comparison of software product line architecture design methods: COPA, FAST, FORM, KobrA and QADA**. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 26.,2004, Scotland, UK. Proceedings. Washington: IEEE Computer Society, 2004, p. 127-136.

McCOMAS, D.; LEAKE, S.; STARK, M. ; MORISIO, M. ; TRAVASSOS, G. H. ; WHITE, M. **Addressing variability in a guidance, navigation, and control flight software product line**. In: SOFTWARE PRODUCT LINE CONFERENCE - SPLC, 1., 2000, Denver, Colorado. Proceedings. Denver, 2000.

MONTENEGRO, D.P. **O sistema de gestão da missão espacial completa brasileira – mecb: uma avaliação de sua contribuição ao desenvolvimento do programa.** Dissertação (Mestrado). Rio de Janeiro: Fundação Getúlio Vargas,1997.

MORAIS, P. Apresenta Programa de Veículos Lançadores de Satélites Cruzeiro Do Sul - O Brasil na Conquista de sua Independência no Lançamento de Satélites, IAE, CTA, 2005. Disponível em: <http://www.aeroespacial.org.br/aab/downloads.php>. Acesso em: 13 jan. 2007.

NORTHROP, L. M. **SEI's software product line tenets**. In: IEEE SOFTWARE, 2002, Los Alamitos, CA: IEEE Computer Society Press, 2002, p. 32-40.

NORTHROP, L. M.; CLEMENTS, P.C. **A framework for software product line practice.** Version 5.0. Pittsburg. Software Engineering Institute, 2007. Disponível em: < http://www.sei.cmu.edu/productlines/framework.html >. Acesso em: 28 abr. 2009.

OLIVEIRA, E. A. A. Q. **Proposta de modelo organizacional de gestão de tecnologia para o setor espacial brasileiro: estudo do caso VLS.** 1998. Tese (Doutorado) - Instituto Tecnológico de Aeronáutica – ITA. Curso de Engenharia Aeronáutica e Mecânica na área de Organização Industrial, São José dos Campos, SP, 1998.

OLIVEIRA, E. A.; GIMENES, I. M.; HUZITA, E. H.; MALDONADO, J. C. **A variability management process for software product lines**. In: CONFERENCE OF THE CENTRE FOR ADVANCED STUDIES ON COLLABORATIVE RESEARCH, 2005, Toranto, Ontario, Canada. Proceedings. New York: ACM, 2005.

OLIVEIRA JUNIOR, E. A. **Um processo de gerenciamento de variabilidade para linha de produto de software**. 2005. Dissertação (Mestrado) - Programa de Pós-Graduação em Ciência da Computação da Universidade Estadual de Maringá, Maringá, 2005.

OMG - Object Management Group, Inc. Needham, MA. Apresenta History of CORBA, 2008. Disponível em:< http://www.omg.org/gettingstarted/history_of_corba.htm>. Acesso em: 25 nov. 2008.

OMG - Object Management Group, Inc. Needham, MA. Apresenta OMG Model Driven Architecture, 2009. Disponível em: < http://www.omg.org/mda/>. Acesso em: 13 de maio de 2009.

PAGE-JONES, M. **Projeto Estruturado de Sistemas**, São Paulo: McGraw-Hill, 1988.

PERKINS, A. **Business rules =meta-data**. In: TECHNOLOGY OF OBJECT-ORIENTED LANGUAGES AND SYSTEMS – TOOLS, 2000, Sydney, Australia. Proceedings. Washington: IEEE Computer Society, 2000.

POHL, K., BÖCKLE, G.; LINDEN, F. J. **software product line engineering: foundations, principles and techniques**. New York: Springer-Verlag, 2005.

PRESSMAN, R.S. **Engenharia de software**, São Paulo: Makron Books, 1995.

REIS FILHO, J.V.B. **Uma abordagem de qualidade e confiabilidade para software crítico**.1995. Dissertação (Mestrado) - Instituto Tecnológico de Aeronáutica – ITA. Curso de Engenharia Eletrônica e de Computação, São José dos Campos, SP, 1995.

RIEHLE, D.; TILMAN, M.; JOHNSON, R.; **Dynamic object model**. In: CONFERENCE ON PATTERNS LANGUAGES OF PROGRAMS – PloP,7., 2000, Monticello Illinois. Proceedings. 2000.

RUIZ, J. C.; KILLIJIAN, M.; FABRE, J.; THÉVENOD-FOSSE, P; 2003. Reflective Fault-Tolerant Systems: From Experience to Challenges. **IEEE Transactions on Computers**, v.52, n.2 , p. 237-254, 2003. DOI= <http://dx.doi.org/10.1109/TC.2003.1176989>

SCHNIEDERS, A. **Variability mechanism centric process family architectures**. In: ANNUAL IEEE INTERNATIONAL SYMPOSIUM AND WORKSHOP ON ENGINEERING OF COMPUTER BASED SYSTEMS - ECBS, 13., 2006a, Potsdam, Germany. Proceedings. Washington: IEEE Computer Society, 2006a, p. 289-298. DOI= <http://dx.doi.org/10.1109/ECBS.2006.72>

SCHNIEDERS, A. **Modeling and implementing variability in state machine based process family architectures for automotive systems**. In: INTERNATIONAL WORKSHOP ON SOFTWARE ENGINEERING FOR AUTOMOTIVE SYSTEMS – SEAS, 2006b, Shanghai, China. Proceedings. New York: ACM, 2006b, p. 37-44.

SPC - Software Productivity Consortium. **Reuse-driven software processes guidebook** (SPC-92019-CMC). Herndon, VA, 1993. Version 02.00.03.

SUN Microsystems, Inc. Apresenta produtos desenvolvidos, 2008. Disponível em: <http://java.sun.com/>. Acesso em: 25 nov. 2008.

SVAHNBERG, M.; BOSCH, J. **Issues concerning variability in software product lines**. In: INTERNATIONAL WORKSHOP ON SOFTWARE ARCHITECTURES FOR PRODUCT FAMILIES, 2000. Proceedings. F. v. Linden, Ed. Lecture Notes In Computer Science, vol. 1951. London: Springer-Verlag, 2000, p.146-157.

SVAHNBERG, M.; VAN GURP, J.; BOSCH, J. **A taxonomy of variability realization techniques**. Sweden: Blekinge Institute of Technology, 2002. (Technical report).

THAO, C.; MUNSON, E. V.; NGUYEN, T. N. **Software configuration management for product derivation in software product families**. In: ANNUAL IEEE INTERNATIONAL CONFERENCE AND WORKSHOP ON THE ENGINEERING OF COMPUTER BASED SYSTEMS – ECBS, 15., 2008. Proceedings. Washington: IEEE Computer Society, 2008, p. 265-274. DOI= <http://dx.doi.org/10.1109/ECBS.2008.53>

THOMÉ, A. C. **Uma arquitetura de software distribuída configurável e adaptável aplicada às várias missões de controle de satélites.** 2004.Tese (Doutorado) – Instituto Nacional de Pesquisas Espaciais - INPE, São José dos Campos, 2004.

TRIGAUX J., HEYMANS P. **Software product lines: state of the art**. Belgium: Institut d'Informatique FUNDP, PLENTY project, 2003. (Technical Report EPH3310300R0462/215315)

UNIVERSITY OF NOVI SAD. Novi Sad, Serbia. Apresenta: 1st IEEE Eastern European Regional Conference on the Engineering of Computer Based Systems – ECBS-EERC. Disponível em:< http://www.ecbs-eerc.org/>. Acesso em: 15 jun. 2009.

VAN GURP, J.; BOSCH, J., SVAHNBERG, M. **On the notion of variability in software product lines**. In: WORKING IEEE/IFIP CONFERENCE ON SOFTWARE ARCHITECTURE, 2001, Amsterdam, The Netherlands. Proceedings. Washington: IEEE Computer Society, 2001.

VOELTER, M. Variabilities - Representing Variability In Software Systems, 2006. Disponível em: <http://www.voelter.de/data/presentations/mdsd-tutorial/09_PLAndVariants.pdf>. Acesso em 26 jul 2009.

VOELTER, M.; GROHER, I. **Product line implementation using aspect-oriented and model driven software development.** In: INTERNATIONAL SOFTWARE PRODUCT LINE CONFERENCE, 11.,2007, Kyoto, Japan. Proceedings. Washington: IEEE Computer Society, 2007.

WELICKI, L., YODER, J.W., WIRFSBROCK, R. **A pattern language for adaptive object models: part I rendering patterns**. In: CONFERENCE ON PATTERN LANGUAGES OF PROGRAMS – PloP, 2007, Monticello Illinois. Proceedings. 2007

YODER, J.W. **Adaptive object-model architecture style "giving users control over their business models".** Tokyo, Japan, National Institute of Informatics, 2007 The Adaptive Object-Model Architecture Style, Palestra, Tokyo, 2007.

YODER, J. W., BALAGUER, F., JOHNSON, R. **Architecture and design of adaptive object-models**. ACM SIGPLAN Notices, Column: Intriguing Technology from Oopsla, vol. 36., 2001.New York: ACM, 2001, p. 50-60. DOI= <http://doi.acm.org/10.1145/583960.583966>.

YODER, J. W.; JOHNSON, R.E. **Architecture and design of adaptive object-models**. São Paulo, Universidade de São Paulo, Instituto de Matemática e Estatística –IME, 23 e 24 ago. 2003. Cursos e Palestras sobre Desenvolvimento de Software Orientado a Objetos São Paulo, 2003.

YODER, J. W.; JOHNSON, R. E. **The adaptive object-model architectural style**. In: WORLD COMPUTER CONGRESS TC2 STREAM / 3RD IEEE/IFIP CONFERENCE ON SOFTWARE ARCHITECTURE: SYSTEM DESIGN, DEVELOPMENT AND MAINTENANCE, 17., 2002. Proceedings, vol. 224. The Netherlands, 2002, p. 3-27.

ANEXO A - Padrão para Descrição de Casos de Uso

1) Nome do Caso de Uso: nome que identifica o caso de uso descrito.

2) Categoria de Reuso: tipo de caso de uso em relação a classificação (comum, opcional ou alternativo) conforme descrita na Seção 6.3 deste trabalho.

3) Sumário: descrição resumida do caso de uso.

4) Ator(es): lista do(s) ator(es) que participa(m) do caso de uso descrito.

5) Evento Iniciador: evento que causa o início do caso de uso descrito.

6) Pré-Condição: condição em que o sistema da linha de produtos de software deve estar antes da execução do caso de uso descrito.

7) Descrição: seqüência de passos para execução do caso de uso descrito. As variações, caso existam, devem ser descritas como ponto de variação.

8) Ponto(s) de Variação: lista os pontos de variação e variantes encontrados nos passos do item anterior.

9) Pós-Condição: apresenta a condição em que o sistema da linha de produtos de software deve estar após a execução do caso de uso descrito.

10) Extensão: caminhos alternativos do caso de uso, incluindo as exceções de um curso normal de eventos.

11) Inclusão: lista de casos de uso usados pelo caso de uso descrito.