

UNIVERSIDADE FEDERAL DE SÃO CARLOS
CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

**UMA ABORDAGEM APOIADA POR LINGUAGENS
ESPECÍFICAS DE DOMÍNIO PARA CRIAÇÃO DE
LINHAS DE PRODUTOS DE SOFTWARE
EMBARCADO**

RAFAEL SERAPILHA DURELLI

ORIENTADORA: PROF^a. DR^a. ROSÂNGELA AP. DELLOSSO PENTEADO

São Carlos - SP
Maio/2011

UNIVERSIDADE FEDERAL DE SÃO CARLOS
CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

**UMA ABORDAGEM APOIADA POR LINGUAGENS
ESPECÍFICAS DE DOMÍNIO PARA CRIAÇÃO DE
LINHAS DE PRODUTOS DE SOFTWARE
EMBARCADO**

RAFAEL SERAPILHA DURELLI

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Engenharia de Software.
Orientadora: Dra. Rosângela Ap. Dellosso Penteadó

São Carlos - SP
Maio/2011

**Ficha catalográfica elaborada pelo DePT da
Biblioteca Comunitária da UFSCar**

D955aa

Durelli, Rafael Serapilha.

Uma abordagem apoiada por linguagens específicas de domínio para criação de linhas de produtos de software embarcado / Rafael Serapilha Durelli. -- São Carlos : UFSCar, 2011.
139 f.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2011.

1. Análise e projeto de sistemas. 2. Linha de produtos de software. 3. Linguagem específica de domínio. 4. Sistemas embarcados I. Título.

CDD: 004.21 (20^a)

Universidade Federal de São Carlos
Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

**“Uma Abordagem Apoiada por Linguagens Específicas
de Domínio para a Criação de Linhas de Produtos
de Software Embarcado”**

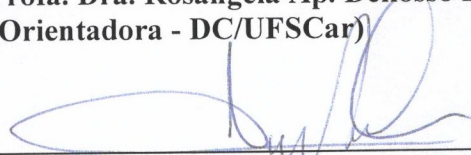
RAFAEL SERAPILHA DURELLI

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Membros da Banca:



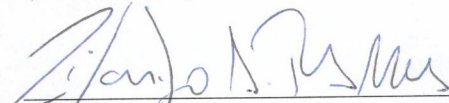
Profa. Dra. Rosângela Ap. Dellosso Penteado
(Orientadora - DC/UFSCar)



Prof. Dr. Valter Vieira de Camargo
(Co-orientador - DC/UFSCar)



Prof. Dr. Denis Fernando Wolf
(ICMC/USP)



Prof. Dr. Ricardo Argenton Ramos
(UNIVASF)

São Carlos
Maio/2011

Dedico este trabalho para minha família, e para minha namorada, Patrícia.

AGRADECIMENTOS

Agradeço primeiramente a Deus, sem ele nada seria possível.

Agradeço também a Professora Rosangela pela dedicação, amizade e confiança em mim depositada. Obrigado por compartilhar seus conhecimentos, serão sempre lembrados.

Ao Professor Valter pela amizade, dedicação e várias correções minuciosas.

Aos meus pais Humberto e Lucia por me apoiarem em todos os momentos de minha vida.

À minha namorada Patrícia, pela compreensão e carinho.

Ao meu irmão, por me mostrar o caminho.

A minha amiga Simone Borges.

Aos colegas de laboratório, Matheus, Paulo, André, DB, Thiago, Hiro.

E a todos os meus colegas do mestrado pela amizade e bons momentos que passamos durante esse período.

A CAPES pelo apoio financeiro.

Saber, todo mundo sabe, querer todo mundo quer mais fácil falar do que fazer.
Humberto Gessinger

Resumo

Sistemas embarcados são utilizados em vários dispositivos que fazem parte da vida cotidiana, de modo que o mercado de tais sistemas tem crescido de maneira expressiva. Esses sistemas sempre foram associados com código de baixo nível, no entanto, essa visão está desatualizada. Nas aplicações embarcadas correntes o software é a principal parcela, embora nenhuma técnica sistemática de reuso seja utilizada para sua concepção. Desse modo ocorre um atraso considerável na produtividade dos sistemas, uma vez que experiências anteriores bem sucedidas não são reaproveitadas, sendo necessário que o desenvolvedor comece do zero toda vez que um software for desenvolvido. Com a crescente complexidade dos sistemas embarcados é necessário utilizar técnicas de reuso para diminuir o atrasado da produção de tais sistemas. Nesse contexto, Linha de Produtos de Software (LPS) é definida como uma técnica de reuso que permite a construção de vários sistemas pertencentes a um mesmo domínio. LPS é aplicável para a geração de produtos específicos de um domínio, mas que possuem um conjunto de características comuns e pontos de variabilidades bem definidos. O Desenvolvimento de Software Orientado a Modelos (do inglês – *Model-Driven Development* - MDD) é outra técnica de reuso na qual tem como principal objetivo reduzir a distância semântica entre o problema do domínio e solução/implementação, fazendo com que o engenheiro não precise interagir diretamente com o código-fonte, podendo se concentrar em modelos que possuem maiores níveis de abstração e posteriormente realizar transformações *Model-To-Code* e/ou *Model-To-Model*. A partir dessas técnicas de reuso é introduzido um processo para o desenvolvimento de linhas de produtos de software no domínio de *Robôs Moveis*. A fim de utilizar o processo proposto foi desenvolvida uma LPS intitulada *LegoRobosMoveis Linha de Produtos de Software* (LRMLPS). Adicionalmente, foi desenvolvida uma linguagem específica de domínio denominada F2MoC que auxilia o engenheiro de aplicação na instanciação automática de membros da LRMLPS.

Palavras-chave: Linha de Produtos de Software, Desenvolvimento de Software Dirigido a Modelos, linguagens específica de domínio, sistemas embarcados, robôs móveis

Abstract

Embedded systems have been used in a myriad of devices that are present in our daily lives, thereby the market for such sort of system has increased significantly over the last few years. These systems were once associated with low-level code, however, this is an outdated view of embedded systems technology. Although the current embedded systems are mostly composed of software, no systematic reuse technique is used in throughout their development. Thus, since previous successful experiences are not reused, forcing the developer to create some of the involved elements from the scratch, there is a considerable delay in the production of these systems. Due to the ever increasing complexity of embedded systems it is necessary to apply reuse techniques in order to lessen the effort needed to develop such systems. Within this context, software product lines (SPL) are reuse techniques that allow the creation of several systems belonging to a certain domain. SPL can be used to generate products of a specific domain that share common features but are each different in a specific way. Model-driven development is another reuse technique whose main objective is to reduce the semantic distance between the domain problem and its solution/implementation; thus, the developer does not need to direct interact with the solution source code, being able to focus on models and transforming those models in source code or yet other models. Based on these techniques, a process for the development of SPL in the domain of mobile robots was developed. In order to properly use the proposed process, a SPL called LegoMobileRobots Software Product Line (LMRSPL) was devised. Moreover, a domain specific language (DSL) was also developed. This DSL, called F2MoC, assists the application engineer in instantiating LMRSPL members.

Keywords: Software Product Line, Model-Driven Development, Domain-Specific Language, Embedded Systems, Mobile Robots

Lista de Figuras

Figura 2-1 - Kit Lego Mindstorms	25
Figura 2-2 – Microcontrolador NXT	25
Figura 2-3 - Sensor de Toque	26
Figura 2-4 - Sensor de Luz.....	26
Figura 2-5 - Sensor Ultrassônico.....	26
Figura 2-6 - Exemplo de um Atuador	26
Figura 2-7 – Exemplo da Utilização da API LeJOS	27
Figura 2-8 – ESPLEP (Adaptado de Gomaa, 2004)	33
Figura 2-9 – PLUS (Adaptado de Gomaa, 2004).....	34
Figura 2-10 – Exemplo de Diagrama de <i>Features</i> (Gomaa, 2004).....	35
Figura 2-11 - Ilustração do processo de criação de software no desenvolvimento orientado a modelos (Lucredio, 2009)	37
Figura 2-12- Visão Geral do Projeto Eclipse <i>Modeling</i>	41
Figura 2-13 - Visão Geral do <i>Pure::Variants</i>	42
Figura 2-14 - Visualização Gráfica do <i>Pure::Variants</i>	43
Figura 2-15 - Feature Modeling plug-in for Eclipse.....	43
Figura 2-16 - Representação da Ferramenta <i>Feature Modeling Tool</i>	44
Figura 2-17 – Visão Geral da Ferramenta XFeature	45
Figura 4-1 – Lista de Índicios.....	64
Figura 4-2 – Diagrama de Classe dos RMs.....	65
Figura 4-3 - Funcionalidades do RMs	67
Figura 4-4 - Diagrama de <i>Features</i> da LRMLPS	71
Figura 4-5 - Modelo Estático da LRMLPS	73
Figura 4-6 - Trechos de código-fonte da variabilidade Ultrassônico	75
Figura 4-7 - Visão Geral da LRMLPS.....	76
Figura 4-8 - Metamodelo da DSL	77
Figura 4-9 - Implementação do Metamodelo (metaclass <i>Ultrasonic</i>).....	78
Figura 4-10 - <i>Template</i> para geração de código.....	79
Figura 4-11 – Exemplo do uso da DSL.....	80

Figura 5-1 Criando um Projeto, passo 1	84
Figura 5-2 Criando um Projeto, passo 2	85
Figura 5-3 Estrutura da DSL	86
Figura 5-4 Criando os Arquivos Necessários da DSL, passo 1	87
Figura 5-5 – Criando os Arquivos Necessários da DSL, passo 2.....	87
Figura 5-6– Criando os Arquivos Necessários da DSL, passo 3.....	88
Figura 5-7 – Visão Geral da DSL Gráfica	89
Figura 5-8 - Criando Relacionamento entre as Features.....	90
Figura 5-9 Diagrama de <i>Features</i> do Robôs Moveis Explorer	91
Figura 5-10 - Validando o Diagrama de <i>Features</i> do Robôs Moveis Explorer...	92
Figura 5-11 - Transformações M2M e M2C	93
Figura 5-12 - Diagrama de Classe do RMs Explorer	94
Figura 5-13 - Estrutura do Código-Fonte gerado do RM Explorer.....	96
Figura 5-14 – Trecho do código da classe <code>TeleOperated</code> gerado a partir do diagrama de <i>features</i> ilustrado na Figura 5-5	96
Figura 5-15 - RM Explorer	97
Figura 5-16 - Diagrama de <i>Features</i> do RM ForkLift.....	99
Figura 5-17 - Diagrama de Classe do RMs ForkLift	100
Figura 5-18 – Estrutura do Código-Fonte gerado do RM ForkLift.....	102
Figura 5-19 - Trecho do código da classe <code>Autonomous</code> gerado a partir do diagrama de <i>features</i> ilustrado na Figura 5-12	102
Figura 5-20 - RM ForkLift.....	103
Figura 5-21 - Diagrama de <i>Features</i> do RM <i>HomeSentry</i>	105
Figura 5-22 - Diagrama de Classe do RMs HomeSentry	106
Figura 5-23 - Estrutura do Código-Fonte gerado do RM HomeSentry	108
Figura 5-24 - Trecho do código da classe <code>ObstacleDetector</code> gerado a partir do diagrama de <i>features</i> ilustrado na Figura 5-18.....	108
Figura A-1 - Kit <i>Lego Mindstorms</i>	123
Figura A-2 - Utilizando a Classe <i>Motor</i>	124
Figura A-3 - Utilizando a Classe <i>SimpleNavigator</i> para controlar um determinado RM.....	125
Figura A-4 – Sensor de Toque	126
Figura A-5 - Instanciando o Sensor de Toque.....	127

Figura A-6 - Método <i>isPressed()</i>	127
Figura A-7 - Sensor Ultrassônico	128
Figura A-8 - Exemplo da utilização do Sensor Ultrassônico	128
Figura A-9 - Sensor de Luz	129
Figura A-10 - Instanciação da classe <i>LightSensor</i>	129
Figura B-11 – Passos para a criação do Projeto GMF.....	132
Figura B-12 - Metamodelo do Diagrama Criado	133
Figura B-13 – Criando as Classes do Diagrama	133
Figura B-14 - Geração do Códigos Java	134
Figura B-15 - Criando Componentes Gráficos.....	134
Figura B-16 - Passos para Criar os Componentes Visuais.....	135
Figura B-17 - Configuração do arquivo <i>.gmftool</i>	135
Figura B-18 - Criando o Arquivo <i>.gmfmap</i>	136
Figura B-19 - Arquivo <i>.gmfmap</i>	136
Figura B-20 - Mapeando a metaclasses <i>Node</i>	137
Figura B-21 - Mapeamento dos Relacionamentos do Diagrama	137
Figura B-22 - Passos para a Criação do Diagrama	138
Figura B-23 - Passos para Criar o Diagrama Desenvolvido.....	138
Figura B-24 - Diagrama Desenvolvido	139

LISTA DE TABELAS

Tabela 2-1 - Classificação dos padrões de projeto (Adaptado de Gamma <i>et al.</i> , 1995)	30
Tabela 3-1 -Exemplo de Tabela de <i>Features</i> Candidatas (TFC).....	53
Tabela 3-2 - Relação entre Tipo de <i>Features</i> e Padrões de Projeto	55
Tabela 5-1 - Requisitos do RM 1	83
Tabela 5-2 - Classes Geradas na Transformação M2C do RM Explorer	95
Tabela 5-3 - Requisitos do RM 2 ForkLift	98
Tabela 5-4 - Classes Geradas na Transformação M2C do RM <i>Forklift</i>	101
Tabela 5-5 - Requisitos do RM HomeSentry.....	104
Tabela 5-6 - Classes Geradas na Transformação M2C do RM HomeSentry....	107
Tabela A-1 - Principais Métodos da Classe <i>SimpleNavigator</i>	125

SUMÁRIO

CAPÍTULO 1 - INTRODUÇÃO	15
1.1 Contextualização	15
1.2 Motivação e Objetivos para Realização do Trabalho.....	18
1.3 Organização da Dissertação.....	20
CAPÍTULO 2 - FUNDAMENTAÇÃO TEÓRICA	22
2.1 Considerações Iniciais	22
2.2 Sistemas Embarcados	22
2.2.1 Robôs <i>Moveis</i>	23
2.3 Padrões e Linguagens de Padrões.....	28
2.4 Linha de Produtos de Software.....	31
2.4.1 Método PLUS	32
2.5 Variabilidades em LPS	34
2.6 Desenvolvimento de Software Orientado a Modelos	36
2.6.1 Linguagem Específica de Domínio	38
2.6.1.1 Eclipse Modeling Project.....	40
2.7 Ferramentas Utilizadas para Modelar Diagramas de <i>Features</i>	41
2.7.1 Pure::variants.....	42
2.7.2 <i>Feature Modeling Plug-in for Eclipse</i>	43
2.7.3 <i>Feature Modeling Tool</i>	44
2.7.4 XFeature	45
2.8 Considerações Finais	46
CAPÍTULO 3 - UM PROCESSO PARA CRIAÇÃO DE LPS NO DOMÍNIO DE ROBÔS MOVEIS.....	47
3.1 Considerações Iniciais	47
3.2 Processo Para o Desenvolvimento Linha de Produtos de Software para <i>Robôs Moveis</i>	48
3.2.1 Engenharia de Domínio do Processo Proposto	49
3.2.1.1 Obter um Conjunto de RM.....	50
3.2.1.2 Inspeccionar os RMs	50

3.2.1.3 Criar o Diagrama de <i>Features</i>	54
3.2.1.4 Criar a Arquitetura da LPS.....	55
3.2.1.5 Implementar os Artefatos da LPS	56
3.2.1.6 Implementar a DSL Gráfica	57
3.2.2 Engenharia de Aplicação do Processo Proposto.....	60
3.3 Considerações Finais	61
CAPÍTULO 4 - LEGOROBOMOVEISSOFTWARE SOFTWARE PRODUCT LINE.	62
4.1 Considerações Iniciais	62
4.2 Obter um Conjunto de RM	62
4.3 Inspeccionar os RMs	63
4.3.1 Selecionar RM	63
4.3.2 Comparar e Identificar as Features da LPS:.....	64
4.4 Criar o Diagrama de <i>Features</i>	70
4.5 Criar a Arquitetura da LPS.....	72
4.6 Implementar os Artefatos da LPS	74
4.7 Implementar a DSL Gráfica	76
4.8 Engenharia de Aplicação da LRMLPS.....	79
4.9 Considerações Finais	81
CAPÍTULO 5 - EXEMPLO DE UTILIZACAO PARA GERAÇÃO DE RMS	
BASEADOS NA DSL DA LRMLPS	82
5.1 Considerações Iniciais	82
5.2 Exemplo 1: Geração do RM " <i>Explorer</i> "	83
5.3 Exemplo 2: Geração do RM " <i>ForkLift</i> ".	97
5.4 Exemplo 3: Geração do RM " <i>HomeSentry</i> "	104
5.5 Considerações Finais	109
CAPÍTULO 6 - CONCLUSÕES.....	110
6.1 Considerações Finais	110
6.2 Contribuições.....	111
6.3 Limitações do Trabalho Efetuado	112
6.4 Sugestões de Trabalhos Futuros.....	113
REFERÊNCIAS BIBLIOGRÁFICAS	114

APÊNDICE A.....	122
A.1 Introdução	122
A.2 Projeto <i>Lego Mindstorms</i>	122
A.3 Controlando os Atuadores do Kit Lego Mindstorms	123
A.4 Controlando os Sensores do Kit Lego Mindstorms	126
A.4.1 Sensor de Toque.....	126
A.4.2 Sensor Ultrassônico	128
A.4.3 Sensor de Luz	129
APÊNDICE B.....	131
B.1 Introdução	131
B.2 Utilizando os <i>Frameworks</i> EMF e GMF	131

Capítulo 1

INTRODUÇÃO

1.1 Contextualização

Atualmente, com a expansão dos computadores e dispositivos eletrônicos, softwares podem ser encontrados em muitos produtos, tais como telefones celulares, DVD *players*, carros, aviões e até mesmo equipamentos médicos. Esse software é dito embarcado e é definido como: “um sistema computacional de propósito especial que é utilizado em uma tarefa específica” (Wolf, 2008).

Até recentemente, sistema embarcado era considerado somente aquele cuja implementação ocorria por meio de linguagem de máquina, como *assembly* (Wehrmeister, 2009). Essa linguagem apresenta vários tipos de problemas para o desenvolvedor como, por exemplo, dificuldade de entendimento do código tanto pelo desenvolver quanto pelo usuário, usabilidade, alto custo de desenvolvimento (incluindo codificação, teste e depuração), além de portabilidade reduzida e praticamente pouco ou nenhum reuso de código na troca de plataforma (Wagner e Carro, 2009). Essa visão está desatualizada, em consequência dos avanços obtidos especialmente na área de microeletrônica, das melhorias ocorridas quanto à capacidade do hardware e do baixo custo de inclusão de mais memória, processador com alto desempenho e baixo consumo de energia.

Com o avanço na utilização de sistemas embarcados, há a preocupação em utilizar técnicas de desenvolvimento de software, que eram aplicadas somente em sistemas convencionais (*Desktop* ou servidores), também em sistemas embarcados (Wehrmeister, 2009). Isso se deve ao fato de que pesquisas mostram que mais da metade dos projetos de software embarcados encontram-se atrasados em relação à produção desse software (Wagner e Carro, 2009). Esse atraso está relacionado principalmente com o desenvolvimento do software embarcado, já que o hardware vem sendo cada vez mais considerado uma *commodity*, desde a introdução de

projetos baseados em plataformas (Sangiovanni-Vincentelli e Martin, 2001; Graaf, Lormans e Toetenel, 2003).

A crescente pressão das indústrias por produtos, em intervalos cada vez menores (*time-to-market*), tem motivado pesquisas por novas metodologias para acelerar o desenvolvimento de software embarcado. Portanto, para diminuir o *time-to-market*, aumentar a produtividade e a qualidade dos softwares dos sistemas embarcados, técnicas de reuso devem ser aplicadas (Graaf, Lormans e Toetenel, 2003).

No desenvolvimento de software não embarcado, técnicas de reuso geralmente são de grande importância para a redução do tempo do projeto. Biggerstaff e Perlis (1989) afirmam que reuso de software é a reutilização de qualquer tipo de conhecimento de um sistema em outros sistemas similares, com o objetivo de reduzir o esforço de desenvolvimento e de manutenção. Técnicas, conceitos e métodos que favorecem ao reuso têm sido propostos ao longo dos últimos anos. Entre esses podem ser citados: engenharia de domínio, *frameworks* de software orientados a objetos, padrões de software, arquitetura de software e desenvolvimento baseado em componentes. Na maioria das vezes, o desenvolvimento baseado em componentes fornece reuso de baixa granularidade. Assim, linha de produtos de software (LPS) surge como uma proposta de construção sistemática para a geração de uma família de produtos.

Em setores como o da indústria aeroespacial, automotiva e de componentes eletrônicos, técnicas de linhas de produtos já vêm sendo exploradas há algum tempo. Embora o software seja desenvolvido de maneira completamente diferente do que a dos automóveis, aviões, ou computadores; engenheiros de software podem utilizar estratégias similares de produção.

Uma LPS é definida como uma técnica de reuso que permite a construção de vários sistemas (produtos) do mesmo domínio, mas que possuem um conjunto de características (*features*) comuns e outras específicas. A parte comum é inserida em todos os produtos da linha e as especificidades de cada um são definidas pelos, seus pontos de variabilidade (Clements e Northrop, 2002). A construção de artefatos¹ para sistemas embarcados pode ser favorecida utilizando conceitos de

¹ Um artefato reutilizável é uma parte do desenvolvimento que pode ser utilizado em mais de um projeto (D'Souza e Wills, 1999).

LPS, considerando que há grande quantidade desses sistemas em vários segmentos do mercado.

De forma semelhante, a abordagem de Desenvolvimento Dirigido a Modelos (*Model-Driven Development - MDD*) tem sido altamente utilizada tanto no âmbito acadêmico quanto industrial (Völter, 2008). Isso se deu, pois essa abordagem permite ao engenheiro de software desenvolver aplicações com menor esforço e acelerar a produtividade no processo de desenvolvimento com a utilização de modelos de alto nível. Em outras palavras, MDD tem o foco na modelagem e nas transformações de modelos em artefatos de implementação ou em modelos mais específicos com o objetivo de fornecer maior automação ao desenvolvimento de software. Czarnecki e Antkiewicz (2005) afirmam que LPS e MDD quando combinados podem alcançar grandes benefícios. Assim, MDD pode representar diferentes pontos da LPS de forma mais abstrata, automatizar a análise e geração de código-fonte de um determinado membro da linha de produtos.

Ferramentas automatizadas para o desenvolvimento de sistemas embarcados facilitam o trabalho do engenheiro de aplicação. Nesse sentido é interessante ferramentas que reúnam os conceitos de LPS e MDD. Exemplos dessas ferramentas são apresentadas no Capítulo 2.

Os sistemas embarcados utilizados neste trabalho são os Robôs Moveis (RMs). Tais sistemas embarcados estão se tornando cada vez mais usuais em ambientes domésticos e profissionais, pode-se citar o aspirador de pó *IRobot Roomba* (Irobot, 2011) e o cortador de grama *Husqvarna Auto-mower* (Husqvarna, 2011) ambos totalmente autônomos.

O desenvolvimento de RMs é uma área de pesquisa multidisciplinar que consiste em estudar técnicas de controle de robôs autônomos e tele operados na sua locomoção em ambientes dinamicamente instáveis. Nesses ambientes, geralmente existem obstáculos móveis e fixos, e, portanto, o robô deve ser capaz de perceber, se localizar e se mover em ambientes complexos, tomando decisões por meio de informações providas dos mecanismos de sensoriamento (Fernandes, 2010).

Os hardwares utilizados para criar os RMs neste trabalho consistem de um kit da Lego *Mindstorms* (Lego, 2010), kit de robótica usualmente utilizado para área

educacional que permite a criação e programação de robôs. Para tal, utiliza-se de blocos de montar e deve-se definir o comportamento desejado por meio de sensores, motores e um microcontrolador. Uma das vantagens do kit *Legó Mindstorms* é que é possível utilizar várias linguagens de programação e API (*Application Programming Interface*) para auxiliar no desenvolvimento dos RMs. Neste trabalho é utilizada a máquina virtual *Legó Java Operating System*² (LeJOS) (Lejos, 2011).

1.2 Motivação e Objetivos para Realização do Trabalho

Computadores pessoais (*Desktop*) e servidores correspondem a 2% do mercado de processadores e são essas máquinas que executam o software tradicional, ou seja, sistemas que não possuem severas limitações como: tamanho de memória, capacidade de processamento e outras características físicas. Portanto, há enorme quantidade de software que deve ser desenvolvido para ser executado nos 98% restantes de processadores disponíveis no mercado. O tempo consumido para o desenvolvimento é alto considerando que a pouco reuso, o que implica na demora deste ser disponibilizado comercialmente.

Para lidar com esse atraso e o esforço despendido no desenvolvimento de software embarcado, projetistas procuram por técnicas de níveis mais altos de abstração para realizar a especificação de sistemas embarcados (Wagner e Carro, 2009). Assim, alguns pesquisadores têm explorado o uso de LPS no contexto de sistemas embarcados (Lee, *et al.*, 2005; Stoermer e Roeddiger, 2001; Kim, 2006; Polzer, Kowalewski e Botterweck, 2009; Niemelä e Ihme, 2001).

Como comentado na seção anterior, LPS pode ser complementada com Desenvolvimento de Software Orientado a Modelos (MDD), para apoiar o desenvolvimento de software embarcado. MDD tem como proposta fazer com que o engenheiro de software não precise interagir diretamente com o código-fonte, se concentrando em modelos de mais alto nível de abstração e protegendo-se das complexidades requeridas de uma determinada plataforma.

² No apêndice A é descrito maiores detalhes sobre LeJOS e sua utilização.

Como motivação para realizar este trabalho, podem ser listados os seguintes pontos: a participação no projeto do Instituto Nacional de Ciência e Tecnologia - Sistemas Embarcados Críticos (INCT-SEC), CNPq; o interesse no desenvolvimento de ferramentas que minimizem o esforço do engenheiro de aplicação e produzam produtos de qualidade, a preocupação com a redução do *time-to-market*, a verificação na prática dos benefícios possíveis com o uso combinado de MDD e LPS para desenvolver software para sistemas embarcados para reduzir o *time-to-market*; além da possibilidade de geração automática de código e modelos para membros de uma linha de produtos embarcados.

A partir da motivação apresentada e dos desafios existentes no desenvolvimento de sistemas embarcados, este trabalho tem como objetivo desenvolver um processo para a criação de uma LPS para o domínio de *Robôs Moveis* (RMs) juntamente com técnicas de MDD. Técnicas de MDD foram utilizadas para criar uma Linguagem Específica do Domínio (*Domain-Specific Language - DSL*) e também para realizar transformações *Model-To-Model* (M2M) e *Model-To-Code* (M2C).

Um engenheiro de aplicação deve modelar os requisitos do RM a ser desenvolvido. Isso pode ser realizado por meio de um diagrama de *features* que representa um produto de uma LPS do domínio que o RM será desenvolvido. Para facilitar esse processo, neste trabalho foi criada uma DSL denominada F2MoC (*Feature-To-ModelorCode*) que reúne os conceitos de LPS e MDD, de modo que o engenheiro de aplicação tenha a sua disposição, de forma gráfica, todas as *features* pertencentes à LPS. Assim, ele escolhe quais dessas *features* são necessárias para o RM a ser gerado.

Após o processo de escolha, é importante também que haja uma documentação relacionada ao modelo criado. Nesse sentido a F2MoC permite que dois artefatos sejam gerados: um diagrama de classes correspondente às *features* escolhidas e a geração do código fonte a ser embarcado no RM.

Algumas das tarefas que devem ser realizadas para a obtenção desse objetivo são: o desenvolvimento de uma linha de produtos de software para o domínio de RMs; a construção de uma DSL gráfica (F2MoC) a partir dos requisitos dos clientes para a elaboração de um diagrama de *features*; a geração automática de código fonte Java para a API LeJOS a partir de um diagrama de *features* (M2C);

geração automática de um modelo de classes a partir de um diagrama de *features* (M2M);

Ao final deste trabalho, espera-se alcançar os seguintes benefícios:

1. Aumentar o percentual de reuso proporcionado pela utilização de técnicas de LPS e MDD no domínio dos *Mobiles Robots*;
2. Reduzir o tempo de desenvolvimento de um determinado membro pertencente à linha de produtos desenvolvida;
3. Facilitar a instanciação de *Robôs Moveis* a partir da DSL gráfica criada;
4. Automatizar a geração de código-fonte e modelos com base na DSL criada, tornando assim grande parte das tarefas do engenheiro de aplicação mais ágil;
5. Reduzir o custo de desenvolvimento dos *Robôs Moveis* pertencentes a linha de produtos;
6. Melhorar a qualidade dos *Robôs Moveis* pertencentes à linha de produtos, pois na teoria todos os artefatos disponíveis da linha já foram implementados e testados várias vezes.

1.3 Organização da Dissertação

Este trabalho está organizado em seis Capítulos, incluindo este, e dois Apêndices, além das referências bibliográficas.

No Capítulo 2 estão reunidos os principais conceitos que foram utilizados para que o objetivo deste projeto fosse alcançado. Dentre esses conceitos estão, sistemas embarcados, *Robôs Moveis*, linha de produtos de software, desenvolvimento de software orientado a modelos e algumas das principais ferramentas utilizadas na literatura para modelar diagramas de *features*.

No Capítulo 3 é apresentado o processo criado para auxiliar na construção de linhas de produtos de software para o domínio de *Robôs Moveis*. No Capítulo 4 esse processo é aplicado para exemplificar a sua utilização, realizando a criação de uma linha de produtos de software denominado *LegoMobileRobot Software Product Line* (LRMLPS), usando a DSL criada, denominada F2MoC.

No Capítulo 5 são descritos três exemplos que descrevem o processo de instanciação de um determinado membro pertencente à LRMLPS com o apoio da DSL gráfica criada.

No Capítulo 6 são apresentadas as considerações finais, as contribuições e limitações deste trabalho, bem como sugestões para trabalhos futuros.

O Apêndice A apresenta uma introdução à máquina virtual LeJOS, API que foi empregada para apoiar a criação dos artefatos durante o desenvolvimento da LRMLPS.

O Apêndice B contém uma introdução aos frameworks GMF e EMF os quais foram utilizados durante a implementação da DSL gráfica.

Capítulo 2

FUNDAMENTAÇÃO TEÓRICA

2.1 Considerações Iniciais

Neste Capítulo são abordados os conceitos fundamentais, necessários para compreensão do trabalho, juntamente com algumas pesquisas que serviram de embasamento. Dessa forma, na Seção 2.2 são apresentados os conceitos básicos sobre sistemas embarcados e *Robôs Moveis*, descrevendo seus propósitos. Na Seção 2.3 o conceito de padrões e linguagens de padrões é apresentado. Conceitos de linha de produtos de software são apresentados na Seção 2.4 e na Seção 2.5 são descritos conceitos sobre variabilidades de linha de produtos de software.

Na Seção 2.6 são abordados de forma concisa os conceitos sobre desenvolvimento de software orientado a modelos. Na Seção 2.7 são apresentadas algumas das ferramentas usualmente utilizadas academicamente para representam e modelar diagramas de *features* computacionalmente. E por fim, na Seção 2.8 são apresentadas as considerações finais.

2.2 Sistemas Embarcados

Wolf (2008) afirma que um sistema embarcado é um sistema computacional com um propósito especial que é utilizado em uma tarefa específica. Tais sistemas são, normalmente, menos poderosos (têm processamento restrito) que os sistemas computacionais de propósito gerais como, por exemplo, *desktop* e servidores.

A seguir são apresentadas as principais características dos sistemas embarcados de acordo com Marwedel (2006):

- Sistemas embarcados normalmente interagem com o ambiente em que se encontram, coletando dados de sensores e modificando o ambiente utilizando atuadores;
- Sistemas embarcados devem ser estritamente confiáveis, uma vez que falhas podem colocar em risco o ser humano;
- Baixo consumo de energia, uma vez que muitos sistemas embarcados possuem a característica de ser móveis e são alimentados por baterias, estes devem ser desenvolvidos para consumir pouca bateria;
- Tamanho dos códigos dos sistemas embarcados deve ser reduzido uma vez que a grande maioria desses sistemas não possui dispositivo de armazenamento;
- Devem respeitar requisitos de tempo real. Não cumprir uma tarefa no tempo correto acarreta a perda de dados e conseqüentemente de qualidade podendo até resultar em uma catástrofe;
- Muitos sistemas embarcados são híbridos, ou seja, possuem partes analógicas e digitais.

Assim, é possível identificar e perceber que atualmente, sistemas embarcados estão presentes em nosso cotidiano, de diversas formas e com diferentes objetivos. Este trabalho tem como foco os robôs móveis (RM). Dessa forma, na próxima seção é apresentada maiores informações sobre tais sistemas.

2.2.1 Robôs Moveis

RMs estão se tornando cada vez mais usuais em ambientes domésticos e profissionais. Podemos citar o aspirador de pó *IRobot Roomba* (Irobot, 2011) e o cortador de grama *Husqvarna Auto-mower* (Husqvarna, 2011) ambos totalmente autônomos.

RMs é uma área de pesquisa multidisciplinar que consiste em estudar técnicas de controle de robôs autônomos e tele operados na sua locomoção em ambientes dinamicamente instáveis. Tais ambientes, geralmente existem obstáculos móveis e fixos, e, portanto, o RM deve ser capaz de perceber, se localizar e se mover em ambientes complexos, tomando decisões por meio de informações providas dos mecanismos de sensoriamento (Fernandes, 2010).

Os RMs podem ser classificados entre três categorias, de acordo com o meio em que atuam, tais como:

- **Robôs aquáticos:** utilizados para explorar ambientes como, lagos, mares, etc. Dessa forma, tais sistemas possuem propulsores, balões de ar e sensores impermeáveis, os quais são utilizados para auxiliar que o robô permaneça submerso;
- **Robôs aéreos:** utilizados para fazer o reconhecimento de um determinado terreno, assim, usualmente são equipados com câmeras;
- **Robôs terrestres:** geralmente possuem esteiras, rodas ou pernas como dispositivo de locomoção. Esteiras são utilizadas em ambientes com solos irregulares, tais como, areias, pedregulhos, etc. Rodas são utilizadas em ambientes onde o solo é mais regular. Já robôs com pernas não só podem andar em superfícies irregulares como também podem subir e descer degraus.

Adicionalmente, tais robôs devem possuir sensores os quais são utilizados para conhecer o ambiente em que estão inseridos, tais sensores podem ser: táteis, de proximidade, de visão, entre outros. Sensores táteis são utilizados para detectar quando há um contato entre eles e um objeto. Sensores de proximidade medem a distância em que estão de um determinado objeto. Sensores de visão podem ser utilizados para auxiliar o robô a inspecionar objetos e obter informações em tempo real do ambiente que o robô está inserido. Existem ainda outros sensores que podem ser usados, como sensores de temperatura, de tensão e corrente elétrica, vazão e pressão de fluídos, etc.

Os RMs alvo de estudo neste trabalho são os robôs terrestres. Assim, os hardwares utilizados neste estudo é o kit da *Lego Mindstorms* (Lego, 2010).

Lego Mindstorms foi escolhido, pois o mesmo possui flexibilidade e usabilidade para tornar o desenvolvimento de um determinado RMs mais fácil. Possui ainda a capacidade de se comunicar utilizando o protocolo *bluetooth* e *wi-fi*. Outro fator importante na escolha do *Lego Mindstorms* é que esse kit é relativamente barato quando comparado com outros kits disponíveis no mercado para o desenvolvimento de RMs, tais como *Pionner P3-DX*, *Pionner 3-AT*, *PowerBot*, etc (Mobile, 2011). *Lego Mindstorms* também é um kit de robótica usualmente utilizado

para área educacional que permite a criação e programação de robôs. Para tal, utiliza-se de blocos de montar e deve-se definir o comportamento desejado por meio de sensores, motores e um microcontrolador.

Na Figura 2-1 é apresentada uma visão geral dos hardwares que acompanham esse kit.



Figura 2-1 - Kit Lego Mindstorms

O microcontrolador do *Lego Mindstorms* consiste de um processador ARM de 32 bits, 256 *kilobytes* de memória *flash* utilizada para armazenar os programas e 64 *kilobytes* de RAM (*Random Access Memory*) para armazenar dados durante a execução dos programas. Adicionalmente, possui *bluetooth* interno, o qual permite a interação entre vários dispositivos, tais como computador, celular, *wii remote*, etc.

Na Figura 2-2 é apresentada uma visão geral do microcontrolador da *Lego Mindstorms*.



Figura 2-2 – Microcontrolador NXT

O kit *Lego Mindstorms* possui quatro sensores (2 sensores de toque, 1 ultrassônico e 1 de luz) e três atuadores (Figura 2-1). O sensor de toque (Figura 2-3) detecta quando ele está sendo pressionado ou quando ele é solto.



Figura 2-3 - Sensor de Toque

Sensor de luz (Figura 2-4) permite ao RMs distinguir ambientes claros e escuros. Permite também identificar a intensidade da luz em um ambiente. Esse sensor possui a habilidade de identificar luzes invisíveis aos olhos humanos, tais como luzes de infravermelho. Esse sensor pode ser utilizado para executar uma variedade de funções, tais como seguir uma linha.



Figura 2-4 - Sensor de Luz

O sensor ultrassônico (Figura 2-5) permite ao RMs identificar a presença de objetos. Usualmente este sensor é utilizado para fazer com que o robô evite colidir com possíveis obstáculos.



Figura 2-5 - Sensor Ultrassônico

Na Figura 2-6 é apresentada uma visão geral de um atuador disponível no kit da *Legó Mindstorms*. Esse é utilizado para fazer com que o RMs atue em um ambiente, por exemplo, fornecer a habilidade do RMs se locomover.



Figura 2-6 - Exemplo de um Atuador

Uma das vantagens do kit *Legó Mindstorms* é que é possível utilizar várias linguagens de programação e API (*Application Programming Interface*) para auxiliar no desenvolvimento de robôs. Neste trabalho é utilizado a máquina virtual *Legó Java Operating System*³ (LeJOS) (Lejos, 2011). Essa máquina virtual permite executar código Java (Java, 2011) nos RMs, fornecendo assim a habilidade do programador utilizar paradigmas já consagrados para desenvolver aplicações no domínio dos

³ No apêndice A é descrito maiores detalhes sobre LeJOS e sua utilização.

RMs, tais como orientação a objetos. Na Figura 2-7 é apresentado um exemplo simples da utilização da máquina virtual LeJOS. Na linha 5 é ilustrado a instanciação da classe “*UltraSonicSensor*”, que representa uma abstração do sensor ultrassônico (Figura 2-4). Nas linhas 6 e 7 é ilustrado o método “*forward()*” o qual faz com que o atuador “*B*” e “*C*” se locomova para frente. Nas linhas 8 à 10 o método “*getDistance()*” da classe “*UltraSonicSensor*” é executado. Assim, enquanto a distancia for maior que 25 os autores “*B*” e “*C*” continuaram se locomovendo para frente, caso contrário o método “*stop()*” de cada atuador é chamado.

```
1 public class Exemplo {
2
3     public static void main(String... args){
4
5         UltraSonicSensor ultra = new UltraSonicSensor(SensorPort.S2);
6         Motor.B.forward();
7         Motor.C.forward();
8         while(ultra.getDistance() > 25)
9             //keep moving forward
10        }
11
12        Motor.B.stop();
13        Motor.C.stop();
14 }
```

Figura 2-7 – Exemplo da Utilização da API LeJOS

Desenvolver aplicações para RMs não é uma tarefa trivial, uma vez que os softwares desses sistemas podem aumentar sua complexidade exponencialmente (Wolf, 2008; Graaf *et al*, 2003). Dessa maneira, surge um desafio no desenvolvimento dos RMs, uma vez que o desenvolvimento desses sistemas usualmente é feito sem nenhuma técnica sistemática de reuso. Isso acarreta em um atraso considerado na entrega de um RMs (longo *time-to-market*), pois experiências anteriores não são reutilizadas fazendo com que o desenvolvedor tenha sempre que começar do zero (*from scratch*). Dessa forma, este trabalho tem como objetivo criar uma LPS no domínio de RMs. Outro objetivo é a criação de uma linguagem específica de domínio na qual é utilizada na atividade de Engenharia de Aplicação da LPS. Essa linguagem tem como objetivo agilizar e auxiliar o engenheiro de aplicação na instanciação de um determinado membro pertencente à LPS.

Na Seção 2-3, 2-4 e 2-5 são apresentados os conceitos fundamentais sobre padrões e linguagens de padrões, linha de produtos de software e desenvolvimento de software orientado a modelos.

2.3 Padrões e Linguagens de Padrões

Construir software não é uma tarefa trivial. Engenheiros de software são obrigados a lidar com muitas decisões durante o desenvolvimento de software. Desenvolvedores experientes conseguem construir software de maneira mais produtiva por possuírem conhecimento de soluções anteriores que podem ser aplicadas em situações similares. Desenvolvedores de software têm uma forte tendência em reutilizar soluções que foram satisfatórias para projetos (Beck *et al.*, 1996). Assim, sempre que um desenvolvedor tiver que lidar com novos problemas, o mesmo irá recorrer a esse repertório obtendo assim informações que já foram utilizados anteriormente. Padrões fornecem para os desenvolvedores exemplos a serem seguidos e artifícios a serem copiados e posteriormente refinados ou estendidos, padrões garantem uniformidade na estrutura do software, aumentando a produtividade no seu desenvolvimento e manutenção.

No entanto, para que tais soluções sejam utilizadas de forma correta por outros desenvolvedores, elas devem ser organizadas e documentadas apropriadamente. Um dos principais objetivos dos padrões é catalogar tais experiências para que todos os desenvolvedores possam utilizar tais experiências em outras aplicações, propiciando assim o desenvolvimento de aplicações de fácil manutenção e reutilizáveis (Buschmann *et al.*, 1996).

Originalmente padrões foram criados na área da arquitetura por Christopher Alexander (1977) que diz: “cada padrão descreve um problema que ocorre repetidas vezes em nosso ambiente e o cerne da sua solução, de tal forma que você possa usar essa solução mais de um milhão de vezes, sem nunca fazê-lo da mesma maneira”. Grand (2002) declara que as ideias apresentadas por Alexander são aplicáveis em vários campos além da arquitetura, incluindo o software.

A essência dos padrões é comunicar uma solução para determinado problema em um contexto (Gamma *et al.*, 1994). Em outras palavras, padrões podem ser considerados como uma maneira de encapsular a experiência de programadores, de forma que seja compreensível para outros programadores (Agerbo e Cornils, 1998).

Diversos formatos foram propostos para documentar os padrões de software, como por exemplo, o de *Portland*, o do *GoF*⁴, o formato de *Appleton* entre outros. No entanto, neste trabalho é apresentado somente o formato *GoF* (Gamma *et al.*, 1994) e é o utilizado quando necessário:

- **Nome e classificação do padrão:** o nome do padrão deve ser claro e conciso.
- **Intenção e objetivo:** tem o intuito de, resumidamente, informar o que o padrão de projeto deve fazer, sua razão, intenção e para qual problema de projeto o padrão propõe uma solução.
- **Também conhecido como:** informar outro nome ou lista de nomes pelo qual o padrão também é conhecido.
- **Motivação:** traz um cenário que ilustra um problema de projeto e como as classes e estruturas de objetos pertencentes ao padrão devem se comportar para solucionar o problema.
- **Aplicabilidade:** apresenta situações nas quais o padrão pode ser aplicado e considerado uma solução plausível.
- **Estrutura:** fornece uma representação gráfica dos participantes do padrão. Podem ser utilizados tanto modelos de classe e modelos de sequência entre outros.
- **Participantes:** ilustra as classes e possivelmente os objetos que participam do padrão de projeto e suas responsabilidades.
- **Colaborações:** ilustra como os participantes colaboram para realizar suas responsabilidades.
- **Consequências:** tem o propósito de informar sobre os custos e benefícios da utilização do padrão.
- **Implementação:** ressaltar as armadilhas, sugestões ou técnicas que devem ser consideradas quando o padrão for implementado.
- **Exemplo de código:** fragmentos de código ilustrando uma possível implementação do padrão.
- **Usos conhecidos:** exemplos do padrão encontrados em sistemas reais.

⁴ *GoF* significa *Gang of Four*; apelido atribuído aos autores do livro "*Design Patterns*" (Gamma *et al.*, 1994)

- **Padrões relacionados:** informa quais padrões de projeto estão intimamente relacionados ao padrão subjacente, quais as diferenças e com quais outros padrões pode se empregar.

De acordo com Gamma (*et al.*, 1994) padrões de projeto devem ser classificados por meio de dois critérios, que são, propósito e escopo. Assim, na Tabela 2-1 é apresentada tal classificação e definição de cada uma das categorias propostas. Vale ressaltar que, padrões de diferentes categorias podem ser utilizados em conjunto.

Tabela 2-1 – Classificação dos padrões de projeto (Adaptado de Gamma *et al.*, 1994)

Classificação			
Propósito		Escopo	
Criação	Padrões relacionados à instanciação de objetos.	Classe	Tratam da relação entre classes e suas subclasses. Ilustram uma configuração estática, definida em tempo de compilação.
Estrutural	Utilizados para composição de classes e objetos.	Objeto	Propõe a relação entre os objetos, podem ser alterados em tempo de execução.
Comportamental	Caracterizam a forma como as classes ou objetos interagem e distribuem as responsabilidades.		

O conjunto de padrões para um domínio de aplicação específico, junto com seus princípios estruturais, se transforma em uma linguagem de alto nível, chamada de linguagens de padrões (Alexander, 1977). Linguagens de padrões representam o conhecimento essencial de um domínio de aplicação específico organizado em uma coleção estruturada de padrões (Brugali e Sycara, 2000). Uma linguagem de padrões inclui regras e diretrizes que explicam como e quando aplicar os seus

padrões para resolver um problema que é maior do que qualquer padrão individual pode resolver (Appleton, 1997).

2.4 Linha de Produtos de Software

Linhas de Produtos de Software (LPS), ou família de produtos de software, corresponde a um conjunto de sistemas de software que compartilham características comuns e gerenciadas e que satisfazem a uma necessidade específica de um segmento particular de mercado, sendo desenvolvida a partir de um conjunto comum de ativos, de forma sistemática (Clements e Northrop, 2002). SPL é representada por produtos que, embora possuam requisitos em comum, ao mesmo tempo exibem variabilidade significativa (Griss, 2000). De acordo com Trifaux e Heymans (2004), as linhas de produtos fornecem para as organizações um aumento na reutilização de seus artefatos e tem uma diminuição dos custos e do tempo no desenvolvimento quando desenvolvem software por meio de LPS.

No desenvolvimento de LPS, são reconhecidas as semelhanças e variabilidade de um conjunto de software em um determinado domínio e é então executada uma Engenharia de Domínio (ED). Na ED são produzidos artefatos comuns (núcleo da linha) dos produtos da linha, a fim de maximizar o nível de reuso dos membros da linha, na atividade de Engenharia de Aplicação (EA). Desta maneira, grande parte da arquitetura pode ser reutilizada. Uma vez que as características comuns e variáveis da linha podem ser reutilizadas evitando assim esforços redundantes no desenvolvimento de cada software independente.

De acordo com Clements e Northop (2002) LPS estão sendo amplamente utilizadas na indústria automotiva, aeronáutica e eletrônica. Citam também que tais indústrias estão tendo grande sucesso com o uso de LPS.

Ao contrário dos métodos convencionais de desenvolvimento de software que desenvolvem os produtos de forma manual, LPS preocupa-se com a industrialização de processo de desenvolvimento. Isto inclui o aprendizado da configuração e montagem de componentes para a produção de produtos similares, integração e automação do processo de produção, desenvolvimento de ferramentas que

configuram e automatizam tarefas repetitivas. Caso esse processo for seguido corretamente, é possível obter os seguintes benefícios (Griss, 2000):

- **Aumento da qualidade:** uma vez que são feitas várias revisões e testes dos artefatos para vários produtos, permitindo que haja maiores chances de detecção e correção de erro.
- **Redução do custo de manutenção:** todas as mudanças que ocorrem nos artefatos são propagadas para vários produtos.
- **Evolução organizada:** novos artefatos geram oportunidades de evolução para vários produtos.
- **Menor complexidade:** utilizando reuso de código diminui a quantidade de código a ser mantido e também fornece suporte para a separação de funcionalidades em artefatos.
- **Novos produtos não são construídos do zero:** cada novo software é resultado da combinação de artefatos previamente desenvolvidos e testados que tornam seu desenvolvimento mais eficiente e sua qualidade melhorada.

Diversas abordagens e métodos para o desenvolvimento de linha de produtos de software existem na literatura, tais como: *Product Line Practice* (PLP) (SEI, 2010), *Product Line UML Based Software Engineering* (PLUS) (Gomaa, 2004), método PuLSE (Bayer *et al.*, 1999), *Feature-Oriented Domain Analysis* (FODA) (Kang, 1990), Family-Oriented Abstraction, Specification and Translation (FAST) (Weiss e Chi Tau, 1999), o método KobrA (Atkinson *et al.*, 2001). Porém neste trabalho é apresentado somente o PLUS, por ser o qual é utilizado neste trabalho.

2.4.1 Método PLUS

O método PLUS (*Product Line UML-Based Software Engineering*) foi proposto por Gomaa (2004) com base na UML (OMG, 2010) e composto pelo processo ESPLEP (*Evolutionary Software Product Line Engineering Process*). O ESPLEP é um modelo de processo de software que elimina a distinção tradicional entre desenvolvimento e manutenção de software, ao invés disso, o software evolui por meio de várias iterações. O processo ESPLEP é constituído por dois principais processos comentados a seguir e exibido na Figura 2-8:

1. **Engenharia de Linhas de Produtos:** As características comuns (*commonalities*) e as variabilidades (*variabilities*) são analisadas do ponto de vista geral dos requisitos da linha de produtos sendo construídos os modelos de casos de uso e de *features*, de análise, de arquitetura e de componentes reutilizáveis para essa linha. Testes são feitos nos componentes, bem como nas configurações da linha de produtos e os artefatos produzidos são armazenados em um repositório;
2. **Engenharia de Aplicação:** Durante esse processo, um membro da linha de produtos é desenvolvido a partir dos artefatos armazenados no repositório citado no processo anterior. Dados os requisitos da aplicação individual, o modelo de casos de uso da linha de produtos é adaptado para derivar o modelo de caso de uso da aplicação. A arquitetura da linha de produtos é adaptada para derivar a arquitetura da aplicação. Tendo a arquitetura da aplicação e os componentes apropriados do repositório, pode-se instanciar a aplicação desejada.

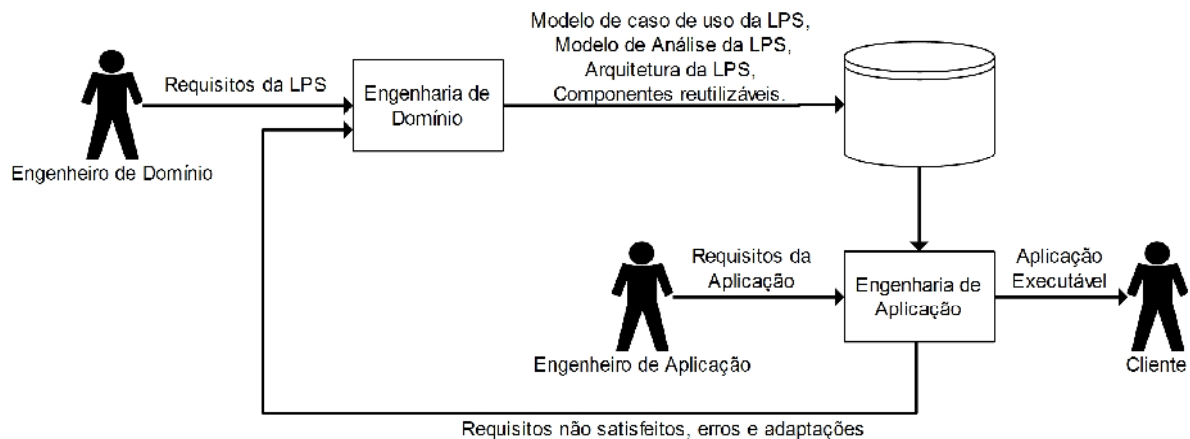


Figura 2-8 – ESPLEP (Adaptado de Gooma, 2004)

As atividades do processo ESPLEP são comentadas a seguir e exibidas na Figura 2-9:

- **Modelagem de requisitos de LPS:** elaboração de um modelo de requisitos que consiste em um modelo de casos de uso e um modelo de *features*. Esse modelo é útil para indicar semelhanças e variabilidades da LPS
- **Modelagem de análise da LPS:** construção de modelos estáticos e dinâmicos, em que o modelo estático define o relacionamento

estrutural entre as classes do domínio e o modelo dinâmico consiste em diagramas de estado e de interação.

- **Modelagem de projeto da LPS:** projeto de arquitetura da LPS e mapeamento do modelo de análise para o ambiente operacional.
- **Implementação incremental dos componentes de software:** a cada iteração, implementa um subconjunto de componentes selecionados, começando pelos casos de uso obrigatórios, seguido dos opcionais e alternativos;
- **Teste da LPS:** consiste na realização de testes de unidade e de integração nos componentes da linha de produtos.

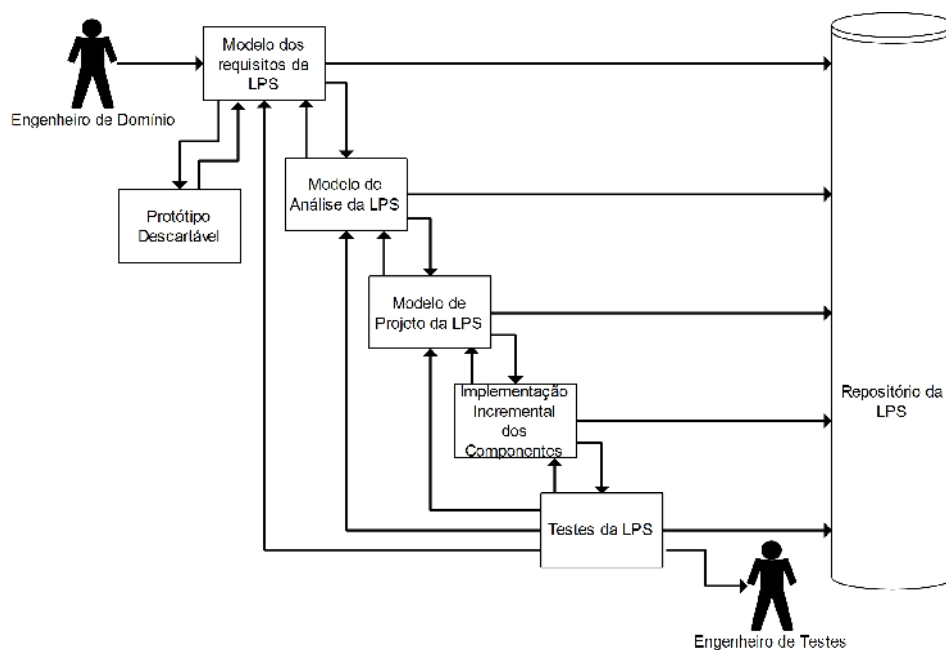


Figura 2-9 – PLUS (Adaptado de Gomaa, 2004)

2.5 Variabilidades em LPS

De acordo com van Gurp *et al* (2001) variabilidade é a forma como os membros de uma LPS podem se diferenciar entre si. A representação explícita de variabilidades torna possível a geração de produtos específicos de uma LPS. Segundo Bosch (2001), as variabilidades podem ser inicialmente identificadas e

representadas por meio do conceito de *features*⁵. Esse conceito teve origem na ED que classifica uma *feature* como uma característica de um sistema que é relevante e visível para o usuário final (Kang, 1990). Informalmente, *features* são informações utilizadas para distinguir um produto do outro. Griss (2000) caracteriza *features* como quaisquer conceitos proeminentes e distintos que são visíveis a vários *stakeholders*.

Usualmente *features* são representados por um modelo de *features* que consiste em uma representação hierarquia baseado em árvores na qual captura os relacionamentos estruturais entre as *features* de um domínio específico. *Features* comuns entre diferentes produtos são consideradas como *features* obrigatórias, enquanto que diferentes *features* podem ser consideradas opcionais ou alternativas. *Features* opcionais podem estar ou não presentes em um produto, quando presentes, adicionam algum valor às *features* obrigatórias de um produto. Duas ou mais *features* podem ser alternativas, porém somente uma pode estar presente em um determinado produto.

Os modelos de *features* são visualmente representados por meio de um diagrama de *features*, por exemplo, na Figura 2-10 é apresenta um diagrama de *features* para o domínio de hotel. Esse diagrama de *features* está representado utilizando a notação PLUS (Gomaa, 2004).

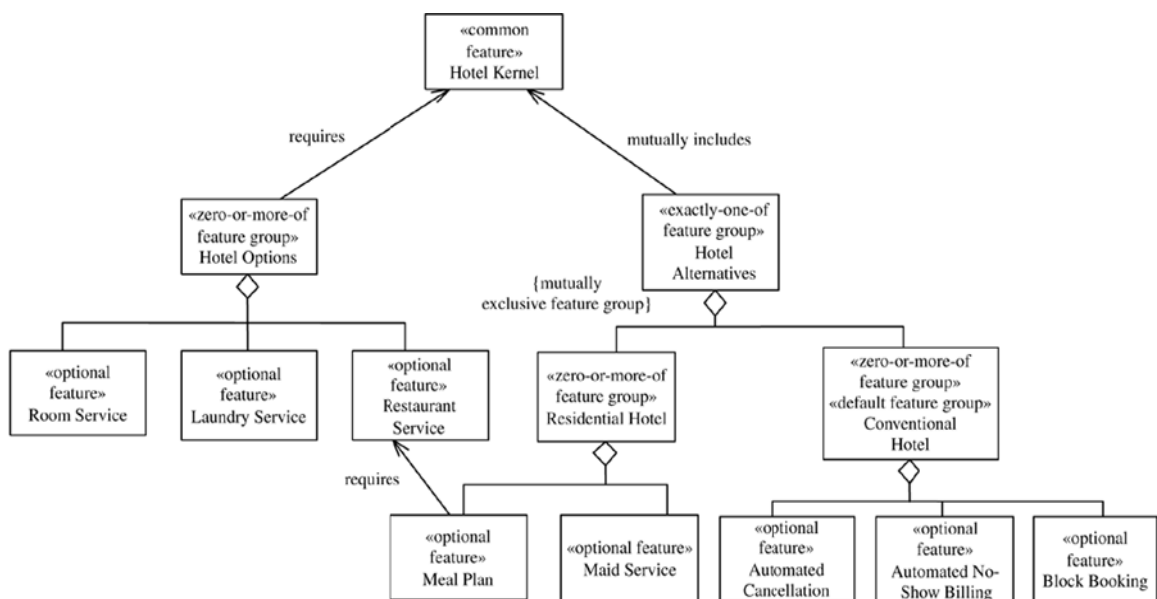


Figura 2-10 – Exemplo de Diagrama de *Features* (Gomaa, 2004)

⁵ Neste trabalho o termo *features* não é traduzido por entender que o correspondente no português, características, não traduz o que tecnicamente o termo *feature* significa.

Como pode ser observado na Figura 2-10 o método PLUS possui estereótipos para classificar e agrupar *features*. O estereótipo **<<common feature>>** classifica as *features* como obrigatórias, os estereótipos **<<optional feature>>** indicam *features* opcionais e o estereótipo **<<alternative features>>** (não apresentado na Figura 2-10) representa *features* alternativas. O método PLUS possui quatro estereótipos utilizados para auxiliar a agrupar as *features*. Por exemplo, o estereótipo **<<zero-or-one-of-feature group>>** indica que nenhuma ou somente uma *feature* pode ser selecionada do grupo, enquanto que **<<zero-or-more-of-feature group>>** representa que nenhuma ou várias *features* podem ser selecionadas. O estereótipo **<<exactly-one-of-feature group>>** indica que somente uma *feature* pode ser selecionada, por fim o estereótipo **<<at-least-one-of-feature group>>** no mínimo uma característica do grupo deve ser selecionada.

2.6 Desenvolvimento de Software Orientado a Modelos

A proposta do desenvolvimento de software orientado a modelos (do inglês *Model-Driven Development* – MDD) é reduzir a distância semântica entre o problema do domínio e a solução/implementação. Dessa forma, o engenheiro de software não precisa interagir inteiramente com o código-fonte, podendo se concentrar em modelos que possuem maiores níveis de abstração. Um mecanismo automático é responsável por gerar automaticamente o código-fonte por meio dos modelos (Figura 2-11). Nesse sentido, o MDD reconhece a importância dos modelos no processo de desenvolvimento de software, não apenas como um “guia” para tarefas de desenvolvimento e manutenção, mas como parte integrante do software (Gronback, 2009; Lucrédio *et al*, 2008; Völter, 2008).

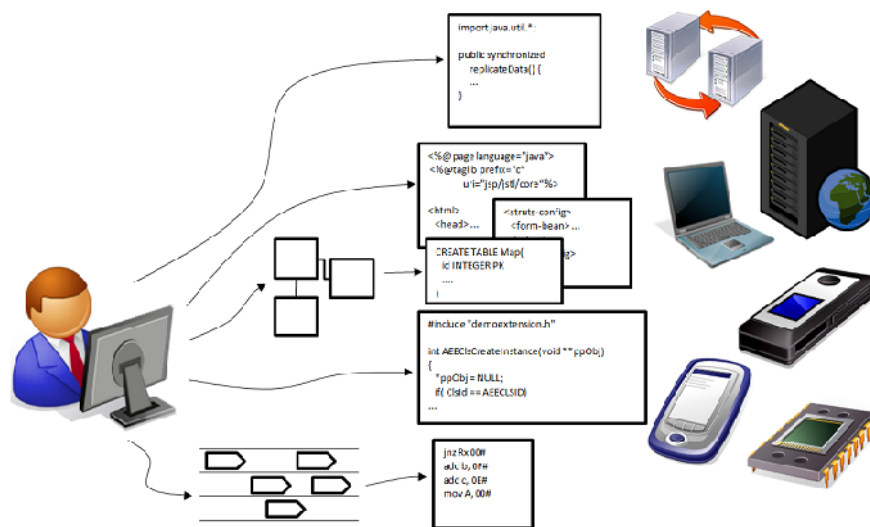


Figura 2-11 - Ilustração do processo de criação de software no desenvolvimento orientado a modelos (Lucrédio, 2009)

De acordo com Kleppe, Warmer e Bast (2003), Bahnot *et al.* (2005) e Mernik, Heering e Sloane (2005) MDD possui as seguintes vantagens:

- **Portabilidade:** O mesmo modelo pode ser transformado em muitas plataformas;
- **Interoperabilidade:** Cada parte do modelo pode ser transformado em código para diferentes plataformas, obtendo assim um software que executa em ambientes heterogêneo;
- **Produtividade:** Tarefas usualmente repetitivas podem ser implementadas como um mecanismo de transformação e geração automática, poupando assim, tempo e esforço que podem ser gastos em tarefas mais importantes;
- **Manutenção e documentação:** Modelos no MDD fazem parte do software, dessa forma, alterações são realizadas diretamente neles, fazendo com que os modelos permaneçam consistentes com o código-fonte. Assim, a documentação se manterá atualizada;
- **Comunicação:** No MDD diferentes profissionais possuem meios mais efetivos para se comunicar, sendo que os modelos geralmente são mais abstratos que código-fonte, não exigindo conhecimento técnico relativo à plataforma;
- **Verificação e otimização:** Modelos são mais fáceis de serem verificados semanticamente. Dessa forma, ocorre a redução de erros semânticos e fornece implementações mais eficientes;

- **Corretude:** Geradores não introduzem erros acidentais, tais como erros de digitação.

De acordo com Thomas (2004) e Ambler (2003) MDD possui algumas desvantagens:

- **Rigidez:** o MDD causa inflexibilidade no software gerado, uma vez que grande parte do código é gerado e fica além do “conhecimento” do desenvolvedor;
- **Desempenho:** Usualmente geradores geram código complexos e muitas vezes desnecessários e, portanto, o resultado pode não ser tão eficiente quando comparado com código escrito à mão;
- **Complexidade:** os artefatos necessários para o desenvolvimento baseado em modelos, tais como ferramentas de modelagem, geradores de código, entre outros, introduzem complexidade adicional ao processo;
- **Curva de aprendizado:** O aprendizado das ferramentas de modelagem e geradores, apesar de não ser relativamente complexos, requer um treinamento dedicado.

De acordo com Lucrécio (2009) para possibilitar a criação de modelos, é necessário uma ferramenta de modelagem. Essa ferramenta é representada por uma linguagem específica de domínio (*Domain-Specific Language* - DSL). Assim, na Seção 2.6.1 é descrito maiores informações sobre DSL.

2.6.1 Linguagem Específica de Domínio

De acordo com Fowler (2010) uma linguagem específica de domínio (DSL) é definida como uma linguagem computacional que é utilizada em um domínio particular com o intuito de realizar tarefas específicas. DSLs são usualmente pequenas, e declarativas, focadas em um domínio de problema em particular (Deursen *et al* 2000). Podem ser classificadas como internas, externas e gráficas.

DSL externa é uma linguagem que pode ser compilada, interpretada ou executada, e possui sintaxe própria ou usa uma representação pré-definida (ex., XML), porém, necessita de um *parser* para processá-las. DSL externas permitem liberdade para criar uma nova estrutura sintática e semântica. Existem várias ferramentas disponíveis na literatura para auxiliar a construção de tais DSL como:

ANother Tool for Language Recognition – ANTLR (ANTLR, 2010), Lex & Yacc (LeYc, 2010).

DSL interna pode ser entendida como uma maneira de se projetar uma API ou uma interface no domínio da aplicação, usando uma linguagem mais natural para o desenvolvedor, com palavras dentro de um contexto ou domínio específico (Fowler, 2010). Geralmente DSLs internas são construídas com o auxílio de linguagens dinâmicas, tais como Lisp, Smalltalk, Ruby, Python e Boo.

DSL gráficas como o próprio nome sugere utiliza formas e linhas para representar informações do domínio ao invés de textos. Tais DSLs são utilizadas para expressar informações de uma maneira concisa e mais abstrata, proporcionando melhor entendimento e visualização do problema. Também fornecem comunicação em alto nível, uma vez que o engenheiro trabalha com elementos visuais e não textuais. É mais fácil de identificar e entender o que acontece em uma DSL gráfica do que uma DSL interna ou externa. Existem várias ferramentas que permitem desenvolver DSL gráficas, tais como, *Generic Modeling Environment* (GME) (GME, 2010), *DSL Tools plugin* para a plataforma *Visual Studio* da *Microsoft* (Visual, 2010) e *Eclipse Modeling Framework* (EMF) (EMF, 2010). Este trabalho descreve somente o framework EMF e é o utilizado quando necessário (Seção 2.6.1.1).

A utilização de DSLs apresenta vantagens e desvantagens. Dentre as vantagens é possível citar (Lucrédio, 2010; Deursen *et al*, 2000):

- DSLs permitem que soluções sejam expressas no nível de abstração do domínio do problema. Conseqüentemente, especialistas do domínio podem compreender, validar, modificar ou mesmo desenvolver seus próprios programas;
- DSLs aumentam a produtividade, confiabilidade, manutenibilidade e portabilidade;
- DSLs são baseadas no conhecimento do domínio, e, portanto, possibilitam sua conversão e reutilização;
- DSLs possibilitam validação e otimização em nível de domínio.

Segundo Deursen *et al* (2000) as desvantagens de se utilizar DSLs são:

- Alto custo ao projetar e manter uma DSL;

- Alto custo na capacitação de usuários que usarão a DSL, ou seja, curva de aprendizado alta;
- Dificuldade de se definir um escopo adequado para uma DSL;
- Para os casos de DSLs executáveis, a perda potencial de desempenho quando comparado com código-fonte desenvolvido à mão.

2.6.1.1 Eclipse Modeling Project

Eclipse Modeling Framework (EMF) (EMF, 2010) é um framework para modelagem e geração de DSLs (Gronback, 2009). O EMF segue um meta-modelo denominado *Ecore* que surgiu como uma implementação desenvolvida pela OMG⁶, porém, evoluiu para um forma mais eficiente, a partir de experiências obtida após alguns anos na construção de ferramentas (Eclipse, 2010). Na Figura 2-12 é apresentada uma visão geral do projeto *Eclipse Modeling* (Eclipse, 2010). Como pode ser observado EMF representa o *core*, ou seja, o *framework* base para o desenvolvimento de uma determinada DSL. A seguir apresentam-se descrições e objetivo de cada um dos *frameworks* do projeto Eclipse utilizados neste trabalho:

- **Graphical Modeling Framework (GMF)**: permite a definição completa de um ambiente de modelagem para uma DSL gráfica. Por meio da definição do meta-modelo da linguagem, da aparência gráfica dos elementos visuais dessa linguagem, e das ferramentas necessárias para criar os elementos, é gerado um ambiente completo. Esse ambiente permite ao engenheiro de software construir modelos utilizando a DSL gráfica definida.
- **Model-To-Text (M2C)**: Fornece a capacidade de realizar transformações de modelos para códigos-fontes, para tal o projeto Eclipse disponibiliza o framework *Java Emitter Templates* (JET). JET consiste de um mecanismo de geração de código-fonte, no qual utiliza *templates* como base. Inclusões de código Java nesses *templates*, permitem a metaprogramação, ou seja, a criação de programas que criam programas. O JET usualmente é acoplado a arquivos XML ou modelos EMF, sendo portanto possível utilizá-lo como gerador de código para uma DSL;

⁶ www.omg.org

- **Model-To-Model (M2M):** Fornece a capacidade de realizar transformações de um modelo mais abstrato para outro modelo específico e vice versa, para tal o projeto Eclipse disponibiliza o framework ATL⁷ (*Atlas Transformation Language*).

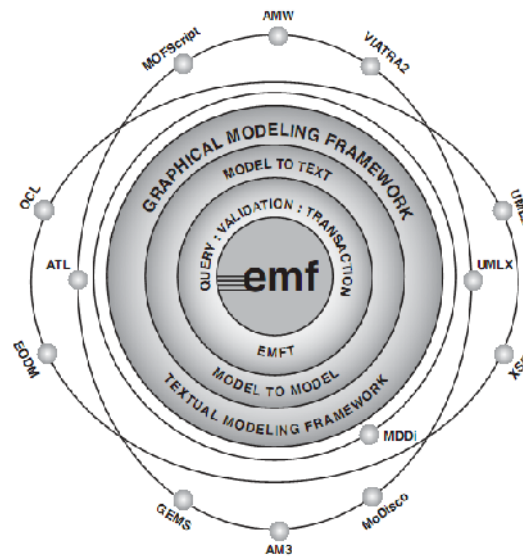


Figura 2-12- Visão Geral do Projeto Eclipse *Modeling*

2.7 Ferramentas Utilizadas para Modelar Diagramas de *Features*

Na literatura é possível identificar várias ferramentas utilizadas para auxiliar na modelagem de diagramas de *Features*. Dessa forma, essa seção apresenta uma visão geral e objetiva de algumas dessas ferramentas as quais foram analisadas com o intuito de obter conhecimento e avaliar qual a melhor forma de se representar diagramas de *features* computacionalmente. Adicionalmente, tais conhecimentos foram utilizados para auxiliar no desenvolvimento de uma DSL (ver Capítulo 5) que é utilizada na atividade de Engenharia de Aplicação na qual auxilia na instanciação de um determinado membro da linha de produtos.

⁷ <http://www.eclipse.org/atl/>

2.7.1 Pure::variants

Pure::variants (Pure, 2011) é um ferramenta utilizada para realizar a modelagem de diagrama de *features*, ela foi criada pela companhia *pure-systems*⁸. Essa ferramenta é feita como base no Projeto *Eclipse Modeling Project* (Seção 2.6.1.1), seu principal objetivo é ser utilizado como uma ferramenta para projetar e modelar arquiteturas de uma respectiva LPS. Na Figura 2-13 é apresentada uma visão geral da ferramenta *Pure::variants*. Como observado todas as *features* são agrupadas em uma lista. Essa lista é composta por várias caixas de seleção (*checkbox*), que são utilizadas para configurar a linha de produtos. Adicionalmente, *Pure::variants* possui a capacidade de representar os modelos graficamente (Figura 2-14), porém esse modelo visual é utilizado somente para ter uma visão geral do diagrama de features da linha de produtos desenvolvida, ou seja, a ferramenta não permite realizar ações (deletar, editar e criar) sobre o diagrama.

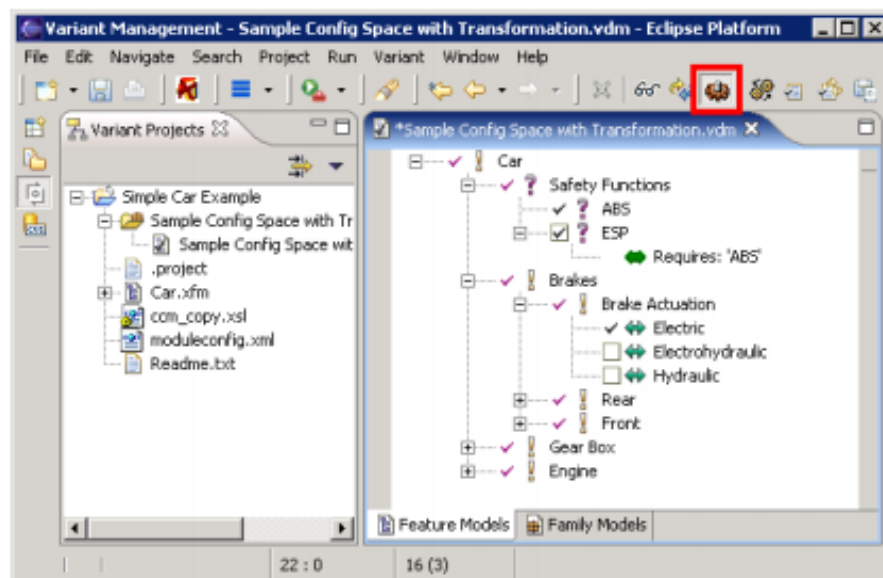


Figura 2-13 - Visão Geral do *Pure::Variants*

⁸ http://www.pure-systems.com/pure_variants.49.0.html

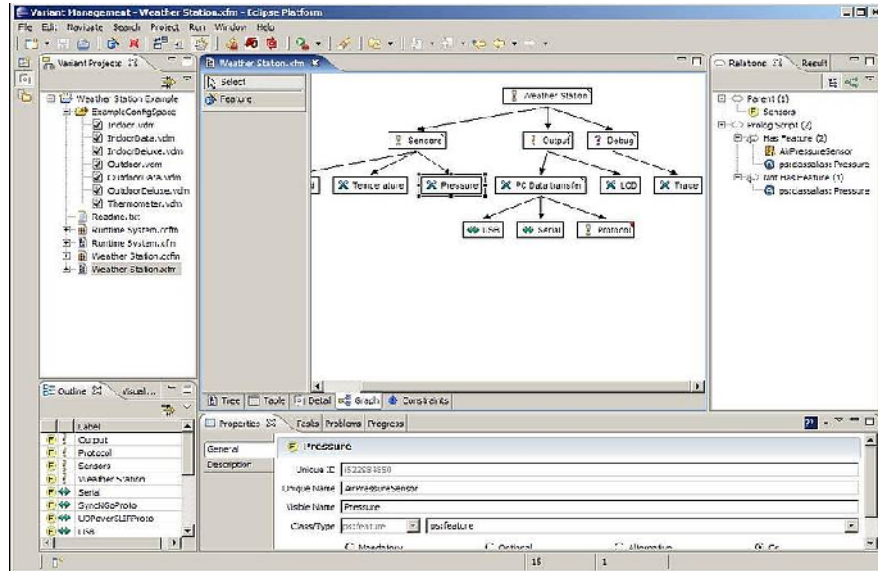


Figura 2-14 - Visualização Gráfica do Pure::Variants

2.7.2 Feature Modeling Plug-in for Eclipse

Essa ferramenta é um *plugin* desenvolvido para ser utilizado também no IDE Eclipse. As *features* são representadas por uma lista, essa lista agrupa todas as *features* disponíveis da linha de produtos. Dessa forma, cada nó da lista representa uma determina *features* e seus relacionamentos são adicionados a outro nó abaixo de uma respectiva *feature*. Na Figura 2-15 é apresenta uma visão dessa ferramenta.

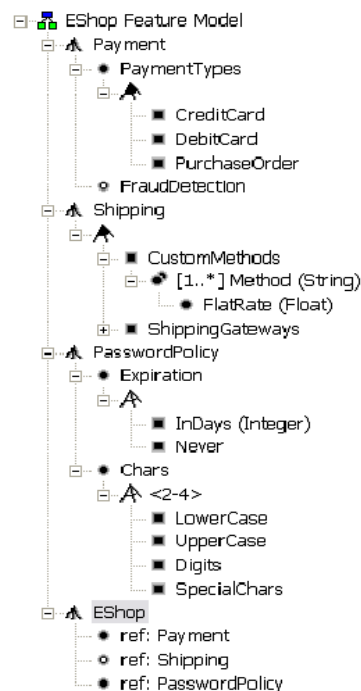







Figura 2-15 - Feature Modeling plug-in for Eclipse

Como pode ser observado na Figura 2-15, o símbolo  representa um conjunto de *features* raiz (“Payment”, “Shipping”, “PasswordPolicy” e “EShop”). O símbolo  representa que a *features* “PaymentTypes” é obrigatória enquanto que símbolo  indica que a *feature* “FraudDectection” é opcional. O símbolo  indica que pelo menos uma *feature* deve ser selecionada, porém pode-se selecionar mais do que uma *feature*. Por fim o símbolo  representa que somente uma *feature* pode ser selecionada (Czarnecki e Antkiewicz, 2004).

2.7.3 Feature Modeling Tool

Essa ferramenta é feita com base no IDE Visual Studio⁹ e permite a criação de diagramas de *features* utilizando a ação arrastar e soltar (*Drag-and-Drop*). Adicionalmente, esta ferramenta possui duas representações do diagrama de *features*, por meio de uma lista, similar às ferramentas já apresentadas e graficamente representadas por componentes. Na Figura 2-16 é apresentada uma visão dessa ferramenta.

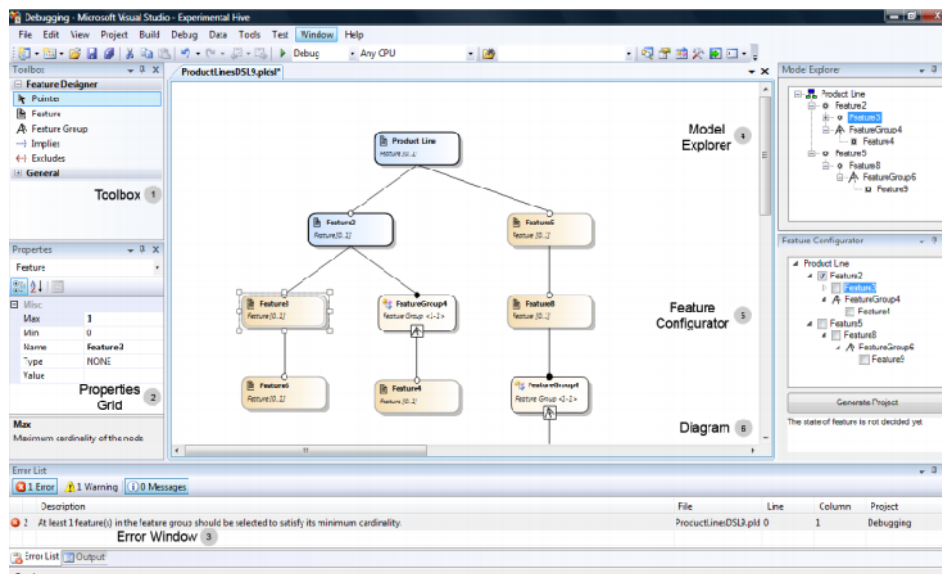


Figura 2-16 - Representação da Ferramenta *Feature Modeling Tool*

A janela principal (Figura 2-16) representa o diagrama de *features* que está sendo desenvolvido, na qual é permitido adicionar, modificar e deletar *features*. A representação do diagrama de *features* é feita com base na notação FODA proposta

⁹ <http://www.microsoft.com/visualstudio/en-us/>

por Kang (1990), ou seja, cada nó representa uma determinada *feature* e cada aresta representa o relacionamento das *features*.

2.7.4 XFeature

A ferramenta XFeature investiga a construção de uma ferramenta para auxiliar e modelar LPS (XFeature, 2011) .

Essa ferramenta também foi desenvolvida utilizando o projeto *Eclipse Modeling*, assim, a mesma deve ser executada somente na IDE Eclipse. XFeature¹⁰ representa o diagrama de *features* utilizando uma representação hierárquica baseada em árvores, onde os nós representam as *features* e as arestas representam a composição hierárquica (Figura 2-17).

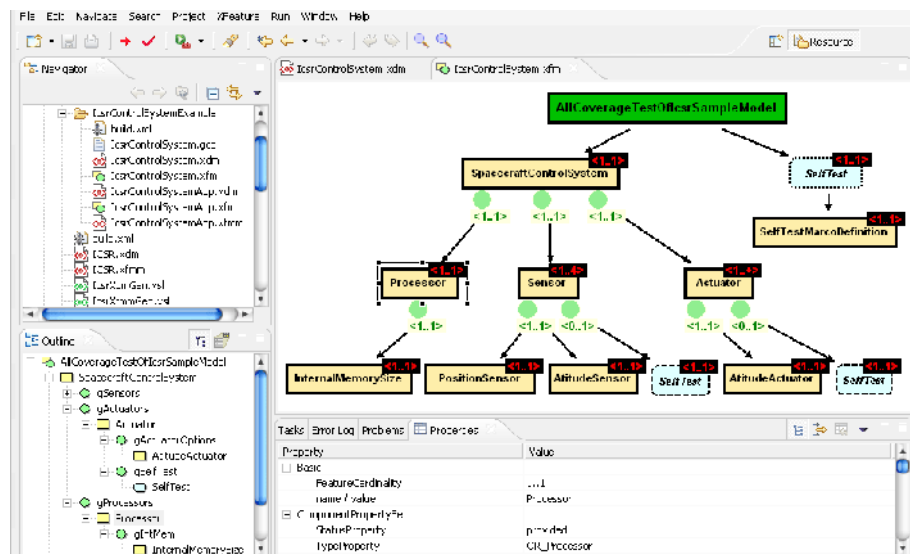


Figura 2-17 – Visão Geral da Ferramenta XFeature

Como pode ser observado na Figura 2-17 o engenheiro deve indicar qual a cardinalidade de uma determinada *features*. Essa cardinalidade informa qual o tipo que a *feature* representa, por exemplo, *features* opcionais possuem a cardinalidade $<0...1>$ e *features* obrigatórias possuem a cardinalidade $<1...1>$.

¹⁰ <http://www.pnp-software.com/XFeature/>

2.8 Considerações Finais

Neste Capítulo foi apresentado uma visão geral e pragmática dos principais conceitos inerentes a sistemas embarcados, *Robôs Moveis*, linha de produtos de software, desenvolvimento de software orientado a objetos. Tais conceitos foram utilizados para realizar este trabalho, ou seja, para a criação da linha de produtos no domínio dos *Robôs Moveis* e criação de um DSL que é utilizada na Engenharia de Aplicação para auxiliar na instanciação automática de um membro da linha.

Esse Capítulo também apresentou uma visão geral do projeto *Eclipse Modeling*, o qual é utilizado para criar uma DSL que servirá de base para auxiliar na instanciação de membros pertencentes à linha de produtos desenvolvida. Adicionalmente, também foram apresentadas algumas ferramentas que são utilizadas para modelar diagramas de *features*. Essa ferramenta serviram de base para auxiliar na criação da DSL criada nesse trabalho.

O Capítulo seguinte apresenta uma abordagem para o desenvolvimento de linha de produtos no domínio de *Robôs Moveis*, apresenta também informações necessárias para compreensão dessa abordagem.

Capítulo 3

UM PROCESSO PARA CRIAÇÃO DE LPS NO DOMÍNIO DE ROBÔS MOVEIS

3.1 Considerações Iniciais

Sistemas embarcados são utilizados em vários dispositivos que fazem parte da vida cotidiana, de modo que o mercado de tais sistemas tem crescido de maneira expressiva. Um subconjunto dos sistemas embarcados que também está se tornando cada vez mais comum em ambientes domésticos e profissionais são os Robôs Moveis (RMs).

O desenvolvimento do software dos RMs é a principal parcela. Dessa forma, ocorre um atraso considerável na sua produção (*time-to-market*), aumento de erros (*error-prone*) e manutenibilidade dos RMs. Esse atraso juntamente com aumento a tendência de erros estão relacionados com a produção desse software, já que o hardware vem sendo cada vez mais considerado uma *commodity*, desde a introdução de projetos baseados em plataformas (Sangiovanni-Vincentelli e Martin, 2001). Desse modo, metodologias/técnicas consagradas de Engenharia de Software estão sendo cada vez mais utilizadas para modelar softwares nesse domínio. Dentre essas técnicas, Linha de Produtos de Software (LPS) (Polzer *et al*, 2009; Kim, 2006; Lee *et al*, 2005; Ubayashi e Nakajima, 2010) tem chamado a atenção, principalmente em consequência dos resultados promissores alcançados em outros domínios (automotivo, aviônico, entre outros). Durante a última década LPS tem sido considerada uma técnica promissora para aumentar a qualidade e reduzir o tempo de produção no desenvolvimento de software (Polzer *et al*, 2009).

Este Capítulo tem por objetivo descrever um processo iterativo e evolutivo para a criação de uma LPS no domínio dos *Robôs Moveis*, com oito atividades detalhadas nas próximas seções.

Na Seção 3.2 é apresentado a Engenharia de Domínio (ED) enquanto que na Seção 3.3 é apresentado a Engenharia de Aplicação (EA) do processo proposto de desenvolvimento de LPS para o domínio de RMs, respectivamente. Na Seção 3.4 são apresentadas as considerações finais do Capítulo.

3.2 Processo Para o Desenvolvimento Linha de Produtos de Software para Robôs Moveis

Esta seção apresenta um processo iterativo e evolutivo (Figura 3-1) para o desenvolvimento de uma LPS para *RMs*, dividido em duas principais atividades: Engenharia de Domínio (ED), que envolve a criação e manutenção dos artefatos da linha, e Engenharia de Aplicação (EA), que utiliza os artefatos criados na atividade anterior para auxiliar a instanciação de produtos (Clements e Northrop, 2002).

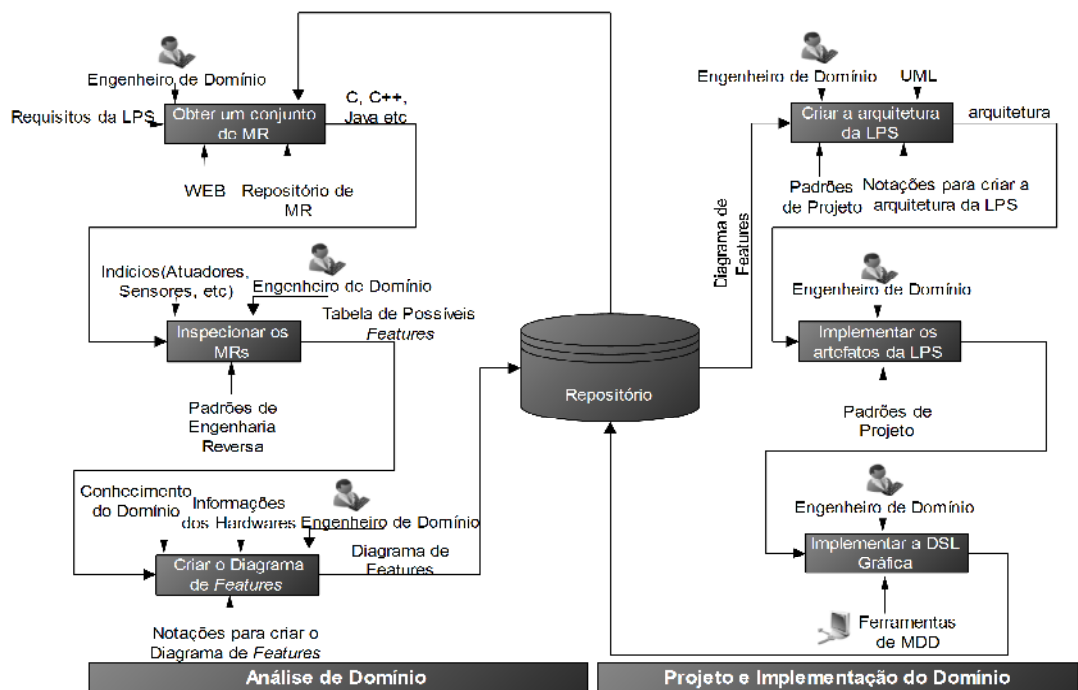


Figura 3-1- Processo para criação de LPS para Robôs Moveis

A notação *Structured Analysis and Design Technique* (SADT) (Ross, 1978) é utilizada para representar graficamente as atividades que devem ser realizadas no processo proposto. Os retângulos representam as atividades, as setas que entram no lado esquerdo de cada retângulo representam as entradas de dados, as setas ao lado direito representam a saída de dados. As setas no topo do retângulo representam os controles que influenciam internamente cada etapa e as setas na base de cada retângulo representam os participantes e as ferramentas utilizadas em cada atividade.

As seções a seguir apresentam as atividades da ED que devem guiar o engenheiro de domínio na criação de uma LPS para o domínio dos RMs.

3.2.1 Engenharia de Domínio do Processo Proposto

A Engenharia de Domínio (ED) é o processo de identificação e organização do conhecimento acerca de uma classe de problemas. O objetivo da ED é sistematizar a criação de modelos do domínio, arquiteturas e conjuntos de artefatos de software para dar suporte à construção de aplicações em um domínio de problema particular (Heymans, 2003).

Do lado esquerdo da Figura 3-1 são apresentados as atividades definidas para a ED, do processo proposto, enquanto que do lado direito são apresentados as atividades para a EA.

As atividades do processo são realizadas iterativamente, conforme apresentado na Figura 3-2. Dependendo da iteração, as atividades têm menor ou maior intensidade, por exemplo, no começo do processo, as atividades “*Obter um conjunto de RM*”, “*Inspecionar os RMs*” e “*Criar o Diagrama de Features*” são mais enfatizadas. Isso ocorre, pois está trata-se da identificação das características comuns e variáveis da LPS. Porém, as atividades “*Criar a arquitetura da LPS*”, “*Implementar os artefatos da LPS*”, e “*Implementar a DSL Gráfica*” são enfatizadas nas últimas iterações do processo.

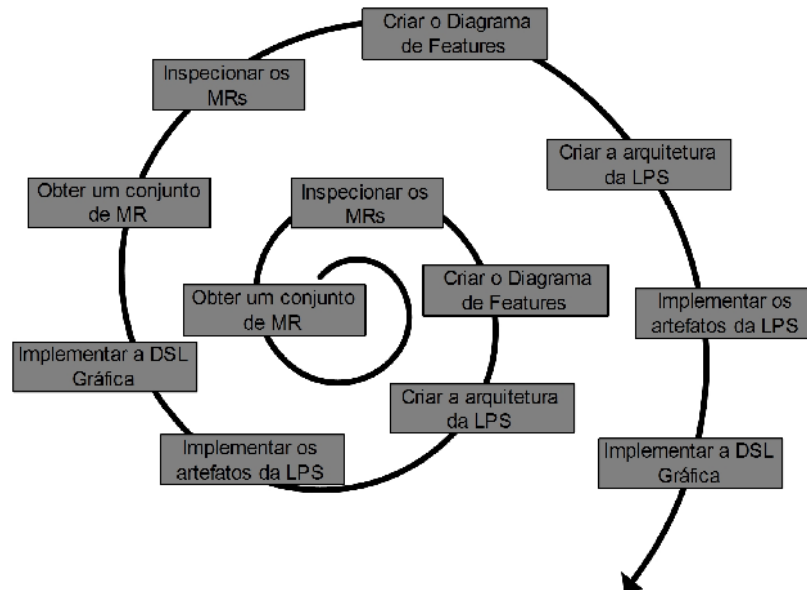


Figura 3-2- Modelo em Espiral do processo; adaptado de (BOEHM, 1986)

3.2.1.1 Obter um Conjunto de RM

A ED inicia-se com a atividade de obtenção de um conjunto de sistemas, utilizado para auxiliar a identificação das *features* comuns e variáveis da LPS. Gomma (2004) afirma que a identificação das *features* não deve ser feita exclusivamente por meio de códigos-fontes. Desse modo, os seguintes documentos podem ser considerados para esta atividade: código fonte, documentos de requisitos, diagrama Simulink e até mesmo casos de uso, etc. Tais artefatos podem ser coletados de repositórios ou por meio da Web (*World Wide Web*).

Para garantir melhor identificação das *features* da LPS pelo menos três sistemas pertencentes ao domínio devem ser coletados (Clements e Northrop, 2002). Se isso não ocorrer, a LPS pode não ter todo o seu potencial alcançado e o custo de sua concepção ser alto e não justificável.

Após coletar os artefatos, o engenheiro do domínio deve começar a identificação das *features* comuns e variáveis entre eles. A próxima seção descreve a atividade responsável pela identificação dessas *features*.

3.2.1.2 Inspeccionar os RMs

Esta atividade tem como objetivo auxiliar na identificação progressivamente das similaridades e funcionalidades dos artefatos coletados na atividade “*Obter um conjunto de RM*”. Usualmente as identificações das similaridades e funcionalidades

são feitas por meio de comparações de aplicações pertencentes ao domínio ou por meio da descrição de caso de usos ou pela execução da própria aplicação (Gomma, 2004).

No processo proposto, como pode ser observado no diagrama SADT da Figura 3-1, padrões de engenharia reversa (Demeyer *et al*, 2002) são utilizados para apoiar a identificação das similaridades da linha. No entanto, a identificação de similaridades não é uma tarefa trivial e requer um bom conhecimento do domínio no qual se pretende desenvolver a linha de produtos. Dessa maneira, na Figura 3-3 são ilustradas esquematicamente três diretrizes propostas que devem auxiliar o engenheiro de domínio no procedimento de identificação das *features* da LPS. Cada acrônimo “*Dn*” representa uma diretriz, detalhada a seguir. Maiores informações sobre essas diretrizes podem ser encontradas em Durelli *et al* (2010).

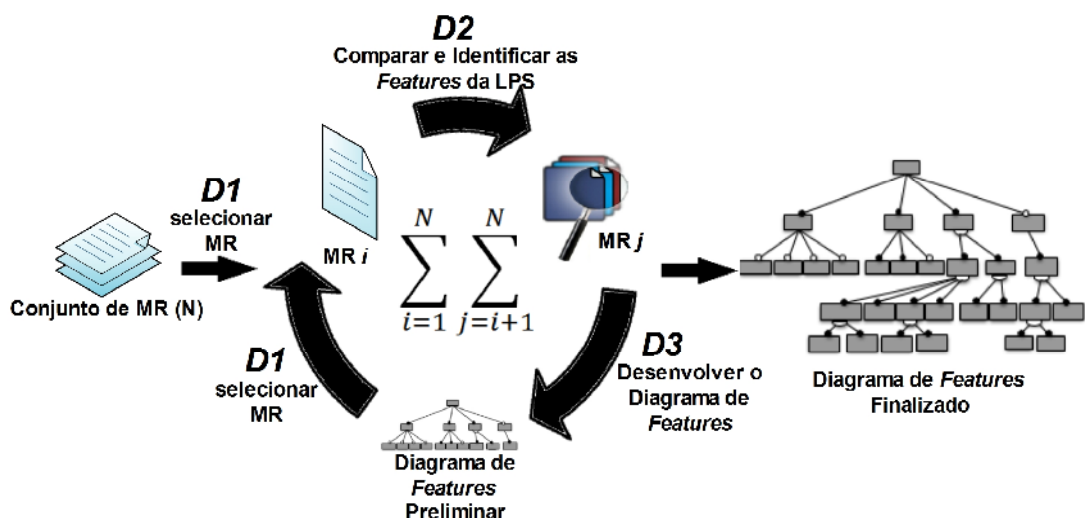


Figura 3-3 - Diretrizes para identificações das *Features* da LPS

- 1. Selecionar RM:** Primeiramente deve-se ter um conjunto de sistemas coletado, denominado por RM_i , sendo $i=1, \dots, N$ com N igual ao número total de sistemas. Em seguida, o engenheiro de domínio cria uma lista de indícios de *features* com base no manual e nos hardwares desses sistemas. Por exemplo, “motor”, “sensores” e “dispositivo de locomoção” são possíveis indícios do domínio de RMs. Em seguida, o engenheiro de domínio deve selecionar o primeiro RM_i para começar a comparação e identificação das *features* da LPS. Posteriormente as

diretrizes D2 e D3 são aplicadas nos RMs coletados como mostra o ciclo ilustrado na Figura 3-3.

2. Comparar e Identificar as *Features* da LPS: A partir da lista de indícios criada anteriormente, o engenheiro de domínio inicia a identificação das possíveis *features* dos sistemas. Essa identificação pode utilizar padrões de engenharia reversa como, *Read All the Code in One Hour* e *Skin the Documentation* (Demeyer *et al.*, 2002). Caso o RMi possua um diagrama de classe, a identificação inicia com a aplicação do padrão *Skim the Documentation* da seguinte forma: (i) para cada classe do modelo do RMi, verifica-se a existência de indícios de acordo com os indícios da lista de indícios – (ii) um indício existe em uma classe se a classe possui uma das seguinte relações: herança, agregação e/ou composição com esse indício. Caso o RMi não possua um diagrama de classe deve-se aplicar o padrão *Read All the Code in One Hour*, para verificar a existência de indícios, para cada classe, com base na lista de indícios anteriormente criada. Nesse caso, uma classe possui um indício, se o mesmo aparece por meio de herança e/ou como tipo de um atributo (composição e agregação). Uma Tabela de *Features* Candidatas (TFC) que agrupa as *features* identificadas no domínio também é um dos artefatos criados nessa diretriz. Adicionalmente, tais *features* devem ser classificadas como obrigatórias opcionais e/ou alternativas: (i) classificam-se as *features* que aparecerem em todos os RMs como obrigatórias; (ii) classificam-se as *features* que aparecem em alguns RMs como opcionais; (iii) classificam-se as *features* como *alternative xor* quando somente uma dessas *features* é encontrada em cada RMi; (iv) classificam-se as *features* como *alternative or* quando duas ou mais dessas *features* são encontradas em pelo menos um dos RMs. Por exemplo, na Tabela 3-1 é apresentado um exemplo do artefato TFC, para ilustrar esse artefato é considerado o domínio de uma industria automobilística, onde é possível escolher as *features* de um determinado automóvel. Cada linha da tabela representa uma *feature*. As colunas contém o nome da *Feature*, sua categoria e o seu tipo o nome de seu pai (Nó_Pai), o

nome de seus filhos (Nó(s) Filho(s)), uma breve descrição das *features* e, por fim, o nome do sistema do qual a *feature* foi extraída. Ressalta-se que a cobertura da identificação das *features* pode ser maior se for feita de maneira incremental. *Features* não identificadas no primeiro ciclo podem ser encontradas nos ciclos seguintes.

- 3. Desenvolver o Diagrama de *Features*:** A partir do artefato TFC o engenheiro de domínio constrói um diagrama de *features*. O processo para essa construção é detalhada na atividade “Criar o Diagrama de *Features*”.

Tabela 3-1 – Exemplo de Tabela de *Features* Candidatas (TFC)

Nome <i>Feature</i>	Categoria <i>Feature</i>	Nó Pai	Nó(s) Filho(s)	Descrição das <i>Features</i>	Aplicações pertencentes do domínio		
					A1	A2	A3
Automóvel	Obrigatória	--	Transmissão/Ar Condicionado	Característica raiz da linha			
Transmissão	Obrigatória	Automóvel	Manual/Automática	Representa o que automóvel possui transmissão			
Ar condicionado	Opcional	Automóvel	--	Representa um determinado Ar Condicionado do Automóvel			
Manual	Alternativa	Transmissão	--	Representa que o tipo da transmissão é manual			
Automática	Alternativa	Transmissão	--	Representa uma transmissão Automática			

Após inspecionar todos os artefatos obtidos e criar a tabela TFC da LPS, deve-se iniciar a modelagem e implementação da linha, por meio das atividades:

“Criar o Diagrama de *Features*”, “Criar a Arquitetura da LPS”, “Implementar os artefatos da LPS”, e “Implementar a DSL gráfica”.

3.2.1.3 Criar o Diagrama de *Features*

Esta atividade é responsável por criar um diagrama de *features* da LPS que represente as *features* existentes nas aplicações de um determinado domínio, com base no artefato TFC juntamente com notações encontradas na literatura. Neste trabalho o diagrama de *features* é utilizado como um modelo semântico para auxiliar o engenheiro de aplicação configurar múltiplos produtos tendo como base a LPS.

O engenheiro do domínio pode usar vários métodos como notação para auxiliar a criação do diagrama de *features*, tais como: o método **Product Line UML based Software engineering** (PLUS) (Gomaa, 2005), FODA (*Feature-Oriented Domain Analysis*) (Kang *et al.*, 1990), entre outros. Neste trabalho a notação proposta por Gomaa (2004) é a utilizada na Figura 3-4 para representar o diagrama de *features* criado com base na Tabela 3-1. PLUS foi escolhido, pois o mesmo tem por base a UML, no qual é uma linguagem amplamente difundida na comunidade de Engenharia de Software, facilitando assim o uso e o entendimento do diagrama de *features*. A *feature* raiz é representada pela primeira linha do artefato TFC. Adicionalmente, o desenvolvimento do diagrama de *features* segue informações contidas nas colunas **Nó Pai**, **Nó(s) Filho(s)** e **Categoria Feature** do artefato TFC. Por exemplo, a *feature* “Automóvel” deve ter um relacionamento com as *subfeature* “Transmissão” e “Ar Condicionado” conforme indica a coluna **Nó(s) Filho(s)**. Posteriormente deve-se utilizar a coluna **Categoria Feature** para classificar tais *subfeatures* entre obrigatórias, opcionais e/ou alternativa. Este processo deve ser realizado por todo o artefato TFC.

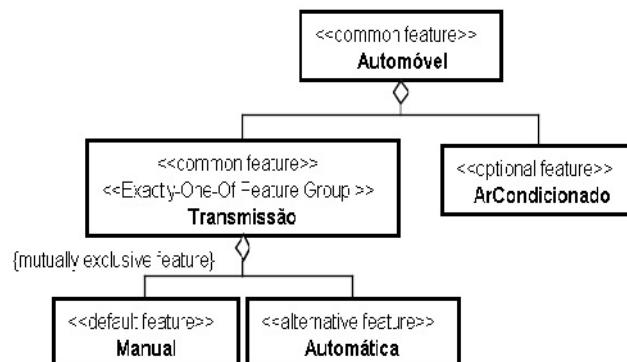


Figura 3-4 - Modelo de *Features* criado com base na Tabela de *Features* Candidatas

Uma vez que o engenheiro de domínio desenvolveu o diagrama de *features*, a próxima atividade consiste no desenvolvimento da arquitetura da LPS, que é descrita na próxima seção.

3.2.1.4 Criar a Arquitetura da LPS

Atividade responsável por auxiliar o engenheiro de domínio a criar a arquitetura, modelo estático, da linha de produtos de software, que neste trabalho é representada pelo diagrama de classe da UML (OMG, 2010). Na arquitetura são representadas as *features* comuns e variáveis existentes em uma linha de produtos (Gomma, 2004). Em um modelo estático de um único sistema, todas as classes são necessárias, já em uma LPS uma classe é necessária para pelo menos um membro da linha, mas pode não ser necessária para todos os membros dessa LPS. Dessa forma, alguma notação específica deve ser utilizada para criar o modelo estático da linha. Neste trabalho a arquitetura da LPS utiliza a notação do método PLUS (Gomaa, 2004).

O desenvolvimento da arquitetura da LPS é feito tendo como base o diagrama de *features* desenvolvido na atividade anterior. Dessa maneira, a tradução do diagrama de *features* para a arquitetura da LPS é feita com o auxílio de padrões de projeto (Gamma *et al.*, 1994) juntamente com a UML. Para tal, Almeida *et al* (2007), descrevem diretrizes de quando e como os padrões de projeto podem ser utilizados para representar as *features* na arquitetura da LPS. Na Tabela 3-2 é apresentada uma relação entre os tipos de *features* e os possíveis padrões de projeto utilizados para representar tais *features* na arquitetura da LPS. A coluna “*Tipo de Feature*” representa os tipos que geralmente uma *feature* pode ser representada e a coluna “*Padrões de Projeto*” representa quais padrões podem ser utilizados para modelar tais *feature* para um modelo estático, o qual representa a arquitetura da linha.

Tabela 3-2 – Relação entre Tipo de *Features* e Padrões de Projeto

Tipo de <i>Features</i>	Padrões de projeto
<i>Features</i> Alternativas	Abstract Factory juntamente com Singleton; Factory Method; Prototype juntamente com Singleton; Strategy e Template Method
<i>Features</i> Opcionais	Builder; Decorator e Observer

Como descrito na coluna “Tipo de *Features*” as *features* alternativas podem ser representadas no modelo estático por meio dos padrões *Abstract Factory* e *Singleton*, *Factory Method*, *Prototype* e *Singleton*, *Strategy* ou *Template Method*. De maneira semelhante, as *features* opcionais podem ser representadas pelos seguintes padrões de projeto: *Builder*, *Decorator* e *Observer*.

Na Figura 3-5 é apresentado o mesmo modelo de *features* ilustrado na Figura 5, porém as *feature* “*Transmissão*” e “*ArCondicionado*” possuem um fragmento que ilustra como é feita a tradução do diagrama de *features* para diagrama de classe, usando a notação PLUS juntamente com as diretrizes propostas por Almeida *et al* (2007).

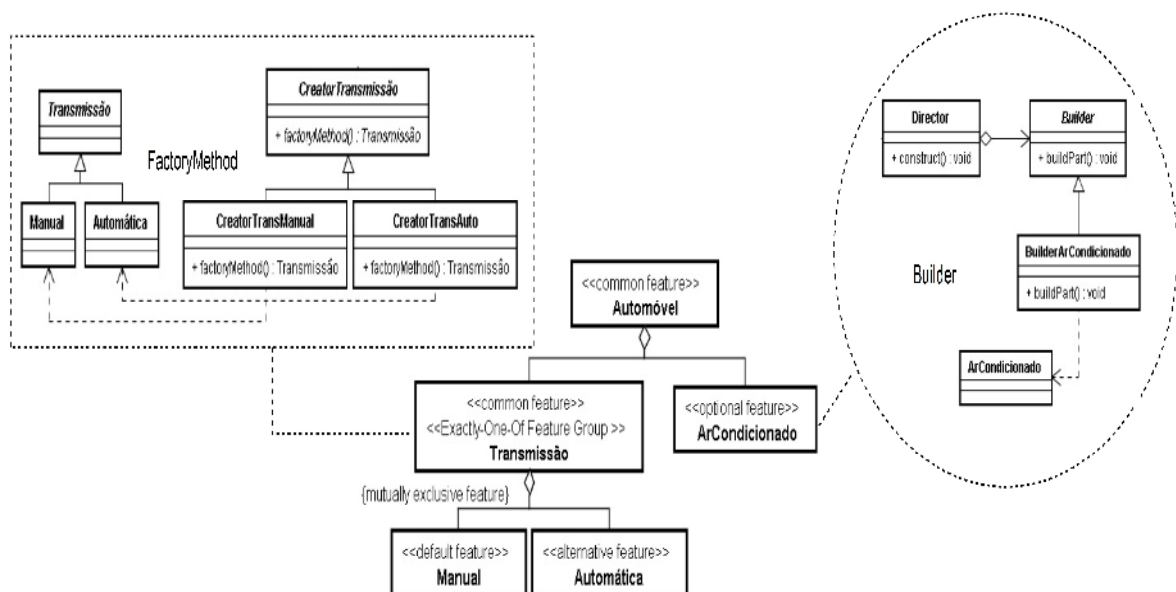


Figura 3-5 - Modelo Estático Modelado Utilizando Padrões de Projeto

Ao finalizar a arquitetura da linha de produtos de software, o engenheiro de domínio deve iniciar a implementação dessa linha de produto, como tratada na próxima seção.

3.2.1.5 Implementar os Artefatos da LPS

A atividade “Implementar os Artefatos da LPS” recebe como entrada a arquitetura da LPS, diagrama de *features* e documentos de requisitos. O engenheiro de domínio deve escolher a linguagem de programação que será utilizada para implementar os *assets* da LPS e também decidir qual abordagem será utilizada para implementar as variabilidades dessa LPS. Na literatura pode-se encontrar várias abordagens para implementar variabilidades como, por exemplo,

Aggregation/Delegation, Inheritance, Parameterization, Overloading, Delphi properties, Dynamic Class Loading, Static Libraries, Dynamic Link Libraries, Conditional Compilation, Frames, Reflection, Aspect oriented programming, e Design Patterns (Anastasopoulos e Gacek., 2001). O processo descrito neste trabalho utiliza padrões de projeto para auxiliar o engenheiro de domínio a implementar as variabilidades e as características comuns da LPS.

O engenheiro de domínio também precisa de uma IDE (*Integrated Development Environment*) para sintetizar os artefatos identificados e criados nas atividades anteriores para código-fonte. A estratégia adotada para implementar as variabilidades da linha segue a seguinte ordem:

1. Implementar os artefatos obrigatórios;
2. Implementar os artefatos alternativos;
3. Implementar os artefatos opcionais.

A saída desta atividade consiste em um conjunto de arquivos que estão contidos nos códigos-fonte e que serão utilizados para a instanciação de um produto dessa linha.

3.2.1.6 Implementar a DSL Gráfica

Esta atividade tem como objetivo preparar um ambiente que facilite o reuso da LPS. Esse ambiente é representado por uma DSL gráfica a qual é utilizada pelo engenheiro de aplicação para instanciar um determinado produto.

O engenheiro do domínio utiliza ferramentas para a construção do metamodelo da DSL e dos seus *templates*. Algumas dessas ferramentas, como o *Graphical Modeling Framework* (GMF) (GMF, 2011) da IDE Eclipse (Eclipse, 2011), permitem tanto a definição dos elementos e dos relacionamentos existentes em um metamodelo, quanto a criação de sua representação gráfica. Adicionalmente devem ser construídos *templates* em uma linguagem de transformação como, por exemplo, *eXtensible Stylesheet Language* (XSL) (XSL, 2011) ou *Java Emitter Templates* (JET) (JET, 2011). Tais *templates* são utilizados posteriormente pela DSL para gerar transformações *Model-To-Model* (M2M) e *Model-To-Code* (M2C).

As principais fontes de informações utilizadas para auxiliar o engenheiro de domínio a criar o metamodelo da DSL são as *features* identificadas nas atividades anteriores.

A construção do metamodelo da DSL é realizada por meio dos seguintes passos:

1. Criar uma metaclasses abstrata chamada *Feature* e outra metaclasses chamada *Relationship*;
2. Inserir no metamodelo da DSL, para cada *feature* pertencente ao diagrama de *features*, uma metaclasses com o mesmo nome da *feature*;
3. Inserir no metamodelo da DSL, os relacionamentos existentes entre os seus elementos tomando por base os relacionamentos já existentes no diagrama de *features*;
4. Inserir no metamodelo da DSL um tipo enumerado que representa o tipo de uma *feature*: obrigatória, opcional e/ou alternativa.

O metamodelo exibido na Figura 3-6 ilustra os passos acima descritos e foi desenvolvido com base no diagrama de *features* apresentado na Figura 3-4.

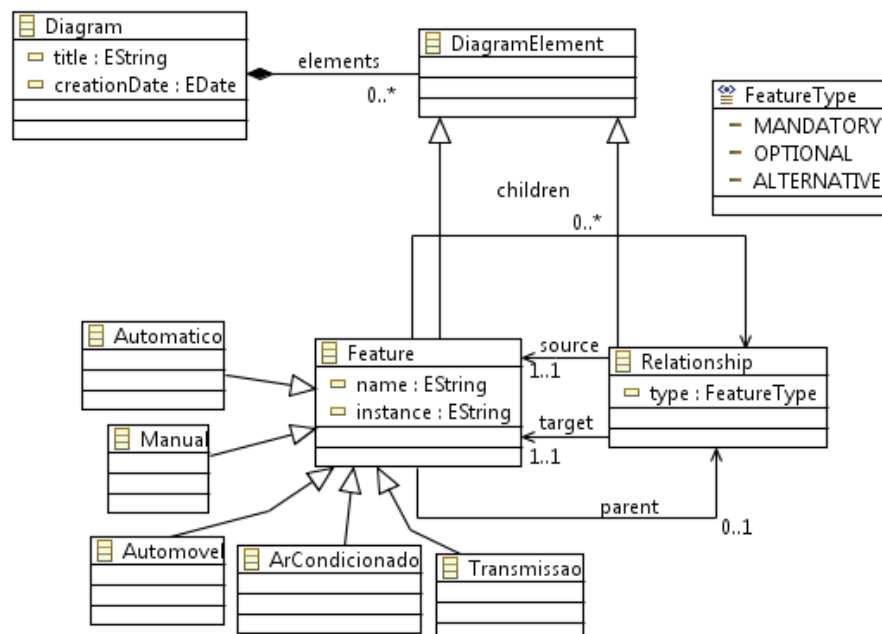


Figura 3-6 – Metamodelo desenvolvido tendo por base o diagrama de features

A metaclasses *Diagram* armazena os dados gerais dos modelos criados a partir desse metamodelo. A metaclasses *DiagramElement* representa todos os elementos que estarão disponíveis na DSL e que serão apresentados graficamente. Nesse caso, os relacionamentos entre as *features* representado pela metaclasses *Relationship* e as *features* ilustradas pelas submetaclasses de *Features*. A metaclasses *Feature* contém informações relacionadas aos nomes das *features* e de

seus respectivos relacionamentos. As submetaclases *Manual*, *Automatico*, *Automovel*, *ArCondicionado* e *Transmissão* representam as *features* do diagrama de *features* apresentado na Figura 3-4. Por fim, a enumeração *FeatureType* representa os tipos que uma determinada *feature* pode ser.

Após a modelagem do metamodelo da DSL deve-se iniciar a construção de como tais metaclases serão representadas graficamente, ou seja, a definição da sintaxe concreta dos elementos da DSL gráfica. Usualmente, formas geométricas, como retângulo e elipse, ou imagens, podem ser utilizadas para representar os elementos da DSL de forma mais intuitiva e natural. As seguintes regras devem ser utilizadas na criação dos elementos da DSL gráfica no processo descrito neste trabalho:

- As *features* do diagrama de *features* devem ser representadas como blocos retangulares;
- E os relacionamentos entre as *features* devem ser representadas por meio de linhas.

No Apêndice B é apresentado uma visão geral da criação dos elementos visuais de uma DSL gráfica.

Após finalizar a construção do metamodelo da DSL deve-se iniciar a construção dos *templates* com a utilização de uma linguagem de transformação.

Os *templates* são dependentes tanto do metamodelo da DSL quanto do conteúdo que podem dar origem, por isso, normalmente, cada *template* está vinculado a um elemento ou a um relacionamento existente no metamodelo da DSL.

Um *template* é formado por partes imutáveis e independentes da aplicação que está sendo instanciada e por partes que variam de acordo com as informações provenientes do modelo da aplicação. A melhor tática para construí-los é analisar o código específico de aplicações para identificar cada uma dessas partes.

Além dos *templates* responsáveis pela geração do código específico da aplicação, podem ser construídos *templates* para a criação do pacote da aplicação bem como a cópia de outros artefatos pré-fabricados que possam ser necessários para o funcionamento da aplicação.

3.2.2 Engenharia de Aplicação do Processo Proposto

A EA (Engenharia de Aplicação) dedica-se ao estudo das melhores técnicas, processos e métodos para a construção de aplicações, tendo como base o reuso de artefatos. Na EA, os componentes de software previamente desenvolvidos na ED são reutilizados para o desenvolvimento das aplicações do domínio do problema considerado (Weiss e Lai, 1999).

Conforme ilustra o diagrama SADT da Figura 3-7, a EA proposta neste trabalho é dividida nas atividades: “Configurar Features” e “Gerar Membro da LPS”. O engenheiro de aplicação utiliza a DSL gráfica, os artefatos e os *templates* da LPS que foram construídos na etapa de ED para dar origem a membros da LPS

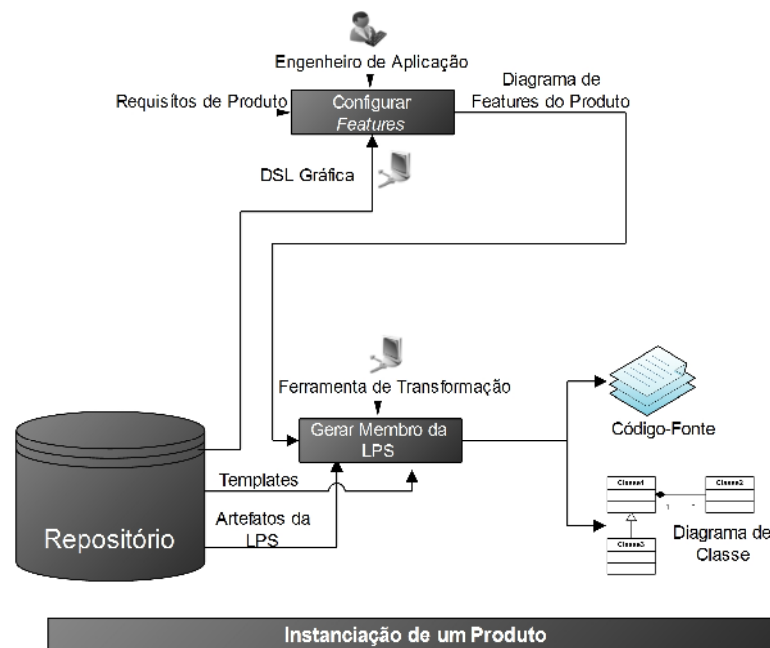


Figura 3-7 - Engenharia de Aplicação

O processo de instanciação de um membro da LPS inicia-se com a obtenção dos requisitos do sistema a ser instanciado. Deve-se verificar se tais requisitos fazem parte da LPS. Se um ou mais requisitos não forem satisfeitos, verifica-se a viabilidade da reconfiguração da LPS para a adição desses requisitos na linha de produtos. Caso seja viável deve-se voltar para a atividade ED novamente tomando tais requisitos como guia para criar as novas *features* comuns e variáveis da linha. Caso não seja viável, cabe ao engenheiro de aplicação juntamente com o engenheiro de domínio decidir se irão desenvolver uma nova LPS ou um *único sistema* para cobrir tais requisitos. No entanto, caso os requisitos pertençam à linha

de produtos criada, o engenheiro de aplicação deve utilizar a DSL criada e derivar um diagrama de *features* mapeando tais requisitos. Assim a DSL gráfica criada utiliza esse diagrama de *features* para realizar transformações M2M e M2C do produto modelado.

3.3 Considerações Finais

A ideia de reuso de software não é nova, em uma conferência da OTAN foram apresentadas as primeiras ideias referentes a reuso de componentes de software produzidos em massa e os possíveis benefícios que tal abordagem traria à chamada “crise de software”. Desde então, a pesquisa sobre reuso de software evoluiu, produzindo uma variedade de trabalhos na área. Infelizmente, muitas dessas técnicas de reuso de software não são aplicadas para SE. Isto cria um grande desafio para o desenvolvimento de SE. Assim, para otimizar o *time-to-market*, a produtividade, e a qualidade do desenvolvimento de SE companhias devem tentar aplicar técnicas consagradas de reuso no domínio de SE. Vários autores (Kim, 2006; Lee *et al.*, 2005; Polzer *et al.*, 2009; Ubayashi e Hirayama., 2010) utilizam LPS para o domínio de SE para tentar alcançar a limitações anteriormente citadas.

Tendo isso em mente, este trabalho apresentou um processo que visa à construção de linha de produtos de software no domínio de RM a partir de produtos previamente desenvolvidos.

O processo apresentada também fornece dicas de projeto e implementação usando ferramentas do ambiente Eclipse, possibilitando a criação de uma linguagem específica de domínio e a geração automática de código e diagrama de classes por meio de *templates*. Este trabalho apresentou a utilização de técnicas de MDD para auxiliar a instanciação automática de um determinado RM. No Capítulo 4 é apresentado uma linha de produtos de software intitulada *LegoRobosMoveis Linha de Produtos de Software* que foi desenvolvida utilizando o processo aqui proposto.

Capítulo 4

LEGO ROBOMOVEIS SOFTWARE SOFTWARE PRODUCT LINE

4.1 Considerações Iniciais

A abordagem apresentada no Capítulo 3 para auxiliar o desenvolvimento de LPS para o domínio de RMs foi utilizado no desenvolvimento da linha de produtos de software intitulada *LegoRobosMoveis Linha de Produtos de Software* (LRMLPS).

Essa linha foi desenvolvida utilizando o Kit da LEGO Mindstorms (Lego, 2011) juntamente com a máquina virtual Lego Java Operating System (LeJOS) (Lejos, 2011).

LEGO Mindstorms é um kit de robótica para área educacional que permite a criação e programação de robôs. Para tal, utiliza-se de blocos de montar e deve-se definir o comportamento desejado por meio de sensores, motores e um microcontrolador. Maiores informações sobre o Lego Mindstorms podem ser encontradas no Capítulo 2, Apêndice A ou em <http://mindstorms.lego.com/en-us/Default.aspx>. Neste contexto, nas seções 4.2 a 4.7 são apresentadas as atividades que compõem o processo apresentado no Capítulo 3. Na Seção 4.8 é apresentado a Engenharia de Aplicação da LRMLPS. E por fim, na Seção 4.9 são apresentadas as considerações finais desse Capítulo.

4.2 Obter um Conjunto de RM

Esta atividade é responsável por obter um conjunto de artefatos, utilizado para auxiliar a identificação das features comuns e variáveis da LPS. Esses artefatos não precisam ser necessariamente aplicações implementadas, podem ser, por

exemplo, por código-fonte, diagrama de Simulink, labVIEW, documentos de requisitos ou de casos de uso, etc..

A construção da LRMLPS segue o processo apresentado no Capítulo 3 e começou com a aquisição de quatro sistemas distintos pertencentes ao domínio de RMs: RM1 – chamado de *Explorer*, utilizado para explorar ambientes hostis, RM2 – chamado de *Measurer* responsável por medir terrenos, RM3 – *Charger* atua como um robo de chão de fábrica e RM4 – *HomeSentry*, utilizado para monitorar uma determinada residência.

As seções seguintes descrevem os passos realizados para a identificação progressiva das similaridades e variabilidades da LRMLPS desses RMs.

4.3 Inspeccionar os RMs

O engenheiro de domínio deve inspeccionar os sistemas coletados na atividade anterior e seguir as diretrizes apresentadas na Figura 3-3 do Capítulo 3 para auxiliar o engenheiro a identificar as possíveis *features* da linha. A seguir é descrito como tais diretrizes foram aplicadas para desenvolver a LRMLPS.

4.3.1 Selecionar RM

Os RMs coletados na atividade anterior foram numerados de 1 até 4. Posteriormente após a consulta ao hardware desses RMs foi criada uma lista de indícios com base nos seus sensores, atuadores, dispositivo de comunicação e de locomoção exibidos na Figura 4-1.

1. Sensores;
 - a. Ultrassônico
 - b. Toque
 - c. Luz
 - d. Câmera
2. Atuadores;
3. Dispositivo de locomoção
 - a. Pneu
 - b. Esteira
 - c. Caster
4. Dispositivo de Comunicação
 - a. *Bluetooth*
 - b. Wi-Fi

Figura 4-1 – Lista de Indícios

Em seguida cada um dos RMs selecionados são comparados com a lista de indícios para a identificação de suas *features* comuns e variáveis da LMSPL, como descrito a seguir.

4.3.2 Comparar e Identificar as Features da LPS:

Diretriz responsável por comparar e identificar as *features* comuns e variáveis dos RMs anteriormente coletados. Os padrões de engenharia reversa *Read All the Code in One Hour* e *Skim the Documentation* (Demeyer *et al.*, 2002) juntamente com a lista de indícios criada anteriormente foram aplicados nos RMs para identificar as *features* do domínio. Assim, para cada diagrama de classe dos RMs apresentados na Figura 4-2 foram aplicados o padrão *Skim the Documentation*.

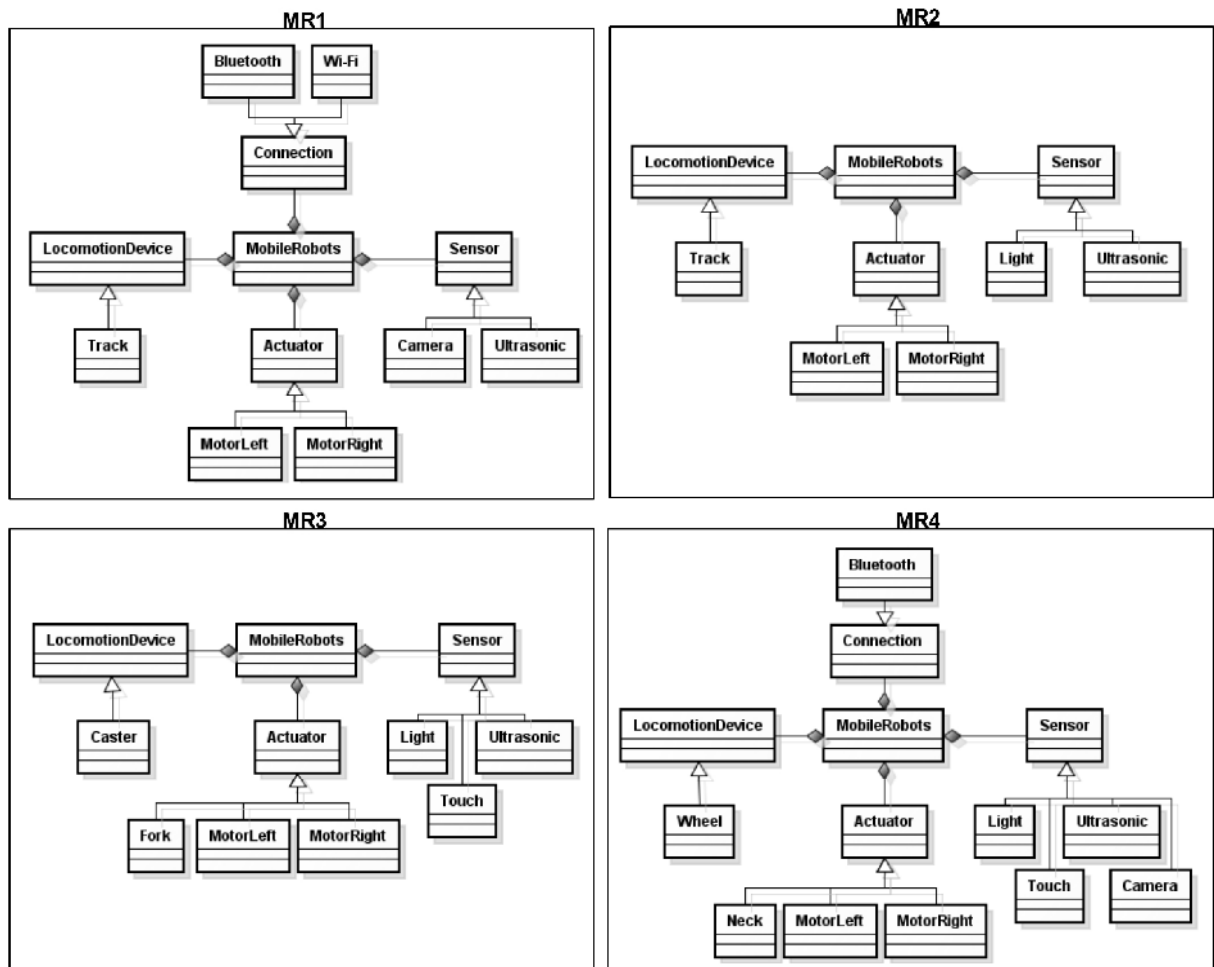


Figura 4-2 – Diagrama de Classe dos RMs

O padrão Skim the Documentation foi aplicado da seguinte forma: (i) em cada classe do modelo, verificou a existência de indícios pertencentes à lista de indícios criada – (ii) um indício existe em uma classe se a classe possui uma das seguintes relações com esse indício: herança, agregação ou composição. O conjunto de *features* da Tabela 4-1 foi identificado. Vale ressaltar que os diagramas de classes foram coletados da web¹¹.

¹¹ <http://www.nxtprograms.com/>

Tabela 4-1 - Features Identificadas

RM <i>i</i>	<i>Features</i> identificadas	RM <i>i</i>	<i>Features</i> identificadas
1	<ol style="list-style-type: none"> 1. Sensores <ol style="list-style-type: none"> a. Ultrassônico; b. Câmera. 2. Dispositivo de Locomoção <ol style="list-style-type: none"> a. Esteira. 3. Atuadores <ol style="list-style-type: none"> a. Motor direito; b. Motor esquerdo. 4. Dispositivo de comunicação <ol style="list-style-type: none"> a. Bluetooth; b. Wi-fi. 	2	<ol style="list-style-type: none"> 1. Sensores <ol style="list-style-type: none"> a. Ultrassônico; b. Sensor de luz. 2. Dispositivo de Locomoção <ol style="list-style-type: none"> a. Esteira. 3. Atuadores <ol style="list-style-type: none"> a. Motor direito; b. Motor esquerdo.
RM <i>i</i>	<i>Features</i> identificadas	RM <i>i</i>	<i>Features</i> identificadas
3	<ol style="list-style-type: none"> 1. Sensores <ol style="list-style-type: none"> a. Ultrassônico; b. Sensor de luz; c. Sensor de toque. 2. Dispositivo de Locomoção <ol style="list-style-type: none"> a. Caster. 3. Atuadores <ol style="list-style-type: none"> a. Motor direito; b. Motor esquerdo. c. Pá carregadeira 	4	<ol style="list-style-type: none"> 1. Sensores <ol style="list-style-type: none"> a. Ultrassônico; b. Sensor de luz; c. Sensor de toque. d. Câmera 2. Dispositivo de Locomoção <ol style="list-style-type: none"> a. Wheel. 3. Atuadores <ol style="list-style-type: none"> a. Motor direito; b. Motor esquerdo. c. <i>Neck</i> 4. Dispositivo de comunicação <ol style="list-style-type: none"> a. Wi-fi.

A primeira e a terceira colunas representam os RMs analisados e, a segunda e quarta colunas representam as *features* identificadas.

Conforme descrito anteriormente deve-se aplicar o padrão *Read All the Code in One Hour* nos RMs coletados a fim de complementar as *features* identificadas

pelo padrão *Skim the Documentation*. Além disso, esse padrão também é útil para que o engenheiro de domínio entenda quais são as funcionalidades de cada *RMi*, uma vez que é feita a leitura parcial do código-fonte. As descrições apresentadas na Figura 4-3 foram obtidas durante a leitura do código-fonte e das documentações dos *RMi*:

MR1 → "explora um determinado ambiente hostil e possui os respectivos sensores: ultrassônico e câmera. O primeiro é utilizado para que o MR evite situações perigosas tal como colidir em objetos. Câmera é utilizada para retornar informações em tempo real do ambiente para um usuário remoto. Utiliza esteira como dispositivo de locomoção e é controlado por um usuário remoto utilizado wi-fi ou Bluetooth"

MR2 → "tem a função de medir um determinado terreno e utiliza alguns sensores tais como ultrassônico e luz. Esse MR utiliza a esteira como dispositivo de locomoção"

MR3 → "tem a capacidade de carregar material de um determinado lugar para outro, os sensores utilizados são: sensor de toque, sensor de luz e ultrassônico e o dispositivo de locomoção utilizado é o *caster*"

MR4 → "monitora uma determinada residência utilizando vários sensores como, ultrassônico, sensor de toque, sensor de luz e uma câmera no qual retorna informações para um usuário e pneus são utilizados como dispositivo de locomoção é controlado utilizando Bluetooth."

Figura 4-3 - Funcionalidades do RMs

Ainda nesta diretriz foi criado o artefato TFC, Tabela 4-2, a qual contém informações das *features* identificadas no domínio de RMs e foi criado com auxílio das diretrizes apresentadas na Seção 3.2.2 do Capítulo 3:

A *feature MobileRobots* representa o domínio em questão e consequentemente não tem Nó Pai. Os Nós-Filhos são criados a partir de um conjunto de *features* genéricas que representam informações desse conjunto. Por exemplo, para as *features* "ultrasonic", "light", "camera" e "touch" foi criada uma *feature* genérica chamada "sensor" que representa o conjunto de todos os sensores identificados. No exemplo também são criadas *features* genéricas para Atuador, Conexão, Comportamento, Dispositivo de Locomoção e Navegação.

A *feature* Conexão é opcional, pois existe somente nos RM1 e 4. Seu Nó Pai é *MobileRobots* e seus Nó(s) Filho(s) são os tipos de conexão existentes para esses RMs como pode ser visto na Tabela 4-2.

Na Tabela 4-2 são mostradas todas as *features* que compõem o artefato TFC. Deve-se também classificar as *features* identificadas como obrigatórias, opcionais e/ou alternativa como descrito na Seção 3.2.2.1 do Capítulo 3.

Na Tabela 4-2 são descritas todas as *features* identificadas do domínio.

Tabela 4-2 – Artefato TFC

Nome <i>Feature</i>	Categoria <i>Feature</i>	Nó Pai	Nó(s) Filho(s)	Descrição das <i>Features</i>	Aplicações pertencentes do domínio			
					RM1	RM2	RM3	RM4
<i>MobileRobots</i>	Obrigatória	--	<i>Connection/Actuator/ /Navigation/VehicleType /Behaviors/LocomotionDevice/ Sensor</i>	Característica raiz da linha				
<i>Connection</i>	Opcional	<i>MobileRobots</i>	<i>Bluetooth/Wi-Fi</i>	Representa que o RM utiliza algum tipo de conexão				
<i>Actuator</i>	Obrigatório	<i>MobileRobots</i>	<i>Neck/Fork /MotorLeft/MotorRight</i>	Representa que o RM possui Atuadores				
<i>Navigation</i>	Obrigatório	<i>MobileRobots</i>	<i>SimpleNavigator/ TachoNavigator</i>	Representa o algoritmo utilizado pelo RM para Navegação				
<i>VehicleType</i>	Obrigatório	<i>MobileRobots</i>	<i>TeleOperated / Autonomous</i>	Representa o tipo do RM				
<i>Behaviors</i>	Obrigatório	<i>MobileRobots</i>	<i>Measurer/CargoShip HomeSentry/Explorer</i>	Representa as funcionalidades que o RM proverá				
<i>Locomotion Device</i>	Obrigatório	<i>MobileRobots</i>	<i>Wheel/CaterpillarTrack/ Caster</i>	Representa qual o tipo de dispositivo e locomoção o RM usa				
<i>Sensor</i>	Obrigatório	<i>MobileRobots</i>	<i>Light/Touch/ Ultrasonic/Camera</i>	Representa que o RM possui sensores				
<i>Bluetooth</i>	Alternativa	<i>Connection</i>	--	Representa que o RM utiliza Bluetooth como conexão				
<i>Wi-Fi</i>	Alternativa	<i>Connection</i>	--	Representa que o RM utiliza wi-fi como conexão				

<i>Neck</i>	Opcional	<i>Actuator</i>	--	Utilizado pelo RM responsável por monitorar um ambiente				
<i>Fork</i>	Opcional	<i>Actuator</i>	--	Utilizado por determinados RMs				
<i>MotorLeft</i>	Obrigatório	<i>Actuator</i>	--	Representa que o RM possui motor				
<i>MotorRight</i>	Obrigatório	<i>Actuator</i>	--	Representa que o RM possui motor				
<i>Simple Navigator</i>	Alternativa	<i>Navigation</i>	--	Representa qual o algoritmo que o RM utiliza para navegação				
<i>Tacho Navigator</i>	Alternativa	<i>Navigation</i>	--	Representa qual o algoritmo que o RM utiliza para navegação				
<i>Tele-Operado</i>	Alternativa	<i>VehicleType</i>	<i>Controller</i>	Representa que o RM é tele-operado				
<i>Autonomous</i>	Alternativa	<i>VehicleType</i>	--	Representa que o RM é autônomo				
<i>Measurer</i>	Alternativa	<i>Behaviors</i>	--	Representa a Funcionalidade do RM				
<i>CargoShip</i>	Alternativa	<i>Behaviors</i>	<i>Charger/ForkLift</i>	Representa a Funcionalidade do RM				
<i>HomeSentry</i>	Alternativa	<i>Behaviors</i>	<i>Alg1/Alg2/Alg3</i>	Representa a Funcionalidade do RM				
<i>Explorer</i>	Alternativa	<i>Behaviors</i>	--	Representa a Funcionalidade do RM				
<i>Wheel</i>	Alternativa	<i>Locomotion Device</i>	--	Representa o dispositivo de locomoção do RM				
<i>CaterpillarTrack</i>	Alternativa	<i>Locomotion Device</i>	--	Representa o dispositivo de locomoção do RM				
<i>Caster</i>	Alternativa	<i>Locomotion Device</i>	--	Representa o dispositivo de locomoção do RM				
<i>Light</i>	Alternativa	<i>Sensor</i>	--	Funcionalidade do sensor de luz				
<i>Touch</i>	Alternativa	<i>Sensor</i>	--	Funcionalidade do sensor de toque				
<i>Ultrasonic</i>	Alternativa	<i>Sensor</i>	--	Funcionalidade do sensor ultrassônico				

<i>Camera</i>	Alternativa	<i>Sensor</i>	--	Funcionalidade de câmera				
<i>Controller</i>	Obrigatório	<i>Tele-operado</i>	<i>Phone/PC/WiiControl</i>	Representa os tipos de controller disponíveis				
<i>Charger</i>	Alternativa	<i>CargoShip</i>	--	Representa o tipo de RM				
<i>ForkLift</i>	Alternativa	<i>CargoShip</i>	--	Representa o tipo de RM				
<i>Alg1</i>	Alternativa	<i>HomeSentry</i>	--	Representa o tipo de RM				
<i>Alg2</i>	Alternativa	<i>HomeSentry</i>	--	Representa o tipo de RM				
<i>Alg3</i>	Alternativa	<i>Homesentry</i>	--	Representa o tipo de RM				
<i>Phone</i>	Alternativa	<i>Controller</i>	--	Representa o tipo de controller				
<i>PC</i>	Alternativa	<i>Controller</i>	--	Representa o tipo de controller				
<i>WiiControl</i>	Alternativa	<i>Controller</i>	--	Representa o tipo de controller				

4.4 Criar o Diagrama de *Features*

Nesta atividade foi desenvolvido o diagrama de *features* da LRMLPS apoiada pela diretriz “D3 - Desenvolver o Diagrama de *Features*” o qual utiliza a notação PLUS (Gomaa, 2004). O desenvolvimento do digrama de *features* é uma atividade iterativa que foi conduzida com auxilio de informações contidas no artefato TFC, apresentado na Tabela 4-2. Por exemplo, a primeira linha da TFC foi utilizada como ponto de partida para o desenvolvimento e agrupamento das *features*. Posteriormente, as colunas “Nome Feature”, “Categoria Feature”, “Nó Pai” e “Nó(s) Filho(s)” foram utilizadas pelo engenheiro de domínio para começar a criação do diagrama de *features*. Dessa forma, a *feature MobileRobots* no qual representa o domínio da LPS foi decomposta em sete *sub-features* (*Connection*, *Actuator*, *Navigation*, *VehicleType*, *Behaviors*, *LocomotionDevice* e *Sensor*). De forma, semelhante à *feature Connection* foi composta de outras duas *features* (*Bluetooth* e *Wi-Fi*). Em seguida, foi feita uma iteração por todo o artefato TFC para o desenvolvimento do diagrama de *features*. Na Figura 4-4 é apresentado o digrama de *features* da LRMLPS. Após o desenvolvimento do diagrama de *features* foi iniciada a atividade “Criar a Arquitetura da LPS” no qual é descrito a seguir.

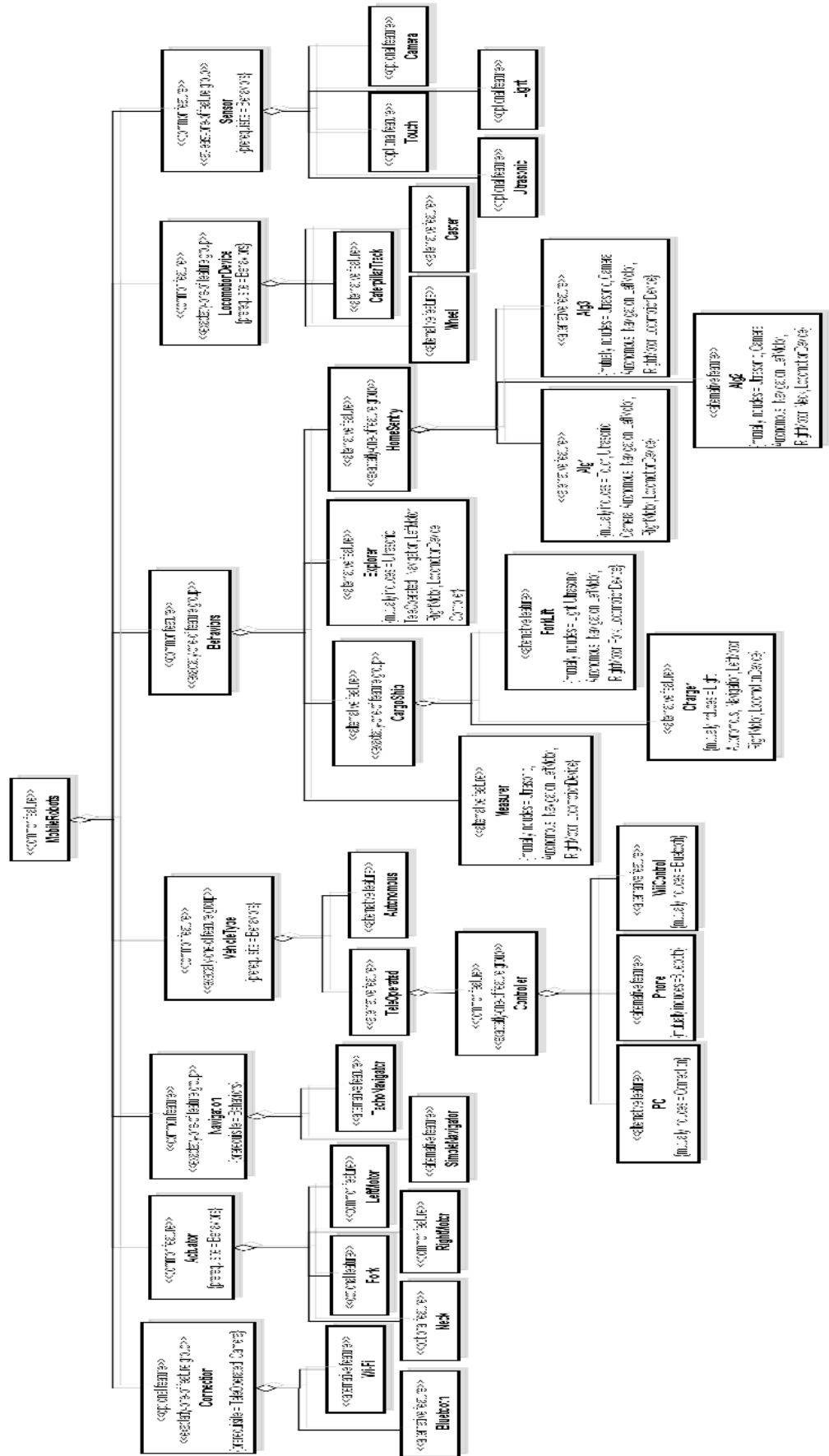


Figura 4-4 - Diagrama de Features da LRMLPS

4.5 Criar a Arquitetura da LPS

Atividade responsável por criar a arquitetura da LRMLPS. O mesmo foi criado utilizando a notação PLUS (Gomaa, 2004) juntamente com as diretrizes propostas por Almeida *et al.*, (2007).

As diretrizes propostas por Almeida *et al.*, (2007) foram úteis para auxiliar o engenheiro de domínio a modelar a estrutura, e a delinear as *features* comuns e variáveis da LRMLPS, ou seja, para a criação da arquitetura da linha. Por exemplo, os conjuntos de *features* *Sensor* (*Ultrasonic*, *Light*, *Touch* e *Camera*) e *Actuator* (*MotorLeft*, *MotorRight*, *Neck* e *Fork*) foram modelados utilizando o padrão *Builder* (Gamma, 2002). *Features* alternativas foram implementadas utilizando o padrão *AbstractFactory* ou *Factory Method*. Dessa forma, na Figura 4-5 é apresentada a arquitetura da LRMLPS.

Após o desenvolvimento da arquitetura da linha de produtos, deu-se início à atividade “Implementar os Artefatos da LPS, descrita a seguir.

4.6 Implementar os Artefatos da LPS

Atividade responsável por implementar as variabilidades e similaridades da LRMLPS. Dessa forma, foi utilizado como entrada o modelo de arquitetura criado anteriormente. O IDE (*Integrated Development Environment*) Eclipse (Eclipse, 2011) foi utilizado como principal ferramenta juntamente com a máquina virtual LeJOS (LeJOS, 2011) para sintetizar as variabilidades e similaridades da LPS em códigos-fontes. A saída desta atividade consiste de um conjunto de arquivos que estão contidos os códigos-fontes.

Conforme descrito por Anastasopoulos e Gacek (2001) existem várias abordagens para implementar variabilidades, no entanto, o hardware do Lego Mindstorms não é tão eficiente para fornecer apoio a algumas dessas abordagens, como AspectJ (AspectJ, 2011). Dessa maneira, padrões de projeto (Gamma, 1994) foram utilizados para implementar as variabilidades e similaridades da LRMLPS. Na Figura 4-6 são apresentados trechos de códigos fontes nos quais foram aplicados o padrão *Builder* para implementar as respectivas *features*: *ultrasonic* e *camera*. A classe *DirectorSensor* (Figura 4-6(a)) é responsável por gerenciar a sequência correta de criação dos sensores da LRMLPS, enquanto que *BuilderSensor*, (Figura 4-6(b)) representa uma classe abstrata a qual tem a responsabilidade de criar sensores da linha. As classes *BuilderUltraSonicSensor* (Figura 4-6(c)) e *BuilderCamera* (Figura 4-6(d)) fornecem uma implementação da classe *BuilderSensor*, as quais auxiliam a criação de instâncias dos seus respectivos sensores.

```

package com.br.ufscar.lejos.sensors.builder.director;

import com.br.ufscar.lejos.sensors.builder.BuilderSensor;

public class DirectorSensor {

    private BuilderSensor builderSensor;

    public void setBuilderSensor(BuilderSensor builderSensor)
    {
        this.builderSensor = builderSensor;
    }

    public Object getSensor(){
        return builderSensor.getSensor();
    }

    public void constructSensor(String port){
        this.builderSensor.buildSensor(port);
    }
}

```

(a)

```

package com.br.ufscar.lejos.sensors.builder;

public abstract class BuilderSensor {

    protected Object sensor;

    public Object getSensor(){
        return sensor;
    }

    public abstract void buildSensor(String sensorPort);
    ...
}

```

(b)

```

package com.br.ufscar.lejos.sensors.builder;

import lejos.nxt.NXTCam;
import lejos.nxt.SensorPort;

public class BuilderCamera extends BuilderSensor{

    @Override
    public void buildSensor(String sensorPort) {
        SensorPort port = null;
        if(sensorPort.equals("S1")){
            port = SensorPort.S1;
        }else if (sensorPort.equals("S2")){
            port = SensorPort.S2;
        }else if (sensorPort.equals("S3")){
            port = SensorPort.S3;
        }
        else{
            port = SensorPort.S4;
        }
        this.sensor = new NXTCam(port);
    }
    ...
}

```

(d)

```

package com.br.ufscar.lejos.sensors.builder;

import lejos.nxt.SensorPort;
import lejos.nxt.UltrasonicSensor;

public class BuilderUltraSonicSensor extends BuilderSensor {

    @Override
    public void buildSensor(String sensorPort) {
        SensorPort port = null;
        if(sensorPort.equals("S1")){
            port = SensorPort.S1;
        }else if (sensorPort.equals("S2")){
            port = SensorPort.S2;
        }else if (sensorPort.equals("S3")){
            port = SensorPort.S3;
        }
        else{
            port = SensorPort.S4;
        }
        this.sensor = new UltrasonicSensor(port);
    }
    ...
}

```

(c)

Figura 4-6 - Trechos de código-fonte da variabilidade Ultrassônico

Na Figura 4-7 é apresentada uma visão geral da estrutura da LRMLPS. A camada de hardware contém todos os sensores e atuadores disponíveis da linha, tais como, ultrassônico, sensor de toque, sensor de luz, câmera e três servos motores. Além dos sensores e atuadores, essa camada também contém um micro controlador. Tais hardwares são utilizados para a construção dos produtos pertencentes à linha. A camada de plataforma oferece ao desenvolvedor uma camada de abstração, implementada em software, entre o hardware e os artefatos. Essa camada deve existir para abstrair os detalhes de baixo nível dos hardwares (sensores e atuadores) permitindo o desenvolvimento de aplicações de maneira mais fácil. A terceira camada representa os artefatos da LPS que foram implementados nessa atividade, ou seja, todas as *features* comuns e variáveis da LRMLPS são armazenadas para serem reutilizadas posteriormente. A quarta camada representa as aplicações que são instanciadas utilizando os artefatos da camada inferior.

Para auxiliar o processo de instanciação de um determinado membro da LRMLPS foi desenvolvida uma DSL gráfica que agiliza o desenvolvimento de

membros da linha. O processo de desenvolvimento da DSL é descrita na próxima seção.

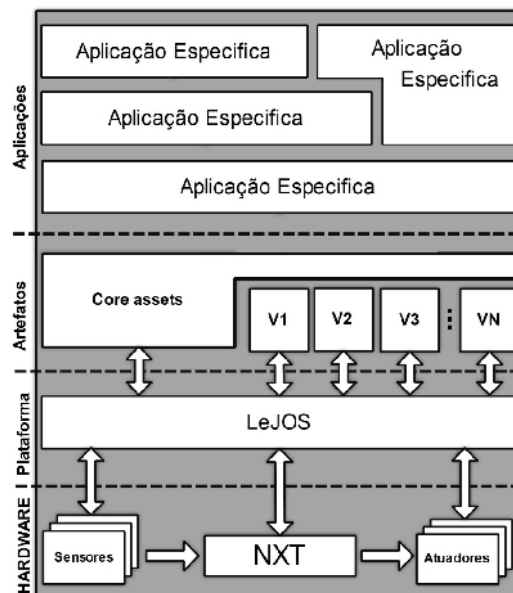


Figura 4-7 - Visão Geral da LRMLPS

4.7 Implementar a DSL Gráfica

Atividade responsável por criar uma DSL gráfica intitulada *Features-To-ModelOrCode* (F2MoC) para auxiliar e agilizar a tarefa do engenheiro de aplicação na instanciação de um membro da linha de produtos.

A partir do diagrama de *features*, o engenheiro de domínio especificou o metamodelo da F2MoC, com suas metaclasses, meta-atributos, e meta-relacionamentos utilizando o meta-metamodelo *Ecore* do *Eclipse Modeling Framework* (EMF) (EMF, 2011).

O *Ecore* fornece uma metalinguagem para definição de metamodelos no IDE Eclipse. Por exemplo, as *features* *Ultrasonic*, *Touch*, *Camera*, *Light*, *MotorLeft*, *MotorRight*, *Neck* e *Fork* ilustradas no diagrama de *features* da Figura 4-4 foram mapeadas para as metaclasses de mesmo nome destacadas no metamodelo da Figura 4-8.

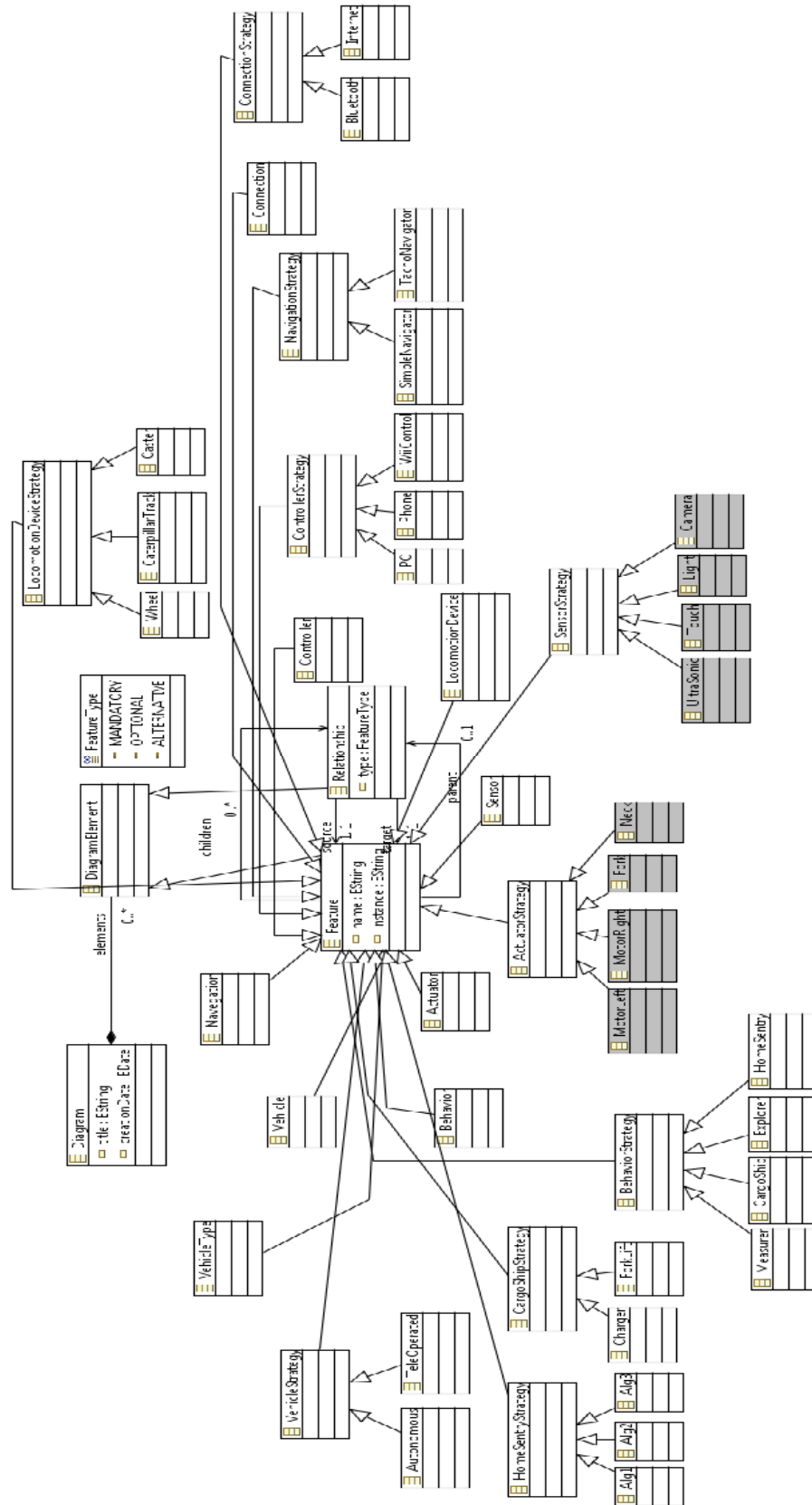


Figura 4-8 - Metamodelo da DSL

Adicionalmente, o EMF possibilita ao engenheiro de domínio gerar código Java a partir do metamodelo e de um editor de modelos que auxilia na especificação

dos membros da LRMLPS na etapa de EA. Nesse editor os modelos são persistidos no formato de *XML metadata Interchange* (XMI), um padrão da *Object Management Group* (OMG) utilizado para representar modelos em *eXtensible Markup Language* (XML). Esse formato define uma estrutura de documentos que considera a relação entre dados e seus correspondentes metadados, o que facilita realizar transformações M2M e M2C. Como exemplo, na Figura 4-9 é apresentado um trecho de código gerado automaticamente o qual corresponde à metaclassa *Ultrasonic*.

```
1 package feature;
2
3 import org.eclipse.emf.common.util.EList;
4
5 public interface Ultrasonic extends DiagramElement {
6
7     String getName();
8
9     Relationship getParent();
10
11    void setParent(Relationship value);
12
13    String getInstance();
14
15    EList<Relationship> getChildren();
16
17    ...
18 }//Ultrasonic
```

Figura 4-9 - Implementação do Metamodelo (metaclassa *Ultrasonic*)

A F2MoC criada é representada na forma de diagrama de *features*. Para instanciar um membro da LRMLPS utilizando a F2MoC o engenheiro de aplicação escolhe um conjunto de *features* disponível da linha. Além disso, tais *features* são utilizadas para realizar transformações M2M e M2C. Assim, a partir das *features* escolhidas a F2MoC cria um diagrama de classe e os códigos-fontes do membro da linha de produtos.

Para realizar tais transformações foram desenvolvidos *templates* utilizando o *framework Java Emitter Template* (JET) (Jet, 2011). Tais *templates* interpretam as *features* selecionadas e conseqüentemente realizam as transformações M2M e M2C. Na Figura 4-10 é apresentado um trecho de um *template* utilizado para realizar transformações M2C. Esse *template* é invocado pelo JET quando for encontrado a *feature Ultrasonic* durante a leitura do XMI. De acordo com a *feature* selecionada um determinado trecho desse *template* será executado, ou seja, quando *feature ultrasonic* for selecionado as linhas 6-11 serão executadas; quando a *feature "Navigation"* for selecionada as linhas 14-17 serão executadas.

Em transformações M2M, o objetivo principal é transformar um modelo de entrada especificado em alto nível em outro modelo mais específico (Czarnecki e Antkiewicz, 2004). Neste trabalho o diagrama de *features* modelado pelo engenheiro de aplicação é transformado em um diagrama de classe. Essa transformação é feita tendo como base o uso de *templates* juntamente com padrões de projeto. Os *templates* desenvolvidos para realizar as transformações M2M utilizaram a ferramenta Papyrus que é uma ferramenta gráfica, open source utilizada para fornecer suporte para UML (OMG, 2011).

```
1 package com.br.ufscar.lejos.groundvehicles.autonomous;
2 import com.br.ufscar.lejos.sensors.builder.BuilderUltrasonicSensor;
3
4 public class Ultrasonic extends Sensor{
5
6 <c:if test =
7 "$actual/parent/source/parent/source/parent/source/children/target
8 [self::Ultrasonic]">
9     private Scheduler scheduler;
10
11 </c:if>
12     protected BehaviorType behaviorType;
13
14     public Ultrasonic (BehaviorType behaviorType, <c:if test =
15 "$actual/parent/source/parent/source/parent/source/children/target
16 [self::Navegation]">ControllerType controllerType, float wheelDiameter,
17 float trackWidth</c:if>){
18
19         super(controllerType, wheelDiameter, trackWidth);
20
21         this.behaviorType = behaviorType;
22     }
```

Figura 4-10 - *Template* para geração de código

Na próxima seção é descrito a Engenharia de Aplicação. Nessa etapa o engenheiro de aplicação deve utilizar a F2MoC, a fim de agilizar o processo de instanciação de um determinado membro pertencente à LRMLPS.

4.8 Engenharia de Aplicação da LRMLPS

Nessa etapa o engenheiro de aplicação utiliza a F2MoC com o intuito de instanciar um determinado membro pertencente à LRMLPS. O procedimento para derivar um produto é ilustrado na Figura 4-11.

A partir de um diagrama de *features* que representa informações de um membro pertencente à linha de produtos é possível realizar transformações M2M e M2C.

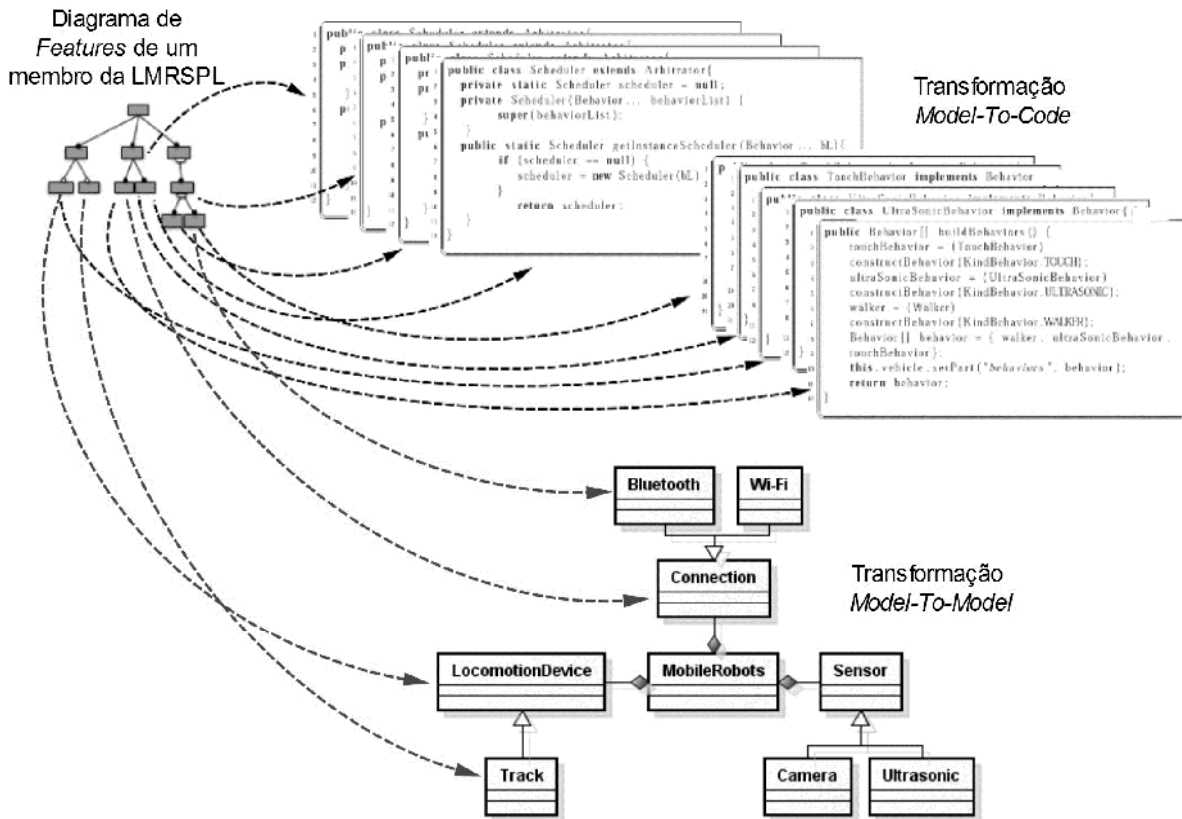


Figura 4-11 – Exemplo do uso da DSL

O processo de instanciação de um membro é da seguinte maneira: (i) dado um conjunto de requisitos, o engenheiro de aplicação executa uma configuração das *features* da aplicação, essa configuração descreve os requisitos por meio de um diagrama de *features* e é apoiada pela F2MoC originada a partir do metamodelo criado na ED. Tais configurações são representadas por um arquivo que utiliza a notação XML, de modo que a extração de suas informações é facilitada – (ii) são feitas transformações M2M e M2C, por meio do *framework* JET. Esse *framework* analisa o arquivo de configuração (XML) a fim de identificar possíveis pontos de variações. Tais pontos são indícios de variações de acordo com as informações fornecidas no diagrama de *features* da aplicação. Dessa forma, os *templates* que foram desenvolvidos na ED são utilizados para transformar o diagrama de *features* em um diagrama de classe (M2M) e em código-fonte (M2C) de um determinado membro da LRMLPS.

No Capítulo 5 são apresentados três casos de uso no qual é ilustrado com maiores detalhes como utilizar a DSL F2MoC desenvolvida neste trabalho e também o processo de instanciação de um determinado membro da LRMLPS.

4.9 Considerações Finais

Graaf, Lormans e Toetenel (2003) afirmam que em geral os projetos de sistemas embarcados encontram-se atrasados devido a falta de reuso neste domínio. Dessa forma, na literatura podem-se encontrar vários trabalhos que utilizam técnicas de linha de produtos de software para o domínio de sistemas embarcados (Polzer *et al*, 2009; Kim, 2006; Lee *et al*, 2005; Ubayashi e Nakajima, 2010). No entanto, nenhum desses trabalhos apresentam diretrizes claras de como desenvolver uma linha de produtos de software para o domínio de sistemas embarcados. Dessa forma, este Capítulo apresentou o desenvolvimento de uma LPS para o domínio de sistemas embarcados mais especificadamente no domínio dos *Robôs Moveis* intitulada *LegoRobosMoveis Linha de Produtos de Software* (LRMLPS). O desenvolvimento da LRMLPS foi feito tendo como base o processo descrito no Capítulo 3. Foi também desenvolvida uma DSL gráfica (F2MoC) na qual auxilia o engenheiro de aplicação na instanciação de membros pertencentes à LRMLPS. Dessa forma, no Capítulo 5 são apresentados três exemplos que descrevem como utilizar a F2MoC.

Capítulo 5

EXEMPLO DE UTILIZAÇÃO PARA GERAÇÃO DE RMs BASEADOS NA DSL DA LRMLPS

5.1 Considerações Iniciais

A DSL apresentada no Capítulo 4, intitulada *FeaturesToModelOrCode* (F2MoC), foi criada com o intuito de auxiliar o engenheiro de aplicação no processo de instanciação de membros da LRMLPS.

As transformações M2C geram o código-fonte da aplicação, que deve ser embarcado no RM para que possa ser executado. Transformações M2M utilizam um modelo com nível de abstração mais alto como entrada e o transforma em um modelo mais específico. Neste trabalho as transformações M2M são realizadas a partir de um diagrama de *features* o qual é utilizado como base para gerar um diagrama de classe de uma determinada aplicação.

Três exemplos foram elaborados para exemplificar a utilização da DSL criada. O primeiro exemplo contempla um RM o qual tem a responsabilidade de explorar um ambiente hostil. O segundo exemplo aborda o desenvolvimento de um RM responsável por carregar material de um determinado lugar para outro, enquanto que o terceiro trata de um RM responsável de monitorar uma determinada residência.

Este Capítulo está organizado da seguinte maneira: na Seção 5.2 é abordado o desenvolvimento do primeiro exemplo; na Seção 5.3 é tratado o segundo exemplo; Na Seção 5.4 é descrita a realização do terceiro exemplo; e, finalmente, na Seção 5.5 estão as considerações finais.

5.2 Exemplo 1: Geração do RM “*Explorer*”.

Conforme descrito anteriormente à instanciação de um determinado RM é conduzida com base em um conjunto de requisitos, definidos pelo cliente, principal interessado na construção do RM. Dessa forma, na Tabela 5-1 contém um conjunto de requisitos que o engenheiro de aplicação deve respeitar para instanciar o RM.

Tabela 5-1 Requisitos do RM 1

#	Descrição
1	O RM tem a responsabilidade de explorar um determinado ambiente hostil;
2	A aplicação consiste em uma estação de controle. Essa estação de controle deve ser representada por um PC no qual deve realizada a comunicação com o RM;
3	O ambiente no qual o RM usualmente é operado possui vários objetos que devem ser evitados pelo RM
4	O RM também deve ser equipado com motores e esteiras;
5	Para seguir a execução de uma missão em tempo real, o RM deve possuir uma câmera, pela qual retorna informações para a estação de controle. Tais informações são utilizadas por um operador humano;
6	A estação de controle deve permitir ao operador transmitir várias instruções ao RM (ex., aumentar/diminuir a velocidade do RM ou alternar uma determinada rota), para a progressão da missão.

O engenheiro de aplicação utiliza o diagrama de *features* da LRMLPS (ver Capítulo 4) para identificar os requisitos solicitados pelo cliente com as *features* disponíveis na linha. Em seguida deve utilizar a DSL F2MoC para modelar o diagrama de *features* correspondente a esses requisitos. Finalmente, com o modelo de *features* modelado e validado é possível realizar automaticamente transformações M2M e M2C.

O engenheiro de aplicação deve ter disponível o IDE Eclipse (Eclipse, 2011) no momento de utilizar a F2MoC, uma vez que essa utiliza o *framework* GMF (*Graphical Modeling Framework*), EMF (*Eclipse Modeling Framework*) e GEF (*Graphical Editing Framework*). No Apêndice B é apresentada uma breve introdução a esses *frameworks*, maiores informações podem ser obtidas em: <http://www.eclipse.org/modeling/>.

A sequência de passos apresentada a seguir é necessária para a utilização da F2MoC:

1. Iniciar o IDE Eclipse;
2. Criar um projeto, o qual conterá as informações geradas pela F2MoC.

Na Figura 5-1(a) é apresentado o procedimento a ser realizado para a criação do projeto. Um *wizard* de criação de projetos é apresentado após a seleção dos seguintes botões: *File – New - Other*. A tela *New* será exibida (Figura 5-1(b)).

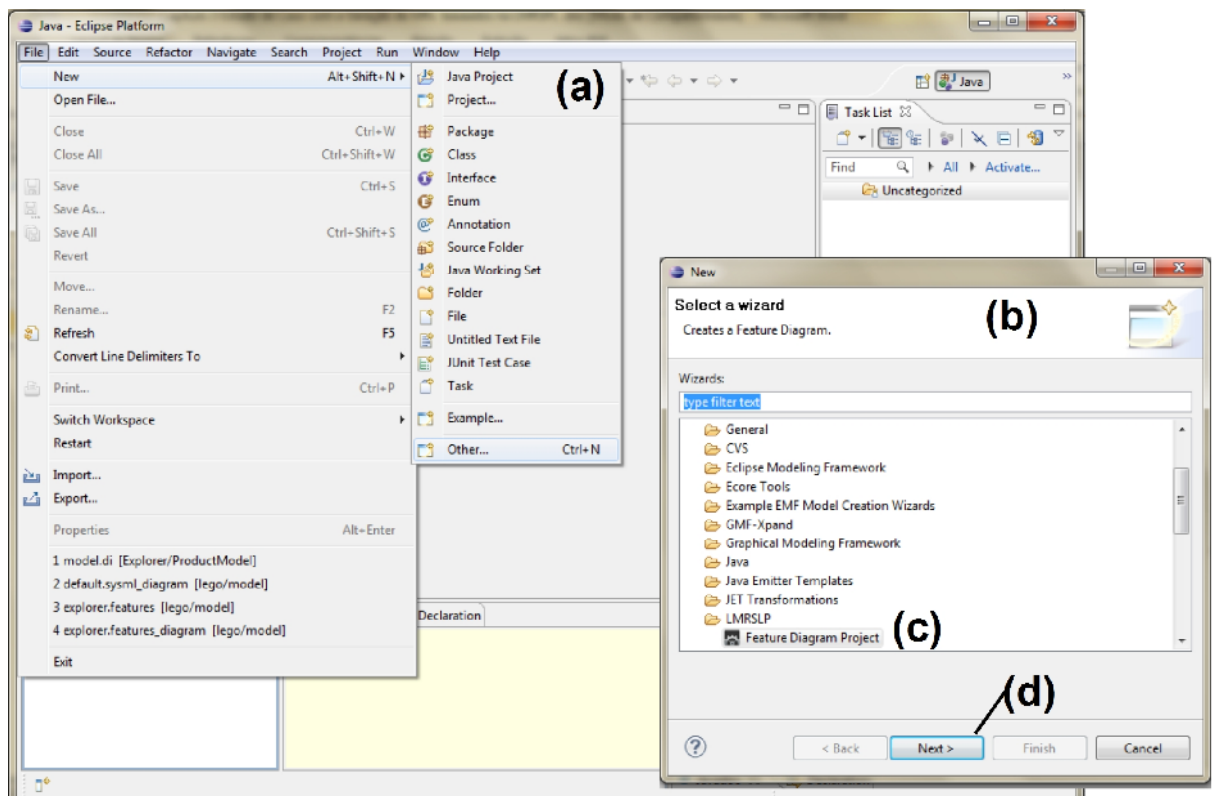


Figura 5-1 Criando um Projeto, passo 1

3. Selecionar o diretório LRMLPS o qual contém o arquivo “*Feature Diagram Project*”, Figura 5-1(c). Após escolher a opção “*Feature Diagram Project*” deve-se clicar em *Next* (Figura 5-1(d)), sendo apresentado outro *wizard*, exibido na Figura 5-2.

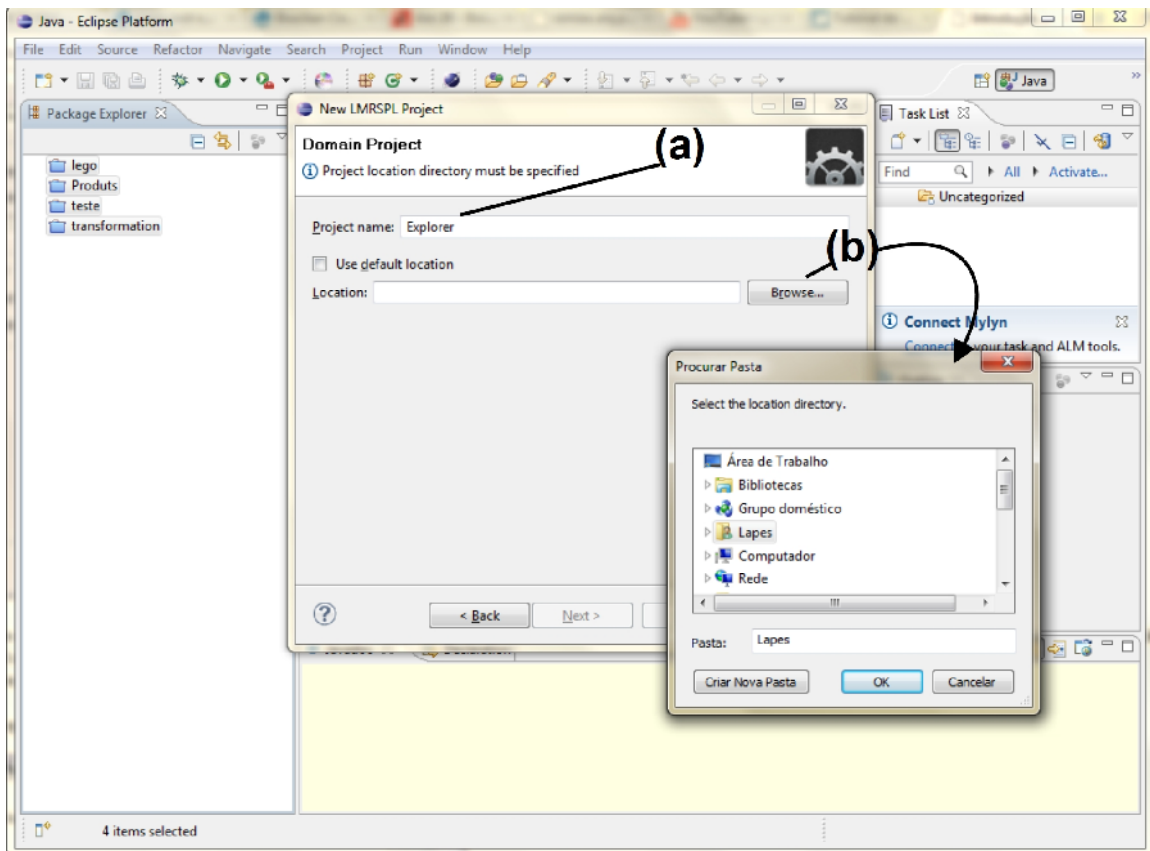


Figura 5-2 Criando um Projeto, passo 2

4. Preencher no campo de entrada “*Project Name*” (Figura 5-2(a)) com o nome do projeto desejado, nesse exemplo, “*Explorer*”.
5. Especificar o diretório onde o projeto será armazenado (Figura 5-2(b)).
6. Clicar em “*Finish*”, para que toda a estrutura necessária da DSL seja criada, como mostrado na Figura 5-3(a). Todas as dependências necessárias para executar a DSL, ou seja, todos os arquivos Java Archive (jar) foram importados automaticamente, como apresentadas na Figura 5-3(b). De maneira semelhante, foi criado um diretório “*src*”, que é responsável por armazenar os códigos-fontes criados pelas transformações M2C (Figura 5-3(c)). O diretório “*model*” também foi criado para armazenar as transformações M2M (Figura 5-3(d)).

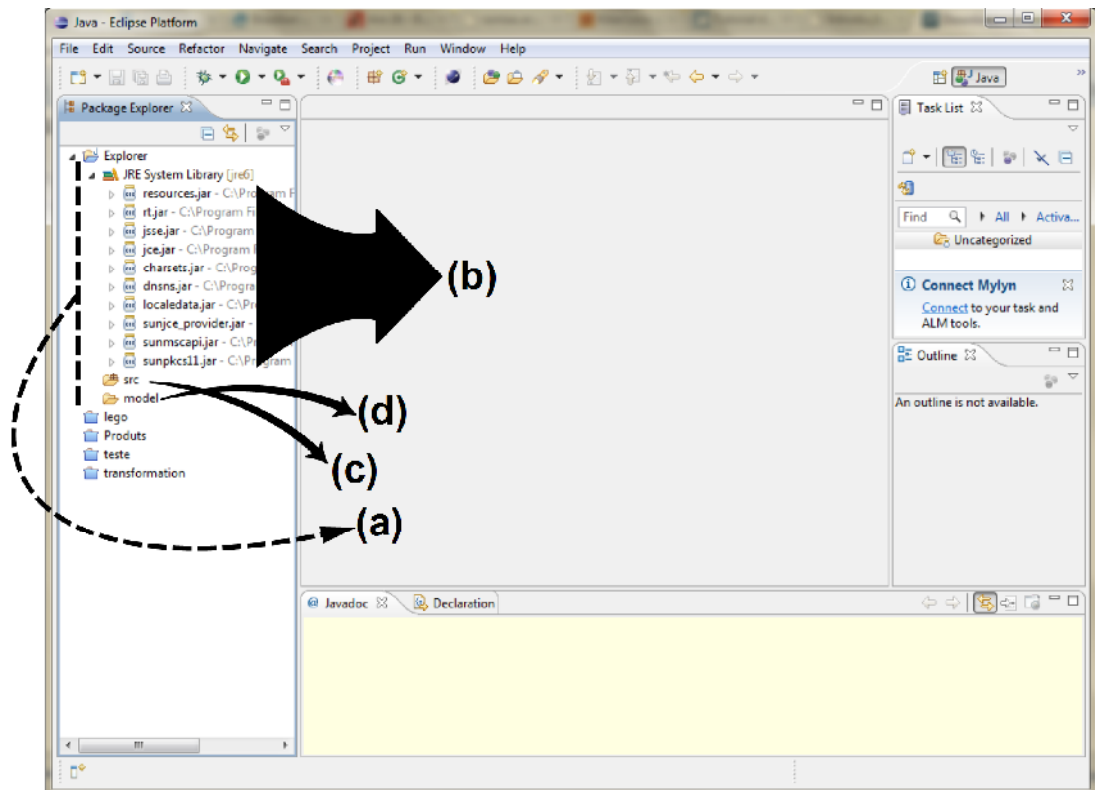


Figura 5-3 Estrutura da DSL

7. Criar os arquivos que contém as abstrações de todas as *features* identificadas na LRMLPS. Isso é realizado selecionando-se os botões File – New – Other. Um wizard como o da Figura 5-4 é criado.
8. Escolher a opção Features Diagram (Figura 5-4(a)), clicar no botão “Next” para apresentar outro wizard responsável em criar o arquivo que contém todas as abstrações das *features* identificadas na LRMLPS (Figura 5-5). Selecionar a pasta “*model*”, onde será persistido o arquivo que contém todas as abstrações das *features* da LRMLPS (Figura 5-5(a)). Atribuir um nome para esse arquivo (Figura 5-5(b)), neste exemplo o arquivo foi nomeado de “*Explorer.features_diagram*”. Esse arquivo representa um ambiente visual que fornece a ação *drag-and-drop*, ou seja, permite ao engenheiro de aplicação clicar em uma respectiva *feature* disponível na LRMLPS e “arrastá-la” a fim de auxiliar na construção do diagrama de *features* de um membro da linha. Posteriormente deve-se clicar no botão “Next”, (Figura 5-5(c)), para que seja apresentado um novo *wizard* como mostrado na Figura 5-6.

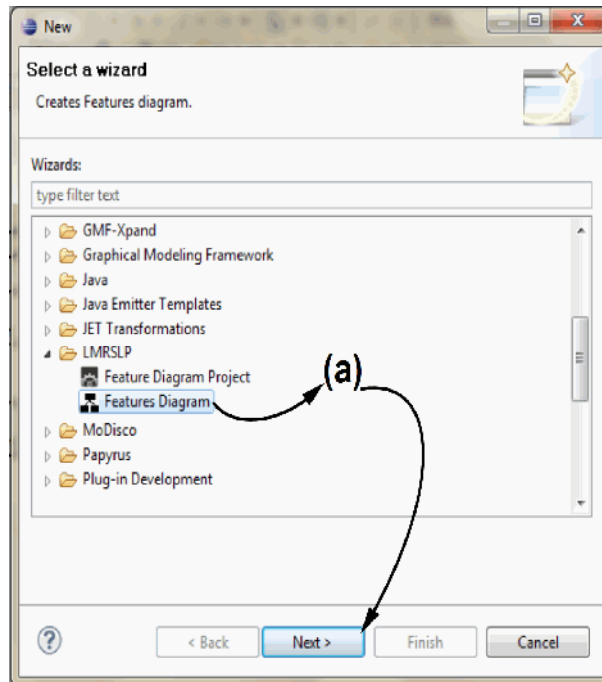


Figura 5-4 Criando os Arquivos Necessários da DSL, passo 1

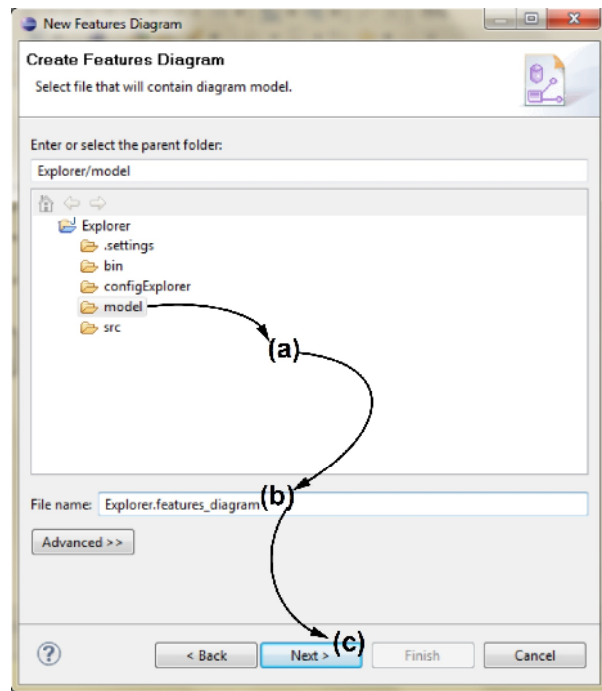


Figura 5-5 – Criando os Arquivos Necessários da DSL, passo 2

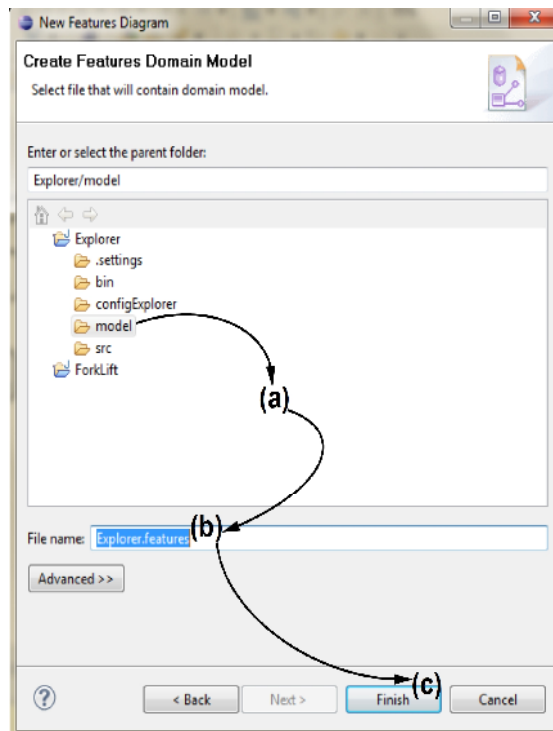


Figura 5-6– Criando os Arquivos Necessários da DSL, passo 3

Esse *wizard* é responsável por criar um arquivo de validação (extensão *.features*) do diagrama de *features* de um determinado membro da linha. Esse arquivo também deve ser armazenado na pasta “*model*” (Figura 5-6(a)), em seguida deve-se atribuir um nome a ele, neste exemplo é o “*Explorer.features*” como apresentado na Figura 5-6(b). Esse arquivo contém as mesmas informações que o arquivo “*Explorer.features_diagram*”, porém sua representação é em uma estrutura de árvore a qual facilita a realização das validações. Não há verificação do diagrama construído em tempo de execução. Dessa forma, o arquivo “*Explorer.features*” é utilizado para a verificação semântica do diagrama de *features* construído pelo engenheiro de aplicação com o auxílio da LRMLPS. Alguns exemplos de verificações realizadas são: se todas as *features* obrigatórias foram preenchidas, se somente uma *feature* alternativa foi escolhida, se os relacionamentos entre as *features* estão corretos. Finalmente deve-se clicar no botão *Finish*, (Figura 5-6(c)), o qual irá criar o ambiente da F2MoC.

Na Figura 5-7 é apresentada uma visão geral do ambiente de desenvolvimento dos diagramas de *features* pertencentes a LRMLPS (Figura 5-7(b)).

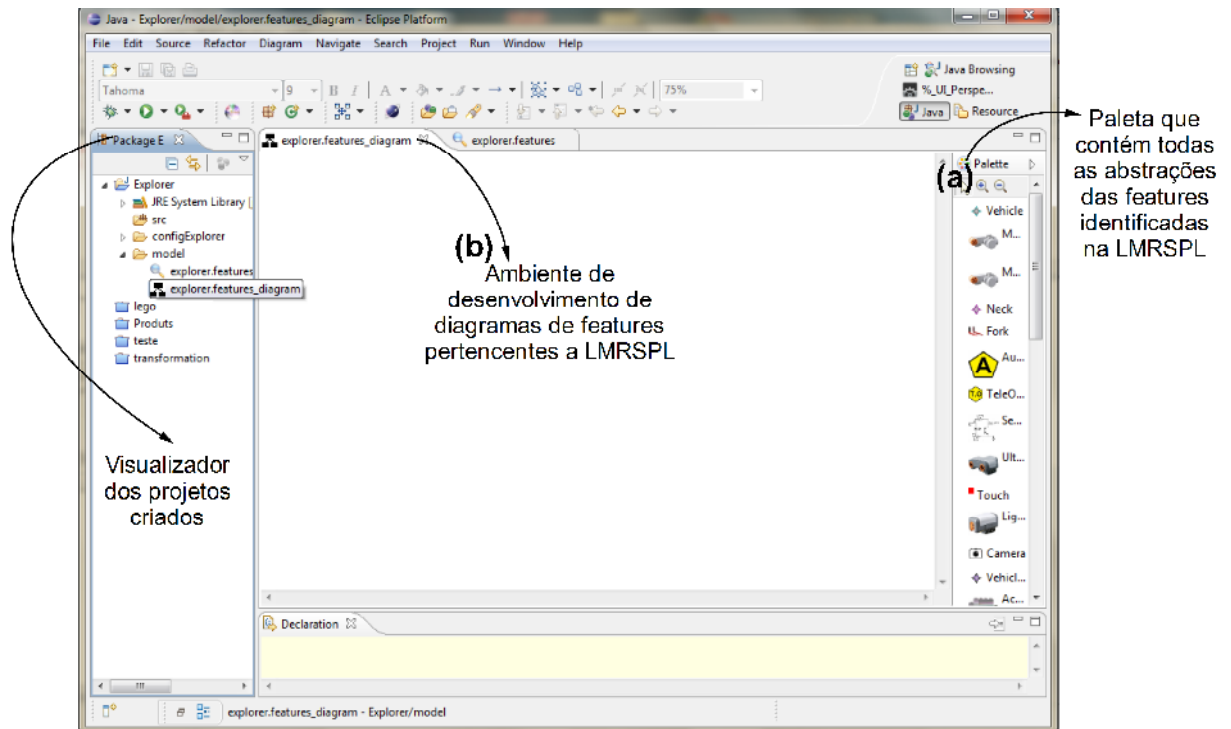


Figura 5-7 – Visão Geral da DSL Gráfica

Com o projeto pronto para ser utilizado, o engenheiro de aplicação deve começar a modelagem do diagrama de *features*. Essa modelagem deve ser conduzida com base nos requisitos listados na Tabela 5-1 e no diagrama de *features* da LRMLPS elaborado, que possui informações sobre os possíveis relacionamentos entre as *features*. O diagrama de *features* da LRMLPS é apresentado no Capítulo 4.

Como apresentado no diagrama de *features* da LRMLPS (ver Capítulo 4), as *features* “Actuators”, “Navigation”, “VehicleType”, “Behavior”, “Locomotion Device” e “Sensor” representam as *features* obrigatórias. Dessa forma, elas devem ser sempre incluídas no diagrama de *feature* de qualquer membro da linha de produtos a ser instanciado. O engenheiro de aplicação deve selecionar uma a uma dessas *features* no conjunto de *features* apresentado na paleta da DSL (Figura 5-7(a)). Após a escolha, essa *feature* deve ser “arrastada” para o ambiente de desenvolvimento de diagrama de *features* (Figura 5-7(b)). Posteriormente devem-se criar os relacionamentos entre as *features* selecionadas, Figura 5-8, deve-se clicar na *feature* base (Figura 5-8(a)), aparecerá uma linha tracejada que representa o relacionamento a ser criado, e essa linha deve ser arrastada até a *feature* que se relaciona com a anterior (Figura 5-8(b)). Em seguida deve-se clicar no botão “create RelationShip” (Figura 5-8(c)). O processo de criação dos relacionamentos deve ser realizado para todas as *features* do diagrama.

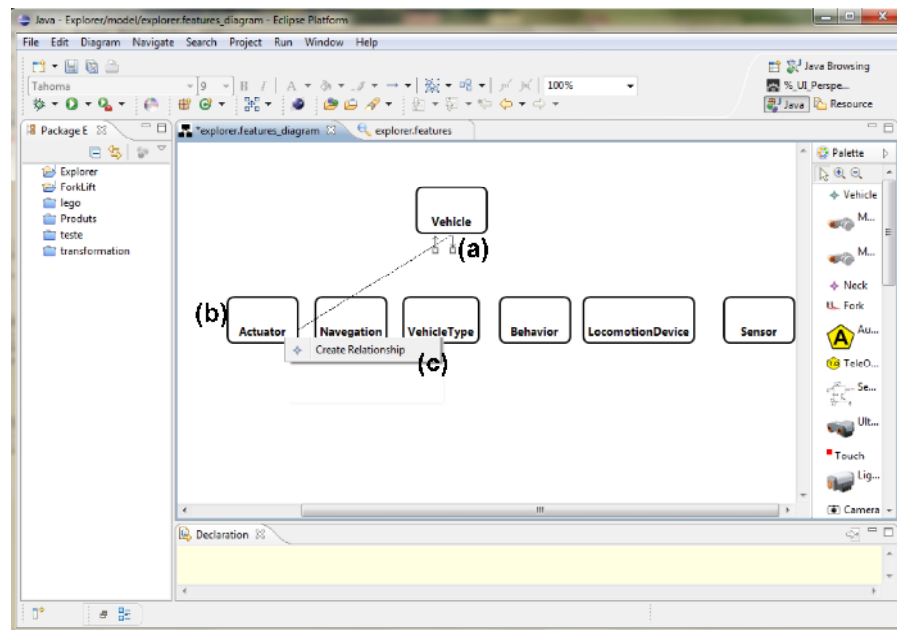


Figura 5-8 - Criando Relacionamento entre as Features

Um dos requisitos solicitados pelo cliente é que o RM possua um dispositivo de comunicação (Tabela 5-1,). A *feature* “*Connection*” foi selecionada, pois representa uma abstração dos dispositivos de comunicação pertencentes à LRMLPS. Como existem dois tipos de comunicação disponível na LRMLPS (ver Capítulo 4), *Bluetooth* ou *Wi-Fi* deve-se selecionar qual atende aos requisitos solicitados. A *feature Bluetooth* deve ser escolhida quando o RM for operar em um ambiente menor, uma vez que tal tecnologia possui um curto alcance. Por outro lado, a *feature Wi-Fi* deve ser selecionada para atuar em ambientes maiores, uma vez que tal tecnologia possui alta potência na transmissão dos dados. Como o RM desse exemplo irá atuar em ambientes relativamente grandes a *feature “Wi-Fi”* (*sub-feature* de “*Connection*”) foi escolhida. A *feature “Explorer”* foi selecionada, uma vez que o RM tem a responsabilidade de explorar um determinado ambiente.

A *feature “TeleOperated”* representa uma abstração de uma estação de controle que é responsável por realizar iteração com o operador humano. Toda estação de controle deve possuir um controlador (“*Controller*”), que na LRMLPS, pode ser do tipo: PC, Celular ou *WiiControl*. Neste exemplo, as *features “TeleOperated”, “Controller”* e “*PC*” foram selecionadas.

As *features “MotorLeft”* e “*MotorRight*” foram selecionadas, pois representam abstrações dos motores do RM. Como descrito na Tabela 5-1 o dispositivo de locomoção utilizado pelo RM é uma esteira, uma vez que o RM será operado em ambientes hostis, ou seja, ambientes que usualmente possuem solos irregulares.

Portanto, a *feature* “CaterpillarTrack” também foi selecionada. Finalmente, os sensores “Ultrasonic” e “Camera” foram selecionados, sendo que a *feature* “Ultrasonic” realizará a identificação de obstáculos.

A *feature* “Camera” tem como principal função retornar informações em tempo real para a estação de controle, fornecendo assim informações necessárias sobre o ambiente para que o operador humano possa tomar decisões em tempo real.

Na Figura 5-9 é apresentado o diagrama completo de *features* desenvolvido utilizando a DSL F2MoC, que atende aos requisitos do RM deste exemplo.

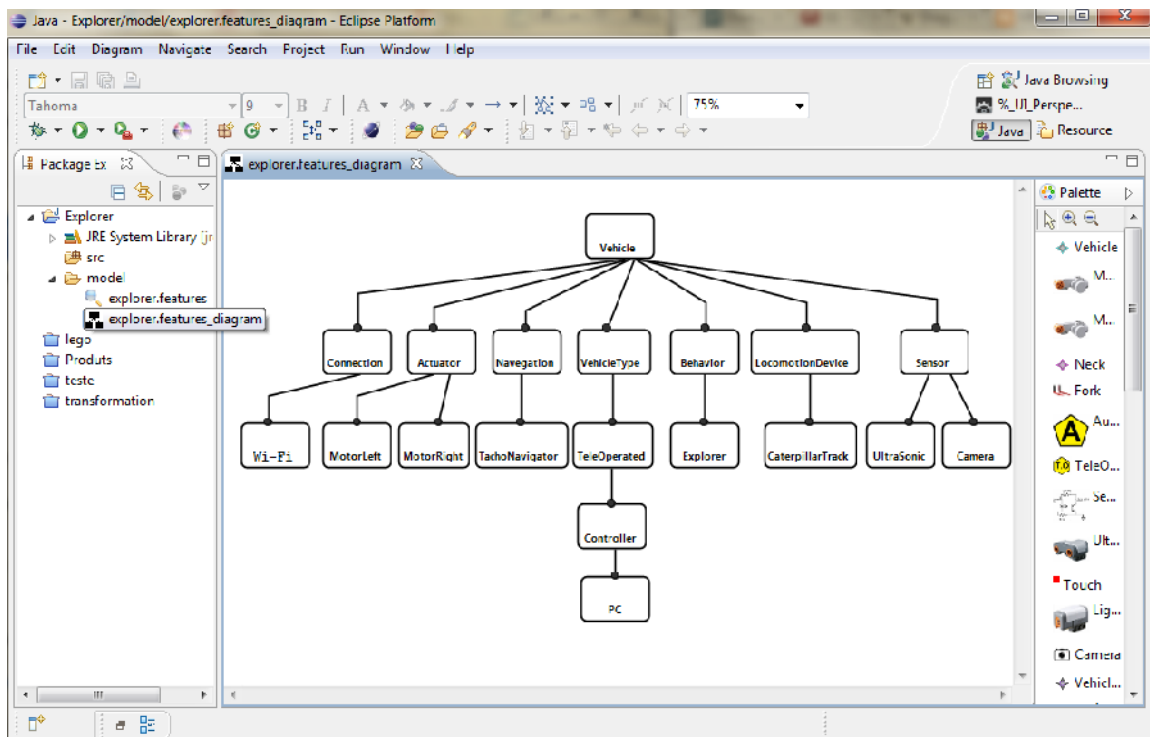


Figura 5-9 Diagrama de *Features* do Robôs Moveis Explorer

Os passos descritos a seguir são utilizados para a validação do diagrama de *features* criado, a fim de verificar se ele está semanticamente correto. Nessa validação deve ser usado o arquivo que possui a extensão “.*features*”, nesse caso, no “*Explorer.features*”.

1. Clicar duas vezes sobre o arquivo “*Explorer.features*”, o qual irá apresentar todas as *features* do diagrama de *features* em uma estrutura de árvore (Figura 5-10(a)).
2. Clicar com o botão direito na aba “*plataforma:/resource/Explorer/model/explorer.features*” (Figura 5-10(b)) e selecionar a opção “*Validade*” (Figura 5-10(c)).

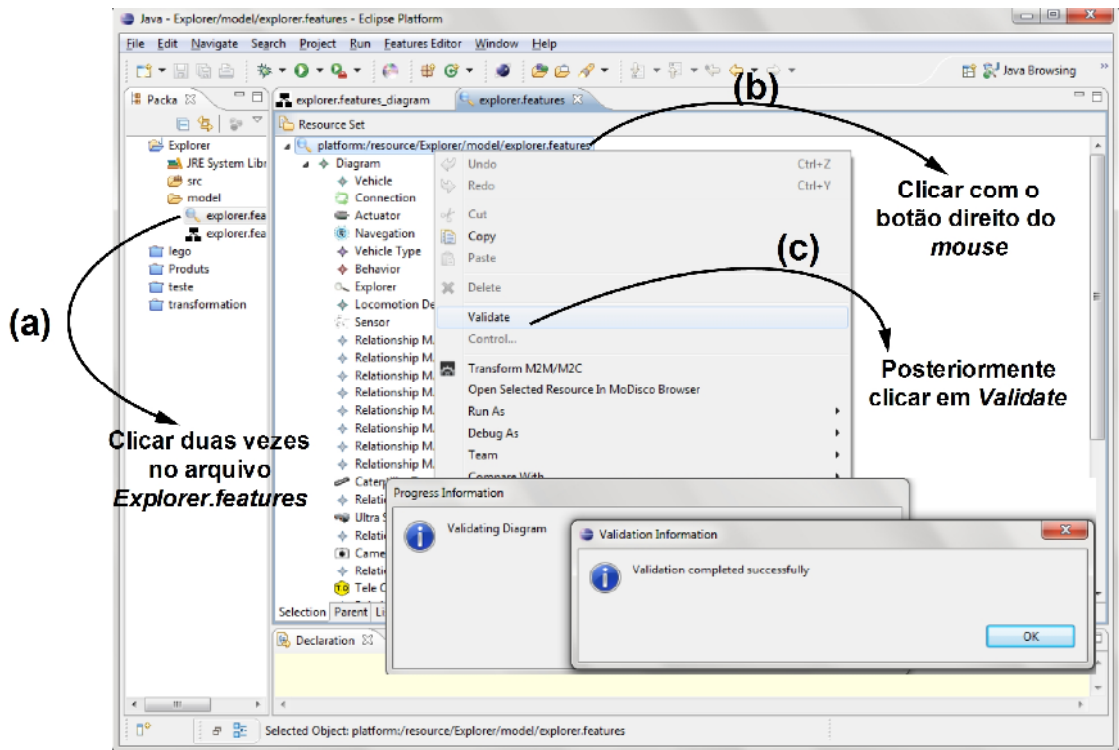


Figura 5-10 - Validando o Diagrama de *Features* do Robôs Moveis Explorer

As janelas “*Progress Information*” e “*Validation Information*” informam ao engenheiro de aplicação se o modelo está ou não semanticamente correto.

Após o desenvolvimento do diagrama de *features* e da validação do mesmo, podem-se iniciar as transformações M2M e M2C.

Para realizar tais transformações o engenheiro de aplicação deve realizar os seguintes passos.

1. Clicar com o botão direito sobre o arquivo “*Explorer.features*” (Figura 5-11(a));
2. Escolher o botão “*Transform M2M/M2C*” (Figura 5-11(b)) para realizar as transformações M2M e M2C respectivamente.

Nas Figuras 5-12 e 5-13 são apresentados, respectivamente, o diagrama de classes e o código fonte gerados automaticamente pela F2MoC para o RM considerado neste exemplo.

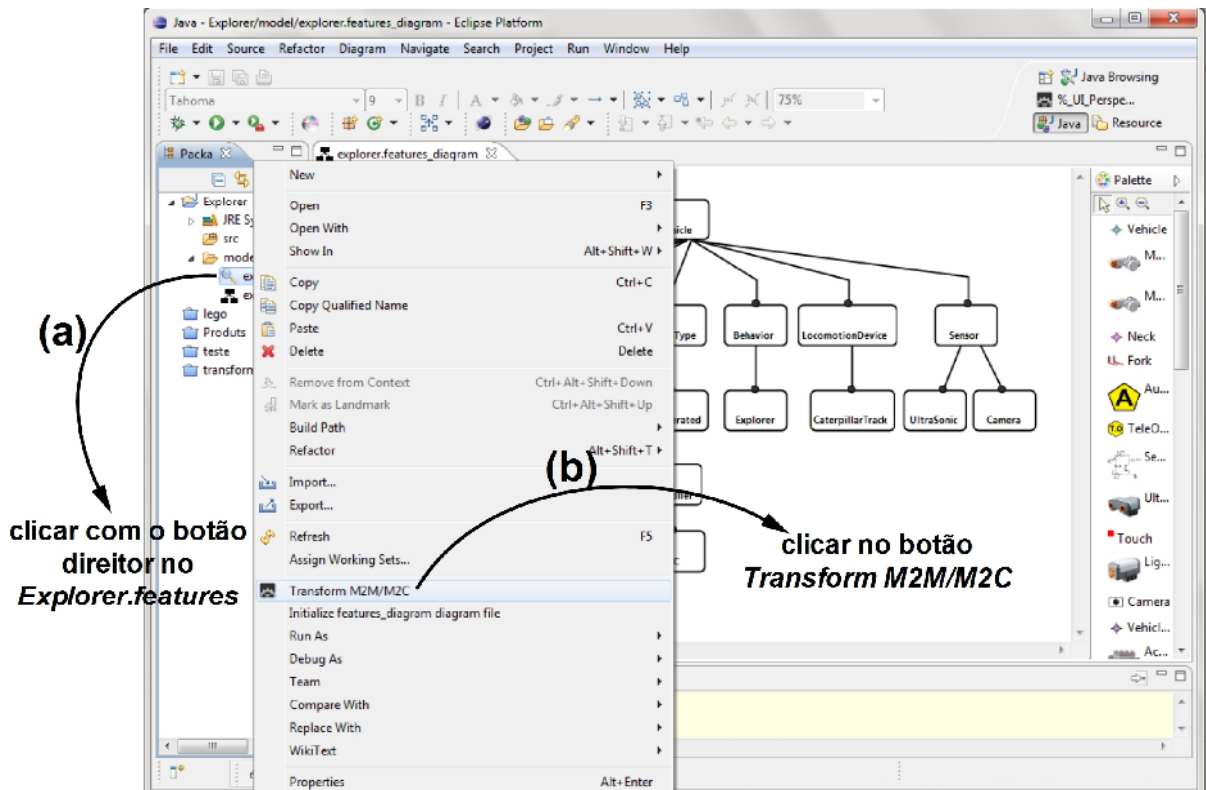


Figura 5-11 - Transformações M2M e M2C

O diagrama de classe gerado tem como principal objetivo aumentar o entendimento do engenheiro da aplicação, diminuir a distância semântica entre as pessoas envolvidas no projeto, uma vez que a UML é uma linguagem unificada, e por fim, servir como documentação para a aplicação gerada.

As *features* ilustradas no diagrama de *features* apresentado na Figura 5-9 foram mapeadas para o diagrama de classes exibido na Figura 5-12, por meio de transformações M2M, sendo que as *features* opcionais e alternativas foram implementadas com o auxílio de padrões de projeto (Gamma, 1994). Por exemplo, o conjunto de *features* “Sensors”, ou seja, as *features* “Camera” e “Ultrasonic” foram implementadas utilizando-se o padrão *Builder*.

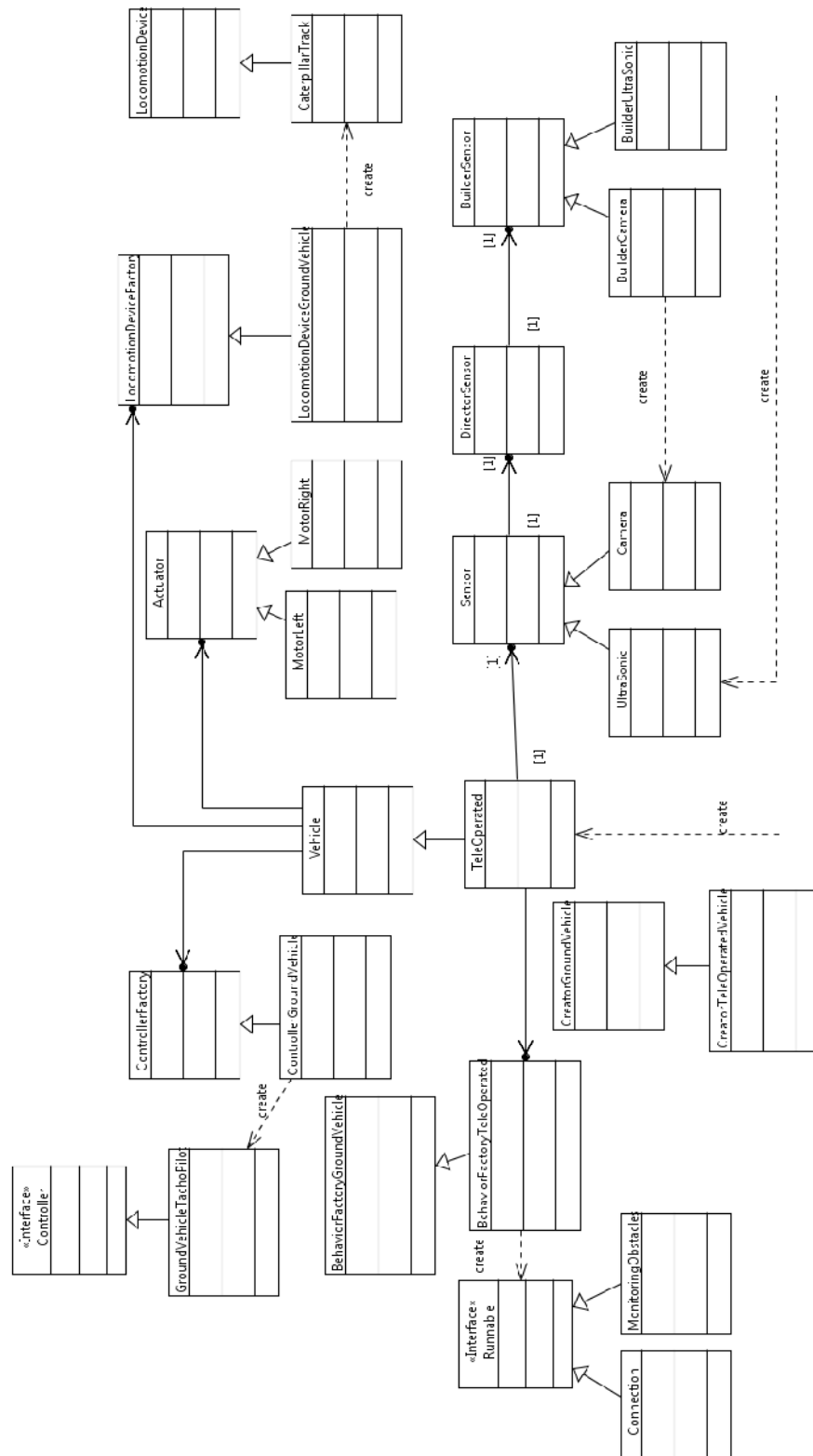


Figura 5-12 - Diagrama de Classe do RMs Explorer

De forma semelhante, a *feature* “CaterpillarTrack” foi implementada com a utilização do *Factory Method*. Maiores informações sobre o mapeamento de diagrama de *features* para diagrama de classes podem ser encontradas em Almeida et al (2007).

Na Figura 5-13 são apresentados os códigos-fontes gerados nas transformações M2C, ou seja, as classes que serão embarcadas no RM “*explorer*”. Na Tabela 5-2 são exibidas as informações das classes que foram geradas, seu nome, a quantidade de NLC¹² e a sua superclasse quando existir.

Tabela 5-2 - Classes Geradas na Transformação M2C do RM Explorer

Classe da Aplicação	NLC	Superclasse
<i>Actuator</i>	10	--
<i>MotorLeft</i>	50	<i>Actuator</i>
<i>MotorRight</i>	50	<i>Actuator</i>
<i>Connection</i>	100	<i>Runnable</i>
<i>MonitoringObstacles</i>	65	<i>Runnable</i>
<i>BehaviorFactoryGroundVehicle</i>	38	--
<i>BehaviorFactoryTeleOperated</i>	31	<i>BehaviorFactoryGroundVehicle</i>
<i>Controller</i>	104	--
<i>GroundVehicleTachoPilot</i>	653	<i>Controller</i>
<i>ControllerFactory</i>	49	--
<i>ControllerGroundVehicle</i>	32	<i>ControllerFactory</i>
<i>CreatorGroundVehicle</i>	45	--
<i>CreatorTeleOperatedVehicle</i>	33	<i>CreatorGroundVehicle</i>
<i>Vehicle</i>	92	--
<i>TeleOperated</i>	103	<i>Vehicle</i>
<i>BuilderSensor</i>	16	--
<i>Sensor</i>	30	--
<i>UltraSonic</i>	65	<i>Sensor</i>
<i>Camera</i>	53	<i>Sensor</i>
<i>DirectorSensor</i>	32	--
<i>BuilderUltraSonic</i>	48	<i>BuilderSensor</i>
<i>BuilderCamera</i>	101	<i>BuilderSensor</i>
<i>DirectorSensor</i>	25	--
<i>Main</i>	27	--

Na Figura 5-14 é apresentado um trecho do código-fonte da classe “*TeleOperated*”. As linhas 3 a 7 e as linhas 9 a 14 correspondem, respectivamente, aos atributos ao construtor dessa classe.

¹² NLC significa número de linhas de código, em inglês, LOC.

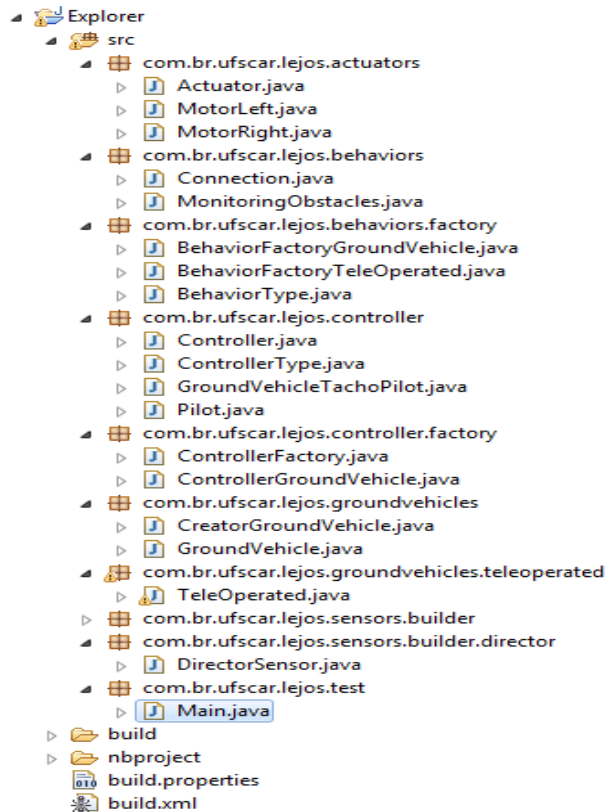


Figura 5-13 - Estrutura do Código-Fonte gerado do RM Explorer

```

1 public class TeleOperated extends Vehicle{
2
3     private BehaviorType behaviorType;
4
5     private List sensors;
6
7     private Runnable[] behaviors;
8
9     public TeleOperated (BehaviorType behaviorType, ControllerType controllerType,
10         float wheelDiameter, float trackWidth, Actuator... actuators){
11         super(controllerType, wheelDiameter, trackWidth);
12         this.behaviorType = behaviorType;
13     }
14 }
15
16 @Override
17 public void constructVehicle(ControllerType controllerType,
18     float wheelDiameter, float trackWidth) {
19     super.constructVehicle(controllerType, wheelDiameter, trackWidth);
20     this.buildSensors(this.behaviorType);
21     this.buildScheduler(this.buildBehaviors());
22 }
23
24 public Runnable[] buildBehaviors(){
25
26     BehaviorFactoryGroundVehicle behaviorFactory = BehaviorFactoryGroundVehicle.getInstance(this);
27     return (Runnable[])behaviorFactory.getBehaviors();
28 }
29
30 public void buildScheduler( Runnable[] behaviors ){
31
32     this.setBehaviors(behaviors);
33 }
34 }
35
36 @Override
37 public void buildActuators() {
38     // TODO Auto-generated method stub
39     super.buildActuators();
40 }
41 ...

```

Figura 5-14 – Trecho do código da classe `TeleOperated` gerado a partir do diagrama de *features* ilustrado na Figura 5-5

O método principal dessa classe é o “*constructVehicle()*”, o qual é responsável por instanciar todas as dependências requeridas para que o RM funcione, ou seja, criar todas as instâncias dos sensores, atuadores, escalonador, etc.

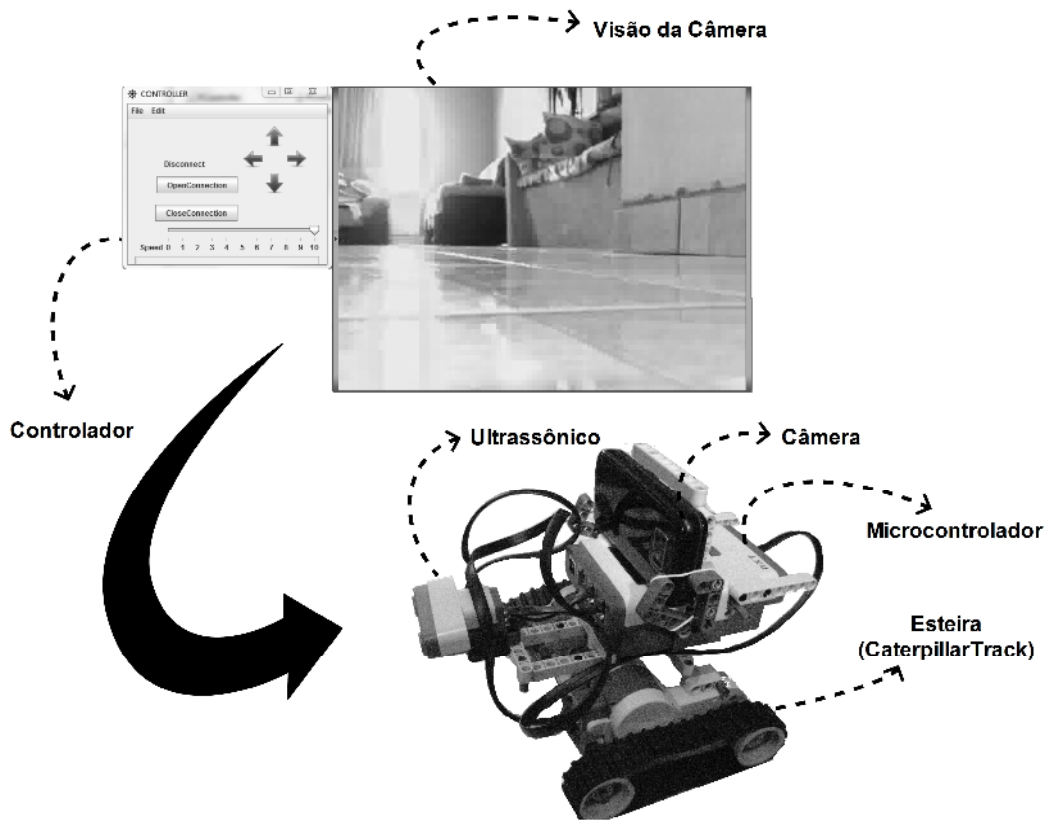


Figura 5-15 - RM Explorer

Após a geração do código-fonte, o mesmo foi embarcado no hardware do RM desenvolvido. Dessa forma, na Figura 5-15 é apresentada uma visão geral do RM e do seu funcionamento.

5.3 Exemplo 2: Geração do RM “*ForkLift*”.

O segundo exemplo aborda o desenvolvimento de um RM industrial. Geralmente RM industriais são utilizados em linha de montagem de produtos, transporte de materiais e nas demais tarefas realizadas dentro de uma indústria.

O RM apresentado neste exemplo tem por função realizar transporte de materiais, ou seja, levar um objeto de um determinado lugar para outro. Na Tabela 5-3 contém os requisitos desse RM.

Tabela 5-3 - Requisitos do RM2 ForkLift

#	Descrição
1	O RM deve pegar um determinado objeto, seguir uma fita adesiva entre dois pontos no qual corresponde ao início e fim do trajeto do RM;
2	O RM deve ter um atuador no qual fornece a funcionalidade de uma empilhadeira;
3	O ambiente no qual o RM usualmente é operado possui vários objetos e pessoas que devem ser evitados pelo RM;
4	O RM também deve ser equipado com motores;
5	O RM deve utilizar como dispositivo de locomoção pneus;
6	O RM deve ser totalmente autônomo.

Os passos para criação de um projeto utilizando a DSL F2MoC são os mesmos descritos anteriormente no exemplo 1.

A criação do diagrama de *features* do RM inicia-se a partir dos requisitos solicitados pelo cliente (Tabela 5-3). Porém, inicialmente, as *features* obrigatórias (“Actuators”, “Navigation”, “VehicleType”, “Behavior”, “Locomotion Device” e “Sensor”) devem incluídas no diagrama de *features*. Nesse ponto, cabe ao engenheiro de aplicação novamente analisar os requisitos a fim de completar o diagrama de *features*, escolhendo as *features* opcionais e alternativas existentes na LRMLPS.

O requisito 1 da Tabela 5-3 informa que fitas adesivas escuras devem ser colocadas no chão da fábrica para marcar o trajeto que o RM deverá realizar. Esse requisito não é atendido pela F2MoC, essa colocação deve ser feita por pessoas que pertencem ao ambiente em que o RM irá atuar.

A *feature* “CargoShip”.do diagrama de *features* da LRMLPS representa uma abstração da funcionalidade de pegar um determinado objeto e leva-lo para outro lugar. Essa *feature* possui duas *subfeatures* “Charger” e “ForkLift”, que podem levar objetos de um lugar para outro, porém a *feature* “ForkLift” adiciona ao RM a capacidade de atuar como uma empilhadeira. Dessa forma, as *features* “CargoShip” e “ForkLift” foram selecionadas para satisfazer o requisito 2 da Tabela 5-3.

A *feature* “Light” tem a funcionalidade de identificar a intensidade de diferentes cores e, neste exemplo, foi selecionada para auxiliar ao RM ForkLift na identificação da fita adesiva e segui-la.

Usualmente, fabricas possuem vários obstáculos e pessoas trabalhando e transitando. Dessa maneira, a fim de evitar que o RM *ForkLift* realize colisões e coloque em risco a segurança dos funcionários da fábrica, o mesmo deve possuir um sensor de proximidade, assim a *feature* “*Ultrasonic*” também foi selecionada.

O RM *ForkLift* deve possuir motores, dessa forma as *features* “*MotorLeft*” e “*MotorRight*” foram selecionadas. De acordo com o requisito 5 da Tabela 5-3 o dispositivo de locomoção do RM deve ser pneu, assim a *feature* “*Wheel*” também foi selecionada. Por fim, a *feature* “*Autonomous*” foi selecionada, pois de acordo com os requisitos o RM deve ser totalmente autônomo. Na Figura 5-16 é apresentado o diagrama de *feature* do RM desenvolvido.

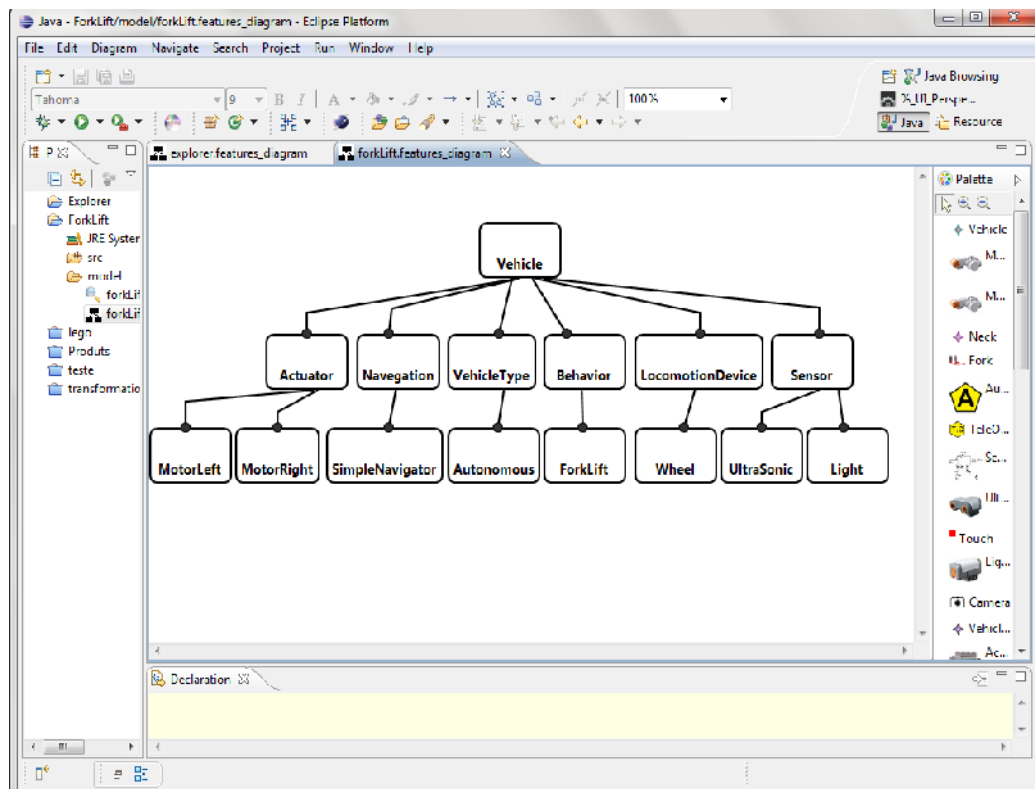


Figura 5-16 - Diagrama de *Features* do RM ForkLift

Após o desenvolvimento do diagrama de *features* do RM, esse deve ser validado, antes de realizar as transformações M2M e M2C.

Após a modelagem e validação do digrama de *features* é possível realizar as transformações M2M e M2C fornecidas pela F2MoC. Os passos necessários para realizar tais transformações são os mesmos já apresentados no exemplo 1 e ilustrados na Figura 5-11.

Tabela 5-4- Classes Geradas na Transformação M2C do RM Forklift

Classe da Aplicação	NLC	Superclasse
<i>Actuator</i>	10	--
<i>MotorLeft</i>	50	<i>Actuator</i>
<i>MotorRight</i>	80	<i>Actuator</i>
<i>Fork</i>	100	<i>Actuator</i>
<i>Builder</i>	56	--
<i>BuilderFork</i>	54	<i>Builder</i>
<i>DirectorFork</i>	65	--
<i>DriveForward</i>	65	<i>Behavior</i>
<i>Observer</i>	52	<i>Behavior</i>
<i>OffLine</i>	48	<i>Behavior</i>
<i>BehaviorFactoryGroundVehicle</i>	33	--
<i>BehaviorFactoryAutonomous</i>	56	<i>BehaviorFactoryGroundVehicle</i>
<i>Controller</i>	104	--
<i>GroundVehicleTachoPilot</i>	653	<i>Controller</i>
<i>ControllerFactory</i>	49	--
<i>CreatorGroundVehicle</i>	45	--
<i>CreatorAutonomousVehicle</i>	34	<i>CreatorGroundVehicle</i>
<i>ControllerGroundVehicle</i>	32	<i>ControllerFactory</i>
<i>Vehicle</i>	85	--
<i>Autonomous</i>	106	<i>Vehicle</i>
<i>BuilderSensor</i>	16	--
<i>Sensor</i>	34	--
<i>LightSensor</i>	45	<i>Sensor</i>
<i>UltraSonic</i>	45	<i>Sensor</i>
<i>BuilderUltraSonic</i>	48	<i>BuilderSensor</i>
<i>BuilderLight</i>	61	<i>BuilderSensor</i>
<i>DirectorSensor</i>	25	--
<i>Main</i>	27	--

Na Figura 5-17 é apresentado o resultado da transformação M2M desse exemplo. Na Tabela 5-4 estão apresentadas as informações das classes que foram geradas durante a transformação M2C, bem como o NLC, o nome de cada classe e o da sua superclasse, quando existir. Na Figura 5-18 é apresentada a estrutura dos

códigos-fontes gerados pela DSL e na Figura 5-19, os trechos do código-fonte da classe “Autonomous”.

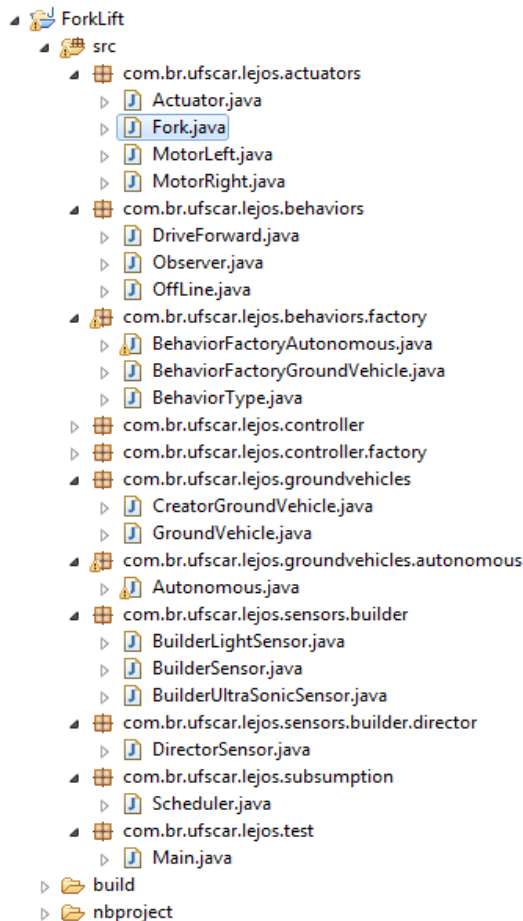


Figura 5-18 – Estrutura do Código-Fonte gerado do RM ForkLift

```

1 public class Autonomous extends GroundVehicle{
2
3     private List sensors;
4     private Scheduler scheduler;
5     protected BehaviorType behaviorType;
6
7
8     public Autonomous (BehaviorType behaviorType, ControllerType controllerType, float
9     wheelDiameter, float trackWidth){
10         super(controllerType, wheelDiameter, trackWidth);
11         this.behaviorType = behaviorType;
12     }
13
14     public List getSensors() {
15         return sensors;
16     }
17
18     public void setSensors(List sensors) {
19         this.sensors = sensors;
20     }
21
22     @Override
23     public void constructVehicle(ControllerType controllerType, float wheelDiameter, float trackWidth) {
24         super.constructVehicle(controllerType, wheelDiameter, trackWidth);
25         buildSensors(this.behaviorType);
26         buildScheduler(buildBehaviors());
27     }
28
29     ...

```

Figura 5-19 - Trecho do código da classe Autonomous gerado a partir do diagrama de features ilustrado na Figura 5-12

Na Figura 5-19 a classe “*Autonomous*” estende a classe “*GroundVehicle*”. Na linha 3 a lista *sensors* contém todas as instancias dos sensores utilizados pelo RM. O atributo *scheduler* (linha 4) representa uma instancia do escalonador do RM, o mesmo é utilizado para permitir que várias tarefas sejam realizadas paralelamente. Por exemplo, o RM deve constantemente obter informações do sensor de luz para seguir as fitas adesivas. Deve também, em paralelo, monitorar as informações obtidas do sensor ultrassônico para evitar qualquer tipo de colisão.

A classe “*BehaviorType*” é responsável por definir o tipo de comportamento do RM e uma instancia dessa classe é a apresentada na linha 5. Nas linhas de 8 a 12 está representado o construtor da classe “*Autonomous*” e nas linhas de 23 a 27 está descrito o método “*constructVehicle(...)*”, responsável por instanciar todos os recursos necessários do RM.

Na Figura 5-20 tem-se uma visão geral dos hardwares do RM *ForkLift* desenvolvidos neste exemplo.

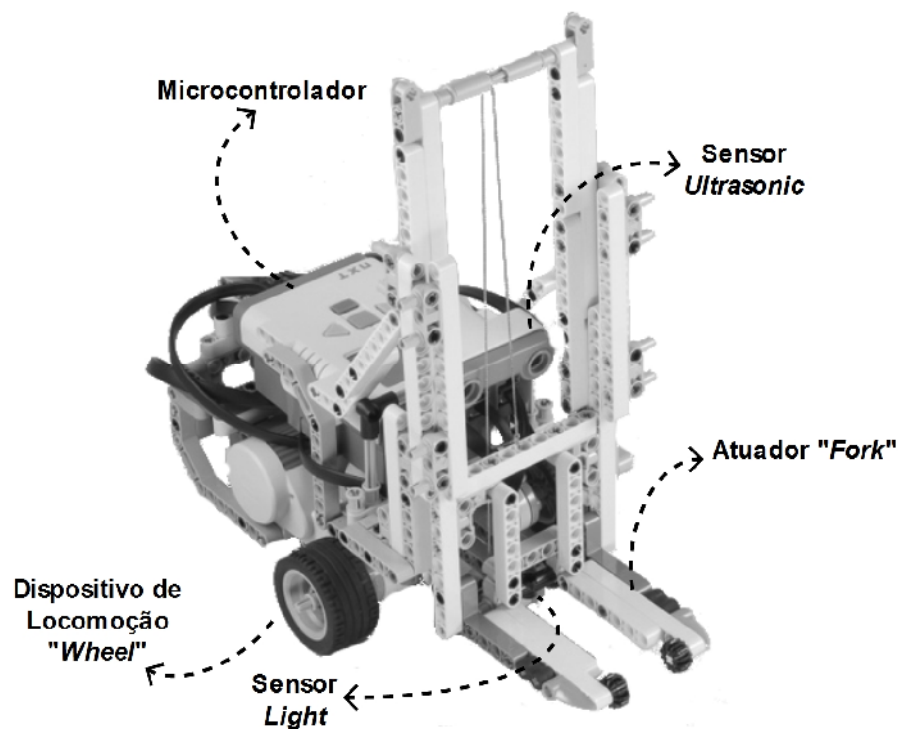


Figura 5-20 - RM ForkLift

5.4 Exemplo 3: Geração do RM “HomeSentry”

O terceiro exemplo refere-se a um RM que tem como funcionalidade monitorar uma determinada residência. Na Tabela 5-5 estão relacionados os requisitos desse RM.

Tabela 5-5 - Requisitos do RM HomeSentry

#	Descrição
1	O RM é responsável por monitorar uma determinada residência;
2	O RM deve utilizar um sensor ultrassônico e um sensor de toque para detectar qualquer anormalidade no ambiente;
3	O RM deve possuir uma câmera que é utilizada para gravar e transmitir informações em tempo real a uma central de vigilância externa;
4	O RM deve possuir um mecanismo de comunicação para transmitir informações obtidas pela câmera em tempo real a uma central de vigilância;
5	O dispositivo de locomoção utilizado pelo RM é pneus;
6	O RM deve ser totalmente autônomo;
7	O RM também deve ser equipado com motores.

Os passos para criação de um projeto utilizando a DSL são os mesmos descritos anteriormente no exemplo 1.

Com base nos requisitos listados na Tabela 5-5, o engenheiro de aplicação inicia a criação do diagrama de *features* do RM. As *features* obrigatórias “Actuators”, “Navigation”, “VehicleType”, “Behavior”, “Locomotion Device” e “Sensor” foram incluídas no diagrama de *features*.

A *feature* “HomeSentry” representa o requisito 1 da Tabela 5-5 e foi selecionada para especificar o comportamento do RM. O requisito 2 é satisfeito com a seleção das *features* “Light” e “Ultrasonic”. As *features* “Wheel”, “Câmera”, “Connection” e “Wi-Fi”, “Autonomous”, “MotorLeft” e “MotorRight” foram selecionadas para atender aos requisitos de 3 a 7. Na Figura 5-21 é apresentado o diagrama de *feature* do RM desenvolvido.

Os passos de validação são os mesmos apresentados para o exemplo 1, e ilustrados na Figura 5-10. Após a realização desses passos, verificou-se que o diagrama de *features* está semanticamente correto.

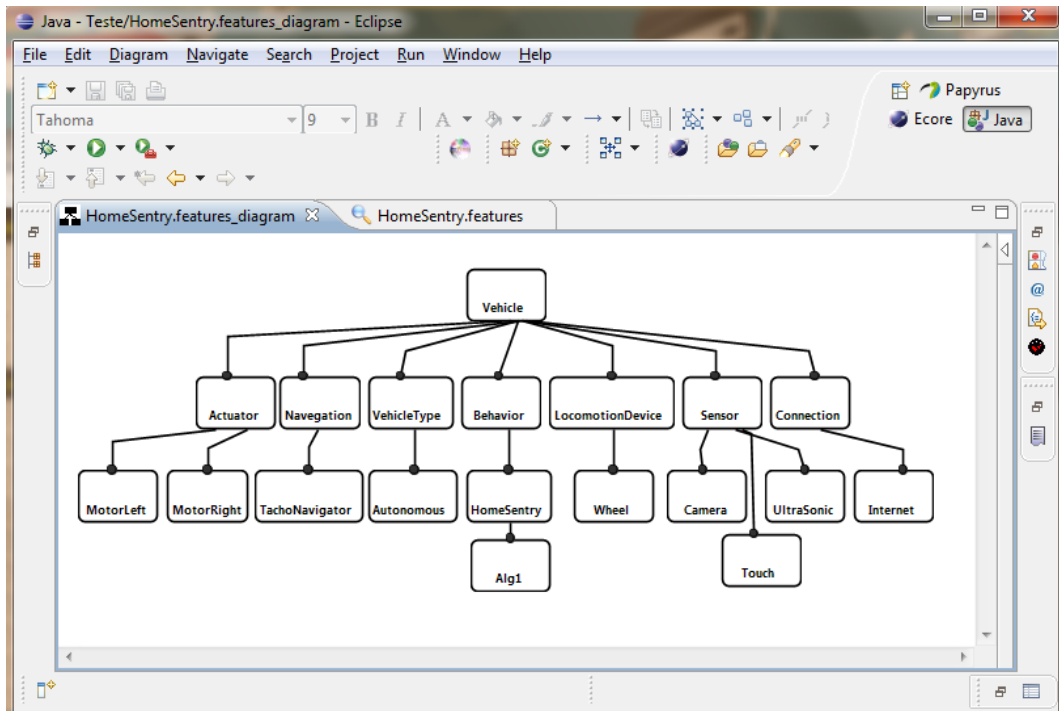


Figura 5-21 - Diagrama de *Features* do RM *HomeSentry*

Com a validação concluída o engenheiro de aplicação pode iniciar as transformações M2M e M2C do RM modelado. A Figura 5-22 apresenta o diagrama de classe gerado com base no diagrama de *features* ilustrado na Figura 5-21.

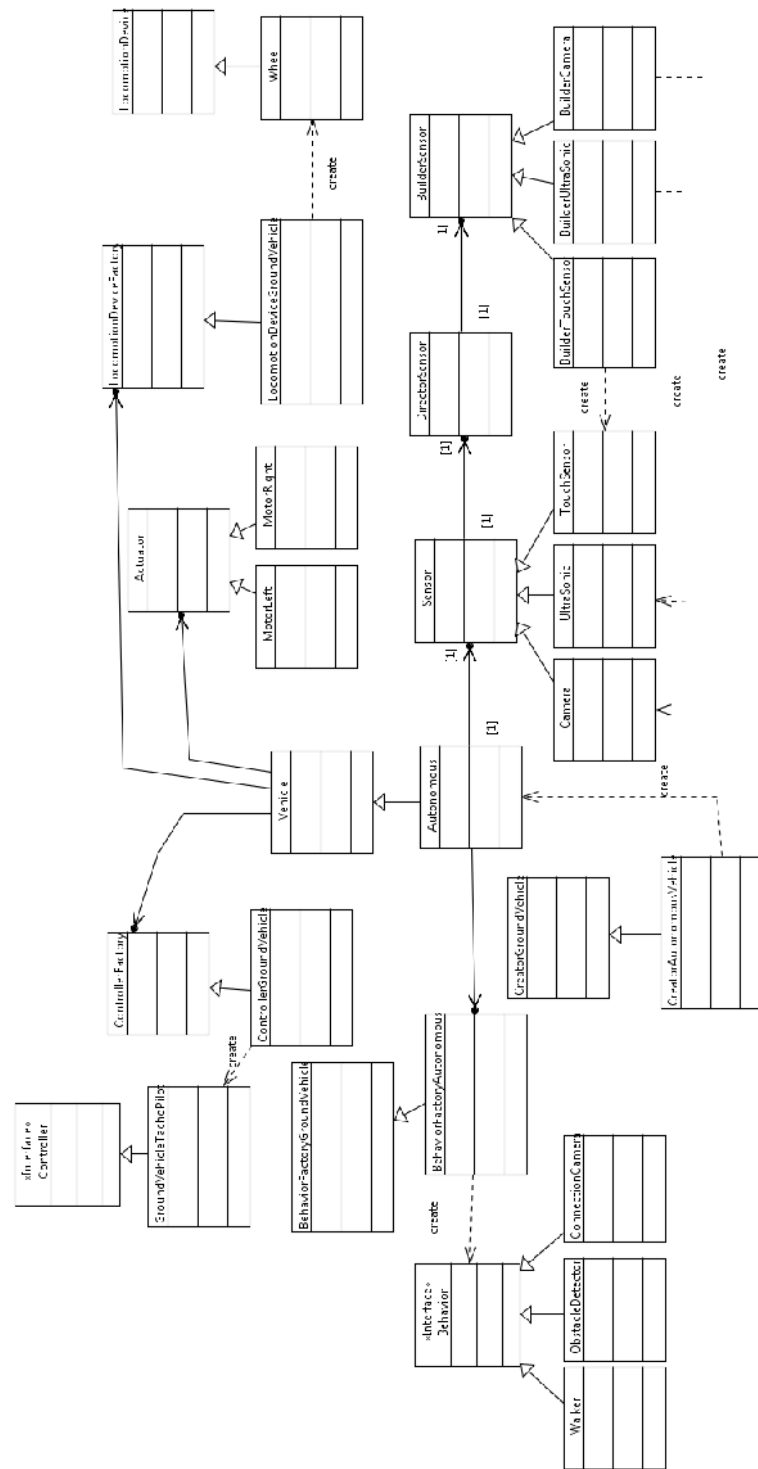


Figura 5-22 - Diagrama de Classe do RMs HomeSentry

Na Tabela 5-6 estão apresentadas informações das classes que foram geradas no decorrer da transformação M2C deste exemplo, bem como o nome da classe, o NLC e seus respectivos relacionamentos.

Nas Figuras 5-23 e 5-24 são apresentados, respectivamente, a estrutura do código-fonte gerado pela DSL e trechos de código-fonte da classe “*ObstacleDetector*”.

A classe “*ObstacleDetector*” (Linha 1 da Figura 5-24) implementa a interface “*lejos.subsumption.Behavior*”, que possui três métodos que devem ser implementados “*action()*” (linhas 12-27), “*boolean takeControl()* (29-33)” e “*suppress()*” (não apresentado). O método *takeControl()* retorna um valor booleano “*true*” se o sensor de toque indicar que o RM colidiu com algum objeto ou se o sensor ultrassônico identificar um possível obstáculo”. Nesse caso, o método *action()* é acionado para executar o comportamento do RM .

Tabela 5-6 - Classes Geradas na Transformação M2C do RM HomeSentry

Classe da Aplicação	NLC	Superclasse
<i>Actuator</i>	10	--
<i>MotorLeft</i>	50	<i>Actuator</i>
<i>MotorRight</i>	80	<i>Actuator</i>
<i>Walker</i>	80	<i>Behavior</i>
<i>ObstacleDetector</i>	56	<i>Behavior</i>
<i>ConnectionCamera</i>	39	<i>Behavior</i>
<i>BehaviorFactoryGroundVehicle</i>	33	--
<i>BehaviorFactoryAutonomous</i>	56	<i>BehaviorFactoryGroundVehicle</i>
<i>Controller</i>	104	--
<i>GroundVehicleTachoPilot</i>	653	<i>Controller</i>
<i>ControllerFactory</i>	49	--
<i>ControllerGroundVehicle</i>	32	<i>ControllerFactory</i>
<i>Vehicle</i>	85	--
<i>Autonomous</i>	106	<i>Vehicle</i>
<i>Sensor</i>	45	--
<i>UltraSonicSensor</i>	32	<i>Sensor</i>
<i>TouchSensor</i>	28	<i>Sensor</i>
<i>Camera</i>	45	<i>Sensor</i>
<i>BuilderSensor</i>	16	--
<i>BuilderUltraSonic</i>	48	<i>BuilderSensor</i>
<i>BuilderTouchSensor</i>	61	<i>BuilderSensor</i>
<i>BuilderCamera</i>	100	<i>BuilderSensor</i>
<i>DirectorSensor</i>	25	--
<i>Main</i>	27	--

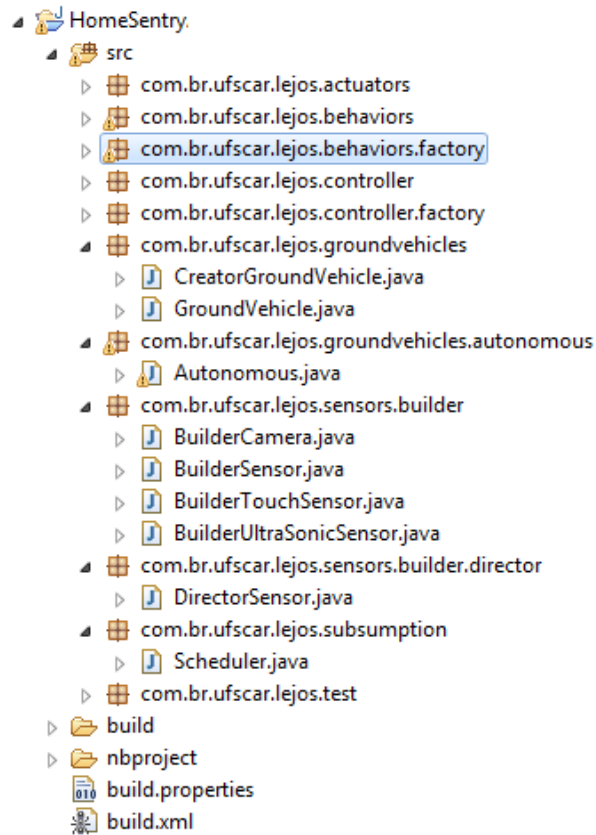


Figura 5-23 - Estrutura do Código-Fonte gerado do RM HomeSentry

```

1 public class ObstacleDetector implements Behavior {
2     private TouchSensor touch;
3     private UltrasonicSensor sonic;
4     private Controller pilot;
5     private static final float[] ANGLE = { 90.0f, 180.0f, 360.0f, -90.0f, -180.0f, -360.0f };
6
7     public ObstacleDetector(List sensors, Controller pilot) {
8         this.sonic = (UltrasonicSensor) sensors.get(0);
9         this.touch = (TouchSensor) sensors.get(1);
10        this.pilot = pilot;
11    }
12    public void action() {
13        // TODO Auto-generated method stub
14        int choice = (int) (Math.random() * 2);
15        if (choice == 0) {
16            this.pilot.rotate(ObstacleDetector.ANGLE[(int) (Math.random() * 6)]);
17            if (pilot instanceof GroundVehicleTachoPilot) {
18                ((GroundVehicleTachoPilot) pilot).setSpeed(800);
19            }
20        } else {
21            Thread.yield();
22            pilot.backward();
23            if (pilot instanceof GroundVehicleTachoPilot) {
24                ((GroundVehicleTachoPilot) pilot).setSpeed(200);
25            }
26        }
27    }
28    @Override
29    public boolean takeControl() {
30        this.sonic.ping();
31        Sound.pause(20);
32        return (this.touch.isPressed() || this.sonic.getDistance() < 25);
33    }
34    ...
35 }

```

Figura 5-24 - Trecho do código da classe ObstacleDetector gerado a partir do diagrama de *features* ilustrado na Figura 5-18

5.5 Considerações Finais

Este Capítulo apresentou em detalhes o processo que deve ser empregado para a instanciação de membros da LRMLPS por meio de uma DSL gráfica desenvolvida neste trabalho (F2MoC). Dessa forma, três exemplos foram conduzidos, para analisar a viabilidade e facilidade de desenvolvimento de membros da linha de produtos LRMLPS juntamente com a DSL gráfica desenvolvida.

Nesses exemplos foram utilizados diferentes requisitos para a instanciação de RMs. O uso da F2MoC facilita o desenvolvimento de um membro da linha, pois o engenheiro de aplicação tem a sua disposição todas as *features* que podem compor essa linha. Dessa forma, a instanciação de um membro é facilitada.

Outra vantagem que pode ser observada é o fato de que a partir do diagrama de *features*, outros dois artefatos podem ser obtidos: o de classes e a geração automática de código fonte. O modelo de classes serve para documentação do sistema em desenvolvimento, pois nem sempre todos os envolvidos conhecem diagramas de *features*. O código fonte gerado automaticamente garante rápida instanciação de membros da linha.

Assim, os exemplos realizados mostraram que MDD e LPS, quando combinados no contexto de SE, proporcionam menor tempo para criação de um sistema, alto nível de abstração e reuso, evitando assim, que o usuário final tenha que se lembrar de detalhes de baixo nível do domínio para instanciar um membro da LPS.

Capítulo 6

CONCLUSÕES

6.1 Considerações Finais

Este trabalho apresentou um processo para o desenvolvimento de linhas de produtos de software para o domínio de *Robôs Moveis* terrestres. Para exemplificar esse processo foi desenvolvida uma LPS intitulada ***LegoMobileRobot Software Product Line*** (LRMLPS) a qual utilizou várias tecnologias tais como o kit LEGO *Mindstorms*, uma máquina virtual chamada *Lego Java Operating System* (LeJOS) e os frameworks *Eclipse Modeling Framework* (EMF) e *Graphical Modeling Project* (GMF). O kit LEGO *Mindstorms* possui um conjunto de sensores e atuadores os quais foram utilizados como hardware dos *Robôs Moveis* da LRMLPS. A LeJOS é uma máquina virtual Java que foi adaptada para fornecer uma abstração aos sensores e atuadores do kit da LEGO *Mindstorms*. Dessa forma, a LeJOS foi utilizada para o desenvolvimento das *features* comuns e variáveis da LRMLPS. Os frameworks EMF e GMF foram utilizados para o desenvolvimento de uma DSL gráfica que auxilia o engenheiro de aplicação na instanciação de um determinado membro da LRMLPS. Essa DSL possui uma abstração de todas as *features* pertencentes ao diagrama de *features* da LRMLPS. Assim, o engenheiro de aplicação deve selecionar um conjunto de *features* de forma sistemática para realizar a instanciação de um determinado membro. Adicionalmente essa DSL permite a realização de transformações M2M e M2C.

As contribuições e limitações deste trabalho são descritas nas Seções 6.2 e 6.3, respectivamente. Na Seção 6.4 são apresentadas sugestões para trabalhos futuros para complementar o aqui apresentado.

6.2 Contribuições

Neste trabalho procurou-se resolver alguns dos problemas encontrados no domínio de sistemas embarcados tais como, diminuir o tempo de desenvolvimento de um sistema embarcado (*time-to-market*), aumentar o índice de reuso nesse domínio e aumentar também a qualidade desses sistemas com a utilização técnicas já consagradas da Engenharia de Software como, linha de produtos de software e desenvolvimento de software orientado a modelo. Estudos e pesquisas nas áreas relacionadas, foram realizados o que resultou em um processo de desenvolvimento de LPS para o domínio de Robôs Moveis. Esse processo guia o engenheiro de domínio desde atividades iniciais de análise até a implementação de artefatos reutilizáveis e de uma DSL gráfica que auxilia o engenheiro de aplicação na instanciação de membros de uma linha desenvolvida. Nesse sentido, é possível ressaltar as seguintes contribuições deste trabalho:

- Um processo sistemático, contendo atividades, entradas e saídas que detalham as tarefas necessárias para que o engenheiro de domínio possa desenvolver uma linha de produtos de software ao domínio de *Robôs Moveis*.
- A constatação da efetividade do uso de linha de produtos de software para o desenvolvimento de sistemas embarcados mais especificadamente para *Robôs Moveis*, gerando a linha de produtos de software intitulada de LRMLPS. Dessa forma, acreditasse que houve melhoria na reusabilidade e diminuição do *time-to-market* de produtos pertencentes a linha de produtos desenvolvida.
- A constatação da integração de técnicas tais como desenvolvimento de software orientado a modelo (MDD) e linha de produtos de software (LPS) no domínio de sistemas embarcados. Assim, pode-se observar que LPS e MDD quando combinados podem ser utilizados no contexto de sistemas embarcados proporcionando menor tempo de criação de um sistema, aumento de reuso e alto nível de abstração fazendo com que o usuário final não tenha que conhecer informações específica do domínio para criar um membro da LPS.

- A DSL desenvolvida realiza transformações M2M e M2C. Nas transformações M2M um diagrama de *features* que corresponde a um determinado membro da linha é utilizado como entrada para a geração de um diagrama de classe. Esse diagrama de classe pode ser utilizado para diminuir a distância semântica entre os envolvidos no projeto, uma vez que a UML é uma linguagem unificada e padronizada. As transformações M2C geram o código-fonte da aplicação, que deve ser embarcada nos *Robôs Moveis* para que possa ser executado. Para realizar essa transformação é utilizado também um diagrama de *features* que corresponde as *features* comuns e variáveis de um determinado membro da LPS.
- Inferisse que pode haver a diminuição do tempo e maior facilidade de desenvolvimento de um determinado membro pertencente a LRMLPS com a utilização da F2MoC. Uma vez que o engenheiro de aplicação desenvolve um membro com base em modelos (diagrama de *features*) ao invés de códigos-fontes poupando o de conhecer detalhes específicos de plataforma.

6.3 Limitações do Trabalho Efetuado

As limitações identificadas neste trabalho são:

- Somente uma LPS foi desenvolvida utilizando o processo proposto. É preciso que se desenvolvam outras linhas de produtos de software utilizando o processo aqui apresentado para que as atividades do mesmo possam ser adaptadas e melhoradas e se convertam em um processo mais completo.
- A realização dos exemplos deste trabalho mostrou que LPS e MDD quando combinados podem diminuir o *time-to-market*, aumentar o reuso de software no domínio de *Robôs Moveis*. Entretanto é necessário realizar uma avaliação quantitativa tanto da LPS quanto da DSL desenvolvida. Assim, será possível fazer uma avaliação mais

precisa sobre o nível de reuso e medir o tempo gasto para a instanciação de membros da linha desenvolvida.

- A DSL desenvolvida neste trabalho é feita como base nos frameworks EMF e GMF do IDE Eclipse (Eclipse, 2011). Assim, a utilização dessa DSL é totalmente dependente do ambiente Eclipse. Por exemplo, não é possível utilizar outro ambiente de desenvolvimento tal como o *Netbeans* (Netbeans, 2011) ou *JBuilder* (JBuilder, 2011) para utilizar a DSL desenvolvida.
- Um kit da Lego *Mindstorms* foi utilizado como base para o desenvolvimento dos *Robôs Moveis* pertencentes à LRMLPS. Porém, esse kit contém poucos sensores e atuadores o qual dificulta e restringi a evolução da linha de produtos de software desenvolvida.

6.4 Sugestões de Trabalhos Futuros

Com o objetivo de dar continuidade ao trabalho aqui desenvolvido, de forma a complementá-lo, algumas linhas de pesquisa que podem ser seguidas são as seguintes:

- Desenvolver outras linhas de produtos de software utilizando o processo aqui proposto, a fim de torna-lo mais completo. Utilizar também outros hardwares para construir novos *Robôs Moveis*, tais como: *PowerBoot*, *Pioneer P3-DX* e *AmigoBot* (Mobile, 2011);
- Adquirir novos sensores e atuadores para aumentar as variabilidades da LRMLPS.
- Realizar avaliações quantitativas tanto da LRMLPS e da DSL desenvolvida a fim de avaliar o nível de reuso obtido.
- Criar um repositório responsável por armazenar todos os artefatos desenvolvidos durante a construção da LPS. Adicionalmente, é importante definir estratégia para controle de versão dos artefatos armazenados nesse repositório.

REFERÊNCIAS BIBLIOGRÁFICAS

AGERBO, E.; CORNILS, A. How to Preserve the Benefits of Design Patterns. In: ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES AND APPLICATIONS (OOPSLA '98), 13th, 1998, New York, USA. **Proceedings...** 1998. p. 134-143.

ALEXANDER, C.; ISHIKAWA, S.; SILVERSTEIN, M. **A Pattern Language: Towns, Buildings, Construction**. 1^a. ed. Oxford University Press, 1977. 1171 p.

ALMEIDA, E. S.; ALVARO, R.; GARCIA, V. C.; NASCIMENTO, R.; MEIRA, S. L., LUCRÉIDIO, D. A Systematic Approach to Design Domain-Specific Software Architectures. **Journal of Software**, vol. 2, n. 2, p. 38-51, 2007.

AMBLER, S. W. Agile model driven development is good enough. **IEEE Software**, p. v. 20, n. 5, p. 71-73, 2003.

ANASTASOPOULOS, M.; GACEK, C. Implementing Product Line Variabilities. In: SYMPOSIUM ON SOFTWARE REUSABILITY: PUTTING SOFTWARE REUSE IN CONTEXT, 2001, New York, USA. **Proceedings...** 2001, 109-117.

ANTLR. **ANother Tool for Language Recognition**. 1989. Disponível em: <<http://www.antlr.org/>>. Acesso em: 23 ago. 2010.

APPLETON, B. **Patterns and Software: Essential concepts and terminology**. 1997. Disponível em: <<http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html>>. Acesso em: 12 jul. 2010.

ASPECTJ. AspectJ crosscutting objects for better modularity. **AspectJ**, 2001. Disponível em: <<http://www.eclipse.org/aspectj/>>. Acesso em: 15 fev. 2011.

ATKINSON, C.; BAYER, J.; MUTHIG, D. Component-Based product line development: the Kobra Approach. In: first conference on Software product lines: experience and research directions: experience and research directions, 1^a, 2001, New Yourk, USA. **Proceedings...** 2001, p. 289-301.

BAHNOT, V; ROMAN, A; TRASK, B; CORPORATION, P. Using Domain-Specific Modeling to Develop Software Defined Radio. In: **OOPSLA on Domain-Specific Modeling**, 5th, 2005, San Diego, **Proceedings...** 2005, p. 33–42.

BAYER, J; Flege, O; Knauber, P; Laqua, R; Muthig, D; Schmid, K; Widen, T; DeBaud, j. PuLSE: A Methodology to Develop Software Product Lines. In: symposium on Software reusability, 5th, 1999, New York, USA. **Proceedings...** 1999, p. 122-131.

BECK, K; Crocker, R; Meszaros, G; Vlissides, J; Coplien, J; Dominick, L; Paulisch, F. Industrial Experience with Design Patterns. In: Conference on Software Engineering, 18th, 1996, Washington, DC. **Proceedings...** 1996 p. 103-114.

BIGGERSTAFF, T. J.; PERLIS, A. J. **Software Reusability: Concepts and Models**. 1 ed. Assn for Computing Machinery, 1989. 460 p.

BOEHM, B. A Spiral Model of Software Development and Enhancement. In: ACM SIGSOFT Software Engineering Notes, 4th, 1986, Nova York, USA. **Proceedings...** 1986 p. 14-24.

BOSCH, J. Software Product Lines: Organizational Alternatives. In: International Conference on Software Engineering (ICSE '01), 23rd, 2001, Washington, DC. **Proceedings...** 2001. p. 91-100.

BRUGALI, D.; SYCARA, K. Frameworks and Pattern Languages: An Intriguing Relationship. **ACM Computing Surveys**, v. 32, n.2 p. 70-86, 2000.

BUSCHMANN, F; MEUNIER, R; ROHNERT, H; SOMMERLAD, P; STAL, M. **Pattern-Oriented Software Architecture**. 1 ed. John Wiley & Sons, 1996. 476 p.

CLEMENTS, P.; NORTHROP, L. **Software Product Lines: Practices and Patterns**. 3 ed. Addison-Wesley Professional, 2002. 608 p.

CZARNECKI, C.; ANTKIEWICZ, M. Model-driven software product lines. In: Conference on Object-oriented programming, systems, languages, and applications, 20th, 2005, New York, USA. **Proceedings...** 2005. p. 126-128.

CZARNECKI, K.; ANTKIEWICZ, M. FeaturePlugin: feature modeling plug-in for Eclipse. In: OOPSLA workshop on eclipse technology eXchange (eclipse '04), 3rd, 2004, New York, USA. **Proceedings...** 2004. p. 67-72.

DEMEYER, S.; DUCASSE, S.; NIERSTRASZ, O. **Object Oriented Reengineering Patterns**. 1. ed. São Francisco: Morgan Kaufmann Publishers, 2002. 282 p.

DEURSEN, A.; KLINT, P.; VISSER, J. Domain-specific languages: an annotated bibliography. **ACM SIGPLAN Notices**, Nova York, v. 35, n. 6, 2000.

DURELLI, R; CONRADO, D; RAMOS, R; PASTOR, O; CAMARGO, V; PENTEADO, R. Identifying Features for Ground Vehicles Software Product Lines by Means of Annotated Models. In: International Workshop on Model Based Architecting and Construction of Embedded Systems, 3rd, 2010, Oslo. **Proceedings...** 2010. p 140-146

ECLIPSE. **Eclipse Modeling Project**. 2010. Disponível em: <<http://www.eclipse.org/modeling/>>. Acesso em: 16 set. 2010.

ECLIPSEIDE. **Eclipse**. 2011. Disponível em: <<http://www.eclipse.org/>>. Acesso em: 13 jul. 2010.

EMF. **Eclipse Modeling Framework**. 2010. Disponível em: <<http://www.eclipse.org/modeling/emf/>>. Acesso em: 16 set. 2010.

FERNANDES, L. C; DIAS, M; OSÓRIO, F; WOLF, D. A Driving Assistance System for Navigation in Urban Environments. In: *Ibero-American conference on Advances in artificial intelligence* (IBERAMIA'10), 12th, 2010, Berlin, Heidelberg. **Proceedings...** 2010. p.1-12.

FOWLER, M. **Domain-Specific Languages**. 1. ed. Addison-Wesley Professional, 2010. 640 p.

GAMMA, E; HELM, R; JOHNSON, R; VLISSIDES, J. **Design Patterns: Elements of Reusable Object-Oriented Software**. 1. ed. Addison-Wesley Professional, 1994. 416 p.

GME. Generic Modeling Environment. **GME**, 2010. Disponível em: <<http://www.isis.vanderbilt.edu/projects/gme>>. Acesso em: 27 Setembro 2010.

GMF. **Graphical Modeling Project**. 2011. Disponível em: <<http://www.eclipse.org/modeling/gmp/>>. Acesso em: 13 jul. 2011.

GOMAA, H. **Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures**. 1. ed. Addison Wesley, 2004. 736 p.

GRAAF, B.; LORMANS, M.; TOETENEL, H. Embedded Software Engineering: The State of the Practice. **IEEE Software**, v. 20, n. 6 p. 61-69, 2003

GRAND, M. **Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML**. 2. ed. Wiley, 2002. 544 p.

GRISS, M. L. Implementing Product-Line Features with Component Reuse. In: International Conference on Software Reuse: Advances in Software Reusability, 6th, 2000, Londres. **Proceedings...** 2000. p 137-152.

GRONBACK, R. C. **Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit**. 1. ed. Addison-Wesley Professional, 2009. 736 p.

HEYMANS, P.; TRIGAUX, J. C. Software product line: state of the art. **IEEE Transactions on Software Engineering**, v. 20, n.6, p. 476-493, 2003.

HUSQVARNA. **Global leader in outdoor power products**. 2011. Disponível em: <<http://www.husqvarna.com/int/international-microsite/home/>>. Acesso em: 23 dez. 2011.

IROBOT. Irobot, **Robots that Make a Difference**. 2011. Disponível em: <<http://www.irobot.com/>>. Acesso em: 14 jan. 2011.

JAVA. **Oracle**, 2011. Disponível em: <<http://www.oracle.com/technetwork/java/javase/downloads/index.html>>. Acesso em: 13 fev. 2011.

JBUILDER. Borland. 2008. Disponível em: <<http://www.borland.com/br/products/jbuilder/>>. Acesso em: 15 jan. 2011.

JET. Java Emitter Template. 2011. Disponível em: <<http://www.eclipse.org/modeling/m2t/?project=jet#jet>>. Acesso em: 15 Jan. 2011.

KANG, K. C. et al. **Feature Oriented Domain Analysis (FODA) feasibility study**. Software Engineering Institute, Carnegie Mellon University, 1990. 148 p. Relatório Técnico. Disponível em: < <http://www.sei.cmu.edu/reports/90tr021.pdf> >. Acesso em: 10 ago. 2010.

KIM, H.-K. Applying Product Line to the Embedded Systems. **Computational Science and Its Applications**, v.19, n. 6 p. 163-171, 2006.

KLEPPE, A.; WARMER, J.; BAST, W. **MDA Explained: The Model Driven Architecture: Practice and Promise**. 1. ed. Addison-Wesley Longman Publishing, 2003. 192 p.

LEE, J. et al. Methodology for Embedded System Development Based on Product Line. In: International Conference on Advanced Communication Technology, 7th, 2005, Dublin. **Proceedings...** 2005. p. 920 - 923.

LEGO. **Lego Mindstorms**. 1998. Disponível em: <<http://mindstorms.lego.com/en-us/Default.aspx>>. Acesso em: 12 set. 2011.

LEJOS. **LeJOS Java for LEGO Mindstorms**. 2001. Disponível em: <<http://lejos.sourceforge.net/>>. Acesso em: 16 ago. 2010.

LEYC. **A Lexical Analyzer Generator and Yet Another Compiler-Compiler**, 1990. Disponível em: <<http://dinosaur.compilertools.net/>>. Acesso em: 10 mai. 2010.

LUCREDIO, D. **Uma Abordagem Orientada a Modelos para Reutilização de Software**. 256 p. Tese (Doutorado em Ciência da Computação) Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos-SP, 2009

LUCRÉDIO, D; FORTES, R; ALMEIDA, E; MEIRA, S. Performing Domain Analysis for Model-Driven Software Reuse. In: International conference on Software Reuse: High Confidence Software Reuse in Large Systems, 10th, 2008, Berlin. **Proceedings...** 2008. p. 200-211.

MARWEDEL, P. **Embedded System Design**. 1. ed. Secaucus: Springer-Verlag New York, 2006. 560 p.

MERNIK, M.; HEERING, J.; SLOANE, A. M. When and How to Develop Domain-Specific Languages. **ACM Computing Surveys**, v.19, n. 7 p. 316-344, 2005.

MOBILE. **Robôs Moveis Autonomos**, 2011. Disponível em: <http://www.mobilerobots.com/Mobile_Robots.aspx>. Acesso em: 15 jan. 2011.

NETBEANS. **Netbeans**, 1996. Disponível em: <<http://netbeans.org/>>. Acesso em: 19 jan. 2011.

NIEMELÄ, E.; IHME, T. Product Line Software Engineering of Embedded Systems. In: *Symposium on Software reusability: putting software reuse in context (SSR '01)*, 3rd, 2001, New York, USA. **Proceedings...** 2001 p. 118-125.

OMG. **Unified Modeling Language**, 1997. Disponível em: <<http://www.uml.org/>>. Acesso em: 18 abr. 2010.

POLZER, ; KOWALEWSKI, ; BOTTERWECK, G. Applying Software Product Line Techniques in Model-Based Embedded Systems Engineering. In: ICSE Workshop on Model-Based Methodologies for Pervasive and Embedded Software, 2009, Washington, DC. **Proceedings...** 2009 p. 2-10.

PURE. **Pure:variants**. 2010. Disponível em: <http://www.pure-systems.com/pure_variants.49.0.html>. Acesso em: 30 nov. 2010.

ROSS, D. T. Structured Analysis (SA): A Language for Communicating Ideas. **IEEE Transactions on Software Engineering**, v. 3, n. 1, p. 16-34, 1977.

SANGIOVANNI-VINCENTELLI, A.; MARTIN, G. Platform-based design and software design methodology for embedded systems. **IEEE Design Test of Computers**, v. 18, n. 6, p. 23-33, 2001.

SEI. **Software Engineering Institute**. 2010. Disponível em: <<http://www.sei.cmu.edu/productlines/>>. Acesso em: 2 jul. 2010.

STOERMER, ; ROEDDIGER, M. Introducing Product Lines in Small Embedded Systems. In: International Workshop on Software Product-Family Engineering, 4th, 2002, Londres. **Proceedings...** 2002. p.101-112.

THOMAS, D. MDA: Revenge of the Modelers or UML Utopia? **IEEE Software**, v. 21, n. 3, p. 15-17, 2004.

TRIGAUX, J. C.; HEYMANS, P. Software product lines: state of the art. In: Product Line Engineering of Food Traceability Software, 4th, 2003, Londres. **Proceedings...** 2003. p.50-67.

UBAYASHI, N.; NAKAJIMA, S.; HIRAYAMA, M. Context-Dependent Product Line Practice for Constructing Reliable Embedded Systems. In: international conference on Software product lines: going beyond, 14th, 2010, Berlin. **Proceedings...** 2010. p.1-15.

VAN GURP, J.; BOSCH, J.; SVAHNBERG, M. On the notion of Variability in Software Product Lines. In: Working IEEE/IFIP Conference on Software Architecture (WICSA 2001), 2001, Washington, DC. **Proceedings...** 2001 p. 45-55.

VISUAL. **VISUAL STUDIO 2010 PRODUCTS**. 2010. Disponível em: <<http://www.microsoft.com/visualstudio/en-us/products/2010-editions/>>. Acesso em: 16 out. 2010.

VÖLTER. M. **MD* Best Practices**, 2008. Disponível em: <<http://www.voelter.de/>>. Acesso em: 14 jun. 2009.

WAGNER, F. R.; CARRO, L. Metodologias e Técnicas de Engenharia de Software para Sistemas Embarcados. In: **Escola Regional de Informática**, 2009, Manaus. **Anáís...** 2009 p. 1-45.

WEHMEISTER, M. A. **Framework Orientado a Objetos para Projeto de Hardware e Software Embarcados para Sistemas Tempo-Real**. 204 p. Tese(Doutorado em Ciência da Computação) – Programa de Pós-Graduação, Universidade Federal do Rio Grande do Sul, Rio Grande do Sul – RS, 2008.

WEISS, D. M.; LAI, C. R. **Software Product-Line Engineering: A Family-Based Software Development Process**. 1. ed. Boston: Addison-Wesley Longman Publishing, 1999. 448 p.

WOLF, W. **Computers as components: principles of embedded computing system design**. 2. ed. San Francisco: Morgan Kaufmann Publishers, 2008. 662 p.

XFEATURE. **XFeature -- Feature Modelling Tool**. 2010. Disponível em: <<http://www.pnp-software.com/XFeature/>>. Acesso em: 20 set. 2010.

XSL. **The Extensible Stylesheet Language Family**. 2011. Disponível em: <<http://www.w3.org/Style/XSL/>>. Acesso em: 14 jan. 2011.

Apêndice A

BREVE INTRODUÇÃO A MÁQUINA VIRTUAL LEJOS VERSÃO 0.8.5

A.1 Introdução

Este apêndice apresenta uma breve introdução à máquina virtual LeJOS versão 0.8.5, a qual fornece uma abstração para os hardwares do Kit da *Legó Mindstorms* (LeJOS, 2011).

LeJOS foi utilizado como a linguagem base para a criação da linha de produtos intitulada LRMLPS, apresentada no Capítulo X. A máquina virtual LeJOS é um projeto de código aberto (*open source*) hospedada em <http://lejos.sourceforge.net/index.php>. Essa máquina possui várias funções que facilitam a interação com o kit da *Legó Mindstorms* (Legó, 2010), como, por exemplo, obter informações dos sensores e fornecer atuação nos atuadores que são explicadas a seguir. Informações adicionais podem ser encontradas em <http://lejos.sourceforge.net/nxt/nxj/tutorial/index.htm>. Detalhes sobre a instalação da máquina virtual são omitidos neste Apêndice, porém, podem ser encontrados em: <http://lejos.sourceforge.net/nxt/nxj/tutorial/Preliminaries/GettingStartedWindows.htm#6>.

A.2 Projeto *Legó Mindstorms*

O kit da *Legó Mindstorms* é um produto educacional projetado pela empresa Legó¹³, com o objetivo de auxiliar desenvolvimento fácil e rápido de vários tipos de robôs. Na Figura A-1 é apresentada uma visão geral do kit da *Legó Mindstorms*

¹³ <http://www.lego.com/en-us/Default.aspx>

utilizado neste trabalho e a seguir são detalhados como a máquina virtual LeJOS interage com os hardware do kit.

Como pode ser observado na Figura A-1, o kit da Lego contém um microcontrolador (*NXT Brick*), que consiste de um processador ARM de 32 bits, 256 *kilobytes* de memória *flash* utilizada para armazenar os programas e 64 *kilobytes* de RAM (*Random Access Memory*) para armazenar dados durante a execução dos programas. Adicionalmente, o kit possui quatro sensores básicos: dois sensores de toque, um sensor de luz e um ultrassônico. O Kit também contém três atuadores os quais são geralmente utilizados para fazer com que o RM se locomova.

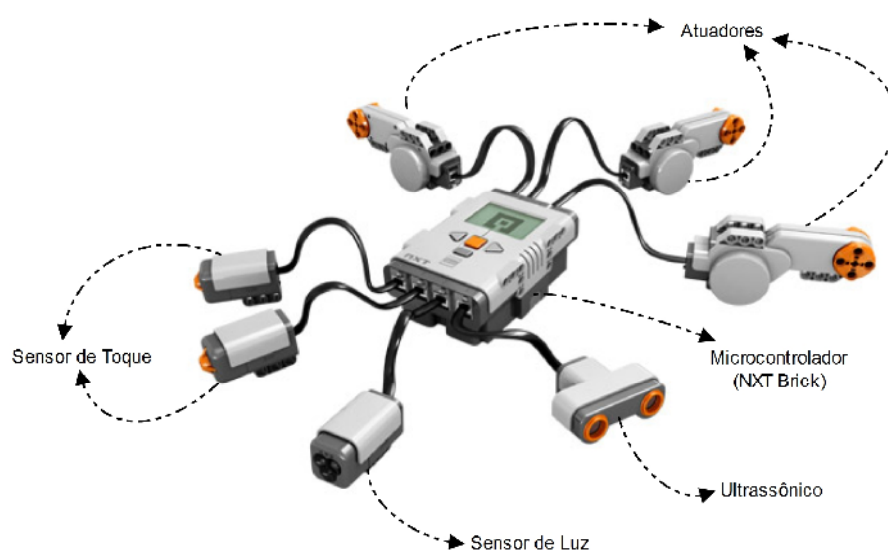


Figura A-1 - Kit *Lego Mindstorms*

A.3 Controlando os Atuadores do Kit *Lego Mindstorms*

Os três atuadores, Figura A-1, podem ser utilizados para diferentes funções, desde fazer um robô se locomover até mesmo atuar como uma empilhadeira. Para facilitar a iteração entre o programador e esses atuadores a máquina virtual LeJOS possui a classe `lejos.nxt.Motor`.

Na Figura A-2 é apresentado um exemplo simples da utilização da classe `lejos.nxt.Motor`. Para utilizar a classe “`lejos.nxt.Motor`” deve-se especificar qual atuador será utilizado. Isso é feito indicando em qual porta (A, B ou C) do micro

controlador o atuador está conectado. Como pode ser observado na linha 15 é chamado o método “*forward()*” o qual faz com que o atuador conectado na porta “A” do micro controlador comece a girar em uma determinada direção. Essa ação se repete até que um determinado botão do micro controlador seja pressionado (linha 17), o qual fará com que o atuador gire na direção oposta a que estava girando (linha 18). Na linha 20, a chamada ao método “*setSpeed(int)*” indica que a velocidade de rotação do atuador deve ser aumentada. Para que esse atuador pare, o método “*stop()*”. (linha 21) é chamado.

```
1
2 import lejos.nxt.*; // this is required for all programs that run on the NXT
3
4
5 /**
6  *Motor runs forward then backward as button is pressed.
7  * @author Durelli
8  */
9 public class BasicMotorTest
10 {
11
12     public static void main(String[] args)
13     {
14
15         Motor.A.forward();
16         LCD.drawString("FORWARD", 0, 0);
17         Button.waitForPress();
18         Motor.A.backward();
19         LCD.drawString("BACKWARD", 0, 1);
20         Motor.setSpeed(800);
21         Motor.A.stop();
22     }
23 }
```

Figura A-2 - Utilizando a Classe *Motor*

Vale ressaltar que a classe “*lejos.nxt.Motor*” deve ser utilizada quando é necessário controlar um determinado atuador de forma independente. Porém, geralmente os RM possuem pelo menos dois atuadores, os quais devem ser sincronizados para fornecer uma melhor locomoção.

LeJOS contém várias classes com métodos associados para controlar os motores dos RMs de forma síncrona. A Figura A-3 ilustra a instanciação da classe “*lejos.robotics.navigation.SimpleNavigator*”, que é uma das classes fornecidas pela LeJOS para controlar os motores dos RMs.

```

1 package com.br.ufscar.lejos.test;
2
3 import lejos.nxt.Motor;
4 import lejos.robotics.navigation.Pilot;
5 import lejos.robotics.navigation.SimpleNavigator;
6 import lejos.robotics.navigation.TachoPilot;
7
8 public class GoForward {
9
10
11     public static void main(String[] args) {
12         Pilot pilot = new TachoPilot(5.6f, 16.0f, Motor.A, Motor.B);
13         SimpleNavigator simple = new SimpleNavigator(pilot);
14
15         simple.forward();
16     }
17
18 }

```

Figura A-3 - Utilizando a Classe *SimpleNavigator* para controlar um determinado RM

Nas linhas 3 a 6 são apresentados as importações necessárias para que o código seja executado normalmente. Como pode ser observado uma das facilidades de se utilizar a máquina virtual LeJOS é que a mesma é tem muita semelhança à linguagem de programação Java. Dessa forma, a curva de aprendizagem é muito pequena, uma vez que Java é uma linguagem amplamente utilizada hoje em dia. As linhas 11 a 16 representam o método *main(String[] args)*, responsável por iniciar a execução do programa. Na linha 12 foi criada uma instancia da classe “*lejos.robotics.navigation.TachoPilot*”, que é responsável por fornecer informações do dispositivo de locomoção do RM. Tais informações são passadas por parâmetro pelo construtor da classe *TachoPilot*. O primeiro parâmetro (nesse caso 5.6f) representa o diâmetro dos pneus que o RM possui, o segundo parâmetro (16.0f) representa a largura do chassi do RM, por fim, o terceiro e o quarto parâmetros representam a porta na qual os motores do RM estão conectados. Na linha 13 é criada uma instancia da classe *SimpleNavigator*, a qual possui vários métodos para auxiliar na locomoção dos RMs. Por fim, na linha 15 é invocado o método *forward()*, o qual faz com que o RM se locomova para frente. Na Tabela A-1 são apresentados os principais métodos da classe *SimpleNavigator*.

Tabela A-1 – Principais Métodos da Classe *SimpleNavigator*

Métodos	Descrição
<i>void setSpeed(int Speed)</i>	Aumenta ou diminui a velocidade dos motores do RMs
<i>void forward()</i>	Faz com que o RM comece a se locomover para frente

<code>void backward()</code>	Faz com que o RM comece a se locomover para trás
<code>void stop()</code>	Faz com que o RM pare de se locomover
<code>void travel(float distance)</code>	Faz com que o RM se locomova uma determinada distância
<code>int getTravelDistance()</code>	Retorna a distância percorrida do RM
<code>void rotate(int angle)</code>	Faz com que o RM vire. Caso o ângulo (<i>angle</i>) seja positivo, o RM vira para a esquerda, caso contrário vira para a direita
<code>boolean isMoving()</code>	Retonar <i>true</i> quando o RM esta se locomovendo, <i>false</i> caso contrário

A.4 Controlando os Sensores do Kit Lego Mindstorms

O kit da *Lego Mindstorms* possui quatro sensores básicos, dois sensores de toque (*Touch*), um sensor ultrassónico (*Ultrasonic*) e um sensor de luz (*Light*). Por sua vez, a máquina virtual LeJOS possui abstrações para todos esses sensores, entre outros sensores de terceiros, tais como sensor de compasso, câmera e infravermelho, etc. A seguir são apresentados exemplos da utilização de cada um dos sensores do kit da *Lego Mindstorms*. Adicionalmente são apresentados alguns dos principais métodos para a utilização desses sensores.

A.4.1 Sensor de Toque

Sensor de toque possui a funcionalidade de identificar quando ele foi pressionado ou quando solto. Na Figura A-4 é apresentada uma visão do sensor de toque.



Figura A-4 – Sensor de Toque

Para usar o sensor de toque simplesmente deve-se criar uma instancia da classe “*lejos.nxt.TouchSensor*”. Na Figura A-5 um exemplo da instanciação da classe *TouchSensor* é apresentado.

```
1 package com.br.ufscar.lejos.test;
2
3 import lejos.nxt.SensorPort;
4 import lejos.nxt.TouchSensor;
5
6 public class GoForward {
7
8
9     public static void main(String[] args) {
10
11         TouchSensor touch = new TouchSensor(SensorPort.S1);
12         ...
13     }
14
15 }
```

Figura A-5 - Instanciando o Sensor de Toque

Para identificar se o sensor de toque foi pressionado deve-se utilizar o método *isPressed()* da classe *TouchSensor*. Esse método retorna *true* quando o sensor estiver pressionado ou *false* caso contrario. Na Figura A-6 é apresentado um exemplo da utilização do método *isPressed()*. Na linha 14 é verificado se o sensor é pressionado, caso retorne *true*, será apresentado “*Pressionado*”, caso contrário apresentará “*Não Pressionado*”.

```
1 package com.br.ufscar.lejos.test;
2
3 import lejos.nxt.LCD;
4 import lejos.nxt.SensorPort;
5 import lejos.nxt.TouchSensor;
6
7 public class GoForward {
8
9
10    public static void main(String[] args) {
11
12        TouchSensor touch = new TouchSensor(SensorPort.S1);
13
14        if(touch.isPressed()){
15
16            LCD.drawString("Pressionado", 0, 0);
17
18        }else
19        {
20            LCD.drawString("Não Presionado", 0, 0);
21        }
22    }
23 }
```

Figura A-6 - Método *isPressed()*

A.4.2 Sensor Ultrassônico

O sensor ultrassônico permite que ao robô calcular a distância e “enxergar” onde os objetos estão. Esse sensor mede a distancia calculando o tempo que leva para uma onda de som bater em um determinado objeto e retornar até o sensor. Na Figura A-7 é apresentada uma visão do sensor ultrassônico do kit da *Lego Mindstorms*.



Figura A-7 - Sensor Ultrassônico

Para utilizar o sensor ultrassônico é necessário criar um instancia da classe “*lejos.nxt.UltrasonicSensor*”. Essa classe possui o método *getDistance()* o qual retorna a distancia do robô até um determinado obstáculo. Na Figura A-8 é apresentado um exemplo da utilização do sensor ultrassônico.

```

1 package com.br.ufscar.lejos.test;
2 import lejos.nxt.Button;
3 import lejos.nxt.LCD;
4 import lejos.nxt.SensorPort;
5 import lejos.nxt.UltrasonicSensor;
6
7 public class GoForward {
8
9
10     public static void main(String[] args) {
11
12         UltrasonicSensor ultrasonic = new UltrasonicSensor(SensorPort.S1);
13
14         while (!Button.ESCAPE.isPressed()) {
15
16             LCD.drawInt(ultrasonic.getDistance(), 0, 0);
17             try{
18                 Thread.sleep(500);
19             }catch (InterruptedException e) {
20
21             }
22         }
23     }
24 }

```

Figura A-8 - Exemplo da utilização do Sensor Ultrassônico

Nas linhas 2 à 5 são feitas as importações necessárias para execução do código. Nas linhas 10 à 23 é apresentado o método *main(String[] args)*. Na linha 12 é instanciado a classe *UltrasonicSensor*, que representa uma abstração do sensor ultrassônico do kit da *Lego Mindstorms*. Na linha 16 é chamado o método

`getDistance()` que retorna um `int` no qual representa a distância de um determinado objeto.

A.4.3 Sensor de Luz

Esse sensor é utilizado para reconhecer a intensidade de luz. O sensor possui um *LED (Light-Emitting Diode)* que emite uma luz no qual é possível fazer distinção da intensidade da luz refletida. Usualmente esse sensor é utilizado para fazer com que o robô possa seguir linhas. Na Figura A-9 é apresentada uma figura que ilustrado o sensor de luz.



Figura A-9 - Sensor de Luz

Para utilizar o sensor de luz deve-se criar uma instancia da classe “`lejos.nxt.LightSensor`”. Essa classe possui vários métodos que auxiliam na utilização do sensor. Na Figura A-10 é apresenta um trecho de código no qual é apresentado como é feita a instanciação e utilização do sensor de luz.

```
1 package com.br.ufscar.lejos.test;
2 import lejos.nxt.LCD;
3 import lejos.nxt.LightSensor;
4 import lejos.nxt.SensorPort;
5
6 public class GoForward {
7
8
9     public static void main(String[] args) {
10
11         LightSensor lightSensor = new LightSensor(SensorPort.S1);
12
13         lightSensor.calibrateHigh();
14
15         LCD.drawInt(lightSensor.readValue(), 0, 0);
16
17     }
18 }
```

Figura A-10 - Instanciação da classe `LightSensor`

Novamente as linhas 2 à 3 descreve as importações necessárias para a execução do código. As linhas 9 à 17 ilustra o método `main(String[] args)`, método

responsável por começar a execução do código. Na linha 11 é feita a instanciação da classe *LightSensor*, na linha 13 o método *calibreteHight()* é responsável por calibrar o sensor, esse procedimento deve ser realizado sempre antes de coletar informações do sensor de luz. Na linha 15 é chamado o método *readValue()* que retorna um inteiro de 0 até 100 no qual representa a intensidade de luz de um determinado ambiente, sendo 0 muito claro e 100 muito escuro.

Maiores informações sobre a API do LeJOS podem ser encontradas em: <http://lejos.sourceforge.net/nxt/nxj/api/index.html>.

Apêndice B

UMA SUCINTA INTRODUÇÃO AOS FRAMEWORKS EMF E GMF

B.1 Introdução

A seguir é apresentada uma introdução sucinta aos *frameworks* EMF e GMF. Essa introdução tem o propósito de facilitar a compreensão de como foi desenvolvida a DSL apresentada no Capítulo 5. A DSL criada neste trabalho tem como objetivo auxiliar na atividade de Engenharia de Aplicação no qual por meio de configurações de diagrama de *features* de um determinado membro da LRMLPS (Capítulo 5) são realizadas transformações M2M e M2C.

O EMF (*Eclipse Modeling Framework*) permite a modelagem do domínio utilizando o metamodelo Ecore. A partir desse metamodelo é possível criar transformações M2M e M2C. O framework GMF (*Graphical Modeling Framework*) por sua vez permite representar o metamodelo criado pelo EMF graficamente. Maiores informações podem ser adquiridas em: <http://www.eclipse.org/modeling/>.

Como pré-requisito para a realização desse Apêndice é necessário possuir o IDE Eclipse, o mesmo pode ser obtido em: <http://www.eclipse.org/download>.

B.2 Utilizando os *Frameworks* EMF e GMF

Na Figura B-1 é apresentada uma visão geral do diagrama que será criado nesse Apêndice.

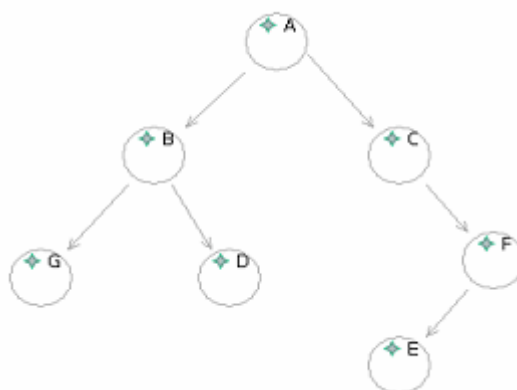


Figura B-1 Visão Geral do Diagrama Criado

Primeiramente deve-se criar um projeto GMF, a Figura B-2 descreve os passos necessários para a criação do projeto.

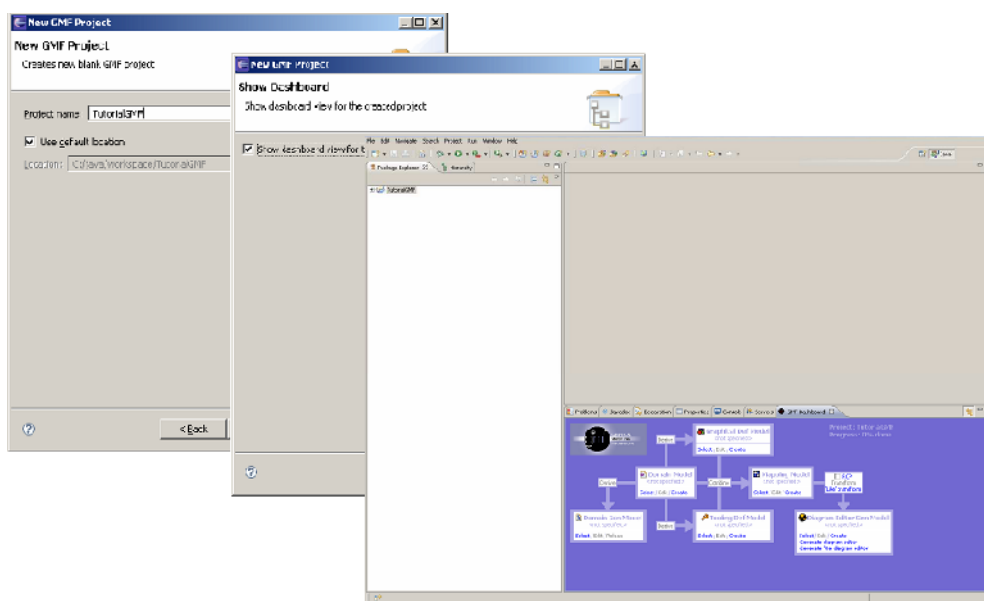


Figura B-11 – Passos para a criação do Projeto GMF

Após criar o projeto GMF deve-se criar um arquivo “.ecore”, esse arquivo representa o metamodelo do diagrama a ser criado. O *framework* GMF disponibiliza um ambiente amigável para criar o metamodelo. Na Figura B-3 é apresentada uma visão geral do metamodelo desenvolvido nesse Apêndice.

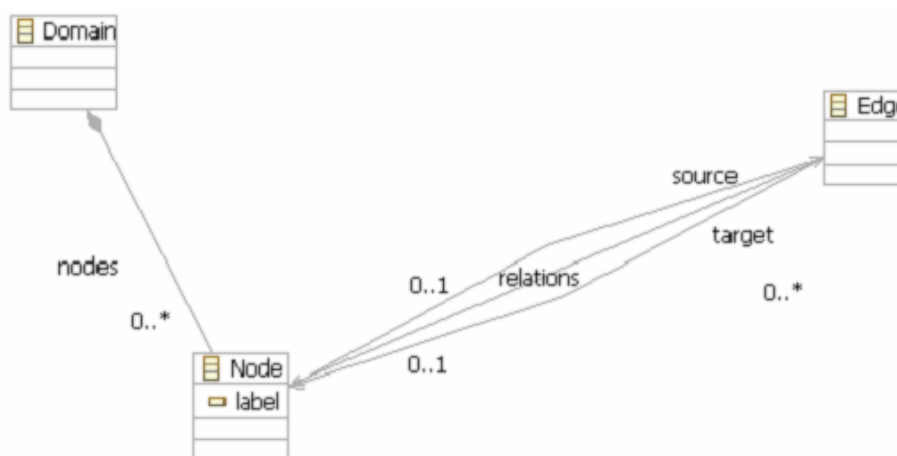


Figura B-12 - Metamodelo do Diagrama Criado

A metaclass *Domain* corresponde ao domínio do projeto, a metaclass *Node* corresponde aos elementos do diagrama por fim a metaclass *Edge* representa uma abstração das ligações entre os elementos do diagrama. Adicionalmente como pode ser observado na Figura B-3 cada nó (*Node*) pode conter uma ligação (*edge*) para outro elemento.

Após criar o *metamodelo* do diagrama é necessário criar as classes que representam o diagrama gerado. Esse processo é feito automaticamente pela IDE Eclipse, no qual é apresentado na Figura B-4.

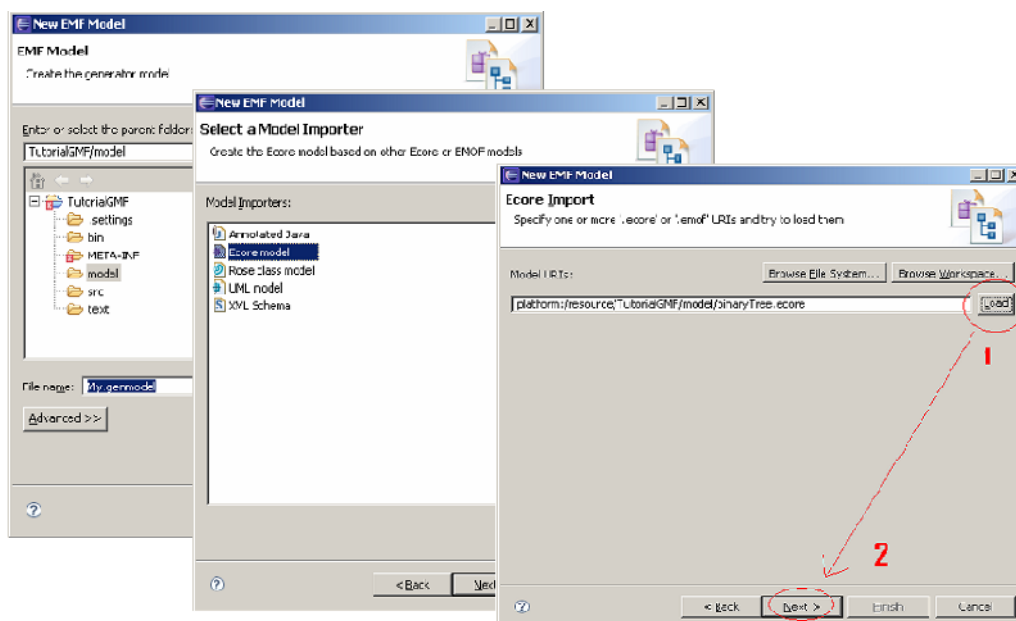


Figura B-13 – Criando as Classes do Diagrama

O arquivo com a extensão *“.genmodel”* é responsável por criar classes Java. Tais classes representam abstrações do metamodelo apresentado na Figura B-3. Para criar essas classes simplesmente deve-se clicar com o botão direito e em

seguida clicar em “*Generate Model Code*” e “*Generate Edit Code*”. Na Figura B-5 é apresentado esse procedimento.

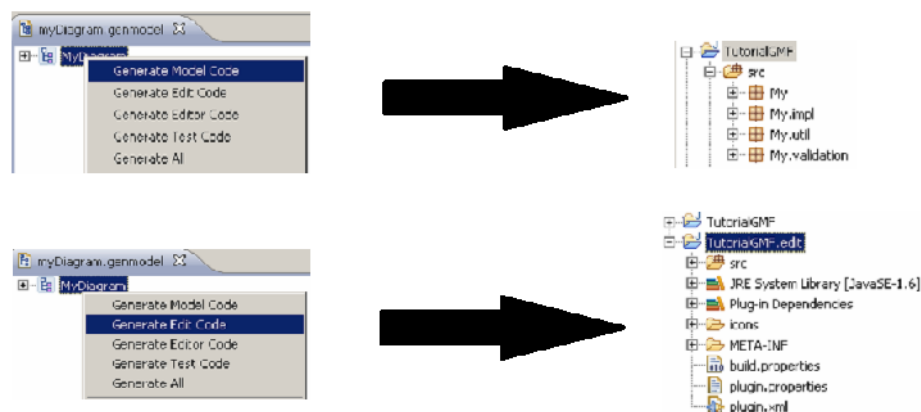


Figura B-14 - Geração do Códigos Java

Após gerar os códigos deve-se informar como os elementos do metamodelo (Figura B-3) serão apresentados visualmente. Para realizar esse processo deve-se criar o arquivo com a extensão “*.gmfgraph*”. A Figura B-6 apresenta esse processo.

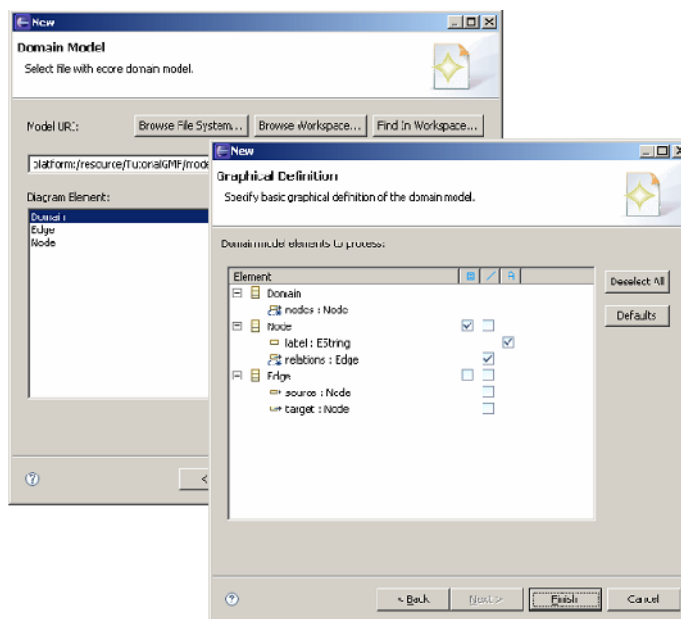


Figura B-15 - Criando Componentes Gráficos

Após clicar em *Finish* é criado o arquivo “*.gmfgraph*”, é nesse arquivo que deve-se especifica como serão representados os metamodelos graficamente. A Figura B-7 apresenta os passos que foram feitos para criar os componentes visuais do metamodelo.

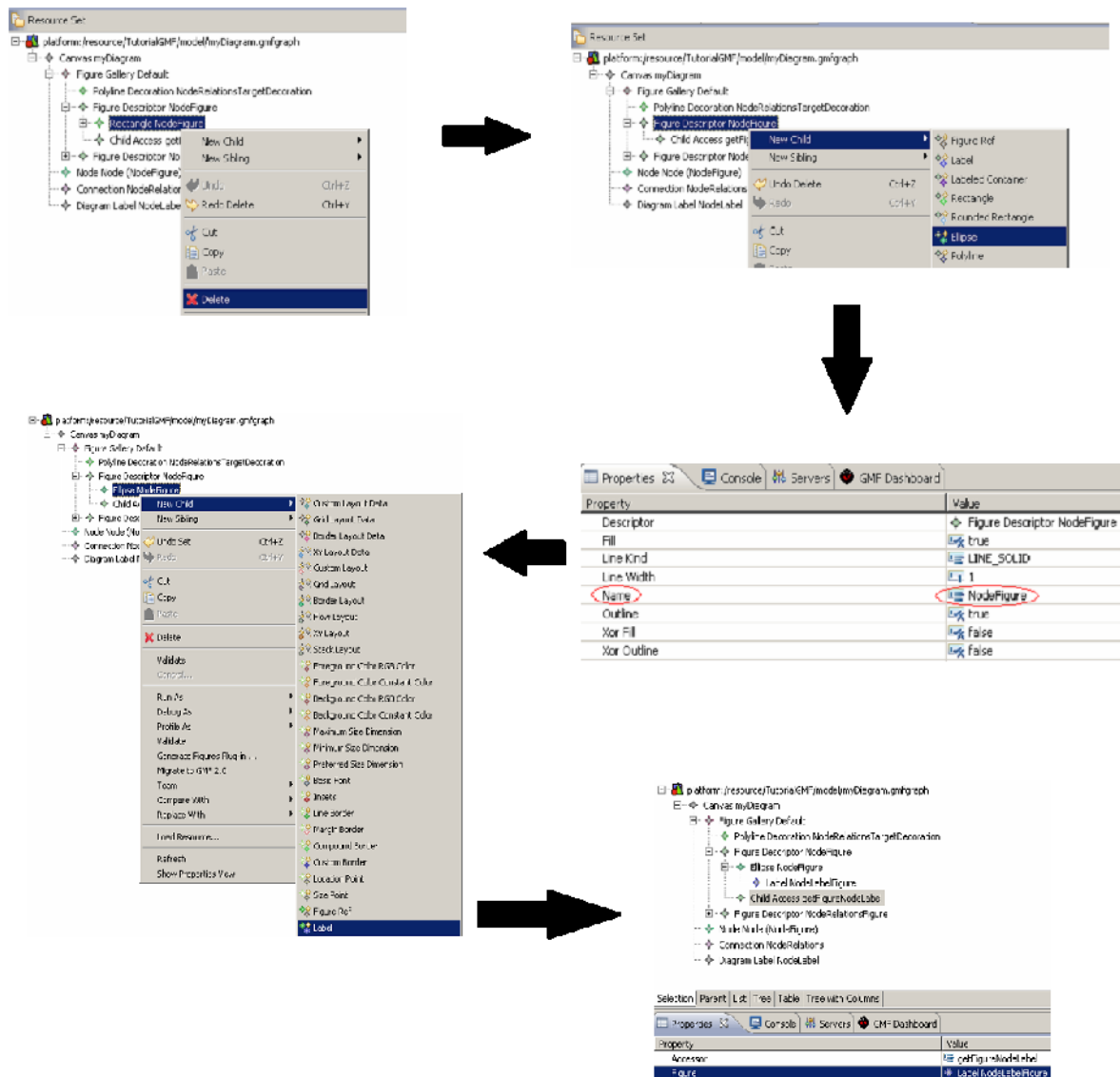


Figura B-16 - Passos para Criar os Componentes Visuais

Após finalizar a criação dos componentes visuais do metamodelo, deve-se criar as paletas da aplicação. Tais paletas são criadas com o arquivo que possui a extensão “.gmftool”. Na Figura B-8 é apresentado esse arquivo, o mesmo é criado automaticamente pelo IDE Eclipse.

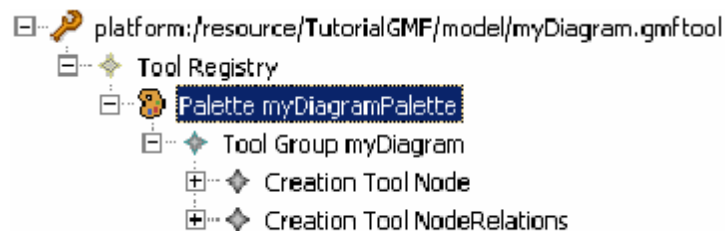


Figura B-17 - Configuração do arquivo .gmftool

Após a criação dos arquivos “.genmodel”, “.gmfgraph” e “.gmftool”, deve-se criar um arquivo “.gmfmap”, esse arquivo representa o mapeamento e fornece uma referencia ao metamodelo, o arquivo “.gmfgraph” e “.gmftool”. Na Figura abaixo são apresentadas uma sequência de *wizards* que devem ser preenchidos para criar o arquivo “.gmfmap”.

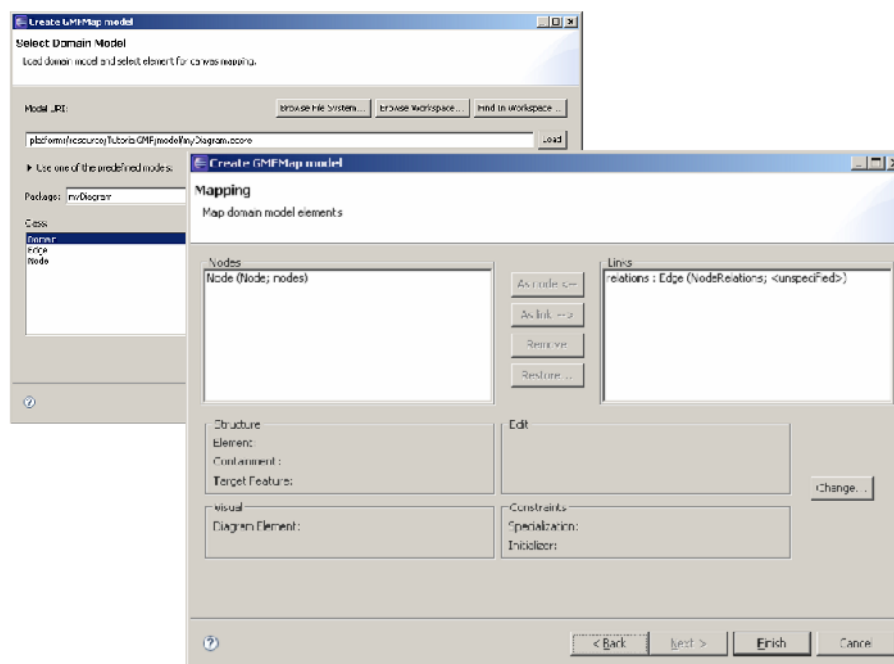


Figura B-18 - Criando o Arquivo .gmfmap

Após a criação do arquivo “.gmfmap” é necessário fazer o mapeamento dos arquivos criados. Na Figura B-10 é apresentado o arquivo “.gmfmap” criado pela IDE Eclipse.

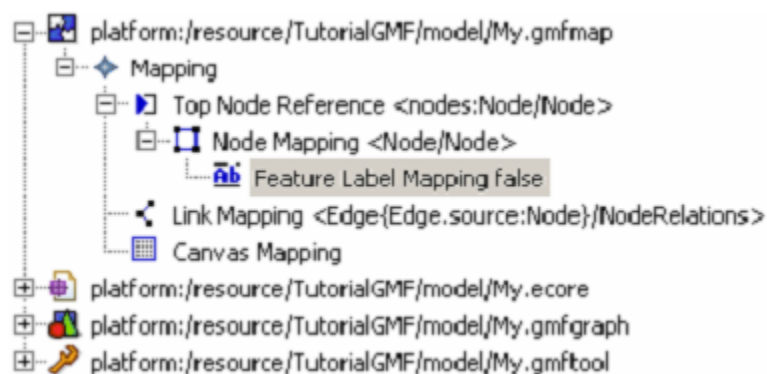


Figura B-19 - Arquivo .gmfmap

Por meio das propriedades de cada nó do arquivo “.gmfmap” é possível conferir se o mapeamento está correto. O nó “*Feature Label Mapping false*” corresponde ao mapeamento do atributo *label* da metaclassa *Node*. A Figura B-11 apresenta o mapeamento que foi realizado na metaclassa *Node*.

Property	Value
Diagram Label	Diagram Label NodeLabel
Edit Method	MESSAGE_FORMAT
Editor Pattern	
Edit Pattern	
Features	label : EString
Read Only	False
View Method	MESSAGE_FORMAT
View Pattern	

Figura B-20 - Mapeamento a metaclassa *Node*

A propriedade “*Diagram Label*” faz uma referencia ao nó “*Diagram Label NodeLabel*” apresentado no arquivo “.*gmfgraph*”, (ver Figura B-7). O nó *Link Mapping* (Figura B-10) é utilizado para representar e fazer o mapeamento dos relacionamentos (*edge*) do diagrama. A Figura B-12 apresenta como foi feita esse mapeamento.

Property	Value
Domain meta information	
Containment Feature	relations : Edge
Element	Edge
Source Feature	source : Node
Target Feature	target : Node
Misc	
Related Diagrams	
Visual representation	
Appearance Style	
Context Menu	
Diagram Link	Connection NodeRelations
Tool	Creation Tool NodeRelations

Figura B-21 - Mapeamento dos Relacionamentos do Diagrama

Após finalizar a configuração do arquivo “.*gmfmap*”, o próximo passo é criar o “.*gmfgen*”, arquivo responsável por realmente criar o diagrama. Para tal, deve-se clicar com o botão direito no nó “*Mapping*” do arquivo “.*gmfmap*”. Dessa forma, será criado o arquivo “.*gmfgen*”, consecutivamente deve-se clicar com o botão direito nesse arquivo na opção “*Generate diagram code*”. Esse processo é ilustrado na Figura B-13.

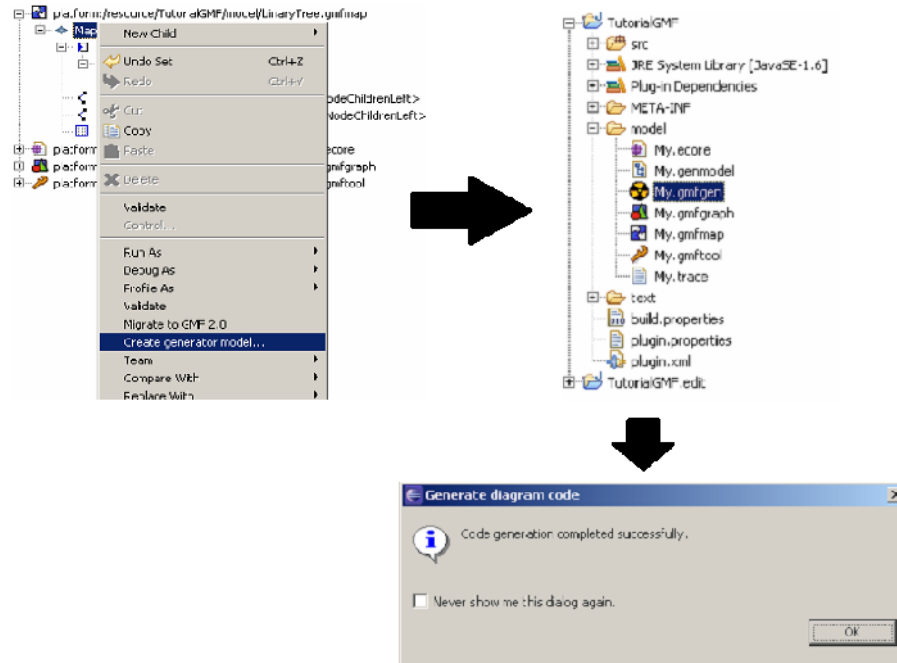


Figura B-22 - Passos para a Criação do Diagrama

Uma vez que o diagrama estiver concluído pode-se utilizar o mesmo. Para tal, é necessário criar um projeto dentro do IDE Eclipse, e posteriormente criar o arquivo que representa o diagrama que foi desenvolvido. Na Figura B-14 é apresentado como esse processo deve ser realizado.

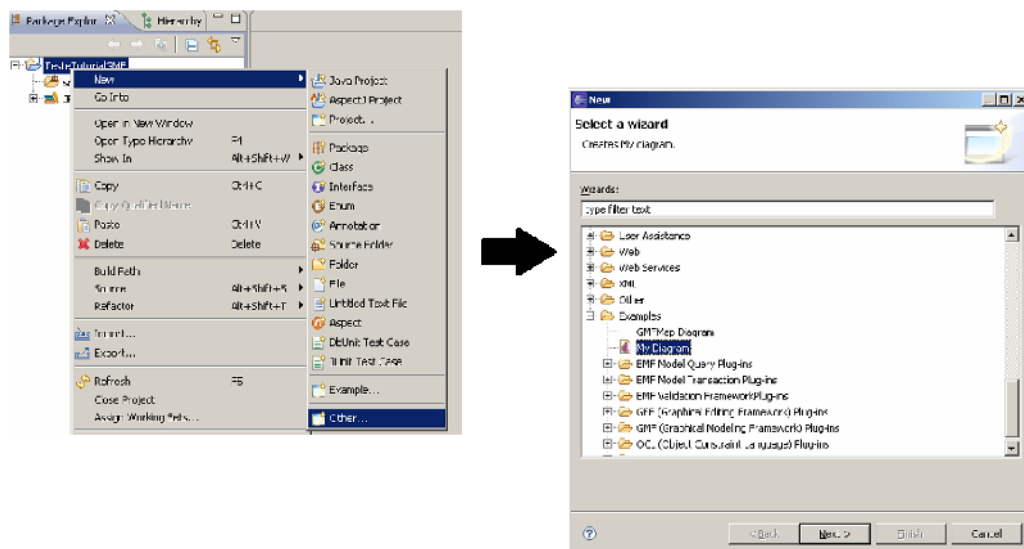


Figura B-23 - Passos para Criar o Diagrama Desenvolvido

Por fim a Figura B-15 apresenta uma visão geral do diagrama desenvolvido nesse Apêndice.

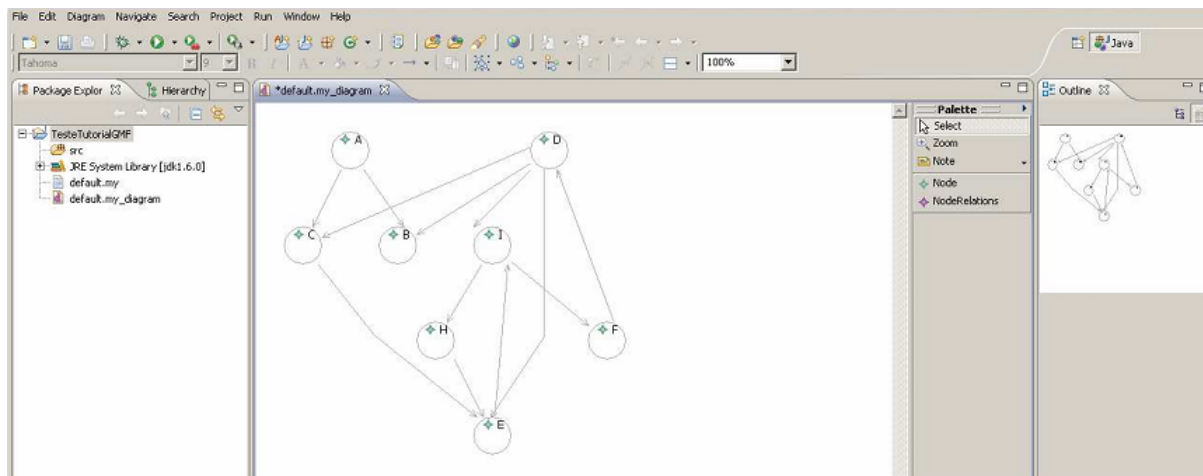


Figura B-24 - Diagrama Desenvolvido