

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

**DESENVOLVIMENTO E REÚSO DE FRAMEWORKS
COM BASE NAS CARACTERÍSTICAS DO DOMÍNIO**

MATHEUS CARVALHO VIANA

ORIENTADORA: PROFA. DRA. ROSÂNGELA APARECIDA DELLOSSO PENTEADO

Co-ORIENTADOR: PROF. DR. ANTÔNIO FRANCISCO DO PRADO

São Carlos - SP
Fevereiro/2014

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

**DESENVOLVIMENTO E REÚSO DE FRAMEWORKS
COM BASE NAS CARACTERÍSTICAS DO DOMÍNIO**

MATHEUS CARVALHO VIANA

Tese 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 Doutor em Ciência da Computação, área de concentração: Engenharia de Software

Orientadora: Dr. Rosângela A. D. Penteado

São Carlos - SP
Fevereiro/2014

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

V614dr

Viana, Matheus Carvalho.

Desenvolvimento e reúso de frameworks com base nas características do domínio / Matheus Carvalho Viana. -- São Carlos : UFSCar, 2014.

212 f.

Tese (Doutorado) -- Universidade Federal de São Carlos, 2014.

1. Engenharia de software. 2. Reuso. 3. Framework (Programa de computador). 4. Característica de domínio. 5. Geração de código. 6. Linguagem específica de domínio. I. Título.

CDD: 005.1 (20^a)

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

**“Desenvolvimento e Reúso de Frameworks com
Base nas Características do Domínio”**

Matheus Carvalho Viana

Tese de Doutorado 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 Doutor em Ciência da Computação.

Membros da Banca:



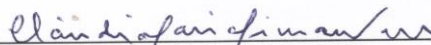
Prof. Dra. Rosângela Apaecida Delosso Penteadó
(Orientadora- DC/UFSCar)



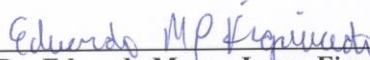
Prof. Dr. Antonio Francisco do Prado
(DC/UFSCar)



Prof. Dr. Daniel Lucrédio
(DC/UFSCar)



Prof. Dra. Claudia Maria Lima Werner
(UFRJ)



Prof. Dr. Eduardo Magno Lages Figueiredo
(UFMG)



Prof. Dr. Paulo Cesar Masiero
(ICMC/USP)

São Carlos
Maio/2014

Dedicada aos meus pais, Mariza e Arnulpho, e aos meus irmãos, Mário Augusto e Ana Paula..

AGRADECIMENTO

Primeiramente, agradeço a Deus pela oportunidade de vir a São Carlos e fazer os cursos de mestrado e de doutorado e muitas outras coisas que me fizeram feliz.

À minha mãe, Mariza, ao meu pai, Arnulpho, ao meu irmão, Mário Augusto e à minha irmã, Ana Paula, por sempre acreditarem no meu potencial e por oferecerem o seu apoio para continuar estudando.

À Professora Dra. Rosângela Penteado por ter me escolhido como seu orientando. Obrigado pela amizade, pela confiança, pela ajuda e por me auxiliar a evoluir como aprendiz e pesquisador.

Ao Professor Dr. Antônio Francisco do Prado por ter acreditado em mim, contribuindo no meu trabalho e sempre me incentivar.

A todos os meus amigos por sofrerem e se divertirem junto comigo. Espero ter sido um bom amigo tanto quanto vocês foram para mim. Tive muita sorte em ter vindo para São Carlos na mesma época que vocês.

Aos meus amigos do Kung Fu Taisan. Além de ganhar novos amigos, pude relaxar a minha mente e aprender a estender os meus limites com concentração e dedicação.

À CAPES pelo apoio financeiro ao meu trabalho.

Eu sempre desejei que o computador fosse tão fácil de usar quanto o telefone.

Agora meu desejo se realizou. Não sei mais usar o telefone.

Bjarne Stroustrup, criador da linguagem C++

RESUMO

Frameworks são artefatos de software que implementam a funcionalidade básica de um domínio. Seu reuso pode aumentar a eficiência do processo de desenvolvimento e a qualidade do código de aplicações. Contudo, frameworks são difíceis de construir e reutilizar, pois necessitam de uma estrutura complexa para implementar as variabilidades do seu domínio e serem adaptáveis o suficiente para poderem ser reutilizados por diversas aplicações. Em vista dessas dificuldades este projeto apresenta duas abordagens: 1) a abordagem *From Features to Frameworks* (F3), na qual o desenvolvedor modela as características de um domínio e uma linguagem de padrões auxilia na implementação de um framework com base nesse modelo; e 2) a abordagem que utiliza uma *Domain-Specific Language* (DSL) construída a partir da identificação e análise das características do domínio do framework para facilitar o reuso desse framework. Uma ferramenta, denominada *From Features to Frameworks Tool* (F3T), também foi desenvolvida para apoiar o uso dessas duas abordagens, fornecendo editores para a modelagem dos domínios e das aplicações e automatizando a implementação de código dos frameworks, das DSLs e das aplicações. Além de facilitar o desenvolvimento e o reuso de framework, experimentos realizados ao longo deste projeto mostraram que essas duas abordagens tornam esses processos mais eficientes e permitem a construção de frameworks e aplicações com menor dificuldade.

Palavras-chave: reuso, framework, padrão, modelo de características, geração de código, *Domain-Specific Languages* (DSLs), abordagem *From Features to Frameworks* (F3), ferramenta F3T.

ABSTRACT

Frameworks are software artifacts that implement the basic functionality of a domain. Its reuse can improve the efficiency of development process and the quality of application code. However, frameworks are difficult to develop and reuse, since they require a complex structure to implement domain variability and be adaptable enough to be reused by different applications. Due to these difficulties, this research presents two approaches: 1) the From Features to Frameworks (F3) approach, in which the developer models the features of a domain and a pattern language helps in implementing a framework based on this model; and 2) the approach that uses a Domain-Specific Language (DSL) built from the identification and analysis of the domain features of a framework to facilitate the reuse of this framework. A tool, called From Features to Frameworks Tool (F3T), was also developed to support the use of these two approaches, providing editors for modeling domains and applications and automating the implementation of code of frameworks, DSLs and applications. In addition to facilitate the development and reuse of frameworks, experiments conducted during this project showed that these two approaches make these processes more efficient and allow the construction of frameworks and applications with less difficulty.

Keywords: reuse, framework, pattern, feature model, code generation, Domain-Specific Languages (DSLs), From Features to Frameworks (F3) approach, F3T.

LISTA DE FIGURAS

Figura 2.1. Exemplo de utilização do padrão <i>Strategy</i>	33
Figura 2.2. Esquema do padrão MVC, adaptado de FREEMAN et al. (2004).	34
Figura 2.3. Grafo do fluxo dos padrões da GRN (BRAGA et al, 1999).	36
Figura 2.4. Modelos de classes do quarto padrão da GRN, Localizar o Recurso.	37
Figura 2.5. Modelo de classe da aplicação para uma locadora de DVDs.	38
Figura 2.6. Arquitetura do Framework GRENJ.	41
Figura 2.7. O reúso do framework GRENJ ocorre com a extensão das classes das camadas de negócio e de interface gráfica.	43
Figura 2.8. Utilização do wizard do framework GRENJ.	43
Figura 2.9. Parte do código da classe Venda com reúso do framework Hibernate.	44
Figura 2.10. Metamodelo do domínio de veículos autônomos do Lego Mindstorms®.	47
Figura 2.11. Exemplo de arquivo XML que armazena dados de vendas de produtos.	49
Figura 2.12. Um (a) template JET e (b) um arquivo de texto gerado a partir dele.	50
Figura 2.13. Modelo de processo de uma LPS. Adaptado de Czarnecki (2005).	51
Figura 2.14. Modelo de características do domínio de veículos autônomos.	51
Figura 2.15. Interface da ferramenta Pure::variants (Pure, 2013).	53
Figura 3.1. Etapas da abordagem F3.	56
Figura 3.2. Modelos F3 são uma extensão dos modelos de características.	57
Figura 3.3. Modelo F3 do domínio aplicações que gerenciam dispositivos de carros.	58
Figura 3.4. Fluxo de execução dos padrões da LPF3.	60
Figura 3.5. Modelos a) do cenário do P01 e b) do projeto da solução.	61
Figura 3.6. Modelos a) do cenário do P02 e b) do projeto da solução.	62
Figura 3.7. Modelos a) do cenário do P03 e b) do projeto da solução.	63
Figura 3.8. Modelos a) do cenário do P04 e b) do projeto da solução.	64
Figura 3.9. Modelos a) do cenário do P05 e b) do projeto da solução.	64
Figura 3.10. Modelo a) do cenário do P06 e b) do projeto da solução.	65
Figura 3.11. Modelos a) do cenário do P07 e b) do projeto da solução.	67
Figura 3.12. Modelos a) do cenário do P08 e b) do projeto da solução.	68
Figura 3.13. Modelos a) do cenário do P09 e b) do projeto da solução.	69
Figura 3.14. Modelos a) do cenário do P10 e b) do projeto da solução.	70
Figura 3.15. Modelos a) do cenário do P11 e b) do projeto da solução.	71

Figura 3.16. Modelos a) do cenário do P12 e b) do projeto da solução.....	72
Figura 3.17. Modelos a) do cenário do padrão e b) do projeto da solução.....	72
Figura 3.18. Modelo do projeto da solução do P14.....	73
Figura 3.19. Modelos a) do cenário do P15 e b) do projeto da solução.....	75
Figura 3.20. Modelo do projeto da solução do P16.....	76
Figura 3.21. Modelos a) do cenário do P17 e b) do projeto da solução.....	78
Figura 3.22. Modelo F3 do domínio de transações de aluguel e de comercialização de recursos.....	81
Figura 3.23. Modelo de classes do framework.....	82
Figura 3.24. Trecho de código da classe ResourceTransaction.....	83
Figura 3.25. Trecho de código da classe Resource.....	84
Figura 3.26. Trecho de código da classe ResourceReservation.....	84
Figura 3.27. Modelo de classes DAO do domínio de transações de aluguel e comercialização.....	85
Figura 3.28. Trecho de código da classe ResourceTransactionDAO.....	86
Figura 3.29. Modelo de classes da camada de modelo da aplicação de biblioteca.....	88
Figura 3.30. Código da classe ObraLiteraria.....	88
Figura 3.31. Modelo de classes da camada de persistência da aplicação de biblioteca.....	89
Figura 3.32. Código da classe ObraLiterariaDAO.....	90
Figura 3.33. Modelo de sequência para carregar um objeto da classe ObraLiteraria a partir do banco de dados.....	91
Figura 4.1. Fases da abordagem para construção e uso de DSLs de frameworks.....	95
Figura 4.2. Etapas da fase de Engenharia da DSL.....	96
Figura 4.3. Modelo de classes do framework H.....	97
Figura 4.4. Metamodelo da DSL do framework H.....	99
Figura 4.5. Definição da notação gráfica da DSL do framework H.....	99
Figura 4.6. Definição dos itens do menu da DSL do framework H.....	100
Figura 4.7. Modelo Gmfmap da DSL do framework H.....	100
Figura 4.8. Exemplo de template principal para o framework mostrado na Figura 4.3.....	101
Figura 4.9. Trecho do template da característica D do framework H.....	102
Figura 4.10. Etapas da fase de Engenharia da Aplicação.....	102
Figura 4.11. Exemplo de utilização da DSL do framework H.....	103
Figura 4.12. Mecanismo de validação das DSLs construídas com o GMF.....	104
Figura 4.13. Código gerado para a classe Dapp.....	104
Figura 4.14. Classes do framework GRENJ que implementam o Padrão 01 da GRN.....	106

Figura 4.15. Operações da classe Resource que devem ser sobrescritas.	107
Figura 4.16. Trecho do código da classe Movie.	108
Figura 4.17. Sequência de criação do metamodelo da DSL do framework GRENJ.	109
Figura 4.18. Conteúdo dos modelos (a) Gmftool e (b) Gmfmap do framework GRENJ.	110
Figura 4.19. Parte do template para subclasses de ResourceRental.	111
Figura 4.20. Modelo da aplicação de Aluguel de Carros.	112
Figura 4.21. Parte do código gerado para a classe CarRental.	113
Figura 4.22. Metamodelo da DSL do framework Hibernate.	114
Figura 4.23. Modelo Gmfgraph da DSL do framework Hibernate.	115
Figura 4.24. Modelos (a) Gmftool e (b) Gmfmap da DSL do framework Hibernate.	115
Figura 4.25. Parte do template que gera as classes de entidade.	116
Figura 4.26. Modelo da aplicação de Aluguel de Carros criado com a DSL do framework Hibernate.	117
Figura 4.27. Parte do código da classe Car com o reuso do framework Hibernate.	118
Figura 5.1. Módulos da F3T.	121
Figura 5.2. Metamodelo do editor pra modelo F3.	122
Figura 5.3. Ligação dos elementos das sintaxes abstrata e concreta dos modelos F3.	123
Figura 5.4. Editor gráfico da F3T para modelos F3.	123
Figura 5.5. Mecanismo de validação dos modelos F3 na F3T.	125
Figura 5.6. Organização dos templates do Módulo de Frameworks da F3T.	126
Figura 5.7. Código (a) de template para as operações definidas pelos padrões Decomposição Opcional e Decomposição Obrigatória da LPF3 e (b) de exemplos de operações geradas a partir desse template.	127
Figura 5.8. Organização dos templates do Módulo de Aplicações.	128
Figura 5.9. Código de template que gera as operações que identificam as classes da aplicação e exemplo de código gerado por esse template.	128
Figura 5.10. Criação de um modelo F3 na F3T.	129
Figura 5.11. Modelo F3 do domínio de clínicas médicas.	130
Figura 5.12. Função de geração do código-fonte e da DSL na F3T.	131
Figura 5.13. Geração do restante do conteúdo dos plug-ins da DSL do framework.	131
Figura 5.14. Projetos da DSL e do código-fonte do framework gerados pela F3T.	132
Figura 5.15. Código da operação calcTotal (a) antes e (b) depois da alteração.	133
Figura 5.16. Criação de um modelo com a DSL do framework de Clínicas Médicas.	133
Figura 5.17. Modelo da aplicação para uma Clinica Veterinária.	134
Figura 5.18. Painel de Propriedades da F3T com a característica AtParticular.	135

Figura 5.19. Geração da aplicação para uma Clínica Veterinária.....	135
Figura 6.1. Representação gráfica de uma distribuição normal de dados.	140
Figura 6.2. Exemplos de gráficos de probabilidade.....	140
Figura 6.3. Boxplot criado a partir dos dados do experimento 1.....	147
Figura 6.4. Gráficos resultantes do teste de normalidade sobre os dados do tempo gasto no desenvolvimento das aplicações.	148
Figura 6.5. Gráficos resultantes do teste de normalidade sobre os dados do número de problemas no código-fonte das aplicações.....	149
Figura 6.6. Boxplot criado a partir dos dados do experimento 2.....	157
Figura 6.7. Gráficos resultantes do teste de normalidade sobre os dados do tempo gasto no desenvolvimento dos frameworks.	158
Figura 6.8. Gráficos resultantes do teste de normalidade sobre os dados do número de problemas nos frameworks desenvolvidos pelos participantes.....	159
Figura 6.9. Boxplot criado a partir dos dados do experimento 3.....	166
Figura 6.10. Gráficos resultantes do teste de normalidade sobre o tempo gasto com as ferramentas Pure::Variants e F3T.	167
Figura A.1. Passos para a criação de um Projeto GMF.	190
Figura A.2. GMF Dashboard.....	190
Figura A.3. Metamodelo do diagrama.	191
Figura A.4. Passos para a criação do arquivo com extensão genmodel.	191
Figura A.5. Metamodelo do diagrama.	192
Figura A.6. Criação do modelo Gmfgraph.....	192
Figura A.7. Passos para criar a notação gráfica do diagrama.	193
Figura A.8. Modelo Gmftool.	194
Figura A.9. Passos para a criação modelo Gmfmap.	194
Figura A.10. Modelo Gmfmap.	195
Figura A.11. Geração do arquivo Gmfgen.....	195
Figura A.12. Geração do plug-in do diagrama.....	196
Figura A.13. Passos para a criação de um modelo.....	196
Figura A.14. Visão geral da ferramenta CASE do diagrama.	197
Figura B.1. Interface da ferramenta Pure::variants.....	198
Figura B.2. Modelo de classes do domínio de carros.....	199
Figura B.3. Interface da pure::variants com o projeto Java da aplicação de carros.....	200
Figura B.4. Criação de um projeto de LPS (Variant Project) com a pure::variants.	200
Figura B.5. Identificação dos componentes da aplicação para o projeto da LPS.	201

Figura B.6. Modelo da arquitetura da LPS.	202
Figura B.7. Criação do arquivo do modelo de características da LPS.	202
Figura B.8. Modelo de características da LPS de carros.	202
Figura B.9. Passos para criação do espaço de configuração da LPS.	203
Figura B.10. Passos para criação do modelo de uma aplicação.	204
Figura B.11. Exemplo de modelo de uma aplicação de carros.	204
Figura B.12. Configurando o modelo de arquitetura para a geração da aplicação modelada.	205
Figura B.13. Gerando o código da aplicação.	206
Figura B.14. Arquivos da aplicação.	206

LISTA DE QUADROS

Quadro 2.1. Requisitos de uma aplicação para uma locadora de DVDs.	38
Quadro 2.2. Relação entre algumas classes da GRN e do framework GRENJ.....	42
Quadro 3.1. Padrões da linguagem de padrões F3.....	60
Quadro 4.1. Lista das características identificadas no framework H.	97
Quadro 4.2. Lista detalhada das características do framework H.....	98
Quadro 4.3. Algumas das características do framework GRENJ.	106
Quadro 4.4. Informações sobre sete características do framework GRENJ.....	108
Quadro 4.5. Requisitos da aplicação de Aluguel de Carros.	111
Quadro 4.6. Características de persistência do framework Hibernate.....	113
Quadro 4.7. Características de persistência do framework Hibernate após análise.	114
Quadro 5.1. Propriedades das características.	124
Quadro 5.2. Propriedades dos atributos.....	124
Quadro 5.3. Propriedades das operações.....	124
Quadro 5.4. Propriedades dos parâmetros das operações.	124
Quadro 5.5. Propriedades das decomposições.....	125
Quadro 6.1. Definição do Experimento 1.	141
Quadro 6.2. Atividades do Experimento 1.....	144
Quadro 6.3. Dados coletados do experimento 1.	146
Quadro 6.4. Definição do Experimento 2.	151
Quadro 6.5. Atividades do Experimento 2.....	153
Quadro 6.6. Dados coletados do experimento 2.	155
Quadro 6.7. Definição do Experimento 3.	161
Quadro 6.8. Atividades do Experimento 3.....	163
Quadro 6.9. Dados coletados do experimento 3.	165
Quadro 6.10. Resultados dos experimentos.	170

LISTA DE ABREVIATURAS E SIGLAS

DSL – Domain-Specific Language

EAF – Enterprise Application Frameworks

EMF – Eclipse Modeling Framework

F3 – From Features to Frameworks

F3T – From Features to Frameworks Tool

GMF – Graphical Modeling Framework

GRN – Gestão de Recursos de Negócios

GRNJ: Gestão de Recursos de Negócios com implementação em Java

IDE – Integrated Development Environment

JET – Java Emitter Templates

JSP – Java Server Pages

LP – Linguagem de Padrões

LPS – Linha de Produtos de Software

M2M – Model-to-Model

M2T – Model-to-Text

MDE – Model-Driven Engineering

MIF – Middleware Integration Frameworks

MVC – Model-View-Controller

OMG – Object Management Group

SIF – System Infrastructure Frameworks

UML – Unified Modeling Language

XML – eXtensible Markup Language

XSL – eXtensible Stylesheet Language

W3C – World Wide Web Consortium

CONVENÇÕES ADOTADAS

Algumas convenções de fontes foram adotadas na escrita desta tese de doutorado para facilitar a leitura e o entendimento dos textos. Essas fontes são:

Arial: padronização adotada para o texto da tese. Nomes em inglês que representam conceitos chave nesta tese de doutorado (framework, template e wizard) seguem essa padronização;

Arial Itálico: adotada em palavras em inglês que não são nomes de conceitos chave utilizados nesta tese;

`Courier New`: adotada em nomes de classes, atributos, operações e outras unidades de código-fonte;

Lucida Sans: adotada em nomes de características de domínios;

Times Itálico: adotada em nomes e extensões de arquivos.

SUMÁRIO

CAPÍTULO 1 - INTRODUÇÃO	21
1.1 Contexto	21
1.2 Motivação	22
1.3 Objetivos.....	25
1.4 Trabalhos Relacionados	26
1.5 Metodologia de Desenvolvimento do Trabalho	28
1.6 Organização da Tese.....	29
CAPÍTULO 2 - FUNDAMENTAÇÃO TEÓRICA.....	31
2.1 Considerações Iniciais	31
2.2 Padrões de Software	32
2.2.1 A Linguagem de Padrões Gestão de Recursos de Negócios.....	35
2.3 Frameworks	38
2.3.1 Framework GRENJ.....	41
2.3.2 Framework Hibernate.....	44
2.4 Engenharia Dirigida por Modelos	45
2.4.1 Graphical Modeling Framework	48
2.4.2 Java Emitter Templates	48
2.5 Linhas de Produto de Software	50
2.5.1 Pure::variants.....	52
2.6 Considerações Finais	53
CAPÍTULO 3 - UMA ABORDAGEM PARA FACILITAR O DESENVOLVIMENTO DE FRAMEWORKS	55
3.1 Considerações Iniciais	55
3.2 A Abordagem From Features to Frameworks (F3).....	56
3.2.1 Modelagem do Domínio	56
3.2.2 Construção do Framework.....	59
3.3 Utilização da Abordagem F3.....	79
3.3.1 Modelagem do Domínio de Transações de Aluguel e de Comercialização de Recursos	79
3.3.2 Construção do Framework para Transações de Aluguel e de Comercialização de Recursos	82

3.3.3 Instanciação do Framework para uma Aplicação de Biblioteca.....	87
3.4 Considerações Finais	91
CAPÍTULO 4 - UMA ABORDAGEM PARA FACILITAR A INSTANCIAÇÃO DE FRAMEWORKS	93
4.1 Considerações Iniciais	93
4.2 Abordagem para a Construção e Uso de DSLs de Frameworks	95
4.2.1 Engenharia da DSL.....	96
4.2.1.1 Identificação das Características.....	96
4.2.1.2 Análise das Características.....	97
4.2.1.3 Projeto da DSL.....	98
4.2.1.4 Construção da DSL.....	101
4.2.1.5 Construção do Gerador.....	101
4.2.2 Engenharia da Aplicação	102
4.2.2.1 Modelagem da Aplicação	103
4.2.2.2 Construção da Aplicação	104
4.3 Exemplos de Utilização da Abordagem de DSLs de Frameworks	104
4.3.1 DSL do Framework GRENJ	105
4.3.1.1 Engenharia da DSL do framework GRENJ: Identificação das Características.....	105
4.3.1.2 Engenharia da DSL do framework GRENJ: Análise das Características.....	106
4.3.1.3 Engenharia da DSL do framework GRENJ: Modelagem da DSL	109
4.3.1.4 Engenharia da DSL do framework GRENJ: Construção da DSL.....	110
4.3.1.5 Engenharia da DSL do framework GRENJ: Construção do Gerador.....	111
4.3.1.6 Engenharia da Aplicação com o Framework GRENJ: Modelagem da Aplicação de Aluguel de Carros	111
4.3.1.7 Engenharia da Aplicação com o Framework GRENJ: Construção da Aplicação de Aluguel de Carros	112
4.3.2 DSL do Framework Hibernate.....	113
4.3.2.1 Engenharia da DSL do Framework Hibernate: Identificação das Características ...	113
4.3.2.2 Engenharia da DSL do Framework Hibernate: Análise das Características	113
4.3.2.3 Engenharia da DSL do Framework Hibernate: Projeto da DSL	114
4.3.2.4 Engenharia da DSL do Framework Hibernate: Construção da DSL	116
4.3.2.5 Engenharia da DSL do framework Hibernate: Construção do Gerador	116
4.3.2.6 Engenharia da Aplicação com o framework Hibernate: Modelagem da Aplicação de Aluguel de Carros	117
4.3.2.7 Engenharia da Aplicação com o framework Hibernate: Construção da Aplicação de Aluguel de Carros	118

4.4 Considerações Finais	119
CAPÍTULO 5 - A FERRAMENTA F3T.....	120
5.1 Considerações Iniciais	120
5.2 Construção da F3T	121
5.2.1 Módulo de Domínios	121
5.2.2 Módulo de Frameworks.....	126
5.2.3 Módulo de Aplicações	128
5.3 Exemplo de Utilização da F3T	129
5.3.1 Engenharia do Domínio: Framework de Clínicas Médicas	129
5.3.2 Engenharia da Aplicação: Clínica Veterinária	133
5.4 Considerações Finais	136
CAPÍTULO 6 - EXPERIMENTOS	138
6.1 Considerações Iniciais	138
6.2 Testes Estatísticos.....	139
6.2.1 Shapiro-Wilk	139
6.2.2 Paried T-Test	140
6.2.3 Wilcoxon Signed Rank.....	141
6.3 Experimento 1: Reúso de Frameworks por Meio de DSLs	141
6.3.1 Definição e Planejamento do Experimento 1.....	141
6.3.1.1 Contexto e Participantes	141
6.3.1.2 Formulação das Hipóteses.....	142
6.3.1.3 Variáveis	142
6.3.1.4 Modelo	143
6.3.1.5 Instrumentação	144
6.3.2 Operação do Experimento 1	144
6.3.2.1 Preparação	144
6.3.2.2 Execução	144
6.3.3 Análise dos Dados do Experimento 1	145
6.3.3.1 Análise Descritiva dos Dados.....	145
6.3.3.2 Teste das Hipóteses	147
6.3.4 Ameaças à Validade do Experimento 1	149
6.4 Experimento 2: Abordagem F3	150
6.4.1 Definição e Planejamento do Experimento 2.....	151
6.4.1.1 Contexto e Participantes	151
6.4.1.2 Formulação das Hipóteses.....	151

6.4.1.3 Variáveis	152
6.4.1.4 Modelo	152
6.4.1.5 Instrumentação	153
6.4.2 Operação do Experimento 2	153
6.4.2.1 Preparação	154
6.4.2.2 Execução	154
6.4.3 Análise dos Dados do Experimento 2	154
6.4.3.1 Análise Descritiva dos Dados.....	155
6.4.3.2 Teste das Hipóteses	157
6.4.4 Ameaças à Validade do Experimento 2	159
6.5 Experimento 3: Ferramenta F3T	160
6.5.1 Definição e Planejamento do Experimento 3.....	160
6.5.1.1 Contexto e Participantes	161
6.5.1.2 Formulação das Hipóteses.....	161
6.5.1.3 Variáveis	162
6.5.1.4 Modelo	162
6.5.1.5 Instrumentação	163
6.5.2 Operação do Experimento 3	163
6.5.2.1 Preparação	163
6.5.2.2 Execução	164
6.5.3 Análise dos Dados do Experimento 3	164
6.5.3.1 Análise Descritiva dos Dados.....	164
6.5.3.2 Teste das Hipóteses	166
6.5.4 Ameaças à Validade do Experimento 3	168
6.6 Considerações Finais	170
CAPÍTULO 7 - CONCLUSÕES	172
7.1 Considerações Finais da Tese de Doutorado	172
7.2 Contribuições.....	172
7.3 Limitações.....	174
7.4 Trabalhos Futuros.....	175
7.5 Artigos Publicados	176
REFERÊNCIAS	178
APÊNDICE A - INTRODUÇÃO AO GMF	189
A.1 Considerações Iniciais.....	189
A.2 Utilizando o GMF.....	189

A.3	Considerações Finais	197
APÊNDICE B - INTRODUÇÃO À PURE::VARIANTS.....		198
B.1	Introdução	198
B.2	Aplicação para o Domínio de Carros	199
B.3	Engenharia de Domínio	200
B.3.1	Criando o projeto da LPS	200
B.3.2	Criando o modelo da arquitetura da LPS.....	201
B.3.3	Criando o modelo de características do domínio da LPS	202
B.3.4	Criando o espaço de configuração dos produtos da LPS	203
B.4	Engenharia de Aplicação.....	203
B.4.1	Criando o modelo de uma aplicação	203
APÊNDICE C - FORMULÁRIOS DOS EXPERIMENTOS		207
Formulário de Caracterização de Participante dos Experimentos 1 e 2		207
Formulário de Coleta de Dados do Experimento 1.....		208
Formulário de Coleta de Dados do Experimento 2.....		209
Formulário de Caracterização de Participante do Experimento 3.....		210
Formulário de Coleta de Dados do Experimento 3.....		211

Capítulo 1

INTRODUÇÃO

1.1 Contexto

O reúso tornou-se uma prática comum no desenvolvimento de software para possibilitar a entrega de aplicações complexas de forma eficiente em termos de tempo e custo (BAUER, 2013; SHIVA e SHALA, 2007). Artefatos de desenvolvimento, como padrões de software, frameworks e geradores representam uma grande oportunidade para o aumento da produtividade e da qualidade do código-fonte das aplicações (MAHMOOD et al., 2013; ANGUSWAMY e FRAKES, 2012).

Padrões de software possibilitam ao desenvolvedor reusar experiência e procedimentos que foram adotados em situações enfrentadas anteriormente por outros desenvolvedores. Os padrões são flexíveis, independentes de tecnologia e podem ser aplicados nos mais variados tipos de aplicações (ZAMANI e BUTLER, 2009; FOWLER, 1997).

Frameworks implementam a funcionalidade básica de um domínio e podem ser instanciados em aplicações (STURM e KRAMER, 2013; FAYAD e JOHNSON, 1999). Nesse caso ocorre o reúso de código e de projeto (ABI-ANTOUN, 2007). Existem frameworks que auxiliam na implementação de requisitos não funcionais, como os para persistência de dados, e há outros que apoiam o desenvolvimento de aplicações pertencentes a domínios específicos, como comércio e indústria.

Geradores realizam transformações de modelo para modelo ou de modelo para código, automatizando parte do processo de desenvolvimento de software e permitindo que os desenvolvedores se dediquem mais ao domínio do problema do que aos detalhes de tecnologia (HUTCHINSON et al., 2011; FRANCE e RUMPE, 2007). Com geradores, o reúso pode ocorrer em diferentes níveis, a depender do tipo de artefato que é gerado (CZARNECKI e HELSEN, 2006).

O reúso de artefatos de desenvolvimento não é uma tarefa trivial e depende da habilidade dos desenvolvedores, uma vez que é necessário selecionar os artefatos apropriados para o software em desenvolvimento e implementar o código-fonte de forma correta para que o reúso possa ser feito corretamente. Os principais problemas que podem ocorrer são as dificuldades no entendimento e na instanciação dos artefatos que podem ser reutilizados. Por exemplo, padrões de software são artefatos reutilizáveis com alto nível de abstração e independentes de tecnologia. Conseqüentemente, seu reúso é diretamente dependente da habilidade do desenvolvedor para identificar o problema e adaptar a solução proposta por eles (HARRISON et al., 2007). Já artefatos de código, como os frameworks, possuem complexidade ainda maior, pois, além de ser necessário entender o seu funcionamento, é preciso conhecer toda a sua estrutura e interface para que possam ser reutilizados de forma correta.

Além do reúso de artefatos pré-existentes, novos artefatos específicos de domínio podem ser construídos para serem reutilizados no desenvolvimento de diversas aplicações de um domínio específico. Contudo, o desenvolvimento de software reutilizável é mais complexo do que o desenvolvimento de uma aplicação, pois esse software deve ser flexível, configurável e possuir interfaces bem definidas que lhe permita ser adaptado a diferentes aplicações (OKANOVIC e MATELJAN, 2011; ROBILLARD, 2009; KIRK et al., 2007).

Técnicas de modelagem e geração de código podem ser utilizadas para auxiliar o desenvolvimento e o reúso de artefatos de código, como frameworks e componentes (ANTKIEWICZ et al., 2009). Diante desse contexto, esta tese de doutorado propõe abordagens que: 1) sejam realizadas de forma sistematizada; 2) separem a lógica do domínio dos problemas de implementação; 3) guiem os desenvolvedores na implementação das unidades de código com base nas características do domínio; 4) permitam a automatização da maior parte do processo de desenvolvimento dos artefatos de código; 5) aumentem a eficiência dos processos de desenvolvimento e a qualidade das aplicações.

1.2 Motivação

Após a realização de estudos envolvendo padrões de software, frameworks e geradores, foram observadas as vantagens proporcionadas pelo reúso desses artefatos no desenvolvimento de aplicações (VIANA, 2009; DURELLI et al., 2008; DURELLI, 2008). A implementação do framework Gestão de REcursos de Negócios em Java (GRENJ) (DURELLI, 2008) foi realizada. O seu domínio atende à linguagem de padrões Gestão de Recursos de Negócios (GRN) (BRAGA et al., 1999) que contempla transações de aluguel,

venda, compra e manutenção de bens e de serviços. Assim, o GRENJ pode instanciar aplicações utilizando os padrões GRN na linguagem de implementação Java.

O processo de instanciação do GRENJ inicia-se com a modelagem de uma aplicação em nível de análise utilizando os padrões da GRN para identificar quais de suas classes serão reutilizadas. Em seguida, o código da aplicação é implementado estendendo as classes identificadas e sobrescrevendo os métodos abstratos dessas classes.

Em estudos de caso realizados (VIANA, 2009) verificou-se que o uso do framework GRENJ proporcionou ganho de eficiência no desenvolvimento das aplicações. Entretanto, percebeu-se, também, que esse ganho é proporcional ao grau de conhecimento que o desenvolvedor possui sobre a arquitetura do framework. A existência de inúmeros pontos variáveis que precisam ser configurados durante a instanciação do framework GRENJ em uma aplicação pode levar o desenvolvedor a cometer erros e inserir defeitos no código específico das aplicações. A correção desses defeitos, para que a aplicação reflita as especificações requeridas pelo cliente, consome esforço considerável e aumenta o tempo de desenvolvimento dessas aplicações.

A inserção de defeitos na parte do código das aplicações referente à instanciação do framework GRENJ foi fortemente amenizada com a construção de um wizard gerador de aplicações. Esse wizard fornece formulários nos quais as informações relativas a uma aplicação, por exemplo, os nomes e os atributos das transações, são preenchidas com base nos padrões da GRN. A partir dos dados preenchidos nesses formulários, o wizard executa a geração do código da aplicação, retirando do desenvolvedor a responsabilidade de configurar os pontos variáveis do framework. Como resultado, além da redução de defeitos no código-fonte das aplicações, o tempo gasto na implementação do código referente ao reúso do framework GRENJ foi reduzido para, aproximadamente, 15% do que era gasto¹ sem o apoio do wizard (VIANA, 2009).

Com o uso do wizard do framework GRENJ para o desenvolvimento de aplicações em estudos de caso realizados com a participação de estudantes de graduação, pôde-se observar que o wizard não elimina a necessidade de modelagem da aplicação. Desse modo, o desenvolvedor é obrigado a definir os elementos de uma aplicação duas vezes: no modelo e nos formulários do wizard. Além disso, os wizards não são intuitivos e não permitem uma visão geral de todos os elementos da aplicação.

Com o intuito de resolver esses problemas, Linguagens Específicas de Domínio (*Domain-Specific Language – DSL*) (RUMPE et al., 2010; GRONBACK, 2009) foram consideradas como uma possibilidade para realizar a instanciação de frameworks. Assim, a geração do código-fonte das aplicações passa a ser realizada diretamente a partir dos

¹ Considerando um desenvolvedor experiente com o uso do framework GRENJ e do *wizard*.

modelos das aplicações criados com a DSL. Além disso, as DSLs permitem uma visualização gráfica de todos os elementos das aplicações, o que as tornam mais intuitivas para desenvolvedores acostumados com modelo gráficos, como os da UML. Esses motivos foram considerados para a criação de uma abordagem para a construção de DSLs que facilitam o reúso de frameworks.

Na literatura podem ser encontrados muitos trabalhos que descrevem o desenvolvimento de um framework específico (SACERDOTIANU et al., 2011; CHAVES et al., 2008; AMATRIAIN, 2007; KIM et al., 2005), porém, há uma carência de trabalhos que propõem abordagens de desenvolvimento de frameworks de forma detalhada. Em geral, as abordagens de desenvolvimento de frameworks encontradas sugerem adaptações das etapas de análise, projeto, implementação e teste para o escopo de frameworks ou se baseiam no estudo de um conjunto de aplicações pertencentes ao domínio desejado (STANOJEVIĆ et al., 2011; AMATRIAIN e ARUMI, 2011; XU e BUTLER, 2006; MARKIEWICZ e LUCENA, 2001; FAYAD e JOHNSON, 1999). Apesar da divisão em etapas dessas abordagens proporcionarem maior organização ao processo de desenvolvimento dos frameworks, o trabalho efetivo de construção do framework é totalmente dependente do conhecimento e da habilidade dos desenvolvedores. Não foram encontradas abordagens que guiam os desenvolvedores com relação às estruturas (classes, interfaces de comunicação com as aplicações, etc.) necessárias para o funcionamento dos frameworks.

Além de meios para facilitar o reúso de frameworks, buscou-se também estudar formas para reduzir as dificuldades do desenvolvimento desse tipo de artefato. Foram feitos estudos no sentido de analisar a estrutura interna de alguns frameworks, como GRENJ (VIANA, 2009; DURELLI, 2008), Hibernate (JBOSS COMMUNITY, 2013) e JHotDraw (JHOTDRAW.ORG, 2013). Alguns trabalhos sugerem o uso de padrões de software para auxiliar a construção de frameworks, de modo que as variabilidades do domínio possam ser tratadas (SRINIVASAN, 1999; FAYAD e JOHNSON, 1999; GAMMA et al., 1995). O framework JHotDraw, por exemplo, foi originalmente construído com o intuito de mostrar como padrões de software podem ser utilizados para facilitar a construção de código adaptável (JHOTDRAW.ORG, 2013).

Para que as dificuldades do desenvolvimento de um framework pudessem ser reduzidas, imaginou-se a criação de uma abordagem que separasse a definição do domínio do framework das complexidades de sua implementação. Para essa abordagem, um tipo de modelo de características (JÉZÉQUEL, 2012; KARATAS et al., 2010; KANG et al., 1990) foi definido para a modelagem do domínio e um conjunto de padrões foi criado com base em alguns padrões de projeto e nas análises realizadas sobre ao código dos frameworks anteriormente citados para guiar a construção do código do framework.

Finalmente, percebeu-se que técnicas de Engenharia Dirigida por Modelos (*Model-Driven Engineering* - MDE) também poderiam ser utilizadas para facilitar a construção de frameworks, minimizar o esforço dos desenvolvedores e reduzir o número de erros gerados manualmente, assim como o que ocorreu com o uso de DSLs no reuso de frameworks. Portanto, foi construída uma ferramenta para automatizar as etapas de implementação das abordagens de construção do código-fonte e das DSLs dos frameworks e do código-fonte das aplicações que reutilizam esses frameworks. Assim, essas abordagens poderiam ser utilizadas de forma eficiente.

1.3 Objetivos

Nesta tese de doutorado os objetivos foram criar:

- 1) a abordagem para **Construção de DSLs de Frameworks** - que se baseia na: identificação das características do domínio de um framework para a criação das sintaxes abstrata e concreta e do gerador de aplicações da DSL;
- 2) a abordagem **From Features to Frameworks (F3)** - na qual padrões auxiliam a construção das unidades de código do framework a partir de um modelo criado com as características de um domínio;
- 3) a ferramenta **From Features to Frameworks Tool (F3T)**, que apoia o uso das duas abordagens anteriores, provendo um ambiente para a modelagem de domínios, a geração do código-fonte e de DSLs de frameworks para esses domínios, a modelagem e a geração de aplicações que reutilizam esses frameworks.

Por meio dessas abordagens e ferramenta, esta tese de doutorado espera proporcionar os seguintes benefícios em relação aos estudos realizados e abordagens existentes:

- O desenvolvimento de frameworks e de aplicações é realizado de forma sistematizada, ou seja, por meio de regras bem definidas que guiam os desenvolvedores;
- Maior eficiência e facilidade no desenvolvimento de frameworks e de aplicações;
- Maior qualidade para os frameworks e as aplicações desenvolvidos;
- Maior separação entre a definição das características do domínio e a implementação do código-fonte do frameworks;
- Maior facilidade na instanciação de frameworks por meio de DSLs que contêm elementos que representam as características dos domínios dos frameworks;
- Automatização da implementação do código-fonte dos frameworks, das DSLs e das aplicações.

1.4 Trabalhos Relacionados

Nesta seção são apresentados alguns trabalhos relacionados com o tema de pesquisa proposto nesta tese.

Ye e Liu (2005) propuseram uma abordagem para a modelagem de características de domínios, considerando duas visões: 1) em árvore, que mostra a hierarquia das características; e 2) das dependências, que define as restrições existentes entre as características. De acordo com os seus autores, o enfoque principal dessa abordagem consiste nas dependências entre as características, uma vez que possuem um impacto considerável na configuração dos softwares pertencentes ao domínio modelado. A modelagem de domínios da abordagem F3 atua de forma diferente, em que o domínio é definido em um único modelo de características, permitindo a geração do código e a DSL framework por meio da F3T.

Almeida et al. (2007) desenvolveram uma abordagem sistemática que apóia o projeto de arquiteturas de software específicas de domínio. Essa abordagem consiste de sete atividades: 1) Decompor os Módulos, em que a arquitetura do domínio é dividida em módulos; 2) Refinar os Módulos, em que os módulos são estruturados com o uso de padrões arquiteturais selecionados de acordo com os requisitos funcionais e com os critérios de qualidade desses módulos; 3) Representar as Variabilidades, por meio de diretrizes que determinam quando e como padrões de projeto podem ser aplicados para que essas variabilidades possam existir na arquitetura do domínio; 4) Agrupar os Componentes, em que são definidos conjuntos de componentes que possuem dependências entre si para compor os módulos da arquitetura do domínio; 5) Identificar os Componentes, em que os componentes identificados de acordo com as especificações do domínio; 6) Especificar os Componentes, em que são criadas as interfaces e especificações dos componentes; e 7) Representar a Arquitetura do Domínio, em que modelos de componentes e outras visões arquiteturais são utilizadas para mostrar os componentes, suas interconexões e suas interfaces. A abordagem F3 usou como base a proposta por Almeida et al. (2007), porém foi criada uma linguagem de padrões, a LPF3, que guia os desenvolvedores na construção dos frameworks a partir das características dos domínios. Cada padrão da LPF3 se relaciona com um cenário encontrado nos modelos dos domínios e indica quais unidades de código devem ser criadas para que o framework possa implementar a funcionalidade indicada por aquele cenário.

Antkiewicz et al. (2009) elaboraram uma abordagem para a construção de DSLs de frameworks a partir de uma análise no código-fonte de aplicações instanciadas por esses frameworks. Três exemplos de frameworks de middleware (MIF, ver Seção 2.3) e suas DSLs são apresentadas por esses autores. A abordagem de Construção de DSLs de Frameworks

proposta nesta tese se adapta melhor a frameworks de domínios específicos (EAF, ver Seção 2.3), porém também pode ser utilizada em MIFs, como mostrado na Seção 4.3.2.

Amatriain e Arumi (2011) propuseram uma abordagem em que a construção da DSL ocorre paralelamente à do framework, em atividades iterativas e incrementais. Nessa abordagem, o metamodelo da DSL é criado simultaneamente aos modelos do framework. A implementação da DSL ocorre na mesma etapa e utilizando a mesma linguagem de programação do código-fonte do framework. A abordagem de Construção de DSLs de Frameworks, proposta neste doutorado, foi elaborada para ser utilizada após a construção do framework e, portanto, pode ser aplicada a frameworks pré-existentes e/ou construídos por terceiros.

Zhang et al. (2009) propuseram a *Feature-Oriented Framework Model Language* (FOFML) para desenvolver frameworks com base na abordagem MDA. Nessa linguagem, os domínios são definidos em dois modelos: um metamodelo que contém as características; e um modelo textual que define as restrições. As classes dos frameworks são implementadas de acordo com as características e restrições desses dois modelos. Em comparação, na abordagem F3 as características e as restrições são definidas em um único modelo, com o intuito de facilitar a geração do código-fonte e da DSL dos frameworks por meio da ferramenta F3T.

Application-based Domain Modeling (ADOM) é uma abordagem que facilita a especificação de frameworks e a forma como são reutilizados nas aplicações (STURM e KRAMER 2013; REINHARTZ-BERGER e STURM, 2009). Nessa abordagem mecanismos de classificação, como, por exemplo, anotações Java, são adicionados no código-fonte do framework para identificar as características e restrições do seu domínio. Modelos de classes anotados também podem ser criados com o mesmo propósito. O código de uma aplicação, as anotações definidas para o domínio são utilizadas para indicar o reuso das classes do framework que representam as características do domínio. Desse modo, o código pode ser validado de acordo com as restrições do domínio. Em comparação com a abordagem de construção de DSL proposta nesta tese, a vantagem da abordagem ADOM é que não é necessário construir uma DSL.

Perovich et al. (2009) apresentaram uma abordagem de LPS que utiliza técnicas de MDE na fase de Engenharia de Domínio para permitir a automatização da fase de Engenharia da Aplicação. Nessa abordagem os componentes da arquitetura são definidos de acordo com as características do domínio e os modelos de transformação são criados a partir desses componentes para dar origem a geradores de aplicação. Os produtos da LPS são obtidos a partir desses geradores. A união das abordagens F3 e de Construção de DSLs de Frameworks, realizada principalmente com a utilização da ferramenta F3T, resulta em um processo semelhante ao da abordagem de Perovich et al. (2009). A F3T apoia a

construção de um framework, que representa a arquitetura do domínio, e de uma DSL, que possui um gerador de aplicações, com base nas características do domínio.

Oliveira et al. (2011) propuseram uma ferramenta que auxilia no reuso de frameworks chamada de ReuseTool. Essa ferramenta faz uso da *Reuse Description Language* (RDL), que foi criada para descrever *hot-spots* de frameworks (Oliveira et al., 2007). Para realizar a instanciação de um framework a Reuse-Tool acessa o modelo de classes do framework e o arquivo RDL que descreve como esse modelo de classes deve ser manipulado para acomodar novos elementos da aplicação. A partir dessas informações, o desenvolvedor pode criar o modelo de uma aplicação utilizando a interface da ReuseTool e gerar o código-fonte dessa aplicação. Assim como a abordagem de Construção de DSLs proposta nesta tese, a RDL e a ReuseTool têm como objetivo facilitar os processos de instanciação de frameworks.

Stanojević et al. (2011) desenvolveram um framework, denominado Oz Framework, para o domínio de aplicações *desktop* com as operações básicas de persistência. A partir dessa experiência, esses autores propuseram uma abordagem de desenvolvimento de frameworks com as seguintes etapas: 1) Análise do Domínio, em que três aplicações exemplo devem ser analisadas para que a arquitetura geral do framework seja definida; 2) Projeto do Framework, em que são definidos os componentes internos do framework e a sua interface de comunicação com as aplicações; 3) Implementação; 4) Testes; e 5) Documentação. Para cada etapa, Stanojević et al. (2011) indicam algumas sugestões de caráter informal que podem ser seguidas pelos desenvolvedores.

A *Common Variability Language* (CVL) é uma padronização do *Object Manager Group* (OMG) utilizada para especificar as variabilidades dos domínios (CZARNECKI et al., 2012; ROUILLE, 2012). Assim como os modelos F3, CVL is é uma extensão dos modelos de características. Contudo, CVL utiliza mecanismos similares a OCL para implementar as restrições do domínio, enquanto que nos modelos F3 essas restrições são definidas por meio de relacionamentos e propriedades dos elementos.

1.5 Metodologia de Desenvolvimento do Trabalho

A tese de doutorado desenvolvida teve caráter teórico-prático e fez uso de técnicas e de ferramentas para: 1) a modelagem de características de domínio e de aplicações; 2) a construção de templates para a geração de artefatos de código; e 3) o reuso de padrões, frameworks e outros artefatos reutilizáveis. A experiência adquirida e as ferramentas desenvolvidas em trabalhos anteriores, que envolveram padrões de software e frameworks,

serviram de apoio para as atividades definidas pela abordagens propostas nesta tese de doutorado.

Alguns frameworks foram utilizados como exemplos para a criação de DSLs. A partir desses frameworks, foram definidas as regras da abordagem de Construção de DSLs de Frameworks. Essas DSLs foram construídas no Eclipse IDE (THE ECLIPSE FOUNDATION, 2013a) com o apoio do *Graphical Modeling Framework* (GMF) (THE ECLIPSE FOUNDATION, 2013b; GRONBACK, 2009), para a criação das sintaxes das DSLs, e do *Java Emitter Templates* (JET) (THE ECLIPSE FOUNDATION, 2013c), para implementação dos templates dos geradores de código.

As regras da abordagem F3 também foram definidas a partir de estudos realizados sobre alguns frameworks, padrões de projeto e abordagem para modelagem de domínios. O Eclipse IDE foi utilizado no desenvolvimento de frameworks nos estudos realizados com a abordagem F3.

O desenvolvimento da F3T ocorreu em três etapas: 1) a construção do editor gráfico para modelagem de domínios; 2) a construção do módulo de geração de frameworks; e 3) a construção do módulo de geração de aplicações. As principais ferramentas utilizadas no desenvolvimento da F3T também foram: o Eclipse IDE, o GMF e o JET. Além disso, os códigos dos frameworks e das aplicações gerados pelos templates são implementados em linguagem Java (ORACLE, 2013a) e utilizam o banco de dados MySQL (ORACLE, 2013b).

Provas de conceito e experimentos foram realizados para avaliar as abordagens e a ferramenta propostas de acordo com os objetivos desta tese. Nos experimentos houve a participação de alunos de graduação ou de pós-graduação dos cursos da área de computação da Universidade Federal de São Carlos (UFSCar). O objetivo desses experimentos foi avaliar as vantagens e desvantagens das abordagens e da ferramenta propostas, principalmente, com relação à eficiência e à praticidade obtidas com o seu uso.

1.6 Organização da Tese

Esta tese está organizada em sete capítulos. Neste primeiro foi apresentada a contextualização do trabalho desenvolvido, sua motivação, os objetivos pretendidos, os trabalhos relacionados e a metodologia.

No Capítulo 2 são discutidos os principais conceitos utilizados para o desenvolvimento da tese de doutorado, entre os quais se destacam: padrões de software, frameworks, engenharia dirigida por modelos e linhas de produtos de software.

No Capítulo 3 é apresentada a abordagem F3, detalhando como são realizadas a modelagem dos domínios e a construção dos frameworks. O desenvolvimento de um

framework para o domínio de transações de aluguel e comercialização de recursos é usado como exemplo de uso da abordagem.

No Capítulo 4 a abordagem para a construção de DSLs de frameworks é detalhada. Também é exemplificado o seu uso mostrando como foram construídas DSLs para os frameworks GRENJ e Hibernate.

No Capítulo 5 é apresentada a ferramenta F3T, que apoia o desenvolvimento e o reúso de frameworks com base nas abordagens mostradas nos Capítulos 3 e 4. Como exemplo de uso dessa ferramenta, foi desenvolvido um framework para o domínio de clínicas médicas e uma aplicação para clínicas veterinárias.

No Capítulo 6 são descritos três experimentos realizados durante a tese de doutorado para avaliar: 1) o uso de DSLs para a instanciação de frameworks; 2) o uso da abordagem F3 no desenvolvimento de frameworks; e 3) o uso da F3T como ferramenta que apoia as fases de Engenharia de Domínio e Engenharia de Aplicação.

No capítulo 7 são apresentados as conclusões, as contribuições, as limitações, os trabalhos futuros e os artigos publicados desta tese de doutorado.

No Apêndice A é apresentado um tutorial de introdução ao *Graphical Modeling Framework* (GMF), em que são ensinados os passos para a construção de DSLs.

No Apêndice B é apresentado um tutorial a ferramenta Pure::variants em que são explicados os passos para criação dos artefatos das fases de Engenharia de Domínio e de Engenharia da Aplicação.

Capítulo 2

FUNDAMENTAÇÃO TEÓRICA

2.1 Considerações Iniciais

Aplicações de software são ferramentas essenciais para a operacionalização dos processos, para a divulgação de informações e para a comunicação entre os interessados pelos serviços oferecidos pelas empresas. O dinamismo do mercado, a alta competitividade e as mudanças tecnológicas demandam que as aplicações sejam desenvolvidas em curto prazo com baixo custo, com qualidade e possuam flexibilidade para se adaptar a mudanças do ambiente. Contudo, na maioria das vezes, o desenvolvedor não consegue construir uma aplicação que atenda a esses requisitos, pois os vê como conflitantes. Ele acredita que não é possível desenvolver um produto flexível e com qualidade em um curto espaço de tempo. Para os prazos serem atendidos, as aplicações deixam de ser documentadas, discutidas e implementadas adequadamente e o custo pode se elevar por causa de fatores, tais como o aumento de pessoal e a realização de horas extras de trabalho.

Das soluções propostas pela Engenharia de Software para resolver esse impasse, há as que se baseiam na prática do reuso, que consiste no reaproveitamento de recursos previamente construídos (ANGUSWAMY e FRAKES, 2012; PRESSMAN, 2009). A partir do reuso de software obtém-se aumento de eficiência no processo de desenvolvimento, pois o software não é desenvolvido a partir do zero e não se perde tempo recriando algo feito anteriormente. Também é notado um aumento da qualidade do produto desenvolvido, considerando que os recursos reutilizados funcionam corretamente (SHIVA e SHALA, 2007).

O reuso pode ocorrer em diferentes níveis (MAHMOOD et al., 2013; FRAKES e KANG, 2005). Copiar/colar é a forma mais simples dessa prática. Linguagens de programação possuem mecanismos que permitem o reuso de código, como subprogramas, composição e herança. Algumas técnicas e ferramentas promovem reuso em níveis mais altos de abstração, como de projeto e de experiência. Além disso, o reuso pode ocorrer de

forma mais produtiva quando atrelado a um domínio de software, que é formado por um conjunto de aplicações que compartilham características em comum (JEZEQUEL, 2012; GOMAA; 2004; KANG et al., 1990). Nesse caso, as aplicações que compartilham características comuns podem ser beneficiadas das vantagens já citadas.

Neste capítulo são apresentadas algumas das soluções propostas pela Engenharia de Software que aplicam reuso além do nível de código e que serviram de embasamento para a realização desta tese de doutorado. Essas soluções estão apresentadas da seguinte forma: na Seção 2.2, padrões de software; na Seção 2.3, frameworks; na Seção 2.4, engenharia dirigida por modelos; na Seção 2.5, linhas de produto de software; e na Seção 2.6, as considerações finais deste capítulo.

2.2 Padrões de Software

Durante o desenvolvimento de um sistema de software, o desenvolvedor se depara com inúmeros problemas. Entretanto, muitos desses problemas podem ser recorrentes e ocorrer independentemente do tipo e do domínio do software (FREEMAN et al., 2004), como, por exemplo, como representar as entidades de uma aplicação por meio de classes, como implementar uma funcionalidade com comportamento variante e como implementar a persistência de dados. Então, para evitar que os desenvolvedores desperdiçassem tempo reinventando soluções ou usando soluções inadequadas, os problemas recorrentes e suas soluções mais adequadas começaram a ser catalogados e passaram a ser conhecidos como padrões de software (SOMMERVILLE, 2007; COPLIEN, 1996).

Um **padrão de software** é uma solução comprovada de sucesso para um determinado problema recorrente (KIRCHER e VÖLTER, 2007; GAMMA et al., 1995). Padrões descrevem boas práticas de desenvolvimento de software, de forma que outras pessoas possam reutilizar a experiência neles documentada em vários projetos (THE HILLSIDE GROUP, 2013; MANOLESCU et al., 2007; FOWLER, 1997). Ao se deparar com um problema documentado por um padrão, o desenvolvedor aplica a solução proposta nesse padrão, adaptando-a ao contexto da aplicação que está desenvolvendo.

Por documentarem problemas e soluções, os padrões de software representam um vocabulário comum entre os desenvolvedores, pois mencionar quais foram os padrões aplicados no desenvolvimento de um software facilita o seu entendimento e a sua manutenibilidade (RASOOL e MÄDER, 2011; LOO e LEE, 2010). Além disso, o uso de padrões de software passou a ser exigido por diversas empresas de desenvolvimento por reduzirem o esforço necessário para a resolução dos problemas e por tornarem as implementações mais concisas (LETHBRIDGE, 2000).

Existem diferentes formatos para documentar padrões. Em geral, os autores procuram apresentar as seguintes informações (ZAMANI e BUTLER, 2009; DRUCKENMILLER et al., 2010; HARRISON et al., 2007; GAMMA et al., 1995):

- Nome: simples, significativo e que represente a essência do padrão.
- Classificação: agrupa os padrões de acordo com uma característica comum. Por exemplo, os padrões de projeto foram classificados de acordo com o seu propósito em três grupos: criação, estrutural e comportamental (GAMMA et al., 1995).
- Motivação: explica o que o padrão faz, o problema que ele resolve e a forma como é feita.
- Aplicabilidade: indica em quais situações o padrão pode se aplicado.
- Estrutura: modelo representativo da solução proposta pelo padrão.
- Implementação: exemplo de código que implementa a solução proposta pelo padrão em uma linguagem de programação específica.

Os padrões de software mais conhecidos são os de projeto, que apóiam o desenvolvimento de projetos e arquiteturas mais flexíveis, extensíveis, reutilizáveis e com maior manutenibilidade (HSUEH et al., 2010; FREEMAN et al., 2004; GAMMA et al., 1995). Essas vantagens permitem que os padrões de projeto sejam úteis no desenvolvimento de artefatos de software que demandam grande adaptabilidade, como, por exemplo, frameworks e componentes (ALMEIDA et al., 2007; LEE e KANG, 2004; BRAGA, 2002; KEEPECE e MANNION, 1999).

Na Figura 2.1 é mostrado um exemplo de uso do padrão de projeto *Strategy*, que auxilia na definição de diferentes comportamentos para uma função (FREEMAN et al., 2004; GAMMA et al., 1995). A classe `DocumentPanel` permite a visualização de documentos de texto simples, do MS Word (*.doc*) ou PDF. Apesar de ser a mesma operação, a forma como cada tipo de documento é visualizado é diferente. A classe `FileStrategy` define uma estratégia genérica para visualização dos documentos e as classes `TxtStrategy`, `DocStrategy` e `PDFStrategy` definem os algoritmos específicos para visualização de cada tipo de arquivo por meio do método `show`. O método `show` de `DocumentPanel` possibilita a visualização dos documentos de acordo com a estratégia escolhida.

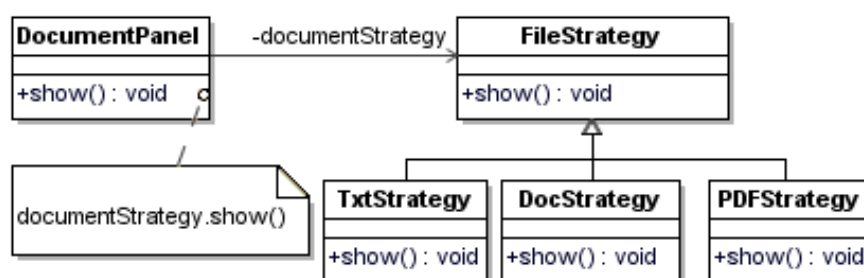


Figura 2.1. Exemplo de utilização do padrão *Strategy*.

Além dos padrões de projeto, existem padrões que atendem a outros propósitos, tais como: de análise, de arquitetura, de implementação (idiomas), de reengenharia, de processo, organizacionais, entre outros (NIWE e STIRNA, 2009; KIRCHER e VÖLTER, 2007; SHALLOWAY e TROTT, 2004). Os padrões de análise, de projeto, de arquitetura e de implementação estão intimamente relacionados com as etapas de desenvolvimento de software e servem de apoio à realização de suas atividades. Por exemplo, um padrão de análise pode indicar quais classes devem ser criadas na construção da lógica de negócio da aplicação, enquanto que um padrão de implementação pode descrever como implementar relações entre componentes de uma determinada linguagem de programação.

A maioria dos padrões são *stand-alone*, pois descrevem soluções pontuais para problemas que ocorrem em contextos específicos. Entretanto, alguns padrões podem ser utilizados em conjunto para resolver problemas mais complexos. Em alguns casos, o uso desses padrões em conjunto se tornou tão comum que passou a ser conhecido como um **padrão composto** (BUSCHMANN et al., 2007). Um exemplo de padrão composto é o *Model-View-Controller* (MVC), que trata da separação entre a lógica do negócio e a interface com o usuário de uma aplicação e é formado pelos padrões de projeto *Observer*, *Composite* e *Strategy* (FREEMAN et al., 2004). Na Figura 2.2 estão ilustrados os relacionamentos entre os padrões que compõem o MVC. Os dados da camada de modelo são alterados a partir de eventos executados por usuários na camada de visão. A camada de controle trata esses eventos e pode realizar modificações na camada de modelo ou diretamente na camada de visão. Quando a camada de modelo tem seus dados alterados, ela atualiza a informação mostrada na camada de visão. A camada de visão também pode requisitar dados da camada de modelo.

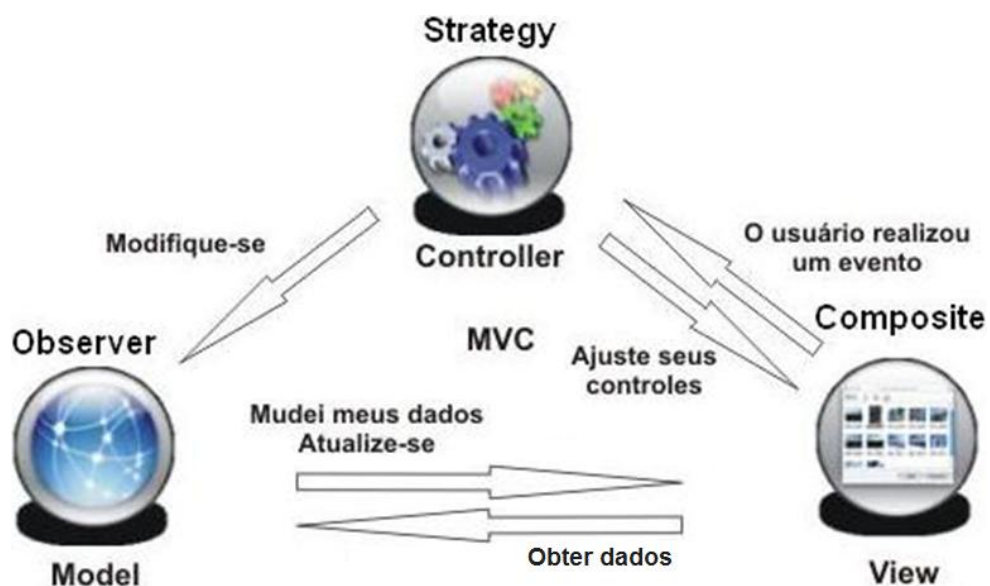


Figura 2.2. Esquema do padrão MVC, adaptado de FREEMAN et al. (2004).

Uma **Linguagem de Padrões (LP)** é uma sequência de padrões com uma relação de dependência entre si, atendendo a um propósito maior. Diferentemente dos padrões compostos, uma LP é específica de domínio, possui um padrão inicial e fornece um guia de como aplicar um padrão após o outro. Além disso, enquanto um padrão resolve apenas um problema isolado dentro de uma aplicação, as linguagens de padrões podem ser aplicadas no desenvolvimento de toda a lógica funcional de um software (THE HILLSIDE GROUP, 2013; ZAMANI e BUTLER, 2009; BUSCHMANN *et al.*, 2007; COPLIEN, 1996).

Os padrões de uma linguagem de padrões compõem um grafo que indica a ordem na qual eles devem ser aplicados. Cada padrão cria o contexto para o próximo padrão a ser utilizado. Dessa forma, uma LP representa a sequência temporal de decisões que guiam o processo de desenvolvimento de software. É possível que alguns padrões sejam opcionais e que haja mais de uma sequência de uso dos padrões. Também é possível que um ou mais padrões sejam utilizados mais de uma vez (ZAMANI e BUTLER, 2009; BRAGA, 2002; BRUGALI e SYCARA, 2000).

2.2.1 A Linguagem de Padrões Gestão de Recursos de Negócios

Nesta seção é apresentada a linguagem de padrões de análise Gestão de Recursos de Negócios (GRN) (BRAGA *et al.*, 1999), utilizada como exemplo em alguns trabalhos. A GRN apoia o desenvolvimento de aplicações no domínio de transações de aluguel, compra, venda e manutenção de bens e serviços, fornecendo reuso de experiência para a identificação das classes que compõem as aplicações.

Na Figura 2.3 é mostrado o grafo do fluxo dos padrões da GRN com a ordem em que eles podem ser aplicados. Ao todo são quinze padrões de análise organizados em três grupos: Identificação do Recurso do Negócio, Transações de Negócio e Detalhes da Transação de Negócio (BRAGA *et al.*, 1999).

O primeiro grupo de padrões da GRN tem como objetivo definir o recurso e a forma como ele deve ser tratado nas aplicações. Três padrões compõem esse grupo:

1. **Identificar o Recurso:** descreve como especificar o recurso e como classificá-lo sobre diversas perspectivas.
2. **Quantificar o Recurso:** especifica que o recurso pode se apresentar de quatro formas: único (simples), instanciável, mensurável ou agrupado em lotes.
3. **Armazenar o Recurso:** padrão opcional que trata de informações relacionadas à forma e ao local de armazenamento do recurso.

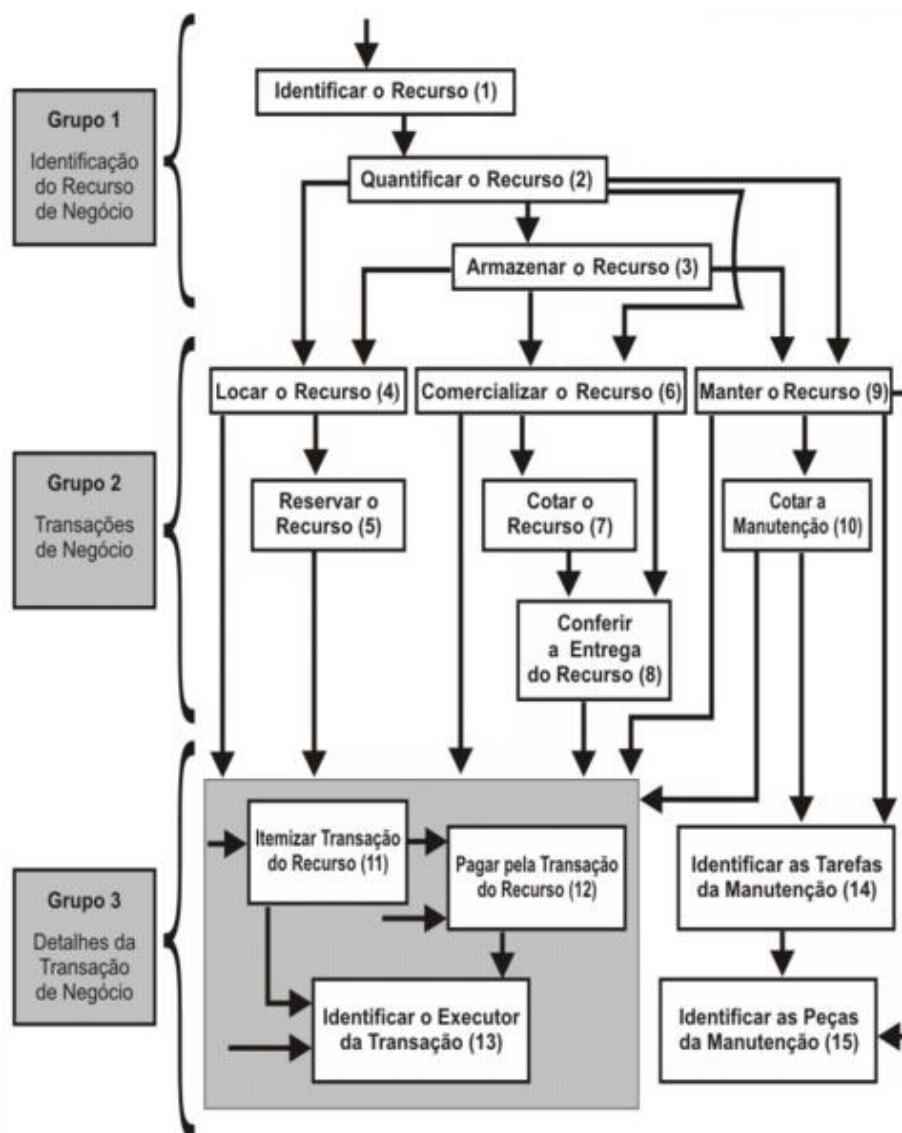


Figura 2.3. Grafo do fluxo dos padrões da GRN (BRAGA et al, 1999).

No segundo grupo estão os padrões responsáveis pelas transações (Braga e Masiero, 2002). Os padrões desse grupo são:

4. **Locar o Recurso:** trata de transações relativas ao aluguel de recursos.
5. **Reservar o Recurso:** possibilita a reserva de recursos para posterior locação.
6. **Comercializar o Recurso:** trata de transações de venda de recursos.
7. **Cotar o Recurso:** contabiliza um valor prévio da comercialização de um recurso.
8. **Conferir a Entrega do Recurso:** trata das informações relacionadas com o envio dos recursos envolvidos nas transações de venda.
9. **Manter o Recurso:** trata das transações de manutenção de recursos.
10. **Cotar a Manutenção:** contabiliza um valor prévio da manutenção de um recurso.

O terceiro e último grupo de padrões da GRN é responsável por definir detalhes relevantes às transações. Os padrões 11 a 13 são opcionais e aplicáveis a todos os tipos de transações tratadas pela GRN, enquanto que os dois últimos são aplicados somente para completar a transação de manutenção. Segue-se a descrição dos padrões do terceiro grupo:

- 11. **Itemizar Transação do Recurso:** define como tratar mais de um recurso por transação e adicionar informações específicas a cada recurso envolvido.
- 12. **Pagar pela Transação do Recurso:** permite gerenciar diferentes formas de pagamento para as transações.
- 13. **Identificar o Executor da Transação:** estabelece como manter informações sobre o usuário do sistema que registram as transações.
- 14. **Identificar as Tarefas da Manutenção:** especifica como registrar informações sobre as tarefas necessárias para realizar a manutenção de recursos.
- 15. **Identificar as Peças da Manutenção:** mantém informações sobre as peças utilizadas nas transações de manutenção de recursos.

Cada padrão da GRN é documentado com uma descrição textual e com dois modelos de classes da UML em nível de análise: um que representa a solução proposta pelo padrão e outro que contém um exemplo de uso desse padrão. Na Figura 2.4.a é mostrado o modelo de classes do quarto padrão da GRN, Locar o Recurso, e na Figura 2.4.b é apresentado um modelo de classes com um exemplo de utilização desse padrão.

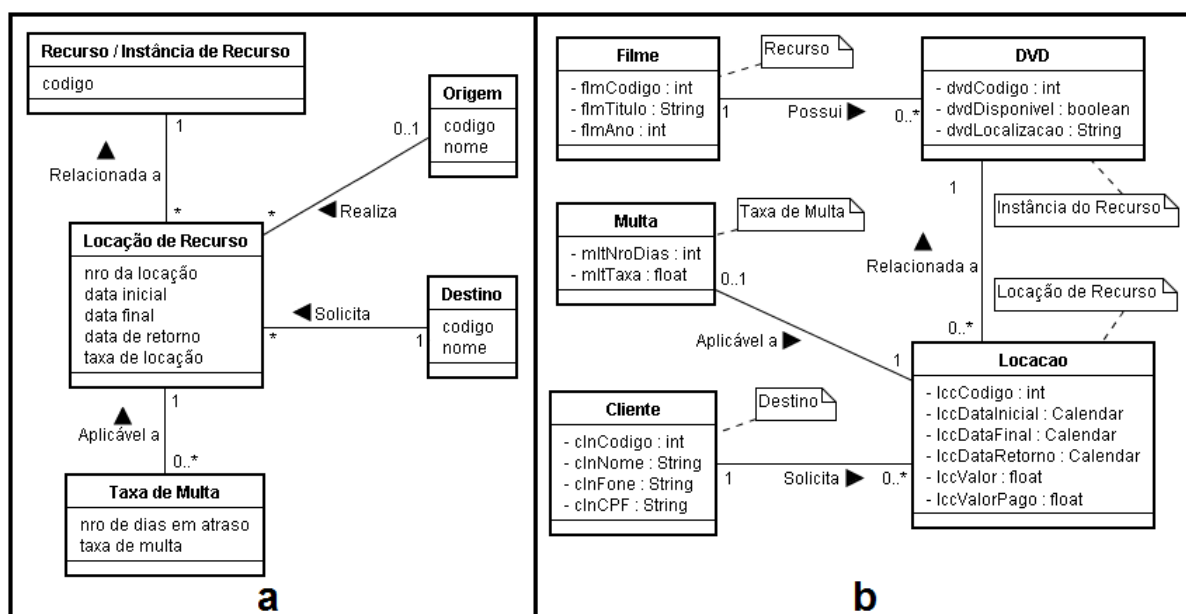


Figura 2.4. Modelos de classes do quarto padrão da GRN, Locar o Recurso.

A fim de ilustrar o uso da GRN, no Quadro 2.1 estão listados os requisitos de uma aplicação para uma locadora de DVDs e na Figura 2.5. Modelo de classe da aplicação para

uma locadora de DVDs. é mostrado o modelo de classes dessa aplicação criado com base no uso desses padrões. As classes desse modelo estão agrupadas de acordo com os padrões da GRN utilizados e os estereótipos dessas classes referem-se aos nomes das classes correspondentes nos modelos desses padrões.

Quadro 2.1. Requisitos de uma aplicação para uma locadora de DVDs.

1	A locadora realiza o aluguel de DVDs de filmes que podem ter uma ou mais cópias. Cada filme possui um código, título e ano.
2	Cada DVD possui código, localização, status de disponibilidade e o filme nele contido.
3	Os filmes são classificados por categoria que indica o valor diário da locação.
4	Os filmes também são classificados por gênero (comédia, terror, ação, etc.).
5	Os DVDs são alugados para os clientes cadastrados da locadora. As informações sobre o cliente são: código, nome, telefone e CPF.
6	As informações de locação são: código, data de locação, data de devolução prevista, código do cliente, DVDs alugados, data de devolução efetiva e valor. Um cliente pode alugar mais de um DVD em uma mesma locação.

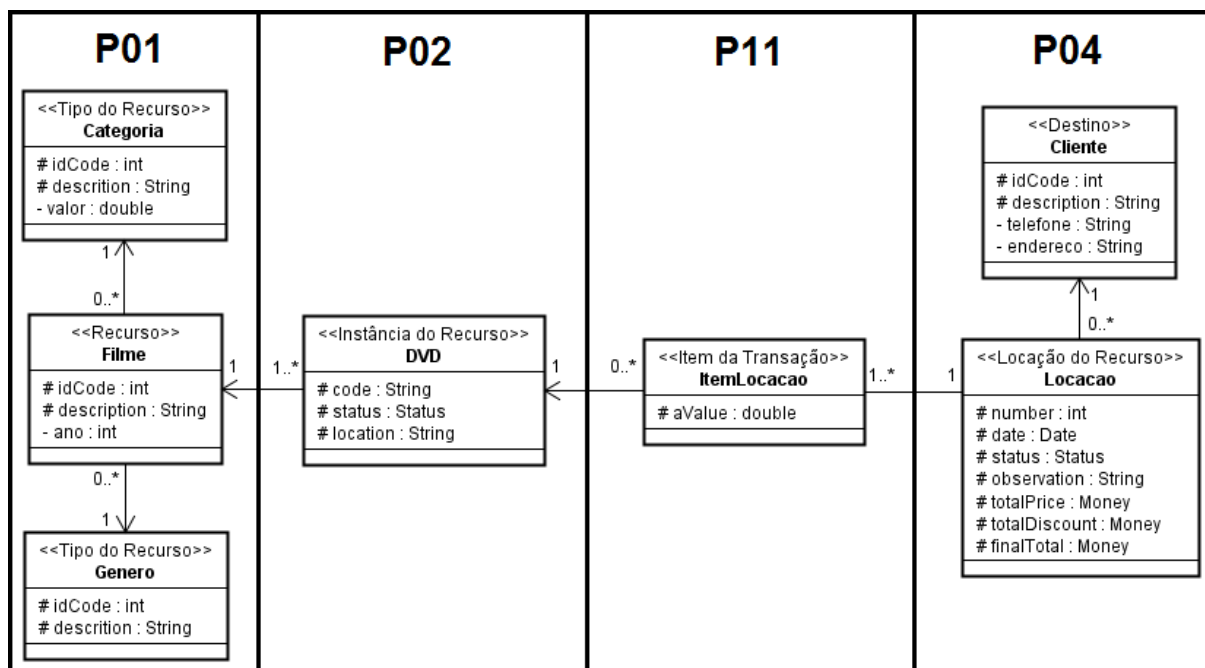


Figura 2.5. Modelo de classe da aplicação para uma locadora de DVDs.

2.3 Frameworks

Um **framework** é um software reutilizável que implementa a estrutura básica de um domínio (FAYAD e JOHNSON, 1999). O desenvolvimento de uma aplicação com o reuso de um framework é feito complementando a funcionalidade do framework com as características específicas dessa aplicação (ANTKIEWICZ et al., 2009; ABI-ANTOUN, 2007; SRINIVASAN, 1999).

Frameworks são compostos de um conjunto de classes abstratas e concretas, assim como bibliotecas de classes (JOHNSON, 1997). Contudo, as classes de uma biblioteca possuem um menor grau de dependência e resolvem problemas de escopo limitado. Já as classes de um framework atuam em conjunto para resolver problemas mais amplos pertencentes ao domínio do framework. Outra diferença é a inversão de controle determinada pelo framework, pois é ele que invoca o código da aplicação, ao contrário do que ocorre com classes de biblioteca, que tem sua utilização determinada pelo código da aplicação (OKANOVIC e MATELJAN, 2011; FREEMAN et al., 2004).

Conceitualmente, frameworks são compostos de duas partes: os pontos fixos (*frozen spots*), que constituem a parte imutável do framework, cujo comportamento se mantém igual independentemente da aplicação que esteja sendo desenvolvida, e os pontos variáveis (*hot spots*), que representam a parte utilizada pelo desenvolvedor para instanciar o framework de acordo com as especificações da aplicação desejada (OLIVEIRA et al., 2011; STANOJEVIĆ et al., 2011; JOHNSON, 1997).

De acordo com a forma de reutilização, frameworks são classificados como: 1) caixa branca, quando o acesso ocorre por meio da herança de suas classes e da sobrescrita de métodos; 2) caixa preta, quando o acesso ocorre pela criação de objetos e/ou pela chamada dos métodos dos componentes do framework; 3) caixa cinza, quando o acesso ocorre das duas maneiras anteriores (AMATRIAIN e ARUMI, 2011; ABI-ANTOUN, 2007; FAYAD e SCHMIDT, 1997).

Em outra classificação, que considera o propósito, os frameworks são classificados como: 1) *System Infrastructure Frameworks* (SIF), quando apoiam o desenvolvimento de softwares que controlam operações de baixo nível, como sistemas operacionais, gerenciadores de janelas gráficas e sistemas embarcados; 2) *Middleware Integration Frameworks* (MIF), quando apoiam a implementação de requisitos não funcionais comuns a muitas aplicações, como persistência e interface web; 3) *Enterprise Application Frameworks* (EAF) ou frameworks de aplicação, quando implementam domínios de negócio e podem ser utilizados como *core assets* em linhas de produtos de software (ver Seção 2.5) (ABI-ANTOUN, 2007; XU e BUTLER, 2006; FAYAD e SCHMIDT, 1997).

O uso de frameworks como apoio ao processo de desenvolvimento de aplicações oferece vantagens como reúso de projeto e de código, robustez e eficiência (STURM e KRAMER, 2013). A desvantagem está na complexidade dessa tarefa. Frameworks são difíceis de se aprender e reusar (BRUCH, 2010; ROBILLARD, 2009; HOU, 2008). É necessário conhecer toda a estrutura do framework para saber identificar quais elementos devem ser reutilizados e como fazer isso em cada aplicação. Portanto, a curva de aprendizado tende a ser íngreme, proporcional à complexidade do framework.

Durante o desenvolvimento de uma aplicação, alguns problemas podem ocorrer e, conseqüentemente, as vantagens fornecidas pelo framework não são obtidas. Pode-se citar como exemplo, a identificação incorreta das classes que devem ser reutilizadas pela aplicação, pois o framework não oferece apoio à etapa de análise. Kirk *et al.* (2007) realizaram uma pesquisa na qual foram apontadas as quatro maiores dificuldades para instanciação de um framework:

1. Mapeamento: como expressar corretamente o problema utilizando o framework.
2. Compreensão da funcionalidade: como entender o comportamento do framework.
3. Compreensão da interação: como entender a comunicação entre as classes do framework.
4. Compreensão da arquitetura: como adaptar os detalhes das aplicações de forma que a integridade e as características não funcionais do framework sejam preservadas.

À medida que se adquire experiência com um framework, as dificuldades para sua utilização são amenizadas. Uma LP em nível de análise pode servir de documentação e auxiliar no entendimento das classes que compõem o domínio desse framework (KIRK et al., 2007; BRUGALI e SYCARA, 2000). Além disso, linguagens específicas de domínio (ver Capítulo 3) podem ser adotadas para facilitar a instanciação de frameworks (ANTKIEWICZ et al., 2009; SANTOS et al., 2008; BRAGA e MASIERO, 2003).

Outra desvantagem que os frameworks apresentam é quanto ao seu desenvolvimento, que é mais complexo do que o de uma aplicação (OKANOVIC e MATELJAN, 2011; ZHANG et al., 2009), pois:

- As características comuns, opcionais e variantes das aplicações que poderão ser instanciadas pelo framework devem ser identificadas;
- O projeto de um framework é formado, principalmente, por classes abstratas;
- A estrutura de um framework deve ser adaptável o suficiente para que possa ser instanciado em diversas aplicações;
- As variabilidades do domínio fazem com que algumas funções possuam diferentes comportamentos a depender de certas condições, aumentando o tamanho do código e o grau de complexidade de implementação;
- Os detalhes das aplicações que reutilizarão o framework não são conhecidos durante o desenvolvimento do framework, sendo necessário adicionar mecanismos que realizam a identificação desses detalhes em tempo de execução;
- Um framework não é um produto final, no sentido de que ele não funciona sem que seja instanciado em uma aplicação, o que dificulta a realização de testes.

Recursos das linguagens de programação orientadas a objetos, como herança e polimorfismo, e de recursos avançados, como anotações, genéricos e *reflection* (SOBEL e FRIEDMAN, 2011), podem fornecer maior adaptabilidade aos frameworks. Além disso,

padrões de projeto, como *Strategy* e *Abstract Factory*, também podem auxiliar na construção das variabilidades dos frameworks (ALMEIDA et al., 2007; SRINIVASAN, 1999).

2.3.1 Framework GRENJ

Nesta seção é apresentado o framework Gestão de REcursos de Negócios com implementação em Java (GRENJ) (VIANA, 2009; DURELLI, 2008), que foi utilizado como objeto de estudo durante a tese de doutorado. Esse framework é um EAF caixa branca que implementa o domínio definido pela linguagem de padrões GRN (BRAGA et al., 1999).

Na Figura 2.6. Arquitetura do Framework GRENJ. é mostrada a arquitetura do framework GRENJ, que é baseada no padrão *Model-View-Controller*. A camada de persistência é responsável pela leitura e escrita no banco de dados das aplicações que reutilizam esse framework. *PersistentObject*, a principal classe dessa camada, foi implementada com base no padrão *Persistent Layer* (YODER et al., 1998). A comunicação com o sistema de banco de dados MySQL (ORACLE, 2013b) ocorre por meio da biblioteca de classes *java.sql* e do conector JDBC.

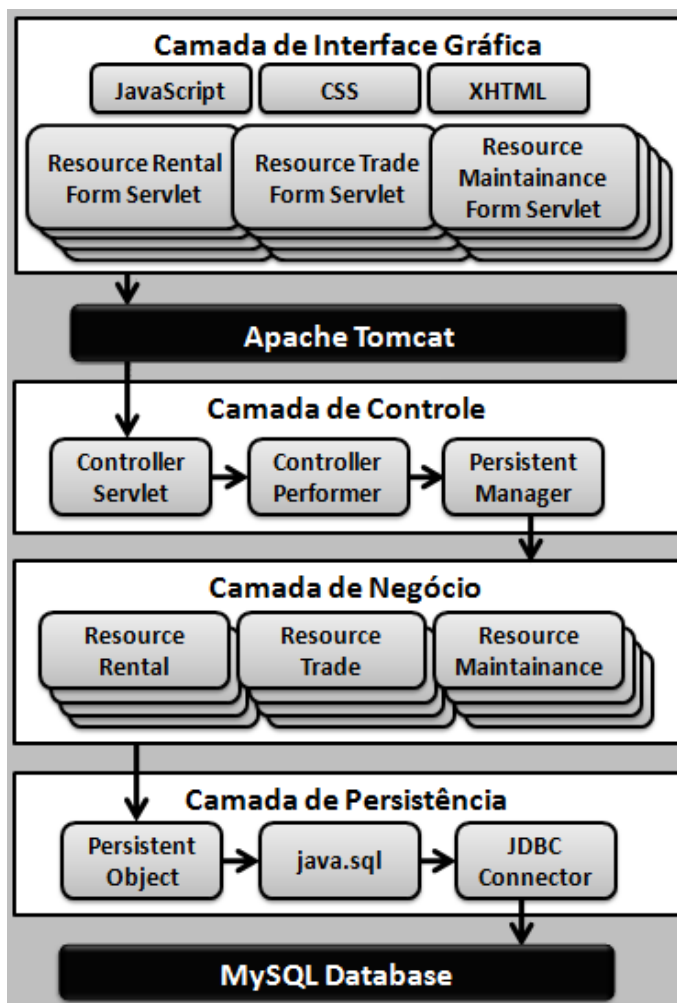


Figura 2.6. Arquitetura do Framework GRENJ.

As classes que representam as características do domínio definido pela linguagem de padrões GRN estão implementadas na camada de negócio (modelo) do framework GRENJ. O desenvolvimento de uma aplicação com o reuso do framework GRENJ ocorre em três etapas: 1) análise, em que os padrões da GRN apoiam a identificação das classes da aplicação de acordo com os requisitos; 2) projeto, em que são identificadas as classes do framework GRENJ que são estendidas pelas classes da aplicação; e 3) implementação, em que ocorre a construção do código-fonte da aplicação com o reuso do framework GRENJ. No Quadro 2.2 são mostrados alguns exemplos de correspondência entre as classes dos padrões da GRN e as do framework GRENJ.

Quadro 2.2. Relação entre algumas classes da GRN e do framework GRENJ.

Padrão da GRN	Classe na GRN	Classe Correspondente no Framework GRENJ	Descrição da Classe
Identificar o Recurso	Recurso	<i>Resource</i>	Bem ou serviço tratado pelas aplicações
	Tipo de Recurso	<i>SimpleType</i>	Tipo simples para classificação dos recursos.
		<i>NestedType</i>	Tipo aninhado para classificação dos recursos.

			Aceita subtipos.
Quantificar o Recurso	Recurso Simples	<i>SingleResource</i>	Indica que o recurso é único.
	Recurso Instanciável	<i>InstantiableResource, ResourceInstance</i>	Representa cópias de um recurso.
Locar o Recurso	Locação do Recurso	<i>ResourceRental</i>	Transação de aluguel.
	Destino	<i>DestinationParty</i>	Destino do recurso nas transações.

Na camada de controle estão as classes que fazem a comunicação entre as camadas de interface gráfica e de negócio. A classe `PersistentManager` e suas subclasses são responsáveis por executar as operações, requisitadas por eventos ocorridos na interface gráfica, manipulando objetos das classes da camada de negócio. A alteração dos dados de um filme na aplicação para uma locadora de DVDs é um exemplo de uma operação requisitada por um evento originado na camada de interface gráfica.

A camada de interface gráfica (visão) é formada, principalmente, por *servlets* que devem ser estendidos para originar as páginas web das aplicações. A maioria desses *servlets* está relacionada com as classes da Camada de Negócio. Por exemplo, no modelo de classes da aplicação para uma locadora de DVDs mostrado na Figura 2.5, a classe `Filme` estende a classe `Recurso` do primeiro padrão da GRN, que corresponde à classe `Resource` no framework GRENJ. De forma equivalente, na Figura 2.7 é mostrado que a classe `FilmeFormServlet` estende `ResourceFormServlet` na camada de interface gráfica para originar a página web que permite o cadastro de filmes.

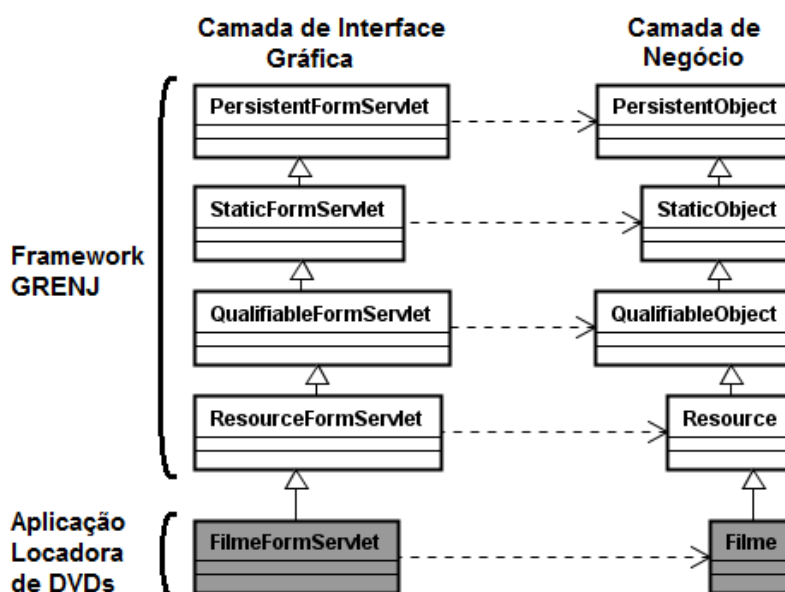


Figura 2.7. O reuso do framework GRENJ ocorre com a extensão das classes das camadas de negócio e de interface gráfica.

O framework GRENJ possui um wizard gerador de aplicações que facilita o desenvolvimento de aplicações (VIANA, 2009). Esse wizard é formado por formulários que

representam as classes do framework GRENJ e são preenchidos de acordo com os requisitos da aplicação. Na Figura 2.8.a é mostrado o formulário relacionado com a classe `Resource` do framework GRENJ preenchido com os dados da classe `Filme` da aplicação para uma locadora de DVDs. A classe `FilmeFormServlet`, que origina a página web mostrada na Figura 2.8.b, é gerada a partir desse formulário.

(a) **Form: Identify Resource - Variant: Resource**

Class Name:

Form Title:

Type	Name
int	ano

Fields:

Add Edit Remove

(b) **Cadastro de Filmes**

IdCode	Description	Categoria	Genero	Ano
--------	-------------	-----------	--------	-----

IdCode:

Description:

Categoria:

Genero:

Ano:

Figura 2.8. Utilização do wizard do framework GRENJ.

2.3.2 Framework Hibernate

O framework Hibernate é um MIF caixa preta que auxilia na implementação da funcionalidade de persistência por meio do mapeamento objeto-relacional entre objetos Java e tabelas de banco de dados relacionais (JBOSS COMMUNITY, 2012). O objetivo desse framework é separar a funcionalidade de persistência dos dados da lógica das aplicações.

O framework Hibernate é uma implementação da Java Persistence API (JPA) 2.0 (ORACLE, 2013c). Essa API determina que uma classe, cujos dados dos objetos devem ser persistidos no banco de dados, é uma Entidade (*Entity*). Além disso, consultas pré-definidas para a recuperação dos dados no banco de dados são denominadas *Named Queries*. Uma particularidade dos frameworks que implementam a JPA 2.0 é que o acesso as suas classes é realizado por meio de anotações, tornando a sua reutilização mais simples.

```

@Entity
@Table(name = "venda", catalog = "vendas", schema = "")
@XmlRootElement
@NamedQueries({
    @NamedQuery(name = "Venda.findAll", query = "SELECT v FROM Venda v"),
    @NamedQuery(name = "Venda.findById",
        query = "SELECT v FROM Venda v WHERE v.id = :id"),
    @NamedQuery(name = "Venda.findByData",
        query = "SELECT v FROM Venda v WHERE v.data = :data"),
    @NamedQuery(name = "Venda.findByStatus",
        query = "SELECT v FROM Venda v WHERE v.status = :status"))
public class Venda implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "id")
    private Integer id;

    @Column(name = "data")
    @Temporal(TemporalType.DATE)
    private Date data;

    @Column(name = "status")
    private Character status;

    @OneToMany(cascade = CascadeType.ALL, mappedBy = "venda")
    private List<Itemvenda> itemvendaList;
}

```

Figura 2.9. Parte do código da classe Venda com reuso do framework Hibernate.

Parte do código da classe `Venda` de uma aplicação é mostrada na Figura 2.9. Nesse código, as anotações definem o reuso do framework Hibernate. A anotação `@Entity` define a classe como uma entidade e `@Table` indica a tabela do banco de dados que armazena os dados dessa classe. A anotação `@NamedQuery` foi utilizada para a recuperação dos dados de todas as vendas ou com base nos atributos `id`, `data` e `status`. A anotação `@Id` indica o atributo `id` como sendo a chave primária da tabela no banco de dados e a anotação `@Column` identifica qual coluna dessa tabela corresponde a cada atributo da classe. O atributo `itemvendaList`, originado de um relacionamento de associação de `Venda` com a classe `ItemVenda` é identificado pela associação `@OneToMany`.

2.4 Engenharia Dirigida por Modelos

Modelagem é o ato de se criar um modelo para obter uma melhor compreensão do que se deseja construir (MCLAUGHLIN et al., 2006). Durante o desenvolvimento de uma aplicação, modelos são criados antes de sua implementação com o intuito de facilitar o entendimento e de servir como documentação dessa aplicação. Esses modelos contêm uma abstração dos elementos que irão compor o código da aplicação, uma vez que são utilizadas linguagens de modelagem voltadas para o domínio de programação de software, por exemplo, a *Unified Modeling Language* (UML) (LARMAN, 2004).

Modelos como os da UML não fazem parte do software em si, embora sejam importantes para o entendimento e a construção. Os desenvolvedores os criam, mas os descartam, implementando as funções de forma manual e realizando manutenções somente no código-fonte. Desse modo, o conhecimento acerca da solução fica criptografado no código-fonte e dificilmente é reutilizado. Portanto, há necessidade de criar modelos que representem esse conhecimento e que possam ser úteis tanto para a documentação quanto para a construção e manutenção de software (LUCRÉDIO, 2009).

A partir dessa ideia, a Engenharia Dirigida por Modelos (*Model-Driven Engineering – MDE*) surge como uma abordagem que propõe reduzir a distância entre o domínio do problema e a implementação de uma solução. Nessa abordagem, o desenvolvimento de software ocorre por meio de modelos que protegem os desenvolvedores das complexidades da implementação e de transformações que originam o código-fonte de maneira automatizada a partir das informações contidas nesses modelos (BEN-AMMAR e MAHFOUDDHI, 2013; LUCRÉDIO, 2009; FRANCE e RUMPE, 2007; SCHMIDT, 2006; MELLOR et al., 2003).

Na MDE o enfoque do desenvolvimento é direcionado aos modelos, ou seja, a modelagem deixa de ser meramente uma forma de planejar o código e passa a ser uma forma de construir o software. O nível de abstração da programação torna-se mais elevado, reduzindo a necessidade do desenvolvedor de interagir manualmente com o código-fonte (BRAGANÇA e MACHADO, 2007). O Object Management Group (OMG, 2013) definiu um modelo de arquitetura para o MDE, conhecido como *Model-Driven Architecture* (MDA), que divide o desenvolvimento de software em níveis de abstração (MAGALHAES et al., 2013; MUKHTAR et al., 2013; FRANCE e RUMPE, 2007):

- Modelo Independente de Computação (*Computation Independent Model – CIM*): descreve o negócio, ou ambiente, no qual o software irá operar. Como a decisão a respeito da informatização é feita por uma pessoa e não por uma ferramenta de

transformação, dificilmente a transformação automatizada desse modelo para o de nível seguinte é implementada;

- Modelo Independente de Plataforma (*Platform Independent Model* - PIM): nível de análise, em que ocorre a definição das características do software ou domínio. Esse modelo pode ser transformado para um ou mais modelos do nível seguinte.
- Modelo Específico de Plataforma (*Platform Specific Model* - PSM): nível de projeto, considera as tecnologias de implementação, devendo existir uma instância desse modelo para cada plataforma de implantação do software. A partir desse modelo ocorre a transformação para o código-fonte;

Com base nos princípios mencionados, a MDE possui as seguintes vantagens (HUTCHINSON et al., 2011; FRANCE e RUMPE, 2007; SCHMIDT, 2006):

- 1) Maior facilidade na criação dos modelos das aplicações, pois é utilizada uma linguagem específica para o domínio do problema;
- 2) Maior produtividade e redução do esforço dos desenvolvedores, pois a maior parte do código-fonte das aplicações pode ser gerado a partir dos modelos;
- 3) Maximização do tempo de vida útil dos modelos e de outros artefatos, pois como são necessários para a geração de código, eles são menos propensos a serem descartados pelos desenvolvedores nos processos de manutenção;
- 4) Flexibilidade de desenvolvimento, pois os modelos independentes de plataforma armazenam a lógica do sistema e são menos sensíveis a essas mudanças;
- 5) O conhecimento a respeito do software não fica exclusivamente na mente dos desenvolvedores e no código, fazendo com que os processos de desenvolvimento e de manutenção fiquem menos vulneráveis às oscilações de pessoal.

Linguagens específicas de domínio (*Domain-Specific Language* - DSL) são utilizadas na MDE para facilitar a modelagem e as transformações. Além de serem mais simples do que as linguagens de propósito geral, as DSLs detêm informações inerentes ao seu domínio (características, regras, restrições), o que facilita a geração de código proposta pelo MDE (DJUKIC et al., 2011; KELLY e TOLVANEN, 2008).

DSLs são constituídas de duas partes: a sintaxe abstrata, na qual são definidos os elementos, os relacionamentos e as regras do domínio; e a sintaxe concreta, que consiste na aparência externa a partir da qual a DSL é utilizada, podendo ter uma notação gráfica, textual, em árvore, etc. (RUMPE et al., 2010; CUADRADO e MOLINA, 2009).

A sintaxe abstrata pode ser definida em um metamodelo a partir das características do domínio da DSL (GRONBACK, 2009). Na Figura 2.10 é mostrado um metamodelo no qual está representada uma adaptação do domínio de veículos autônomos do kit Lego

Mindstorms® NXT 2.0². A metaclass `MindStorm` é o elemento raiz que contém todas as características utilizadas nos modelos dos veículos e `Feature` é a super metaclass abstrata de todas as características do domínio. As demais metclasses representa as características do domínio. Os relacionamentos de associação com multiplicidade mínima igual a 1 indicam que a característica alvo é obrigatória e os com multiplicidade mínima igual a 0 indicam que a característica alvo é opcional. A multiplicidade máxima indica a quantidade máxima de instâncias das características em cada veículo.

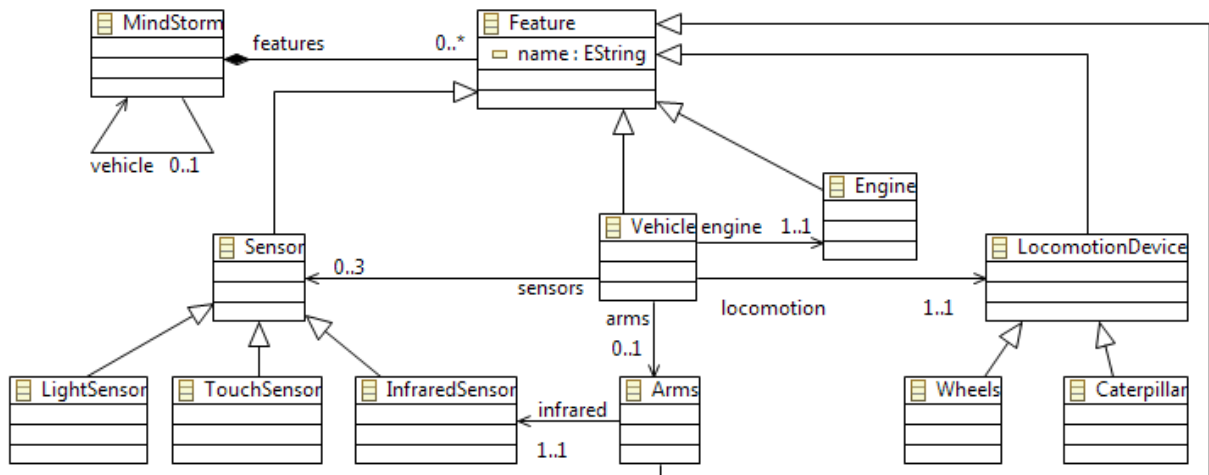


Figura 2.10. Metamodelo do domínio de veículos autônomos do Lego Mindstorms®.

Na sintaxe concreta são definidos os elementos que compõem a notação da DSL e a forma como eles se organizam nessa notação (GRONBACK, 2009). Os passos para a construção da sintaxe concreta dependem da ferramenta utilizada para esse fim e da notação escolhida.

Transformações podem ser realizadas nos modelos criados a partir de DSLs com o intuito de gerar outros artefatos, como, por exemplo, código em linguagem de programação, páginas web, código de testes e outros modelos (MAGALHÃES et al., 2013; RUMPE et al., 2010). Essas transformações podem ser realizadas de modelo para modelo (*Model to Model* - M2M) ou de modelo para texto (*Model to Text* - M2T) (CZARNECKI e HELSEN, 2006). Algumas ferramentas de transformação são específicas de domínio e geram sempre os mesmos tipos de artefatos. Contudo, são mais utilizadas ferramentas de transformação que podem ser configuradas por meio de templates, os quais funcionam como esqueletos que definem o formato resultante das transformações. Nesses casos, a ferramenta de transformação acessa as informações dos modelos e as combina com os templates para gerar novos artefatos (GRONBACK, 2009).

² <http://mindstorms.lego.com/en-us/products/default.aspx/#t>

Um template é formado por partes fixas e variantes (SARASA-CABEZUELO et al., 2012). As partes fixas são copiadas e inseridas, sem alterações, em todos os artefatos gerados a partir desse template. As partes variantes são formadas por comandos da linguagem de transformação utilizada na implementação do template que referenciam informações extraídas dos modelos das aplicações. O conteúdo inserido nos artefatos gerados varia conforme essas informações. Por exemplo, em linguagens de transformação com formato XML, como *eXtensible Stylesheets Language* (XSL) (W3C, 2013) e *Java Emitter Templates* (JET) (GRONBACK, 2009), as partes fixas dos templates consistem de textos e as partes variantes consistem de *tags* que representam comandos da linguagem.

2.4.1 Graphical Modeling Framework

O *Graphical Modeling Framework* (GMF) é uma ferramenta pertencente ao Eclipse IDE que permite a construção de DSLs. É formado pela junção de outras duas ferramentas: o *Eclipse Modeling Framework* (EMF), para a construção de ferramentas CASE que permitem a modelagem de dados em formato XML; e o *Graphical Editing Framework* (GEF), que permite a construção de editores gráficos e interfaces para o Eclipse IDE (THE ECLIPSE FOUNDATION, 2013b).

Com o GMF, a sintaxe abstrata de uma DSL é construída por meio da modelagem do metamodelo e da geração do código do editor dessa DSL, que permite salvar as informações a respeito das aplicações em arquivos XML. A sintaxe concreta é construída com a definição das figuras gráficas e da caixa de menu da DSL e com a junção desses itens com o metamodelo. Como resultado desse processo, o GMF gera uma ferramenta, acoplada ao Eclipse IDE, para a criação de modelos com base na DSL construída (GRONBACK, 2009). Maiores detalhes a respeito do uso e funcionamento do GMF são apresentados no Apêndice A.

2.4.2 Java Emitter Templates

Java Emitter Templates (JET) é uma linguagem de transformação M2T fornecida pelo Eclipse IDE para a construção de templates (THE ECLIPSE FOUNDATION, 2013c). JET é semelhante à linguagem *JavaServer Pages* (JSP) (ORACLE, 2013d), pois seus templates consistem de um código XML que pode conter instruções Java e, ao serem compilados, são traduzidos para arquivos *bytecode* de Java (*.class*). A junção desses arquivos *bytecode* com arquivos da biblioteca do JET resulta em uma ferramenta de transformação M2T específica de domínio que funciona como um *plug-in* do Eclipse IDE e é capaz de acessar informações

armazenadas em arquivos XML para originar arquivos de texto, cujo formato é definido nos templates (GRONBACK, 2009).

Na Figura 2.11 é mostrado um exemplo de arquivo XML, cujo acesso pode ser por um gerador M2T construído com o uso de JET.

```
<?xml version="1.0" encoding="utf-8"?>
<AppVendas xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="dc.ufscar.br/gdms/grenj"
  xsi:schemaLocation="dc.ufscar.br/gdms/grenj/appvenda.ecore">
  <venda id="XYZ001" cliente="//@cliente.0"
    produtos="//@produto.0//@produto.1//@produto.2"/>
  <venda id="XYZ002" cliente="//@cliente.1"
    produtos="//@produto.1//@produto.2"/>
  <cliente nome="JOAO" endereco="ENDERECO DE JOAO"/>
  <cliente nome="MARIA" endereco="ENDERECO DE MARIA"/>
  <produto descricao="LEITE" preco="25.0"/>
  <produto descricao="CAFE" preco="12.5"/>
  <produto descricao="PAO" preco="35.0"/>
</AppVendas>
```

Figura 2.11. Exemplo de arquivo XML que armazena dados de vendas de produtos.

As partes fixas dos templates JET são formadas por texto e as partes variantes dos podem ser formadas por dois tipos de instruções: 1) *tags* XML definidas pela linguagem JET e que realizam comandos semelhantes aos de uma linguagem de programação, por exemplo, iterações, comandos de decisão, declaração e leitura de variáveis, etc.; e 2) *scriptlets*, que são instruções Java demarcadas pelos símbolos “<%” e “%>”. Instruções Java não demarcadas pelos símbolos “<%” e “%>” são consideradas como partes fixas do template e, portanto, não são interpretadas.

Na Figura 2.12.a é apresentado o template JET que, quando combinado com o arquivo XML mostrado na Figura 2.11, origina o arquivo de texto mostrado na Figura 2.12.b. As partes fixas desse template consistem de textos, destacados em negrito, e as partes variantes são formadas pelas *tags* do JET e instruções Java embutidas.

<pre> <%! private count = 0; %> <c:iterate select="/AppVenda/venda" var="venda"> Venda <c:get select="\$venda/@id"/> Cliente: <c:get select="\$venda/cliente/@nome"/> <c:iterate select="\$venda/produtos" var="produto"> Produto: <c:get select="\$produto/@descricao"/> </c:iterate> </c:iterate> Nro. Total de Vendas: <%= ++count %> </pre>	a
<pre> Venda XYZ001 Cliente: JOAO Produto: LEITE Produto: CAFE Produto: PAO Venda XYZ002 Cliente: MARIA Produto: CAFE Produto: PAO Venda XYZ003 Cliente: MARIA Produto: LEITE Produto: PAO Nro. Total de Vendas: 3 </pre>	b

Figura 2.12. Um (a) template JET e (b) um arquivo de texto gerado a partir dele.

2.5 Linhas de Produto de Software

Algumas empresas de software se especializam em um determinado nicho de mercado, desenvolvendo diversos produtos específicos de um domínio. Essas empresas criam um ambiente análogo aos das linhas de produção da indústria, no qual softwares que compartilham diversas características em comum são construídos de forma mais eficiente. Esse tipo de ambiente compõe uma Linha de Produtos de Software (LPS) (MOHAMED-ALI e MOAWAD, 2010; POHL et al., 2005; CLEMENTS e NORTHROP, 2001).

Diversas abordagens de LPS podem ser encontradas na literatura, como *Application-based DDomain Modeling* (ADOM) (KRAMER e STURM, 2010), *Product Line UML-based Software engineering* (PLUS) (GOMAA, 2004), *Product Line Software Engineering* (PuLSE) (BAYER et al., 1999) e *Family-oriented Abstraction, Specification, and Translation* (FAST) (WEISS e LAI, 1999). Apesar das diferenças na forma como tratam o problema, em geral, as abordagens de LPS seguem o modelo de processo mostrado na Figura 2.13, no qual são identificados dois ciclos (SCHMID e ALMEIDA, 2013; CZARNECKI, 2005): **Engenharia do Domínio**, em que ocorre a definição do domínio e a construção dos recursos (DSLs, ferramentas, softwares reutilizáveis e documentos) da LPS; e **Engenharia da Aplicação**, na qual os produtos são desenvolvidos com o apoio dos recursos da LPS.

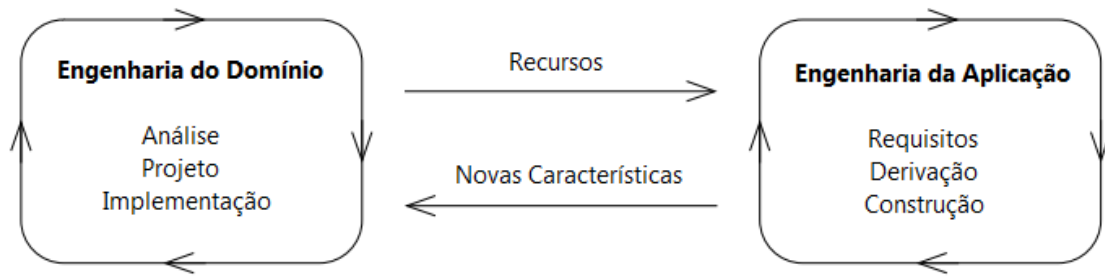


Figura 2.13. Modelo de processo de uma LPS. Adaptado de Czarnecki (2005).

De maneira similar ao processo de desenvolvimento de um único software, a engenharia de domínio também é dividida em análise, projeto e implementação, porém, com enfoque em um conjunto de softwares (RAMOS et al., 2013; PEROVICH et al., 2009; CZARNECKI, 2005). Na etapa de análise da engenharia de domínio ocorre a definição das características que compõem o domínio da LPS. Uma característica é uma propriedade que agrega valor aos usuários de um software e são documentadas em um modelo de características (*Feature Model*) (JÉZÉQUEL, 2012; SEPÚLVEDA et al., 2012; BENAVIDES et al., 2010; KANG et al., 1990) do domínio. Nesse tipo de modelo, as características são organizadas no formato de árvore, sendo que a raiz representa os produtos da LPS. As características são classificadas como obrigatórias ou opcionais e podem ter variantes (CZARNECKI e WASOWSKI, 2007).

Na Figura 2.14 é mostrado um modelo de características no qual está representada uma adaptação do domínio de veículos autônomos do kit Lego Mindstorms® NXT 2.0³. Nesse domínio um veículo (*Vehicle*) é decomposto em quatro características: duas opcionais, sensor (*Sensor*) e braços (*Arms*); e duas obrigatórias, motor (*Engine*) e dispositivo de locomoção (*Locomotion Device*). O veículo pode ter até três sensores: infravermelho (*Infrared Sensor*), sensor de luz (*Light Sensor*) e sensor de toque (*Touch Sensor*). Além disso, o veículo poder usar um dos dois tipos de dispositivo de locomoção: rodas (*Wheels*) ou esteira (*Caterpillar*).

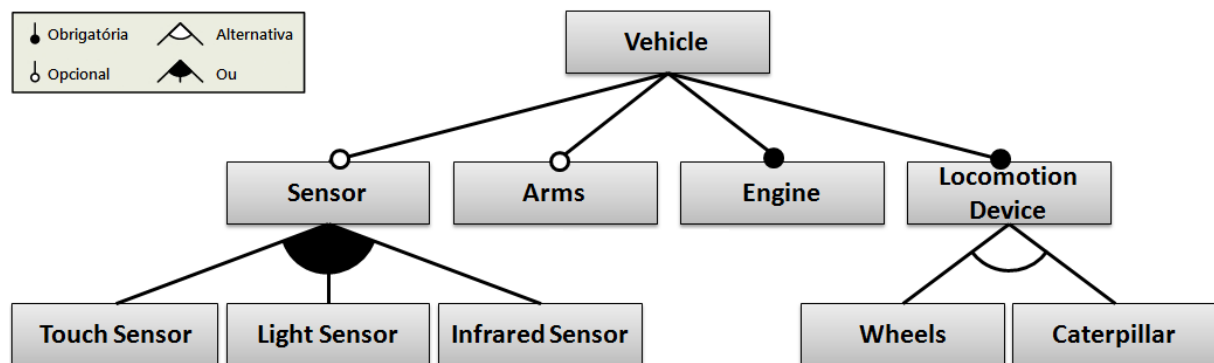


Figura 2.14. Modelo de características do domínio de veículos autônomos.

³ <http://mindstorms.lego.com/en-us/products/default.aspx#t>

Na etapa de projeto são definidos os recursos reutilizáveis com base nas características do domínio. Normalmente, a arquitetura comum a todos os produtos da LPS é representada por um framework (STURM e KRAMER, 2010; KIM et al., 2004; WEISS e LAI, 1999). Também podem ser criados metamodelos para compor o escopo de DSLs e geradores. Além disso, outros recursos pré-existentes, específicos ou não de domínio, podem ser incorporados a LPS, como por exemplo, repositórios, ferramentas de desenvolvimento e frameworks de *middleware*. Por fim, na etapa de implementação, todos os recursos reutilizáveis específicos da LPS são construídos.

A engenharia de aplicação é composta pelas etapas de requisitos, derivação e construção (SCHMID e ALMEIDA, 2013; CZARNECKI, 2005). Na etapa de requisitos um subconjunto das características do domínio e suas variantes são selecionados com base nos requisitos do produto em desenvolvimento. Características específicas do produto também podem ser acrescentadas. Assim, o modelo do produto com base na DSL da LPS é criado como uma instância do modelo de características do domínio. Na etapa de derivação, cria-se um modelo que considera os recursos que serão reutilizados para desenvolver o produto. Por fim, na etapa de construção ocorre a construção do produto com o reuso dos recursos da LPS e com a criação das suas partes específicas.

Como também é mostrado na Figura 2.13, existe uma troca bidirecional de informações/recursos entre ambos os ciclos. A engenharia de domínio supre a engenharia de aplicação com os recursos reutilizáveis da LPS, enquanto que a engenharia de aplicação fornece um *feedback* sobre novas características que podem ser incorporadas ao domínio. Portanto, o domínio de uma LPS pode evoluir a medida que novos produtos são desenvolvidos (RAMOS et al., 2013; CZARNECKI, 2005).

Na literatura são encontradas diversas ferramentas que apoiam o desenvolvimento de LPSs (PEREIRA et al., 2013). Na Subseção 2.5.1 é apresentada a ferramenta Pure:variants, que apoia todas as etapas da Engenharia de Domínio e Engenharia de Aplicação.

2.5.1 Pure::variants

Pure:variants (PURE SYSTEMS, 2013) é uma ferramenta de uso proprietário criada com base no Eclipse IDE, que permite a criação de LPSs a partir de uma aplicação base. Os produtos desenvolvidos com a LPS são variantes dessa aplicação base, formados por um subconjunto das suas características.

A interface da Pure:variants é apresentada na Figura 2.15. As características são organizadas em uma notação de árvore, mostrada no centro da figura. No modelo à direita é possível definir a arquitetura da LPS por meio de uma modelagem dos componentes que

implementam cada característica do domínio. Caixas de seleção no modelo de características permitem selecionar as características que se aplicam a um produto e gerar o seu código-fonte com base no modelo de componentes. Um manual da ferramenta Pure::variants para construção de uma LPS a partir do código-fonte de uma aplicação é apresentado no Apêndice B.

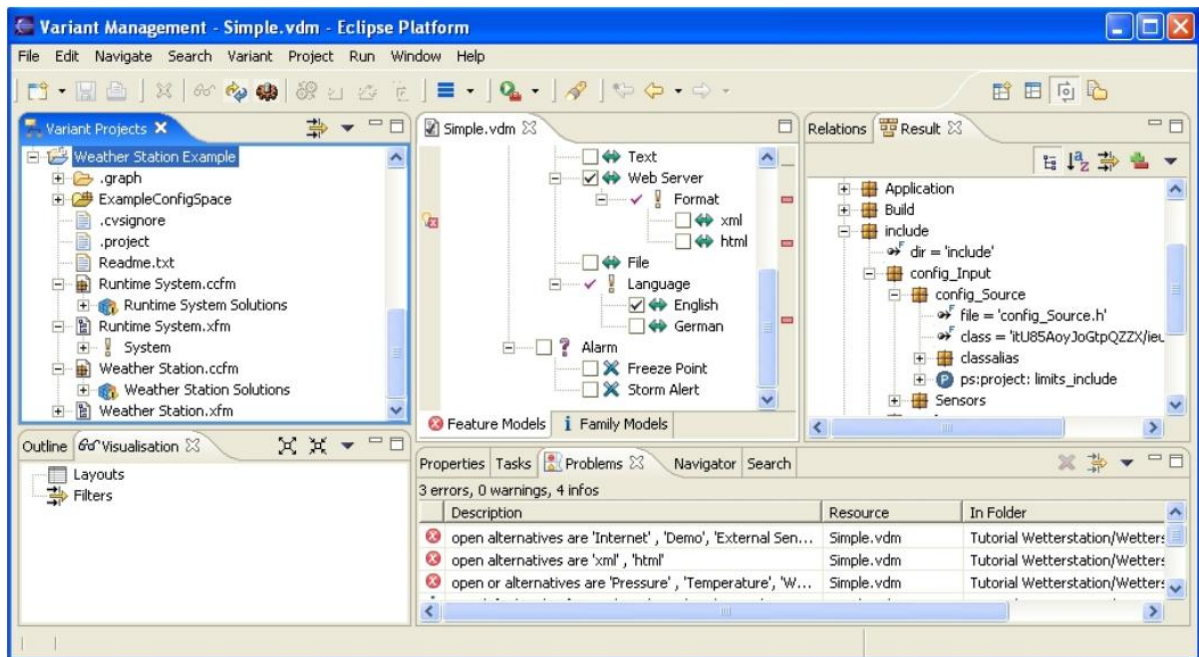


Figura 2.15. Interface da ferramenta Pure::variants (Pure, 2013).

2.6 Considerações Finais

Neste capítulo foram apresentados os conceitos aplicados na tese de doutorado proposta. Esses conceitos proporcionam diferentes formas de reúso, que vão além daquelas fornecidas pelas linguagens de programação. Além disso, seu emprego aperfeiçoa os processos de desenvolvimento de software, reduzindo o tempo e o custo de desenvolvimento, aumentando a qualidade no produto gerado, entre outras vantagens.

LPS constituem um ambiente de desenvolvimento de software construído a partir do gerenciamento das características comuns e variantes desses softwares. Os seus conceitos e regras são úteis a empresas que desenvolvem diversos softwares semelhantes.

MDE promove o uso de DSLs para facilitar a modelagem das aplicações e de ferramentas de transformação para reduzir o tempo gasto e as complexidades na implementação do código-fonte. As ferramentas de transformação podem ser configuradas

por meio de templates, expandindo tanto o escopo de domínios que podem ser contemplados quanto à diversidade de tipos de artefatos que podem ser gerados.

O desenvolvimento de software pode ser facilitado com o reuso de frameworks. Contudo, o desenvolvimento de um framework que atenda ao domínio do negócio de uma empresa não é um processo trivial, pois exige o uso de técnicas e recursos que permitam a implementação das variabilidades do domínio. Portanto, a abordagem proposta nesta tese de doutorado fornece meios para facilitar o desenvolvimento e o reuso de frameworks no desenvolvimento de software.

Além dos frameworks, padrões também apoiam o desenvolvimento de software em diferentes níveis de abstração. Padrões não são regras rígidas, mas ferramentas que podem ser utilizadas para resolver problemas recorrentes em diversas situações. É necessário identificar o problema e adaptar o uso do padrão correto conforme a necessidade da aplicação. Nesta tese, padrões apoiam a construção de frameworks a partir de modelos de características.

No Capítulo 3 é apresentada uma abordagem para facilitar o desenvolvimento de frameworks e no Capítulo 4 é apresentada uma abordagem de construção de DSLs que reduzem as dificuldades encontradas durante a instanciação de frameworks.

Capítulo 3

UMA ABORDAGEM PARA FACILITAR O DESENVOLVIMENTO DE FRAMEWORKS

3.1 Considerações Iniciais

No Capítulo 1 foi comentado que o reuso é uma das principais práticas para se obter maior eficiência nos processos de desenvolvimento de software e na qualidade dos artefatos produzidos (BAUER, 2013; SHIVA e SHALA, 2007). Nesse contexto, os frameworks são alguns dos artefatos de desenvolvimentos mais utilizados com o intuito de obter reuso de projeto e de código (KIRK et al., 2007; FAYAD e JOHNSON, 1999).

Um framework pode ser utilizado com o propósito de servir como a arquitetura básica de uma LPS, quando se deseja desenvolver diversas aplicações pertencentes ao mesmo domínio ou como um facilitador para a implementação de alguma funcionalidade computacional recorrente, tal como o controle de acesso aos usuários (ABI-ANTOUN, 2007; XU e BUTLER, 2006). Em ambos os casos, pode não ser possível obter um framework com as características desejadas a partir de terceiros, sendo, portanto, necessário desenvolvê-lo.

No Capítulo 2, Seção 2.3, foram citadas algumas das dificuldades quanto ao desenvolvimento de um framework, entre as quais, a necessidade de um projeto/código adaptável a diversas aplicações em um domínio, a implementação de classes e operações com comportamentos variável e o uso de técnicas avançadas de programação (*reflection*, genéricos, etc.) (OKANOVIC e MATELJAN, 2011; ZHANG et al., 2009). Alguns autores propuseram o uso de padrões de projeto conhecidos da literatura para apoiar o desenvolvimento de frameworks e outros artefatos reutilizáveis que implementam características de domínios (STANOJEVIC et al., 2011; LUCRÉDIO et al., 2010; LEE e KANG, 2004; WU-DONG et al., 2003; SRINIVASAN, 1999). Contudo, esses padrões não

são específicos para o desenvolvimento de frameworks, exigindo habilidade de adaptação por parte dos desenvolvedores.

Com o intuito de reduzir as dificuldades de desenvolvimento de frameworks, neste capítulo é apresentada a abordagem ***From Features to Frameworks (F3)***, com a qual o desenvolvimento de um framework deve ser realizado em duas etapas: Modelagem do Domínio, em que as características do domínio do framework são definidas; e Construção do Framework, em que uma linguagem de padrões auxilia o projeto a implementação de um framework caixa branca de acordo com as características do domínio modelado (VIANA et al., 2013b; VIANA et al., 2013c).

Com o uso da abordagem F3, a definição do domínio e da funcionalidade do framework ocorre em nível de modelagem, enquanto que as complexidades do projeto e a implementação do framework são amenizadas por uma linguagem de padrões específicos para esse propósito.

Além desta seção, este capítulo é composto por mais três seções: na Seção 3.2 é apresentada a abordagem F3; na Seção 3.3 a abordagem F3 é exemplificada para a construção de framework para o domínio de transações de aluguel e comercialização de recursos; e na Seção 3.4 são apresentadas as considerações finais deste capítulo.

3.2 A Abordagem From Features to Frameworks (F3)

A abordagem F3 tem o objetivo de facilitar o desenvolvimento de frameworks. Como frameworks são sistemas de software específicos de domínio com o propósito de serem reutilizados no desenvolvimento de aplicações, pode-se considerar a F3 como uma abordagem de engenharia de domínio. As duas etapas dessa abordagem estão ilustradas na Figura 3.1. e são descritas em detalhes nas Seções 3.2.1 e 3.2.2.

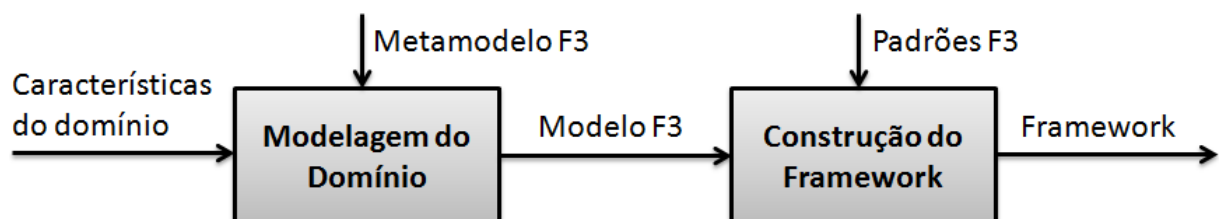


Figura 3.1. Etapas da abordagem F3.

3.2.1 Modelagem do Domínio

A primeira etapa da abordagem F3, Modelagem de Domínio, tem como objetivo definir as características que compõem o domínio do framework. O resultado dessa etapa é um **modelo F3** criado com base nessas características.

Modelos F3 são uma extensão dos modelos de características que incluem propriedades existentes em metalinguagens discutidas na Seção 2.4. Modelos de características convencionais são muito abstratos e metalinguagens, tal como a *Ecore* do *Eclipse Modeling Framework* (EMF) (THE ECLIPSE FOUNDATION, 2013; GRONBACK, 2009), são genéricas demais, de modo que algumas restrições entre os elementos teriam que ser definidas pelos desenvolvedores. BENAVIDES et al. (2010) comentam sobre versões extendidas de modelos de características. O modelo F3 foi criado com o intuito modelar informações suficientes para o desenvolvimento de frameworks. Na Figura 3.2 é mostrada a relação entre modelos F3, modelos de características e metalinguagens.

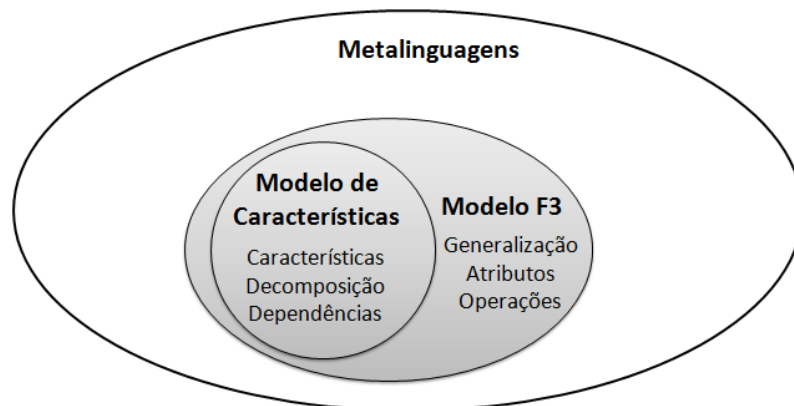


Figura 3.2. Modelos F3 são uma extensão dos modelos de características.

Como nos modelos de características convencionais, as características nos modelos F3 são organizadas em uma estrutura de árvore. Contudo, modelos F3 não constituem, necessariamente, uma árvore, pois uma característica pode se relacionar com outra no mesmo nível, consigo mesma ou com outra em um nível superior. Além disso, a notação gráfica dos modelos F3 é similar a dos modelos de classe da UML. Essa notação foi adotada para permitir que os modelos F3 possam ser criados com o apoio de ferramentas UML. Os elementos e relacionamentos que podem ser definidos nos modelos F3 são:

- **Característica:** elemento que representa uma entidade abstrata ou concreta do domínio. Deve ter um nome e pode conter atributos e operações.
- **Decomposição:** relacionamento que indica que uma característica é composta por outra. Sua multiplicidade mínima indica se a característica alvo do relacionamento é opcional (0) ou obrigatória (1) para a característica de origem. Já sua multiplicidade máxima indica quantas instâncias da característica alvo podem se associar com cada instância da característica de origem. Em frameworks caixa branca, uma instância de uma característica é uma classe da aplicação que estende a principal classe do framework que implementa essa característica. A multiplicidade máxima pode ser: simples (1), para uma única instância da característica; múltipla (*), para várias ocorrências da mesma instância; e composta (**), para diferentes instâncias.

- **Composição:** variante da decomposição em que a característica alvo é parte da característica de origem do relacionamento.
- **Generalização:** relacionamento em que uma característica é uma variação de outra.
- **Dependência:** relacionamento que define uma condição para a instanciação de um característica, podendo ser de dois tipos:
 - **Requerente (requires),** quando uma instância da característica de origem do relacionamento pode ser criada em uma aplicação somente se a característica alvo também for;
 - **Excludente (excludes),** quando uma instância da característica de origem pode ser criada em uma aplicação se e somente se não houver nenhuma instância da característica alvo.

Um exemplo de modelo F3 para o domínio de aplicações que gerenciam dispositivos de carros é mostrado na Figura 3.3. A característica Carro é a principal do domínio, portanto, a raiz da árvore. As características Motor, Cambio e Freios são obrigatórias em relação a Carro, pois a multiplicidade mínima da decomposição é 1. Acessorio é opcional porque multiplicidade mínima da decomposição que aponta para essa característica é 0. Cambio possui duas variantes, Automatico e Manual, assim como Acessorio, que possui as variantes ArCondicionado, Alarme e Trava. O modelo define também que um Carro pode ter várias instâncias de Acessorio (multiplicidade máxima *), portanto, mais de uma variante dessa característica pode ser utilizada. Contudo, como a multiplicidade máxima de Cambio é 1, apenas uma de suas variantes pode ser utilizada. Quando Alarme é escolhida, Trava também deve ser porque um alarme necessita (requires) de uma trava para funcionar.

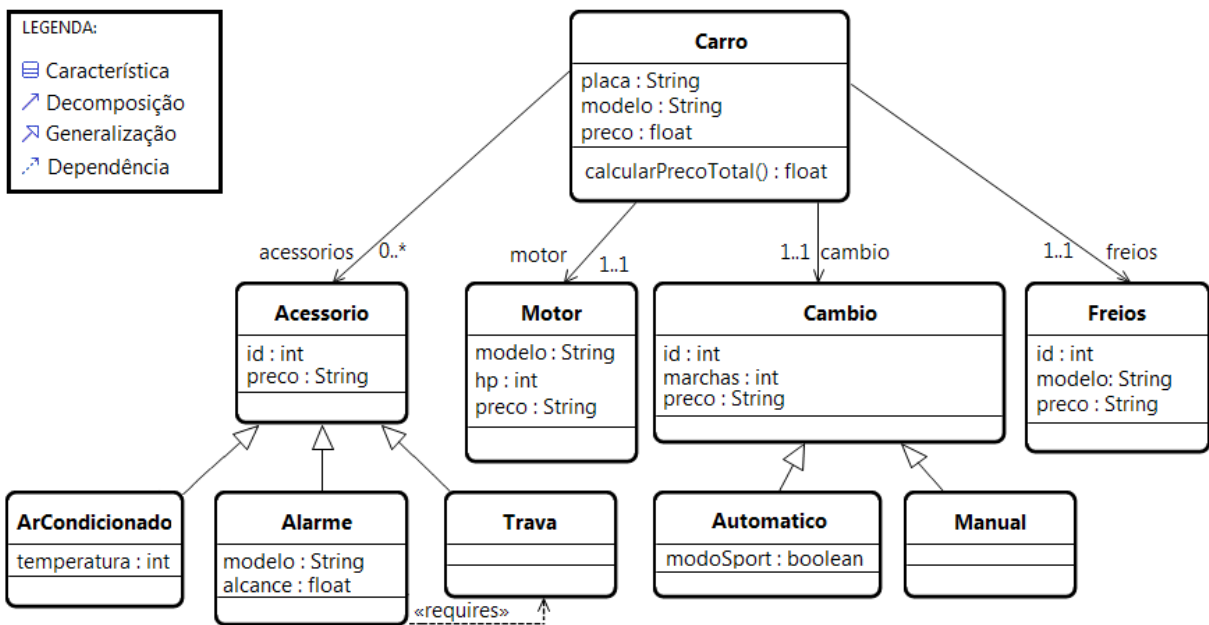


Figura 3.3. Modelo F3 para o domínio de aplicações de dispositivos de carros.

3.2.2 Construção do Framework

A etapa de Construção do Framework tem o objetivo de projetar e implementar um framework caixa branca a partir do modelo F3 criado na etapa anterior. Essa etapa é realizada com o apoio da **Linguagem de Padrões F3 (LPF3)**, que guia os desenvolvedores indicando quais unidades de código devem ser criadas no framework a partir dos elementos e relacionamentos encontrados nos modelos F3.

A LPF3 foi criada com base em estudos realizados em frameworks existentes, em especial, os frameworks GRENJ (VIANA, 2009; DURELLI, 2008), Hibernate (JBoss COMMUNITY, 2013) e JHotDraw (JHOTDRAW.ORG, 2013) e em padrões de projeto, como *Abstract Factory* e *Data Access Object (DAO)* (FREEMAN et al., 2004; GRAND, 2001; GAMMA et al., 1995). Os padrões da LPF3 estão documentados da seguinte forma:

- **Nome:** identifica o padrão e sumariza o seu propósito;
- **Classificação:** agrupa o padrão de acordo com o propósito das unidades de código construídas a partir dele. Os padrões F3 são classificados em dois grupos: **estrutural**, quando estão relacionados com a estrutura básica do framework; **interface**, quando estão relacionados com a comunicação entre o framework e as aplicações que o reutilizam; e **persistência**, quando estão relacionados com a persistência de dados;
- **Padrões relacionados:** indica outros padrões da LPF3 cujo cenário envolve elementos semelhantes ao do padrão em questão;
- **Contexto:** descreve uma função do framework com a qual o padrão está relacionado;
- **Problema:** descreve um comportamento desejado para o framework;
- **Propósito:** descreve o que o padrão faz;
- **Cenário:** descreve uma combinação de características e relacionamentos em um modelo F3 que implica no uso do padrão;
- **Solução:** indica as unidades de código que devem ser criadas para implementar o cenário relacionado com o padrão;
- **Modelos:** mostra uma representação gráfica genérica do cenário e outra da solução;
- **Implementação:** exibe um fragmento de código, em uma linguagem de programação, que ilustra como a solução deve ser implementada.

No Quadro 3.1 estão listados os padrões da LPF3. Esses padrões se baseiam na programação orientada a objetos e sua documentação contém exemplos de implementação em linguagem Java. Contudo, é possível adaptar a implementação dos padrões da LPF3 para outras linguagens orientadas a objetos.

Quadro 3.1. Padrões da linguagem de padrões F3.

Padrões F3		Classificação
#	Nome	
P01	Fábrica de Características	Estrutural
P02	Característica de Domínio	
P03	Variante de Característica	
P04	Atributo de Característica	
P05	Operação de Característica	
P06	Modularização Hierárquica	
P07	Decomposição Simples	
P08	Decomposição Múltipla	
P09	Decomposição Opcional	Interface
P10	Decomposição Obrigatória	
P11	Decomposição Composta	
P12	Característica Requerente	Persistência
P13	Variante Requerente	
P14	Estratégia de Persistência	
P15	Característica Persistente	
P16	Fábrica de DAO	
P17	DAO de Característica	

A ordem de aplicação dos padrões da LPF3 é o apresentado pelo fluxo de execução da Figura 3.4. Contudo, ressalta-se que muitos desses padrões não são obrigatórios, pois são aplicados somente quando o cenário descrito por eles é encontrado em um modelo F3. A descrição completa de cada padrão é apresentada em seguida.

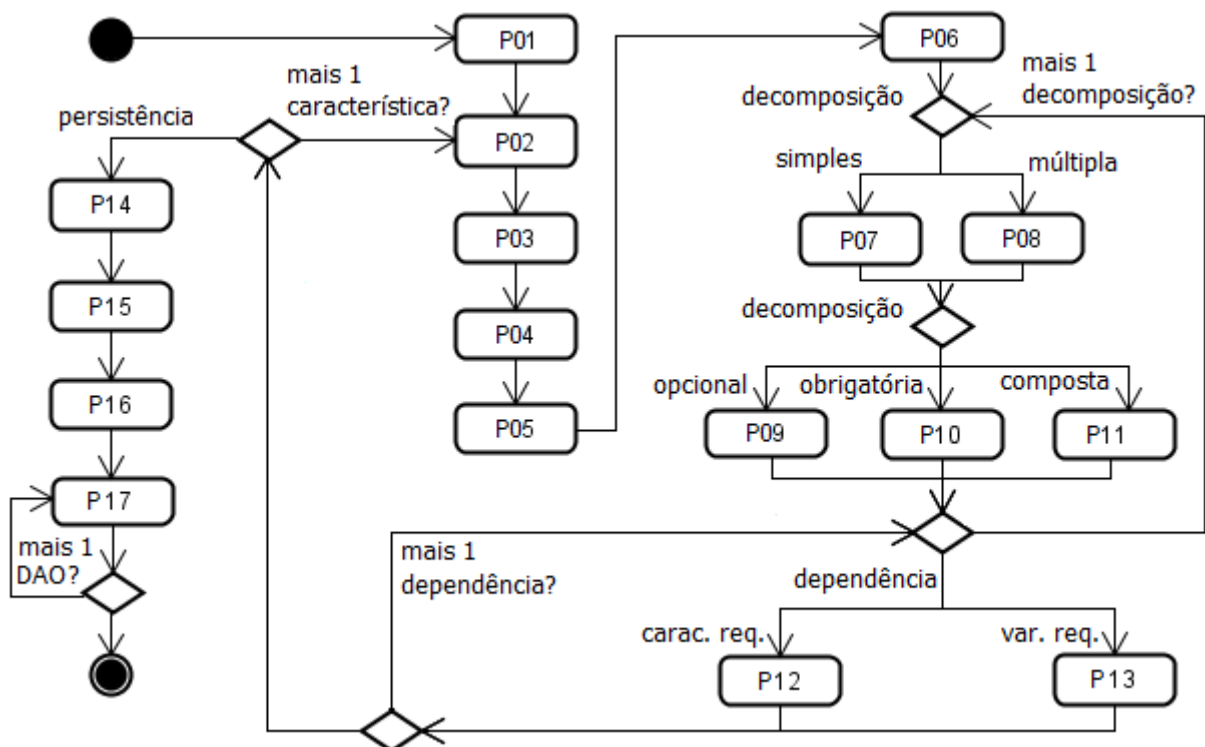


Figura 3.4. Fluxo de execução dos padrões da LPF3.

Padrão 01: Fábrica de Características

Classificação: Estrutural.

Padrões relacionados: Estratégia de Persistência (P14), Característica Persistente (P15) e Fábrica de DAO (P16).

Contexto: criação de objetos das classes das aplicações que reutilizam o framework.

Problema: o framework deve fornecer um meio comum para a criação de objetos das classes que representam as características do seu domínio.

Propósito: indica quais unidades de código devem ser construídas para facilitar a criação de objetos das classes das características.

Cenário: um domínio é formado por um conjunto de características.

Solução: deve ser criada uma classe abstrata que é estendida, direta ou indiretamente, por todas as classes das características. Opcionalmente, nessa classe podem ser incluídos atributos e operações comuns a todas as características. Também deve ser criada uma classe abstrata que implementa a fábrica de objetos das classes das características. Nas aplicações uma classe concreta estende a fábrica do framework para retornar objetos das classes específicas dessas aplicações.

Modelo: Na Figura 3.5 é mostrado a) o cenário do padrão e b) do projeto da solução.

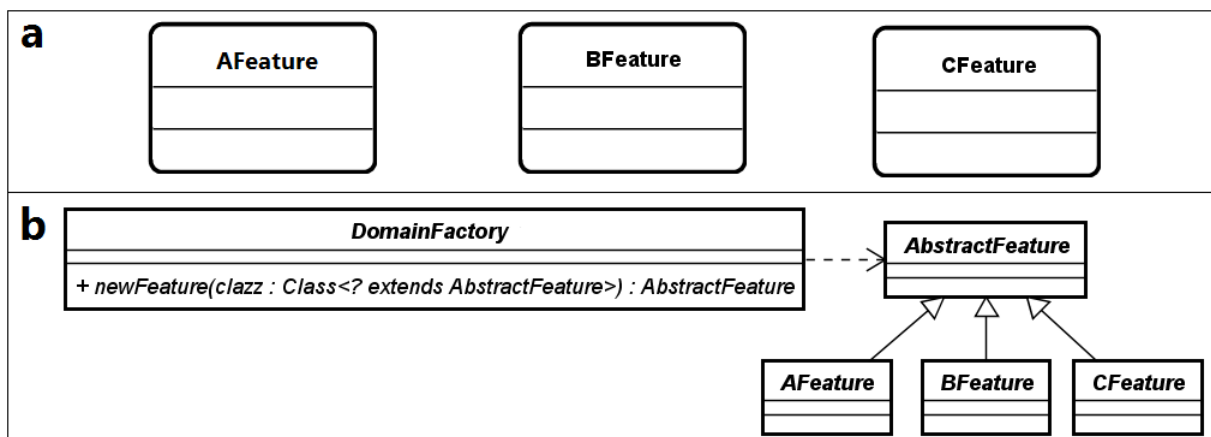


Figura 3.5. Modelos a) do cenário do P01 e b) do projeto da solução.

Implementação: solução em linguagem Java.

```

public abstract class AbstractFeature {

    //atributos (opcional)
    //Caso tenha atributos, incluir um construtor e getters/setters
    //operações (opcional)
}

public abstract DomainFactory {

    public abstract AbstractFeature
        newFeature(Class<? extends AbstractFeature> clazz);

}
  
```


Padrão 2: Característica de Domínio

Classificação: estrutural.

Padrões relacionados: Atributo de Característica (P04), Operação de Característica (P05) e DAO de Característica (P17).

Contexto: implementação das características do framework.

Problema: as classes do framework devem implementar as características do seu domínio.

Propósito: indica quais unidades de código devem ser construídas para uma característica.

Cenário: uma característica foi encontrada no modelo.

Solução: deve ser criada uma classe abstrata correspondente à característica encontrada. Essa classe deve estender `AbstractFeature` e possuir um construtor sem argumentos.

Modelo: Na Figura 3.6 é mostrado a) o cenário do padrão e b) do projeto da solução.

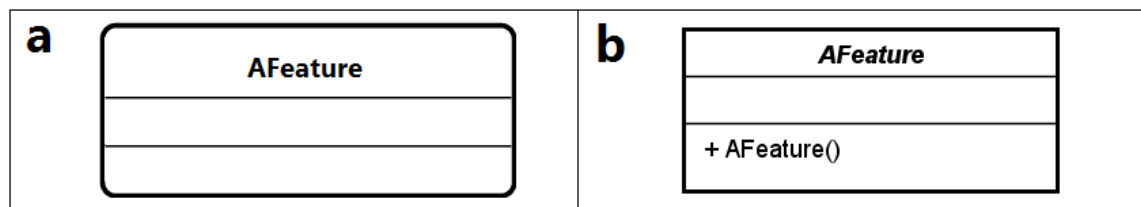


Figura 3.6. Modelos a) do cenário do P02 e b) do projeto da solução.

Implementação: solução em linguagem Java.

```
public abstract class AFeature extends AbstractFeature {

    public AFeature() {
        super();
    }

}
```

Padrão 3: Variante de Característica

Classificação: estrutural.

Padrões relacionados: Atributo de Característica (P04), Operação de Característica (P05) e DAO de Característica (P17).

Contexto: implementação das variantes das características.

Problema: uma característica do framework pode ter variantes com atributos, operações e comportamentos diferentes.

Propósito: indica quais unidades de código devem ser construídas quando uma característica é uma generalização de uma ou mais variantes.

Cenário: uma característica é alvo de relacionamentos de generalização.

Solução: deve ser criada uma classe abstrata para a característica generalizada e outra para cada uma de suas variantes. Todas as classes devem ter seus próprios construtores, com os construtores das variantes invocando o construtor da característica generalizada.

Modelo: Na Figura 3.7 é mostrado a) o cenário do padrão e b) do projeto da solução.

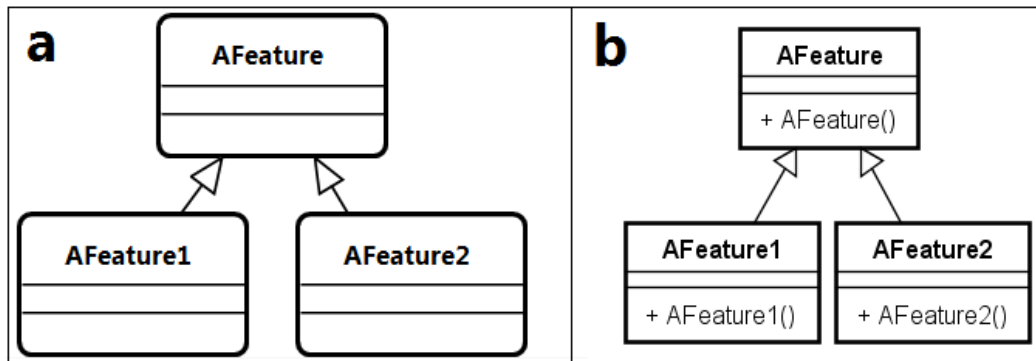


Figura 3.7. Modelos a) do cenário do P03 e b) do projeto da solução.

Implementação: solução em linguagem Java.

```

public abstract class AFeature extends AsbtractFeature {

    public Afeature() {
        super();
    }

}

public abstract class AFeature1 extends AFeature {

    public Afeature1() {
        super();
    }

}

public abstract class AFeature2 extends AFeature {

    public Afeature2() {
        super();
    }

}
  
```

Padrão 4: Atributo de Característica

Classificação: estrutural.

Padrões relacionados: Característica de Domínio (P02) e Variante de Característica (P03).

Contexto: implementação dos atributos das características.

Problema: uma classe que representa uma característica deve conter os atributos definidos para essa característica no modelo do domínio.

Propósito: indica quais unidades de código devem ser construídas para um atributo.

Cenário: um atributo foi encontrado em uma característica do domínio.

Solução: o atributo deve ser implementado na classe da característica. Operações *getter/setter* também devem ser criadas para que o atributo possa ser acessado. O construtor vazio da classe deve inicializar o atributo com um valor *default*.

Modelo: Na Figura 3.8 é mostrado a) o cenário do padrão e b) do projeto da solução.

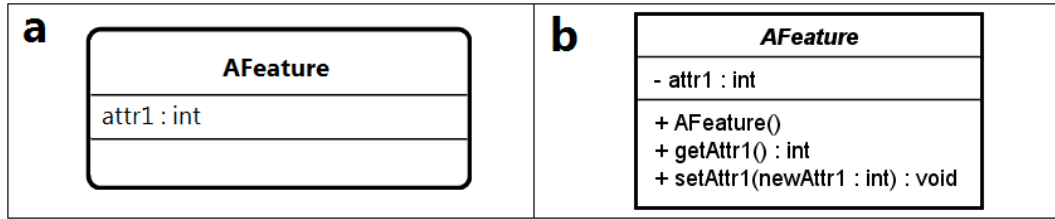


Figura 3.8. Modelos a) do cenário do P04 e b) do projeto da solução.

Implementação: solução em linguagem Java.

```
public abstract class AFeature extends AbstractFeature {

    private int attr1;

    public AFeature() {
        super();
        attr1 = 0;
    }

    public int getAttr1() {
        return attr1;
    }

    public void setAttr1(int newAttr1) {
        attr1 = newAttr1;
    }
}
```

Padrão 5: Operação de Característica

Classificação: estrutural.

Padrões relacionados: Característica de Domínio (P02) e Variante de Característica (P03).

Contexto: implementação das operações das características.

Problema: uma classe que representa uma característica deve conter as operações definidas para essa característica no modelo do domínio.

Propósito: indica quais unidades de código devem ser construídas para uma operação da característica.

Cenário: uma operação foi encontrada em uma característica do domínio.

Solução: a operação deve ser implementada na classe correspondente da característica.

Modelo: Na Figura 3.9 é mostrado a) o cenário do padrão e b) do projeto da solução.

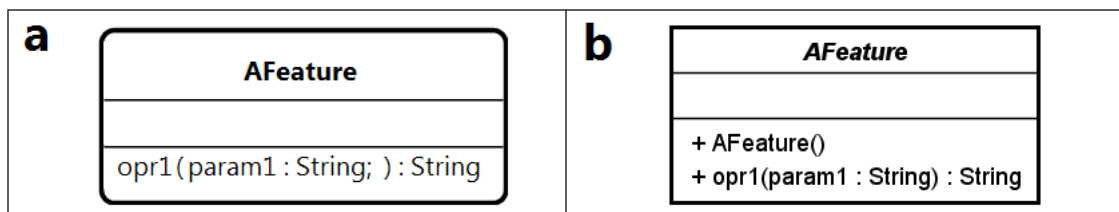


Figura 3.9. Modelos a) do cenário do P05 e b) do projeto da solução.

Implementação: solução em linguagem Java.

```
public abstract class AFeature extends AbstractFeature {

    public AFeature() {
        super();
    }

    public String opr1(String param1) {
        //código interno da operação.
    }

}
```

Padrão 6: Modularização Hierárquica

Classificação: estrutural.

Padrões relacionados: Característica de Domínio (P02) e DAO de Característica (P17).

Contexto: hierarquia das classes que compõem as características.

Problema: os atributos e as operações recorrentes devem estar modularizados em classes do framework que representam generalizações abstratas.

Propósito: indica quais unidades de código devem ser construídas quando diversas características possuem atributos e/ou operações em comum. Em geral, esse padrão deve ser usado para atributos e/ou operações com maior relevância em relação ao domínio.

Cenário: diversas características no modelo possuem atributos e/ou operações em comum.

Solução: crie uma classe abstrata e desloque os atributos e as operações semelhantes para ela. Essa classe é estendida pelas classes originais dos atributos e das operações. Desloque também as operações *getter/setter* dos atributos deslocados. Operações com trechos de código semelhantes também podem ser generalizadas.

Modelo: Na Figura 3.10 é mostrado a) o cenário do padrão e b) do projeto da solução.

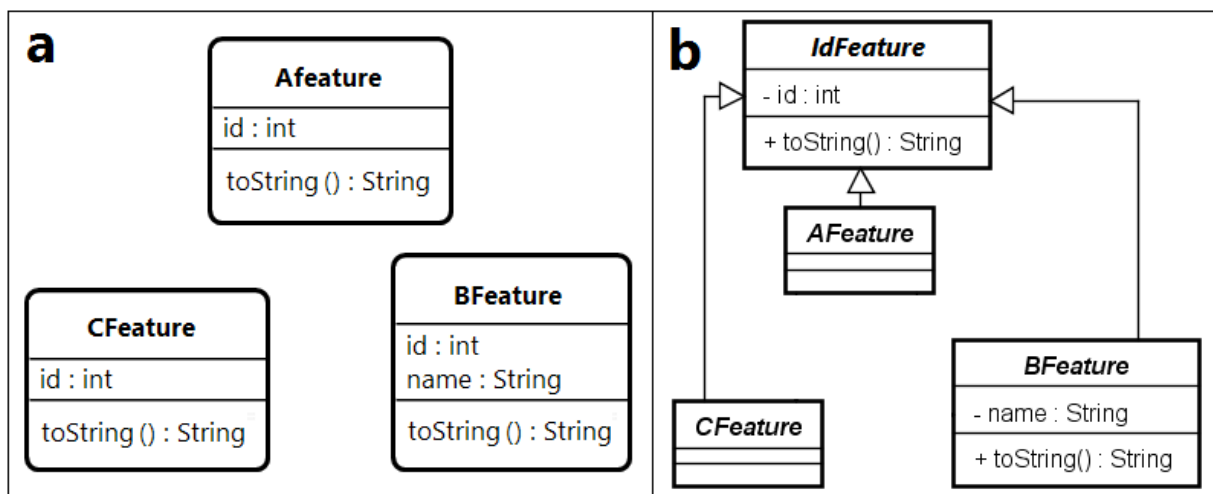


Figura 3.10. Modelo a) do cenário do P06 e b) do projeto da solução.

Implementação: solução em linguagem Java.

```
public abstract class IdFeature extends AbstractFeature {

    private int id;

    public IdFeature() {
        super();
        id = 0;
    }

    public int getId() {
        return id;
    }

    public void setId(int newId) {
        id = newId;
    }

    public String toString() {
        return "\nID: " + id;
    }

}

public abstract class AFeature extends IdFeature {

    public AFeature() {
        super();
    }

}

public abstract class BFeature extends IdFeature {

    private String name;

    public BFeature() {
        super();
        name = null;
    }

    public String getName() {
        return name;
    }

    public void setName(int newName) {
        name = newName;
    }

    public String toString() {
        return super.toString() + "\nNAME: " + name;
    }

}

public abstract class CFeature extends AbstractFeature {

    public CFeature() {
        super();
    }

}
```

Padrão 7: Decomposição Simples

Classificação: estrutural.

Padrões relacionados: Decomposição Opcional (P09) e Decomposição Obrigatória (P10).

Contexto: relacionamento de decomposição simples entre características.

Problema: um objeto da classe de uma característica pode se relacionar com um único objeto da classe de outra característica.

Propósito: indica quais unidades de código devem ser construídas quando uma característica se relaciona com outra por meio de uma decomposição simples.

Cenário: uma característica é a origem de um relacionamento de decomposição com uma característica alvo com multiplicidade máxima igual a 1 (um).

Solução: a classe correspondente à característica de origem do relacionamento de decomposição deve possuir um atributo do tipo da classe correspondente à característica alvo e as operações *getter/setter* desse atributo.

Modelo: Na Figura 3.11 é mostrado a) o cenário do padrão e b) do projeto da solução.

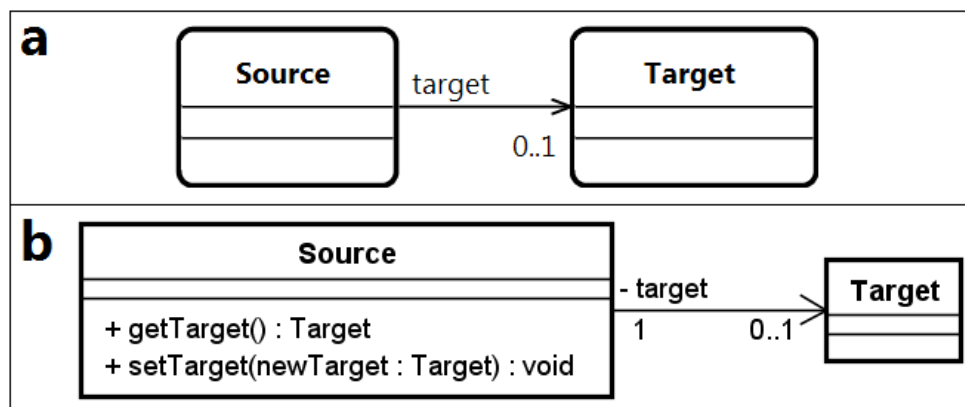


Figura 3.11. Modelos a) do cenário do P07 e b) do projeto da solução.

Implementação: solução em linguagem Java.

```

public abstract class Source extends AbstractFeature {

    private Target target;

    public Source() {
        super();
        target = null;
    }

    public void getTarget() {
        return target;
    }

    public void setTarget(Target newTarget) {
        target = newTarget;
    }

}
  
```

Padrão 8: Decomposição Múltipla

Classificação: estrutural.

Padrões relacionados: Decomposição Opcional (P09) e Decomposição Obrigatória (P10).

Contexto: relacionamento de decomposição múltipla entre características.

Problema: um objeto da classe de uma característica pode se relacionar com um conjunto de objetos da classe de outra característica.

Propósito: indica quais unidades de código devem ser construídas quando uma característica se relaciona com outra por meio de uma decomposição múltipla.

Cenário: uma característica é a origem de um relacionamento de decomposição com uma característica alvo com multiplicidade máxima igual a * (múltipla).

Solução: a classe correspondente à característica de origem do relacionamento de decomposição deve possuir um atributo que é uma lista para objetos da classe correspondente à característica alvo e as operações *getter/setter* desse atributo.

Modelo: Na Figura 3.12 é mostrado a) o cenário do padrão e b) do projeto da solução.

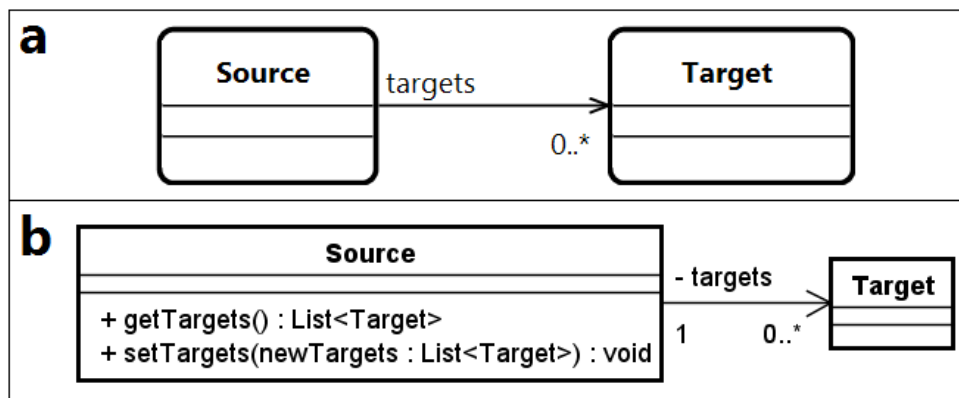


Figura 3.12. Modelos a) do cenário do P08 e b) do projeto da solução.

Implementação: solução em linguagem Java.

```
public abstract class Source extends AbstractFeature {

    private List<Target> targets;

    public Source() {
        super();
        targets = new ArrayList<Target>();
    }

    public void getTargets() {
        return targets;
    }

    public void setTargets(List<Target> newTargets) {
        targets = newTargets;
    }

}
```

Padrão 09: Decomposição Opcional

Classificação: interface.

Padrões relacionados: Decomposição Simples (P07), Decomposição Múltipla (P08) e Decomposição Composta (11).

Contexto: obrigatoriedade de uma característica.

Problema: uma característica é opcional em relação a outra.

Propósito: indica quais unidades de código devem ser construídas quando uma característica se relaciona com outra por meio de uma decomposição opcional.

Cenário: uma característica é a origem de um relacionamento de decomposição com uma característica alvo com multiplicidade mínima igual a 0 (zero).

Solução: a classe da característica de origem do relacionamento de decomposição deve possuir uma operação que indica a classe correspondente à característica alvo. O código dessa operação deve retornar nulo (`null`), pois a característica alvo não é obrigatória.

Modelo: Na Figura 3.13 é mostrado a) o cenário do padrão e b) do projeto da solução.

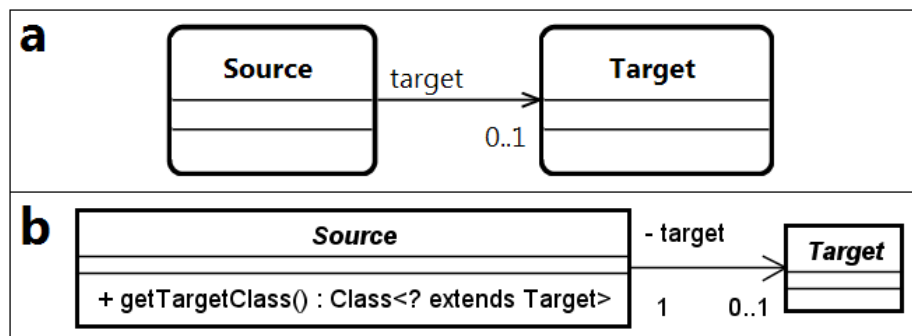


Figura 3.13. Modelos a) do cenário do P09 e b) do projeto da solução.

Implementação: solução em linguagem Java.

```

public abstract class Source extends AbstractFeature {

    public Class<? extends Target> getTargetClass() {
        return null;
    }

}
  
```

Padrão 10: Decomposição Obrigatória

Classificação: interface.

Padrões relacionados: Decomposição Simples (P07), Decomposição Múltipla (P08) e Decomposição Composta (11).

Contexto: obrigatoriedade de uma característica.

Problema: uma característica é obrigatória em relação a outra.

Propósito: indica quais unidades de código devem ser construídas quando uma característica se relaciona com outra por meio de uma decomposição obrigatória.

Cenário: uma característica é a origem de um relacionamento de decomposição com uma característica alvo com multiplicidade mínima igual a 1 (um).

Solução: a classe da característica de origem do relacionamento de decomposição deve possuir uma operação abstrata que retorna a classe da característica alvo.

Modelo: Na Figura 3.14 é mostrado a) o cenário do padrão e b) do projeto da solução.

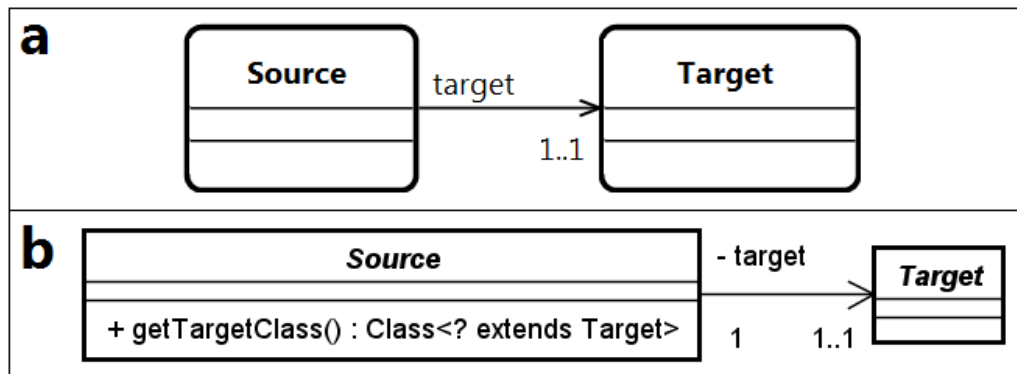


Figura 3.14. Modelos a) do cenário do P10 e b) do projeto da solução.

Implementação: solução em linguagem Java.

```
public abstract class Source extends AbstractFeature {
    public abstract Class<? extends Target> getTargetClass();
}
```

Padrão 11: Decomposição Composta

Classificação: interface.

Padrões relacionados: Decomposição Opcional (P09) e Decomposição Obrigatória (P10).

Contexto: diversas associações com a mesma característica.

Problema: uma característica possui um relacionamento de decomposição com outra característica que pode ocorrer mais de uma vez no modelo de uma aplicação.

Propósito: indica quais unidades de código devem ser construídas quando uma característica possui uma decomposição composta para outra característica. Essas unidades de código simulam associações entre a classe da característica de origem e uma ou mais instâncias da característica alvo.

Cenário: uma característica possui um relacionamento de decomposição composta.

Solução: a classe correspondente à característica de origem do relacionamento de decomposição deve possuir um atributo que é uma lista para objetos da classe correspondente à característica alvo e as operações *getter/setter* desse atributo. Além disso, deve ser criada uma operação abstrata que retorna um *array* com as classes que implementam a característica alvo nas aplicações. Também deve existir uma operação que retorna um *array* com os nomes dos atributos resultantes de decomposições variantes.

Modelo: Na Figura 3.15 é mostrado a) o cenário do padrão e b) do projeto da solução.

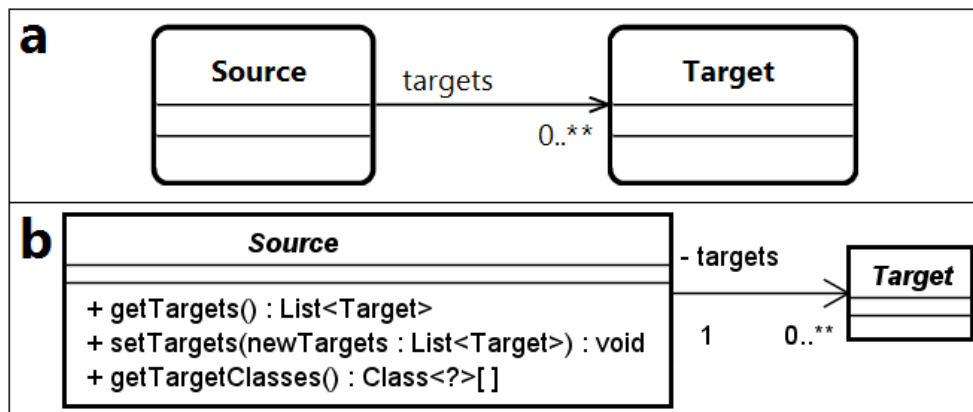


Figura 3.15. Modelos a) do cenário do P11 e b) do projeto da solução.

Implementação: solução em linguagem Java.

```
public abstract class Source extends AbstractFeature {
    private List<Target> targets;

    public Source() {
        super();
        targets = new ArrayList<Target>();
    }

    public void getTargets() {
        return targets;
    }

    public void setTargets(List<Target> newTargets) {
        targets = newTargets;
    }

    public abstract Class<?>[] getTargetClasses();
}

```

Padrão 12: Característica Requerente

Classificação: interface.

Padrões relacionados: Característica de Domínio (P02).

Contexto: implementação de característica com dependência requerente.

Problema: uma característica faz uso de outra, por exemplo, por meio de uma operação, apesar de não haver um relacionamento de decomposição entre elas. Desse modo, a característica alvo torna-se obrigatória para a característica de origem do relacionamento.

Propósito: indica quais unidades de código quando uma característica requer outra.

Cenário: uma característica possui uma dependência requerente para outra característica.

Solução: a classe da característica que possui a dependência requerente deve possuir uma operação abstrata que retorna uma instância da característica alvo dessa dependência.

Modelo: Na Figura 3.16 é mostrado a) o cenário do padrão e b) do projeto da solução.

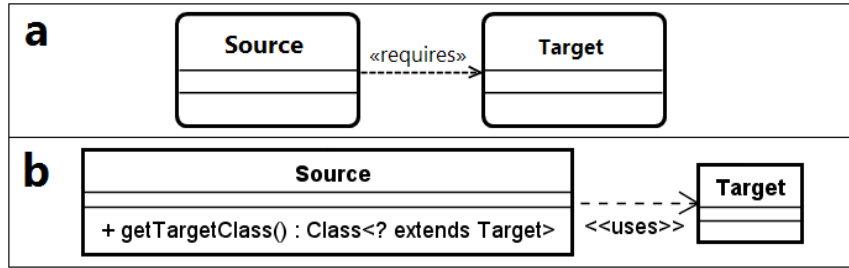


Figura 3.16. Modelos a) do cenário do P12 e b) do projeto da solução.

Implementação: solução em linguagem Java.

```
public abstract class Source extends AsbstractFeature {
    public abstract Class<? extends Target> getTargetClass();
}
```

Padrão 13: Variante Requerente

Classificação: interface.

Padrões relacionados: Variante de Característica (P03) e Decomposição Opcional (P09).

Contexto: implementação de dependência específica de variante.

Problema: uma variante de característica torna obrigatória uma característica que é opcional para a característica generalizada dessa variante.

Propósito: indica quais unidades de código devem ser construídas quando uma variante requer uma característica que é opcional para a generalização dessa variante.

Cenário: uma característica possui uma decomposição opcional para uma característica alvo. Contudo, uma das variantes da primeira característica possui uma dependência requerente para a característica alvo, tornando-a obrigatória para essa variante.

Solução: a classe da variante que possui a dependência requerente deve possuir uma operação abstrata que retorna uma instância da característica alvo dessa dependência. Essa operação sobrescreve a operação da característica generalizada.

Modelo: Na Figura 3.17 é mostrado a) o cenário do padrão e b) do projeto da solução.

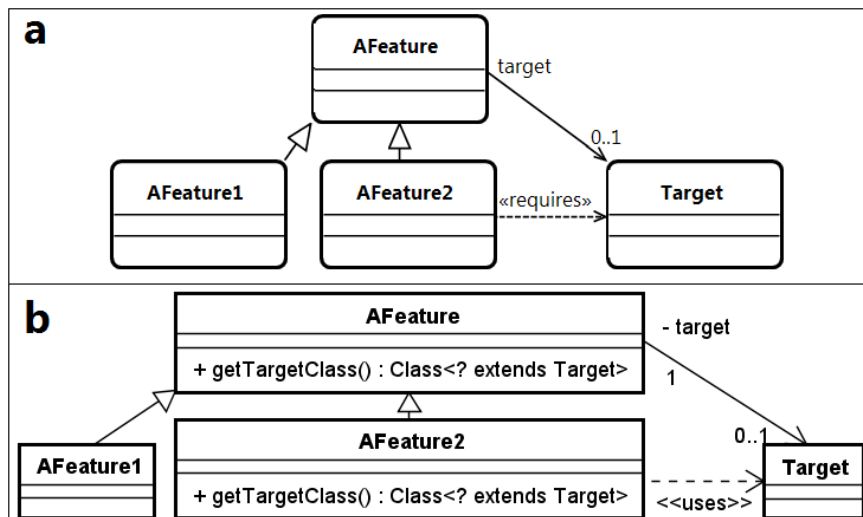


Figura 3.17. Modelos a) do cenário do padrão e b) do projeto da solução.

Implementação: solução em linguagem Java.

```
public abstract class AFeature {

    public Class<? extends Target> getTargetClass() {
        return null;
    }

}

public abstract class AFeature1 extends AFeature { }

public abstract class AFeature2 extends AFeature {

    public abstract Class<? extends Target> getTargetClass();

}
```

Padrão 14: Estratégia de Persistência

Classificação: persistência.

Padrões relacionados: Fábrica de Características (P01), Característica Persistente (P15) e Fábrica de DAO (P16).

Contexto: implementação da comunicação entre o framework e as bases de dados SQL.

Problema: é necessário definir um mecanismo de comunicação entre as classes do framework e as base de dados SQL das aplicações.

Propósito: indica quais unidades de código devem ser construídas para implementar a comunicação entre o framework e as bases de dados.

Cenário: independente de cenário.

Solução: deve ser criada uma classe para definir a estratégia de persistência do framework. Essa classe é estendida nas aplicações para conter as informações específicas sobre as bases de dados dessas aplicações.

Modelo: Na Figura 3.18 é mostrado o projeto da solução.

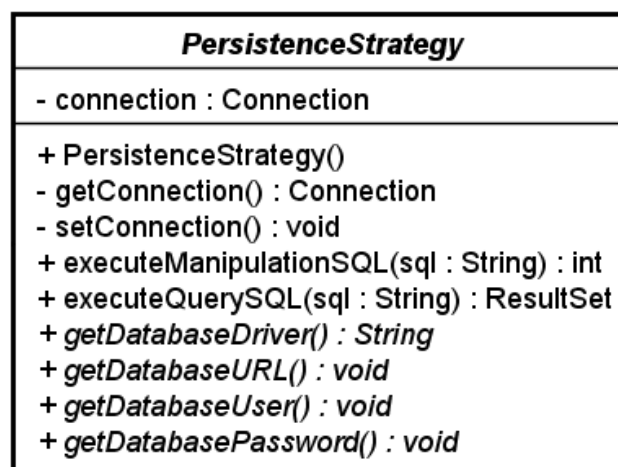


Figura 3.18. Modelo do projeto da solução do P14.

Implementação: solução em linguagem Java.

```
public abstract class PersistenceStrategy {

    private static Connection connection;

    public PersistenceStrategy() {
        connection = null;
    }

    private Connection getConnection() {
        if (connection == null) setConnection();
        return connection;
    }

    private void setConnection() {
        try {
            Class.forName(getDatabaseDriver());
            connection = DriverManager.getConnection(
                getDatabaseURL(), getDatabaseUser(), getDatabasePassword());
        }
        catch (ClassNotFoundException e) {
            // tratar exceção de driver não encontrado
        }
        catch (Exception e) {
            //tratar outras exceções
        }
    }

    public int executeManipulationSQL(String sql) {
        try {
            Statement statement = getConnection().createStatement();
            int result = statement.executeUpdate(sql);
            statement.close();
            return result;
        }
        catch ( SQLException e ) {
            //tratar exceção
        }
    }

    public ResultSet executeQuerySQL(String sql) {
        try {
            Statement statement = getConnection().createStatement();
            return statement.executeQuery(sql);
        }
        catch ( SQLException e ) {
            // tratar exceção
            return null;
        }
    }

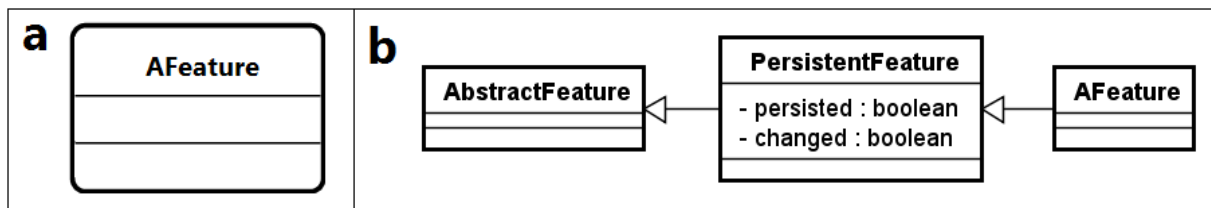
    public abstract String getDatabaseDriver();

    public abstract String getDatabaseURL();

    public abstract String getDatabaseUser();

    public abstract String getDatabasePassword();

}
```

Padrão 15: Característica Persistente**Classificação:** persistência.**Padrões relacionados:** Característica de Domínio (P02) e Variante de Característica (P03) e DAO de Característica (P17).**Contexto:** gerenciamento do estado de persistência dos objetos das classes.**Problema:** objetos das classes das características devem ter mecanismos para controlar o estado de persistência de seus dados.**Propósito:** indica quais unidades de código devem ser construídas para controlar o estado de persistência dos objetos das classes das características.**Cenário:** uma característica ou variante.**Solução:** a classe abstrata `PersistentFeature` estende `AbstractFeature` e possui atributos que controlam o estado de persistência dos objetos das classes das características. As classes das características estendem `PersistentFeature`.**Modelo:** Na Figura 3.19 é mostrado a) o cenário do padrão e b) do projeto da solução.**Figura 3.19. Modelos a) do cenário do P15 e b) do projeto da solução.****Implementação:** solução em linguagem Java.

```
public abstract class PersistentFeature extends AbstractFeature {

    private boolean persisted;
    private boolean changed;

    public AbstractFeature() {
        persisted = false;
        changed = false;
    }

    //getter/setters ...

}

public class AFeature extends PersistentFeature {

    public AFeature() {
        super();
    }

}
```

Padrão 16: Fábrica de DAO**Classificação:** persistência.**Padrões relacionados:** Fábrica de Características (P01) e Estratégia de Persistência (P14).

Contexto: criação de objetos das classes de persistência das características.

Problema: manter a funcionalidade de persistência separada da do domínio.

Propósito: indica quais unidades de código devem ser construídas para definir, de forma geral, as operações de persistência das características.

Cenário: independente de cenário.

Solução: deve ser criada uma classe abstrata com uma implementação das classes de persistência (*Data Access Object* – DAO) das características. Também é criada uma classe que implementa a fábrica de objetos dessas classes de persistência.

Modelo: Na Figura 3.20 é mostrado o projeto da solução.

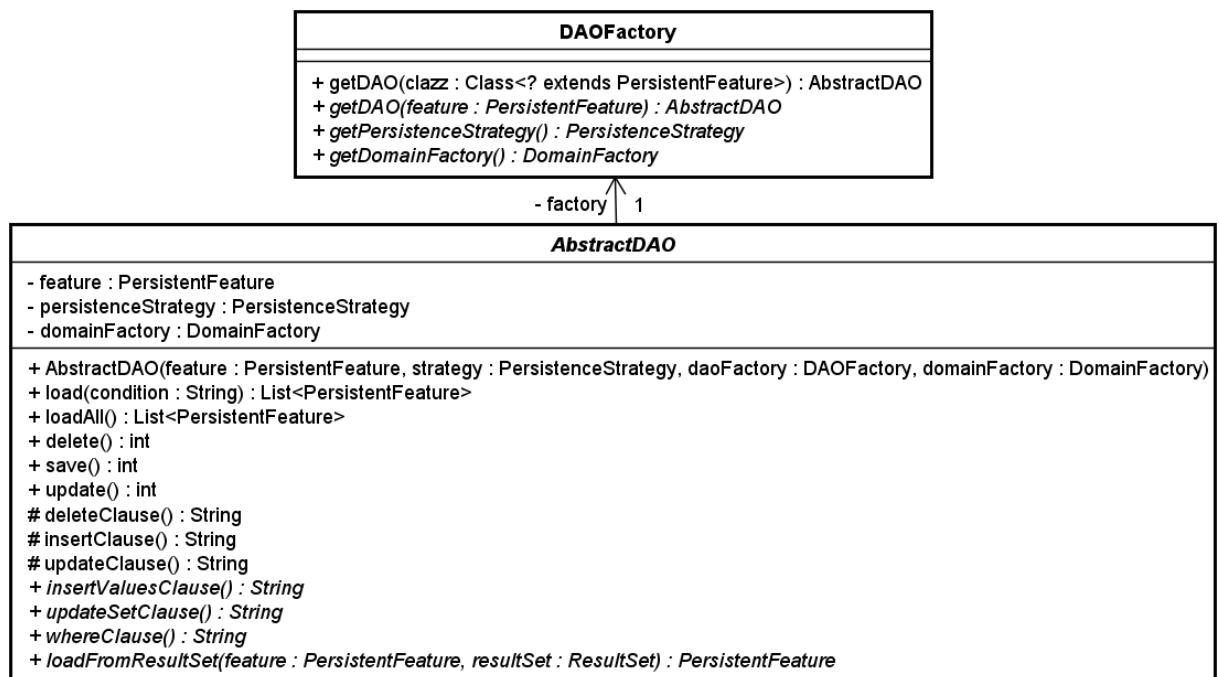


Figura 3.20. Modelo do projeto da solução do P16.

Implementação: solução em linguagem Java.

```

public abstract class DAOFactory {

    public abstract AbstractDAO getDAO(PersistentFeature feature );

    public AbstractDAO getDAO( Class<? extends PersistentFeature> clazz ) {
        return getDAO( DReflection.newInstance(clazz) );
    }

    public abstract PersistenceStrategy getPersistenceStrategy();

    public abstract DomainFactory getDomainFactory();

}

public abstract class AbstractDAO {

    private AbstractFeature feature;

    private PersistenceStrategy persistenceStrategy;
  
```

```
private DAOFactory daoFactory;

public AbstractDAO(AbstractFeature feature, PersistenceStrategy
    strategy, DAOFactory daoFactory, DomainFactory domainFactory) {
    this.feature = feature;
    this.persistenceStrategy = strategy;
    this.daoFactory = daoFactory;
    this.domainFactory = domainFactory;
}

//getters dos atributos...

public List<AbstractFeature> load( String condition ) {

    String sql = "SELECT * FROM " + feature.getClass().getSimpleName();
    if ( condition != null && !condition.equals( "" ) )
        sql += " WHERE " + condition;
    ResultSet rs = getPersistenceStrategy().executeQuerySQL( sql );
    List<AbstractFeature> result = new ArrayList<AbstractFeature>();

    try {
        while ( rs.next() ) {
            result.add( loadFromResultSet(
                domainFactory.newFeature( feature.getClass() ), rs ) );
        }
        rs.close();
    }
    catch (SQLException e) {
        //tratar exceção de ResultSet
    }
    return result;
}

public List<AbstractFeature> loadAll() {
    return load( null );
}

public int delete() {
    return getPersistenceStrategy().
        executeManipulationSQL( deleteClause() );
}

public int save() {
    return getPersistenceStrategy().
        executeManipulationSQL( insertClause() );
}

public int update() {
    return getPersistenceStrategy().
        executeManipulationSQL( updateClause() );
}

protected String deleteClause() {
    return "DELETE FROM " + getFeature().getClass().getSimpleName()
        + " WHERE " + whereClause() + ";";
}

protected String insertClause() {
    return "INSERT INTO " + getFeature().getClass().getSimpleName()
        + " VALUES (" + insertValuesClause() + " );";
}
```



```
protected String updateClause() {
    return "UPDATE TABLE " + getFeature().getClass().getSimpleName()
        + " SET " + updateSetClause() + " WHERE " + whereClause() + ";";
}

public abstract String insertValuesClause();

public abstract String updateSetClause();

public abstract String whereClause();

public abstract PersistentFeature loadFromResultSet(
    AbstractFeature feature, ResultSet rs );
}
```

Padrão 17: DAO de Característica

Classificação: persistência.

Padrões relacionados: Característica de Domínio (P02), Variante de Característica (P03) e Característica Persistente (P15).

Contexto: implementação da funcionalidade de persistência de uma característica.

Problema: uma característica ou variante deve ter seus dados persistidos.

Propósito: indica quais unidades de código devem ser construídas para realizar a persistência dos dados das classes das características.

Cenário: uma característica ou variante.

Solução: criar uma classe de persistência (DAO) para a característica ou variante. Essa classe estende `AbstractDAO`, sobrescrevendo o construtor e as operações `loadFromResultSet`, `insertValuesClause`, `updateSetClause` e `whereClause`, caso a característica ou variante correspondente possua atributos e/ou associações. Outras operações que carregam objetos do banco de dados (`load`) a partir de atributos(s) identificador(es) ou de relacionamentos de composição também podem ser criadas.

Modelo: Na Figura 3.21 é mostrado a) o cenário do padrão e b) do projeto da solução.

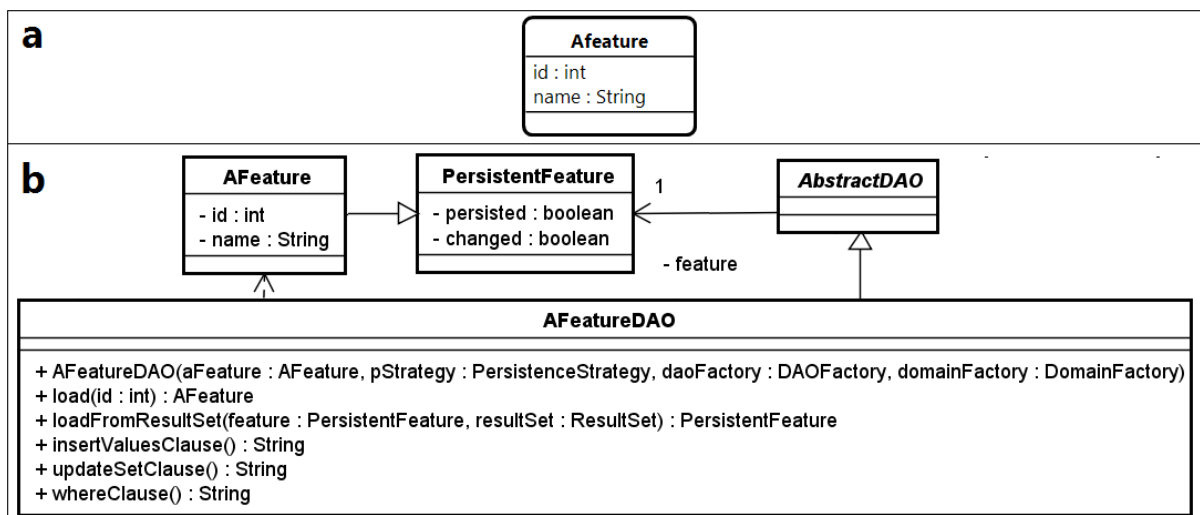


Figura 3.21. Modelos a) do cenário do P17 e b) do projeto da solução.

Implementação: solução em linguagem Java.

```

public abstract class AFeatureDAO extends AbstractDAO {

    public AFeatureDAO(AFeature aFeature, PersistenceStrategy strategy,
        DAOFactory daoFactory, DomainFactory domainFactory) {
        super(aFeature, strategy, daoFactory, domainFactory);
    }

    public AFeature load(Integer id) {
        List<PersistentFeature> features = load("id = " + id);
        return features.isEmpty() ? null : (AFeature) features.get(0);
    }

    @Override
    public PersistentFeature loadFromResultSet(
        PersistentFeature feature, ResultSet rs) {
        try {
            ((AFeature) feature).setId( rs.getInt( "id" ) );
            ((AFeature) feature).setName( rs.getString( "name" ) );
        }
        catch(SQLException e) { /*tratar exceção*/ }
        return feature;
    }

    @Override
    public String insertValuesClause() {
        return ((AFeature) getFeature()).getId() )
            + ", \"" + ((AFeature) getFeature()).getName() ) + "\"";
    }

    @Override
    public String updateSetClause() {
        return "name = \"" + ((AFeature) getFeature()).getName();
    }

    @Override
    public String whereClause() {
        return "id = " + ((AFeature) getFeature()).getId();
    }
}

```

3.3 Utilização da Abordagem F3

Nesta seção é apresentado um exemplo de utilização da abordagem F3 para o desenvolvimento de um framework. As etapas de Modelagem do Domínio e Construção do Framework são descritas nas Seções 3.3.1 e 3.3.2.

3.3.1 Modelagem do Domínio de Transações de Aluguel e de Comercialização de Recursos

O domínio escolhido para o desenvolvimento de um framework utilizando a abordagem F3 é o de transações de aluguel e comercialização de recursos. Esse domínio é

uma adaptação extraída da linguagem de padrões GRN (Seção 2.2.1) e possui as seguintes características:

- **Transação (ResourceTransaction)** é uma ação que transfere a posse de um ou mais recursos para um destino. Transação possui os atributos número, data e desconto. O valor final é calculado com base nos itens da transação e no desconto. Há três tipos de transação:
 - **Aluguel (ResourceRental)**, em que ocorre uma transferência temporária do recurso para um destino obrigatoriamente definido. Possui como atributos adicionais a data de devolução esperada e para o retorno do recurso e a data de devolução efetiva. Uma transação de aluguel pode ser antecedida por uma reserva;
 - **Reserva (ResourceReservation)** é uma solicitação prévia de uma transação aluguel. Uma reserva possui os seguintes atributos adicionais: número identificador, data da reserva, data de retirada, data de devolução e valor;
 - **Comercialização (ResourceTrade)**, quando ocorre uma transferência definitiva do recurso para um destino opcionalmente definido.
- **Item de Transação (TransactionItem)** indica um recurso envolvido na transação e, se for o caso, uma instância desse. Possui os atributos valor e quantidade.
- **Destino (DestinationParty)** é o solicitante de uma transação para o qual o(s) recurso(s) envolvido(s) é(são) entregue(s). Seu registro é opcional, exceto para aluguel e reserva. Destino possui os atributos número identificador e nome.
- **Recurso (Resource)** é a entidade envolvida na transação e possui os atributos número identificador e nome. Além disso, um recurso pode ser de dois tipos:
 - **Único (SingleResource)**, recurso singular que não pode participar de duas ou mais transações simultâneas. O atributo status que indica a sua disponibilidade.
 - **Instanciável (InstatiableResource)**, que representa uma entidade com várias instâncias, sendo que cada **Instância do Recurso (Resource Instance)** possui seu próprio status e número identificador, podendo participar de uma transação independentemente das demais.
- **Tipo de Recurso (ResourceType)** classifica um recurso com base em alguma referência. Os atributos de tipo de recurso são número identificador e nome. Diversos tipos podem ser definidos para classificar um recurso.
- Podem ser registrados os **Pagamentos (Payment)** das transações, cujos atributos são número identificador, data, valor e documento (código de boleto, por exemplo). Um pagamento pode ser de quatro tipos: **Dinheiro (Cash)**; e **Cartão (Card)**, com número, bandeira, nome do titular e data de vencimento.

O modelo F3 criado a partir das características do domínio de transações de aluguel e de comercialização de recursos é mostrado na Figura 3.22. O processo de criação de um modelo F3 é semelhante ao de um modelo de classes da UML, com a característica raiz do domínio, ResourceTransaction, sendo inserida em primeiro lugar. Em seguida, foram inseridas as suas variantes, ResourceRental, ResourceReservation e ResourceTrade. Como só podem ser realizadas reservas para transações de aluguel, um relacionamento de dependência do tipo *requires* foi definido entre ResourceReservation e ResourceRental e outro do tipo *excludes* foi definido entre ResourceReservation e ResourceTrade. Como TransactionItem é obrigatória para ResourceTransaction, o relacionamento de decomposição entre essas características tem multiplicidade mínima igual a 1. O mesmo ocorre para Resource em relação a TransactionItem. Por outro lado, DestinationParty e Payment são opcionais, de modo que a multiplicidade mínima dos relacionamentos de decomposição entre TransactionResource e essas características é igual a 0. As variantes de TransactionItem, denominadas SRTransItem e RITransItem, foram acrescentadas no modelo porque as variantes de Resource, SingleResource e InstantiableResource, influenciam no comportamento dos itens de transação. ResourceType está ligado a Resource por meio de um relacionamento de decomposição composta, pois é possível que sejam definidos diferentes tipos para um recurso em uma aplicação. Como os recursos instanciáveis devem possuir pelo menos uma instância, o relacionamento de decomposição entre InstantiableResource e ResourceInstance possui multiplicidade mínima igual a 1 e máxima igual a *.

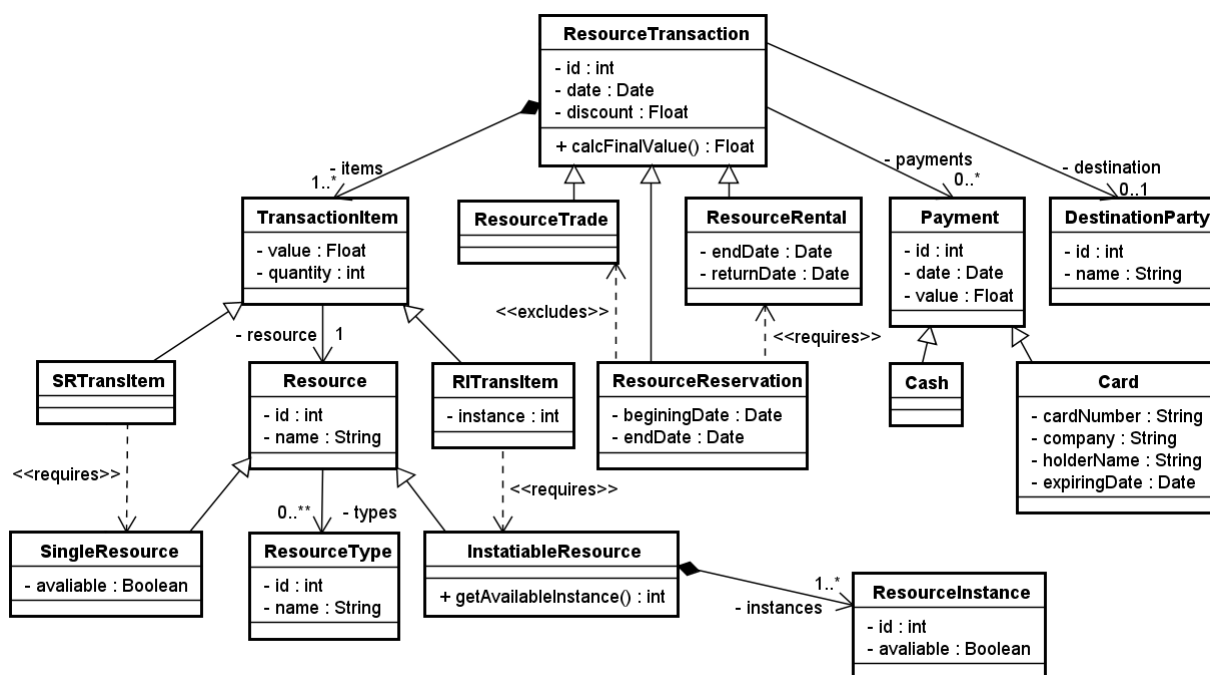


Figura 3.22. Modelo F3 do domínio de transações de aluguel e de comercialização de recursos.

3.3.2 Construção do Framework para Transações de Aluguel e de Comercialização de Recursos

Após a elaboração do modelo F3 do domínio, mostrado na Figura 3.22, foi realizada a etapa de Construção do Framework com o uso dos padrões da LPF3 sobre esse modelo. Na Figura 3.23 é mostrado o modelo de classes resultante do uso dos padrões estruturais da LPF3.

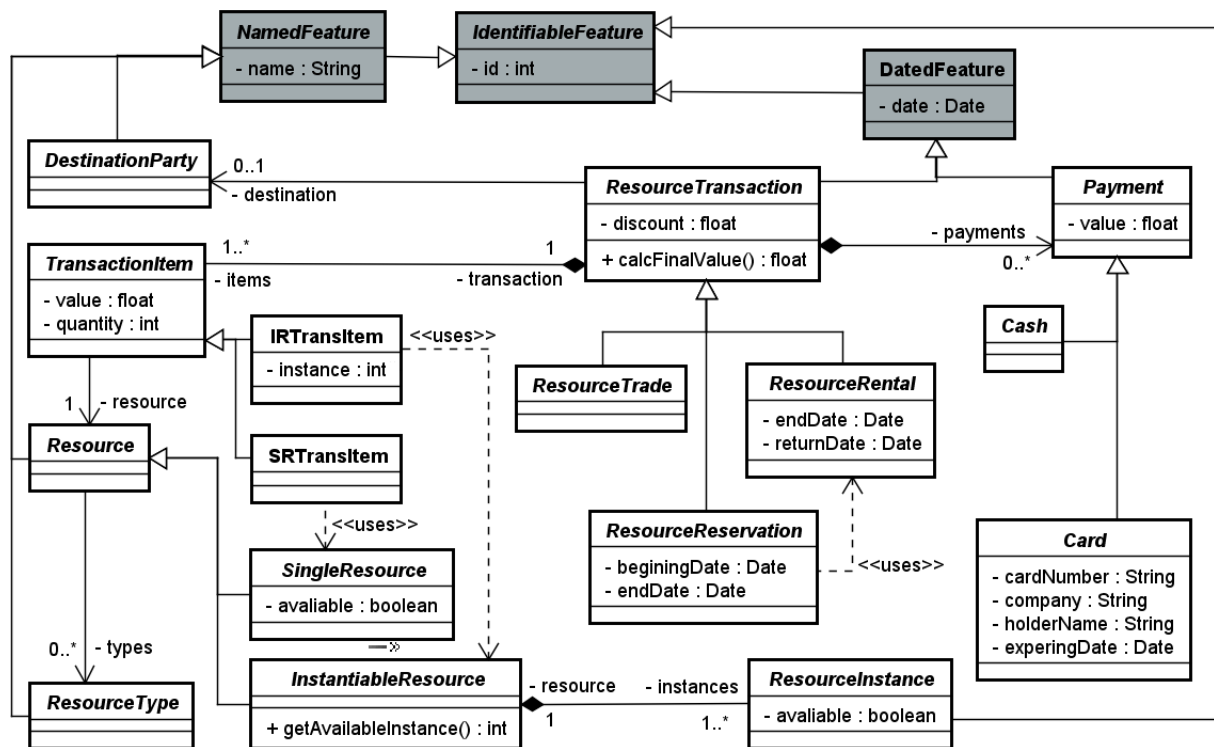


Figura 3.23. Modelo de classes do framework.

Os Padrões 2, 3, 4 e 5 foram aplicados para criar as classes das características e variantes e os respectivos atributos e operações dessas classes. Os construtores e *getters/setters* das classes foram omitidos, uma vez que estão presentes em qualquer aplicação. Também por motivos de simplificação, as classes `DomainFactory` e `AbstractFeature`, criadas com o uso do Padrão 1 exatamente como mostrado na Figura 3.5.b, não aparecem nesse modelo. Todas as classes da Figura 3.23 estendem, direta ou indiretamente, a classe `AbstractFeature`.

Os Padrões 6 e 7, Decomposição Simples e Decomposição Múltipla, foram aplicados para incluir no modelo da Figura 3.23 os relacionamentos de associação das classes do framework originados a partir das decomposições das características do domínio. Por exemplo, partindo da característica `ResourceTransaction` (Figura 3.22), a decomposição simples para `DestinationParty` e as decomposições múltiplas para `TransactionItem` e `Payment` deram origem aos relacionamentos de associação entre as classes dessas

características (Figura 3.23). A implementação dessas associações corresponde aos atributos `destination`, `items` e `payments` da classe `ResourceTransaction`, linhas 4 a 6 da Figura 3.24.

```
1 public class ResourceTransaction extends DatedFeature {
2
3     private float discount;
4     private DestinationParty destination;
5     private List<TransactionItem> items;
6     private List<Payment> payments;
7
8     public ResourceTransaction() {
9         discount = 0;
10        destination = null;
11        items = new ArrayList<TransactionItem>();
12        payments = new ArrayList<Payment>();
13    }
14
15    //...getters/setters
16
17    public abstract Class<? extends TransactionItem> getTransactionItemClass();
18
19    public Class<? extends DestinationParty> getDestinationPartyClass() {
20        return null;
21    }
22
23    public Class<? extends Payment> getPaymentClass() {
24        return null;
25    }
26
27 }
```

Figura 3.24. Trecho de código da classe `ResourceTransaction`.

O Padrão 8, Modularização Hierárquica, foi aplicado para evitar que as instruções de persistência relacionadas com os atributos `id`, `name` e `date` se repetissem em diversas classes DAO. Por isso, foram criadas as classes `IdentifiableFeature`, `NamedFeature` e `DatedFeature`, destacadas na cor cinza na Figura 3.23.

Os Padrões 9 e 10, Decomposição Opcional e Decomposição Obrigatória, foram aplicados de acordo com o valor da multiplicidade mínima dos relacionamentos de decomposição das características no modelo F3 da Figura 3.22. Nas linhas 17 a 25 da Figura 3.24 são mostradas as operações que solicitam as classes que estendem `TransactionItem`, `DestinationParty` e `Payment` nas aplicações. Esses métodos são abstratos quando a característica correspondente à classe solicitada é obrigatória ou retornam `null` quando essa característica é opcional.

O Padrão 11, Decomposição Composta, foi aplicado a partir do relacionamento entre as características `Resource` e `ResourceType` na Figura 3.22. Esse relacionamento indica que uma implementação de `Resource` pode se associar com várias implementações de

ResourceType. Desse modo, a classe resultante da característica Resource deve possuir uma lista do tipo ResourceType como atributo e uma operação que obtém as classes que estendem ResourceType nas aplicações, conforme é mostrado na Figura 3.25.

```
1 public class Resource extends NamedFeature {
2
3     private List<ResourceType> types;
4
5     public Resource() {
6         super();
7         types = new ArrayList<Payment>();
8     }
9
10    //...getters/setters
11
12    public Class[]<? extends ResourceType> getResourceTypeClasses() {
13        return null;
14    }
15
16 }
```

Figura 3.25. Trecho de código da classe Resource.

O padrão 12, Característica Requerente, foi aplicado para as características ResourceReservation, SRTransItem e IRTransItem, que requerem (*require*), respectivamente, as características ResourceRental, SingleResource e InstantiableResource. As classes que implementam as características requerentes contêm uma operação abstrata que solicita a classe que implementa a respectiva característica requerida. Na Figura 3.26 é mostrado parte do código da classe ResourceReservation que ilustra essa operação.

```
1 public class ResourceReservation extends ResourceTransaction {
2
3     //Atributos, construtor, getters e setters
4
5     public abstract Class<? extends ResourceRental> getResourceRentalClass();
6
7 }
```

Figura 3.26. Trecho de código da classe ResourceReservation.

Os padrões de persistência da LPF3 foram aplicados após os padrões de estrutura e de interface. Os padrões Estratégia de Persistência (P14), Característica Persistente (P15) e Fábrica de DAO (P16) são independentes de domínio, portanto, tanto sua modelagem quanto a sua implementação são semelhantes às mostradas na descrição desses padrões. Já o padrão DAO de Característica (P17), foi aplicado uma vez para cada classe da Figura 3.23.

Na Figura 3.27 é mostrado um modelo com as classes DAO do domínio de transações de aluguel e comercialização de recursos. As classes desse modelo estendem, direta e indiretamente, a classe `AbstractDAO`, descrita no Padrão 16 da LPF3. As classes criadas com o Padrão 6 para facilitar a modularização dos atributos e operações das classes das características também devem possuir suas classes DAO correspondentes, como pode ser visto com as classes `IdentifiableFeatureDAO`, `NamedFeatureDAO` e `DatedFeatureDAO`.

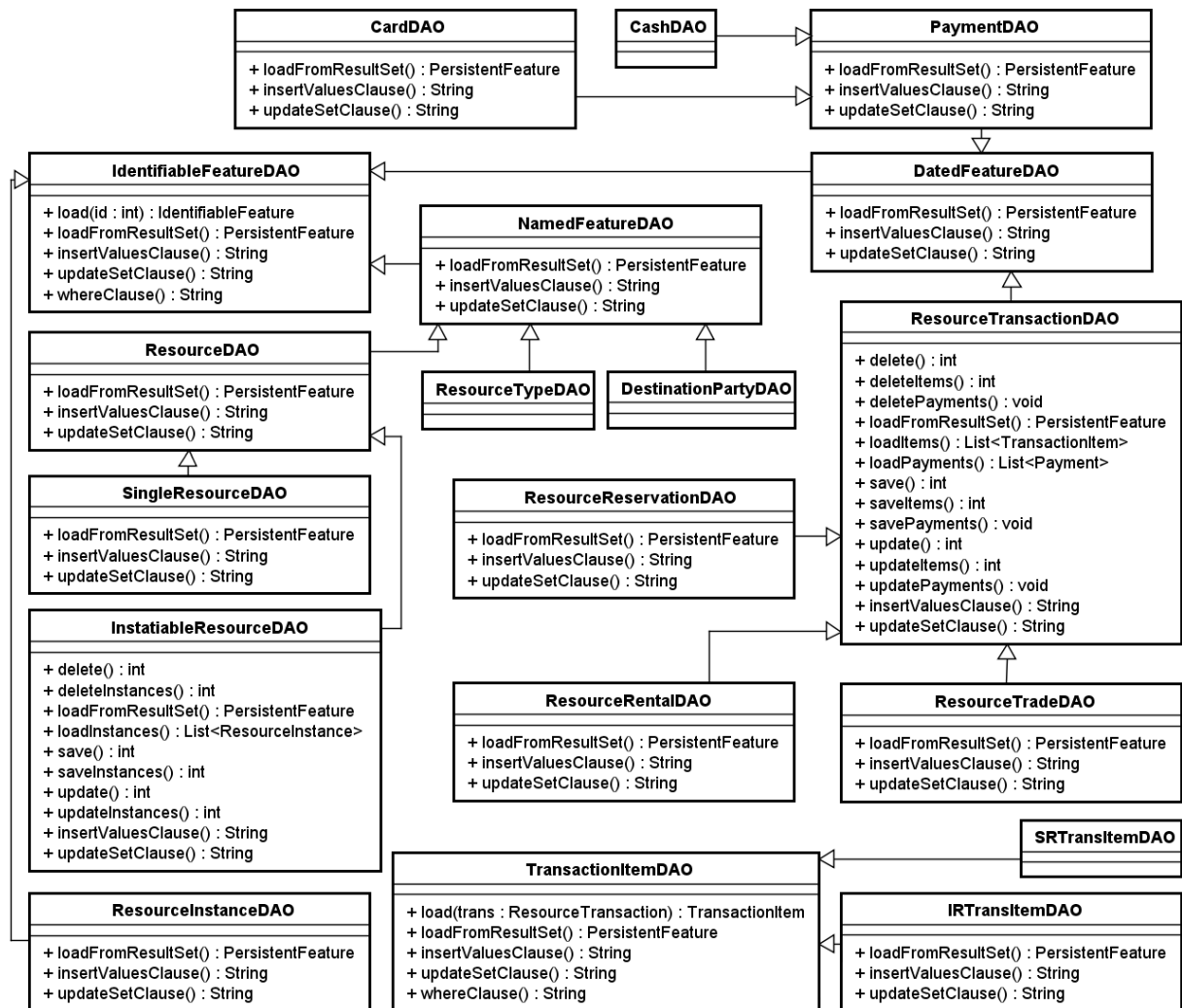


Figura 3.27. Modelo de classes DAO do domínio de transações de aluguel e comercialização.

As classes DAO, cuja classe de característica possui atributos, devem sobrescrever as operações:

- `loadFromResultSet`, que carrega dos dados de um `ResultSet` para um objeto da classe da característica;
- `insertValuesClause` e `updateSetClause`, que criam as partes das instruções `insert` e `update` da SQL que são específicas de cada classe; e

- `whereClause`, que cria a cláusula *where* das instruções SQL para as classes com atributos identificadores;
- `load`, que carrega um objeto da classe da característica. Essa operação é sobrescrita somente se a classe da característica possuir atributos identificadores, como, por exemplo, as classes `IdentifiableFeature` e `TransactionItem`.

Na Figura 3.23, `ResourceTransaction` possui relacionamentos de composição com `TransactionItem` e `Payment`. Por causa disso, `ResourceTransactionDAO`, na Figura 3.27, possui operações para carregar, apagar e salvar os itens e pagamentos de uma transação no banco de dados. Além disso, as operações que carregam, salvam e atualizam as transações foram sobrescritas para também serem executadas sobre os itens e os pagamentos da transação.

```

1 public abstract class ResourceTransactionDAO extends DatedFeatureDAO {
2
3     public PersistentFeature loadFromResultSet(
4         PersistentFeature feature, ResultSet resultSet) {
5
6         ResourceTransaction trans = (ResourceTransaction) feature;
7         trans = super.loadloadFromResultSet( trans, resultSet );
8
9         try {
10            trans.setDiscount( resultSet.getFloat( "discount" ) );
11
12            Class destClazz = trans.getDestinationPartyClass();
13            if ( destClazz != null ) { //tem DestinationParty?
14                DestinationPartyDAO destDAO = daoFactory.getDAO( destClazz );
15                DestinationParty destination = (DestinationParty)
16                    dpDAO.load( resultSet.getInt( "destination" ) )
17                trans.setDestination( destination );
18            }
19        }
20        catch( SQLException e) {
21            e.printStackTrace();
22        }
23        loadItems( trans );
24
25        if ( trans.getPaymentClass() != null ) //tem Payment?
26            loadPayments( trans );
27        return trans;
28    }
29
30    public void loadItems( ResourceTransaction trans ) {
31        TransactionItemDAO transactionItemDAO = (TransactionItemDAO)
32            getDAOFactory().getDAO( trans.getTransactionItemClass() );
33
34        List<TransactionItem> items = transactionItemDAO.load( trans );
35        trans.setItems( items );
36    }
37 }

```

Figura 3.28. Trecho de código da classe `ResourceTransactionDAO`.

Na Figura 3.28 é mostrado o código das operações `loadFromResultSet` e `loadItems` da classe `ResourceTransactionDAO`. Na linha 7 a operação `loadFromResultSet` da classe `DatedFeature` é invocada para que os atributos herdados dessa classe sejam carregados do banco de dados. Na linha 10 é carregado o valor do atributo `discount` e nas linhas 12 a 18 é carregado o valor do atributo `destination`. Como o atributo `destination` foi criado a partir de uma característica opcional, na linha 13 é verificado se essa característica é utilizada na aplicação. Na linha 14, é criado um objeto da classe `DestinationPartyDAO` para que, nas linhas 15 e 16, seja carregado do banco de dados um objeto da classe `DestinationParty` a partir do seu código identificador. Na linha 23 todos os itens da transação são carregados. O mesmo é feito para os pagamentos na linha 26, porém, como essa característica é opcional, anteriormente é feita uma verificação se está sendo utilizada. Nas linhas 30 a 36, a operação `loadItems` carrega um objeto da classe `TransactionItemDAO` e o utiliza para carregar do banco de dados todos os itens da transação.

3.3.3 Instanciação do Framework para uma Aplicação de Biblioteca

Nesta seção é apresentado um exemplo de instanciação do framework para transações de aluguel e comercialização de recursos em que foi desenvolvida uma aplicação de biblioteca. Os requisitos dessa aplicação são:

- Uma biblioteca empresta **obras literárias** para estudantes cadastrados. Uma obra literária possui código identificador, título, ano, editora e categoria;
- **Editora** possui código identificador, nome, endereço e *web site*;
- **Categoria** possui código identificador e nome;
- Uma obra literária pode ter um ou mais **exemplares**, cada um com código identificador, localização e disponibilidade;
- **Estudante** possui código identificador, nome, endereço e email;
- Um **empréstimo** possui código identificador, data, data de devolução prevista, data de devolução efetiva, estudante e uma ou mais obras emprestadas;
- Uma **reserva** de obras literárias pode ser realizada antes do empréstimo. Cada reserva possui código identificador, data da reserva, data do empréstimo, data de devolução prevista, estudante e uma ou mais obras reservadas. Uma reserva pode, ou não, originar um empréstimo.

Na Figura 3.29 é mostrado o modelo de classes da camada de modelo da aplicação de biblioteca elaborado com base nos requisitos dessa aplicação. Esse modelo inclui as classes específicas da aplicação (cor cinza) e as classes reutilizadas do framework (cor branca) para o domínio de transações de aluguel e de comercialização.

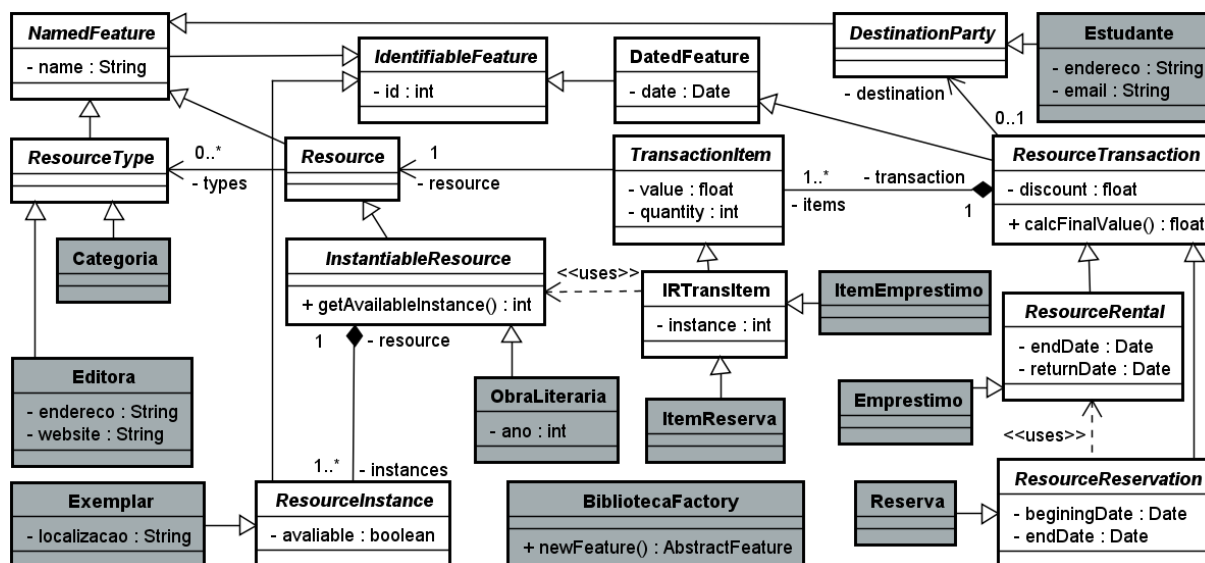


Figura 3.29. Modelo de classes da camada de modelo da aplicação de biblioteca.

Na Figura 3.30 é ilustrado o código da classe `ObraLiteraria`, que estende `InstantiableResource` do framework. A operação `getResourceInstanceClass` foi sobrescrita para informar ao framework a classe da aplicação que implementa as instâncias do recurso. Já a operação `getResourceTypeClasses` foi sobrescrita para informar ao framework as classes da aplicação que implementam tipos do recurso.

```

1 public class ObraLiteraria extends InstantiableResource {
2
3     private int ano;
4
5     public ObraLiteraria() {
6         super();
7         ano = 0;
8     }
9
10    public int getAno() {
11        return ano;
12    }
13
14    public void setAno(int ano) {
15        this.ano = ano;
16    }
17
18    public Class<? extends ResourceInstance>[] getResourceInstanceClass() {
19        return Exemplar.class;
20    }
21
22    public Class<?>[] getResourceTypeClasses() {
23        return new Class<?>[] { Categoria.class, Editora.class };
24    }
25 }

```

Figura 3.30. Código da classe `ObraLiteraria`.

Na Figura 3.31 é mostrado o modelo de classes da camada de persistência da aplicação de biblioteca. As operações das classes DAO do framework (cor branca) foram omitidas para simplificar o modelo. As classes DAO da aplicação (cor cinza) precisam somente sobrescrever as operações `loadFromResultSet`, `insertValuesClause` e `updateSetClause` para que a funcionalidade de persistência passe a considerar os atributos específicos das classes das aplicações. Como somente as classes `ObraLiteraria`, `Editora`, `Exemplar` e `Estudante` possuem atributos próprios, somente as suas classes DAO precisam sobrescrever essas operações. A classe `BibliotecaDAOFactory` é responsável por retornar objetos das classes DAO a partir de objetos das classes da camada de modelo da aplicação. `BibliotecaDAOFactory` também possui acesso à fábrica de objetos das classes da camada de modelo, por meio da operação `getDomainFactory`, para os casos em que os objetos DAO precisam carregar objetos a partir de dados do banco de dados. Também é em `BibliotecaDAOFactory` que o desenvolvedor indica qual é a classe que implementa a estratégia de persistência da aplicação por meio da operação `getPersistenceStrategy`.

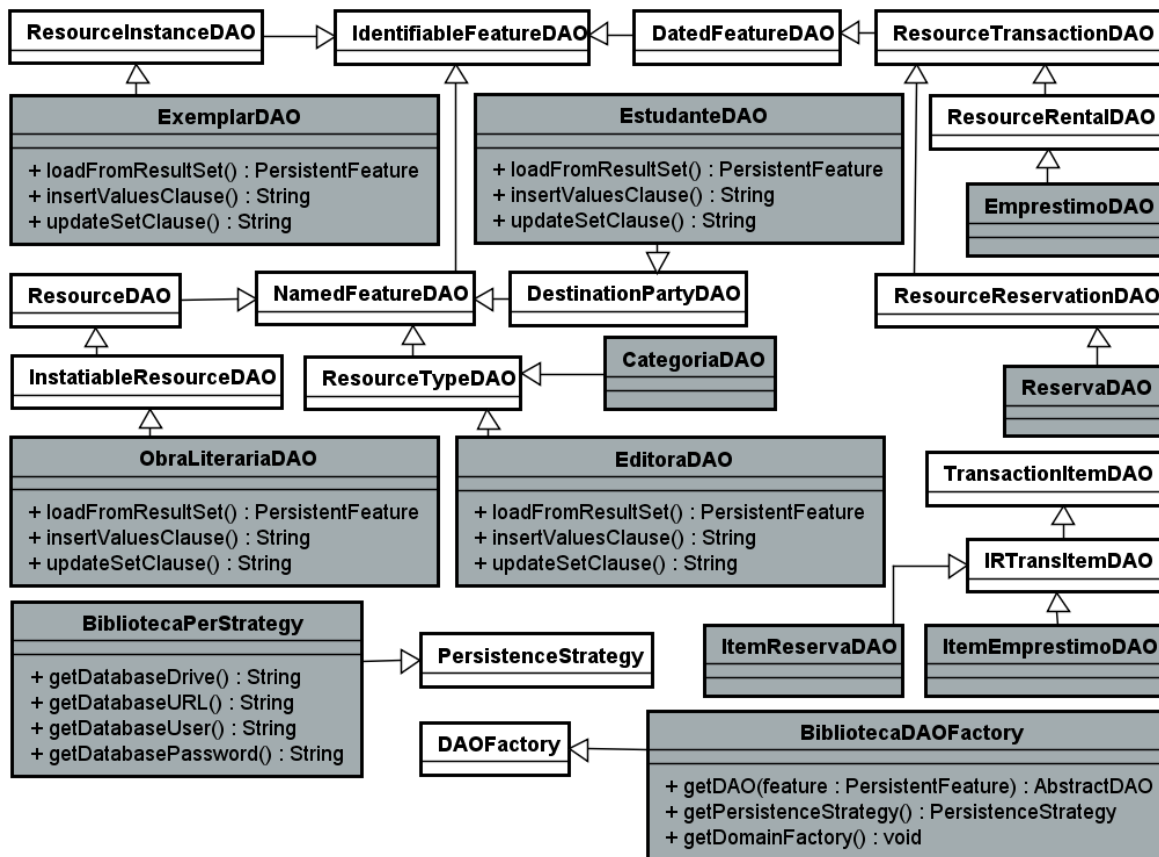


Figura 3.31. Modelo de classes da camada de persistência da aplicação de biblioteca.

Na Figura 3.32 é ilustrado o código da classe `ObraLiterariaDAO`, que estende `InstantiableResourceDAO` do framework. A operação `loadFromResultSet` carrega

do banco de dados o valor dos atributos herdados da classe do framework na linha 5 e do atributo `ano` na linha 8. Nas linhas 16 a 24 as operações `insertValuesClause` e `updateSetClause` incluem `ano` nas instruções SQL de inserção e atualização de dados.

```
1 public class ObraLiterariaDAO extends InstatiableResourceDAO {
2
3     public AbstractFeature loadFromResultSet (
4         PersistentFeature feature, ResultSet rs) {
5         super.loadFromResultSet(feature, rs);
6         ObraLiteraria obra = (ObraLiteraria) feature;
7         try {
8             obra.setAno( rs.getInt( "ano" ) );
9         }
10        catch( SQLException e) {
11            e.printStackTrace();
12        }
13        return feature;
14    }
15
16    public String insertValuesClause() {
17        return super.insertValuesClause()
18            + ", " + ((ObraLiteraria)getFeature()).getAno();
19    }
20
21    public String updateSetClause() {
22        return super.updateSetClause()
23            + ", ano = " + ((ObraLiteraria)getFeature()).getAno();
24    }
25 }
```

Figura 3.32. Código da classe ObraLiterariaDAO.

Um modelo de sequência para carregamento de uma obra literária a partir do banco de dados é exibido na Figura 3.33:

1. a aplicação utiliza uma objeto da classe `ObraLiterariaDAO` para invocar a operação `load` que recebe como argumento o valor do atributo identificador de uma obra literária;
2. o valor do atributo identificador é utilizado como uma condição de busca no banco de dados que é repassada para uma outra versão da operação `load`;
3. a busca por uma obra com base no código identificador é efetuada pela estratégia de persistência da aplicação e o resultado é retornado na forma de um objeto `ResultSet`;
4. a fábrica de objetos da aplicação é acionada;
5. a fábrica retorna um objeto da classe `ObraLiteraria`;
6. a operação `loadFromResultSet` preenche os atributos do objeto da classe `ObraLiteraria` com os valores armazenados no objeto `resultSet`.

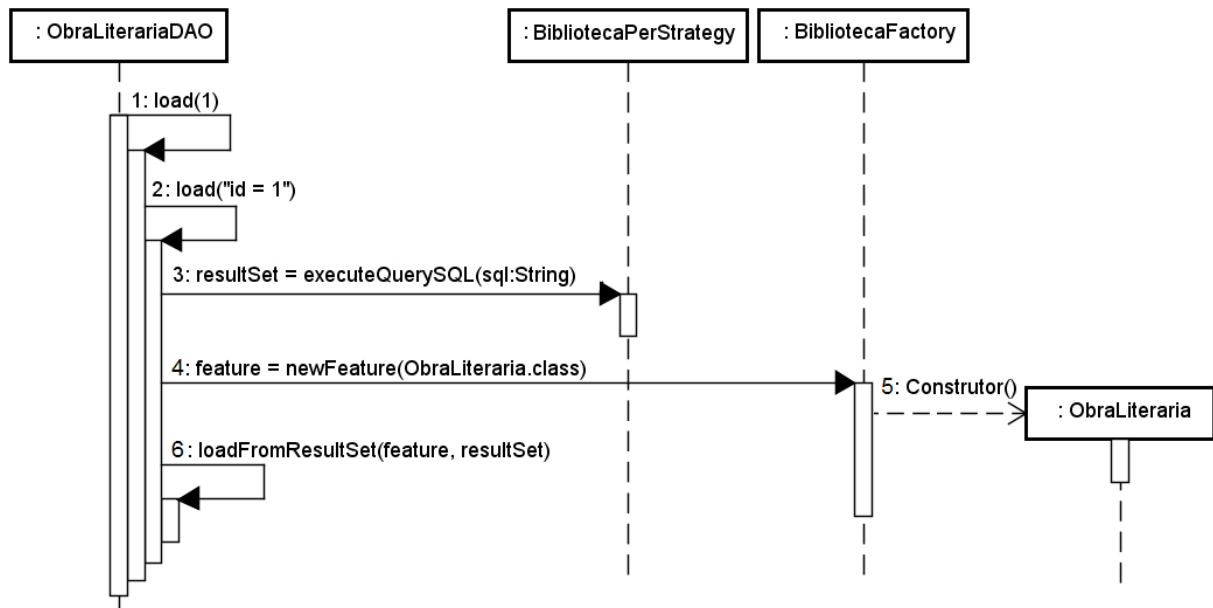


Figura 3.33. Modelo de sequência para carregar um objeto da classe ObraLiteraria a partir do banco de dados.

3.4 Considerações Finais

Neste capítulo foi apresentada a abordagem F3, que auxilia o desenvolvedor no projeto e na implementação de um framework a partir de um modelo com as características do domínio.

Na abordagem F3 é utilizada uma versão estendida do modelo de características, denominada modelo F3. Esse modelo permite a definição das características do domínio e também de atributos e operações que compõem a funcionalidade desse domínio. Os modelos F3 foram elaborados com uma notação gráfica semelhante à dos modelos de classes da UML, de modo a facilitar o seu entendimento por parte dos desenvolvedores. Além disso, essa semelhança também permite que modelos F3 possam ser criados com o apoio de ferramentas para modelagem UML.

O projeto e a implementação de um framework a partir de um modelo F3 é realizada com o apoio de uma linguagem de padrões, a LPF3. Cada padrão da LPF3 é aplicado com base em um cenário identificado nos modelos F3 e a solução proposta pelo padrão indica as estruturas de código que devem ser construídas para que a funcionalidade das características envolvidas nesse cenário seja implementada no framework. Os padrões da LPF3 são classificados de acordo com o propósito das estruturas indicadas na solução proposta. Essa classificação influencia a ordem de execução dos padrões da LPF3, pois primeiramente são aplicados os padrões que definem a estrutura do framework (estrutural),

seguidos pelos padrões que definem a comunicação do framework com a aplicação (interface) e, por fim, os padrões relacionados com a persistência de dados (persistência).

Durante o desenvolvimento de um framework os desenvolvedores podem sentir dificuldade em definir os elementos que compõem a estrutura e a interface de comunicação desse framework com as aplicações. A abordagem F3 permite que esse desenvolvimento seja realizado com maior eficiência e facilidade por dois motivos: 1) os modelos F3 permitem que as características e as regras do domínio do framework sejam definidas sem se preocupar com complexidades de implementação; e 2) os padrões da LPF3 guiam os desenvolvedores indicando quais unidades de código devem ser criadas para que o framework implemente o domínio definido no modelo F3. Com isso, há também um ganho de eficiência, uma vez que os desenvolvedores têm maior consciência de como proceder e cometem menos erros durante a implementação do código-fonte do framework.

Desenvolvedores pouco familiarizados com a abordagem F3 podem gastar um tempo considerável consultando a documentação dos padrões da LPF3 durante o desenvolvimento de um framework. Contudo, o ganho de eficiência no desenvolvimento de frameworks pode aumentar a medida que os desenvolvedores obtêm maior familiaridade com os modelos F3 e os padrões da LPF3. Os modelos F3 utilizam conceitos comuns a outras abordagens de modelagem de domínio. A documentação dos padrões contém descrições textuais e modelos dos cenários que ajudam os desenvolvedores identificarem quando esses padrões devem ser aplicados. Além disso, os padrões Fábrica de Características (P01), Estratégia de Persistência (P14), Característica Persistente (P15) e Fábrica de DAO (P16) são independentes de domínio, portanto, são mais fáceis de serem aplicados e as estruturas de código sugeridas por esses padrões podem ser construídas uma única vez e reutilizadas em diversos frameworks. Além disso, os padrões da LPF3 são baseados em outros padrões encontrados na literatura e conhecidos pelos desenvolvedores, o que facilita o seu aprendizado e uso.

Uma desvantagem da abordagem F3 é que os seus padrões cobrem somente as camadas de persistência e de modelo dos frameworks. A construção das camadas de controle e de visão devem ser realizadas pelo desenvolvedor sem o apoio da abordagem F3.

Capítulo 4

UMA ABORDAGEM PARA FACILITAR A INSTANCIÇÃO DE FRAMEWORKS

4.1 Considerações Iniciais

Como foi comentado no Capítulo 2, a utilização de frameworks no desenvolvimento de aplicações tem como principais benefícios a maior eficiência ao processo de desenvolvimento e maior qualidade para as aplicações desenvolvidas. Isso ocorre porque essas aplicações são construídas a partir de estrutura de código previamente testada e, normalmente, construída com base em padrões de software (STURM e KRAMER, 2013; ABI-ANTOUM, 2007; FAYAD e JOHNSON, 1999).

Estudos têm destacado as vantagens obtidas a partir do reuso de frameworks no desenvolvimento de aplicações (DURELLI et al., 2010; ZHANG et al., 2009; HOU, 2008; KIRK et al., 2007; LOPES et al. 2005). Contudo, esses estudos também afirmam que o reuso de frameworks não é uma tarefa simples, pois demanda conhecimento detalhado do domínio, da arquitetura, das classes, da interface e da linguagem de programação utilizada no framework. Além disso, a complexidade de um framework é proporcional à amplitude e à variabilidade do seu domínio. Mesmo desenvolvedores com experiência em reuso de um determinado framework podem cometer equívocos durante a sua instanciação e desenvolver aplicações que apresentam falhas. Conseqüentemente, a eficiência e a qualidade esperadas com o reuso do framework não são alcançadas.

Antkiewicz et al. (2009) afirmaram que algumas soluções têm sido propostas com o intuito de amenizar essas dificuldades, entre as quais manuais, tutoriais, linguagens de padrões, *cookbooks*, exemplos de aplicações desenvolvidas com a instanciação de frameworks, entre outras. Contudo, para esses autores, essas soluções são passivas, no sentido de que apenas auxiliam ao desenvolvedor quanto ao procedimento a ser realizado.

A tarefa efetiva de instanciação do framework ainda recai sob a responsabilidade do desenvolvedor, sendo realizada sem apoio computacional especializado.

Alguns trabalhos mostraram que a inserção de defeitos na parte do código das aplicações referente à instanciação de um framework pode ser fortemente amenizada com a construção de um wizard gerador de aplicações (VIANA 2009; BRAGA et al., 2003). A utilização de um wizard reduz a responsabilidade do desenvolvedor quanto à configuração de pontos variáveis do framework e, conseqüentemente, também reduz a inserção de defeitos no código e o tempo gasto com a implementação. Contudo, a modelagem da aplicação é necessária mesmo com o uso de wizard, o que faz com que o desenvolvedor informe as características de uma aplicação duas vezes: no modelo e nos formulários do wizard (VIANA, 2009). A principal razão disso é que formulários de wizards são menos intuitivos do que modelos gráficos, tanto no preenchimento das informações das aplicações quanto na visualização posterior dessas informações.

Diante desse contexto, neste capítulo é apresentada uma abordagem para a construção de DSLs que facilita e torna mais eficiente a instanciação de frameworks (VIANA et al., 2013a; VIANA et al., 2012; VIANA et al., 2011). Para construir a DSL de um framework, as características do domínio desse framework são identificadas por meio da análise em seu código-fonte, em sua documentação e no código-fonte das aplicações que o reutilizam. Em seguida, cada característica é analisada individualmente, de modo que os elementos necessários para a sua modelagem sejam incluídos na DSL e o gerador do código-fonte das aplicações seja construído. Essa abordagem também pode ser aplicada aos frameworks construídos com a abordagem F3, apresentada no Capítulo 3.

As vantagens do uso de DSLs em relação às outras abordagens citadas são três: maior facilidade no reúso de frameworks, pois os elementos da DSL são mais intuitivos e ocultam as complexidades de implementação referentes à instanciação do framework; maior eficiência no desenvolvimento de aplicações, pois o código-fonte é gerado a partir dos modelos dessas aplicações; e melhor qualidade nas aplicações, pois a DSL contém restrições e mecanismos de validação que reduzem a possibilidade dos desenvolvedores de modelarem as aplicações de forma incorreta e o código-fonte ser gerado com defeitos.

Com o intuito de apresentar a abordagem de uma maneira mais prática, neste capítulo a construção das sintaxes abstrata e concreta da DSL é realizada com uso do GMF (THE ECLIPSE FOUNDATION, 2013b; GRONBACK, 2009) e a implementação dos templates é feita com o uso da linguagem de transformação JET (THE ECLIPSE FOUNDATION, 2011c). Porém, ressalta-se que a abordagem pode ser aplicada com o uso de outras tecnologias, como, por exemplo, o *Generic Modeling Environment* (GME) (ISIS, 2013) para construção da DSL e *Acceleo* (THE ECLIPSE FOUNDATION, 2011c) para desenvolvimento do gerador do código das aplicações.

As demais seções deste capítulo estão organizadas da seguinte forma: na Seção 4.2 é apresentada a abordagem de construção e uso de DSLs de frameworks; na Seção 4.3 são apresentadas duas provas de conceito que retratam a construção de DSLs para os frameworks GRENJ e Hibernate; e, na Seção 4.4 são feitas as considerações finais a respeito da abordagem proposta neste capítulo.

4.2 Abordagem para a Construção e Uso de DSLs de Frameworks

A abordagem para construção e uso de DSLs de frameworks é composta de duas fases, Engenharia da DSL e Engenharia da Aplicação, como mostrado na Figura 4.1.

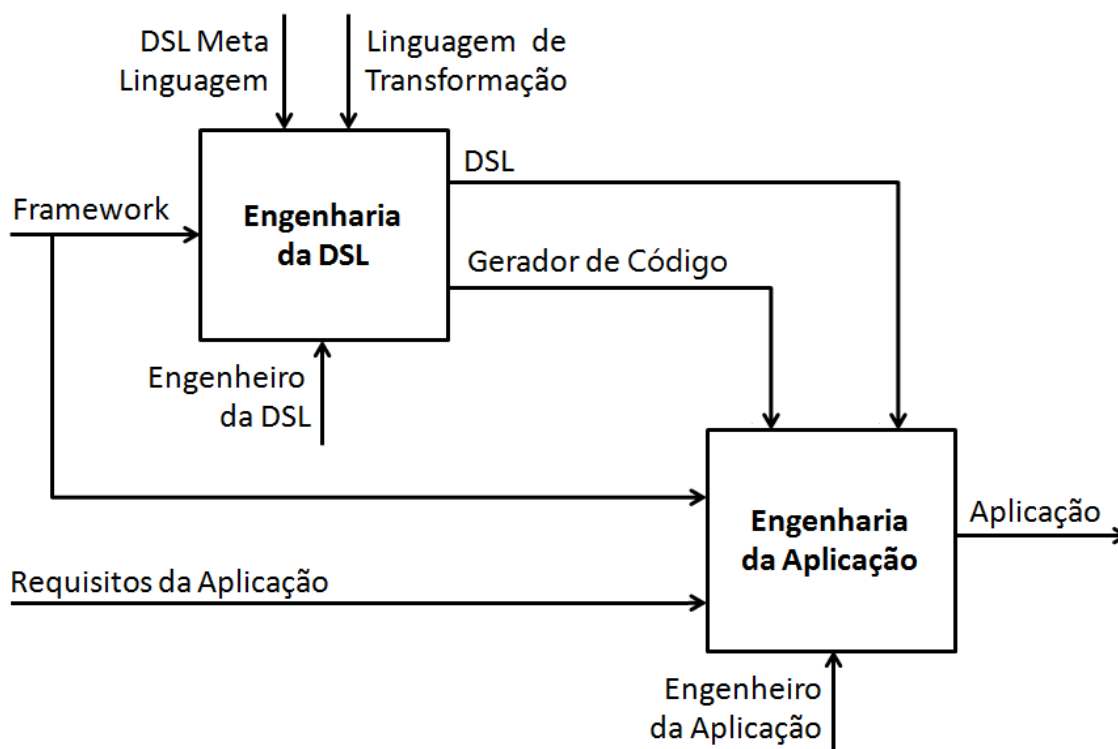


Figura 4.1. Fases da abordagem para construção e uso de DSLs de frameworks.

A notação de Structured Analysis and Design Technique (SADT) (MARCA e McGOWAN, 1988) é utilizada na Figura 4.1. A fase de Engenharia da DSL, apresentada em detalhes na Seção 4.2.1, tem como entrada um framework (código-fonte e documentação). Nessa fase o Engenheiro da DSL faz uso de uma metalinguagem para construir a DSL e uma linguagem de transformação para desenvolver o gerador de código das aplicações com base na análise desse framework. Na fase de Engenharia da Aplicação, apresentada em detalhes na Seção 4.2.2, o Engenheiro da Aplicação utiliza a DSL para criar um modelo com base nos requisitos de uma aplicação. Em seguida, o gerador é utilizado para produzir o código-fonte da aplicação a partir desse modelo.

4.2.1 Engenharia da DSL

Na fase de Engenharia da DSL as características do domínio do framework são analisadas para que a DSL seja projetada e implementada. As cinco etapas que compõem essa fase são mostradas na Figura 4.2: 1) Identificação das Características; 2) Análise das Características; 3) Projeto da DSL; 4) Construção da DSL; e 5) Construção do Gerador.

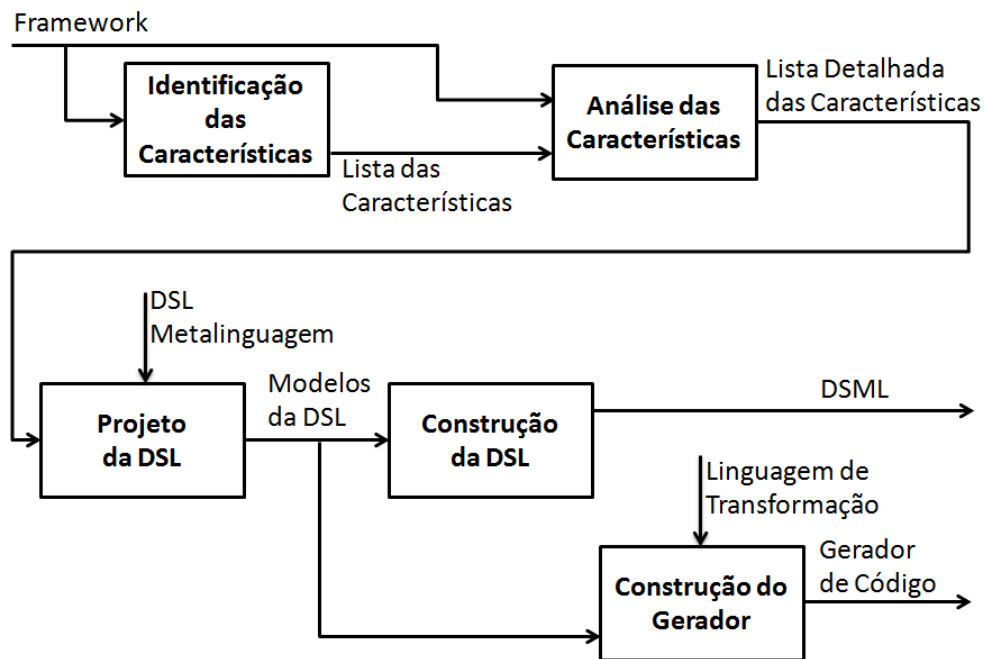


Figura 4.2. Etapas da fase de Engenharia da DSL.

4.2.1.1 Identificação das Características

Esta etapa tem como objetivo analisar a documentação disponível e o código-fonte do framework com o intuito de obter uma visão geral das características do seu domínio e as dependências existentes entre elas. No contexto dessa abordagem, uma característica representa uma entidade do domínio do framework e é identificada, principalmente, por uma classe do framework que pode ser diretamente reutilizada pelas aplicações, seja por herança (caixa branca) ou por composição (caixa preta). Os relacionamentos entre as características são identificados a partir dos relacionamentos e operações das classes do framework que as identificam e de suas superclasses.

Um modelo de classes em nível de projeto do framework H, hipotético, é mostrado na Figura 4.3. Nenhuma característica é identificada a partir da classe A porque essa classe foi criada somente para reunir atributos e operações comuns às classes C e D e não é diretamente reutilizada pelas aplicações. Contudo, características são identificadas a partir das classes B, C e D, pois essas classes podem ser diretamente reutilizadas pelas aplicações para as quais o framework H é instanciado.

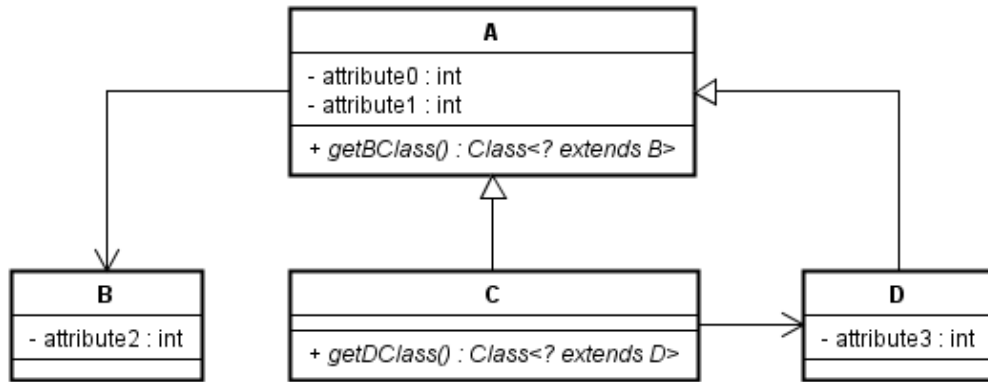


Figura 4.3. Modelo de classes do framework H.

A sequência em que as características devem ser tratadas nas etapas posteriores para a construção da DSL é definida com base nos relacionamentos existentes entre as classes que as implementam. Uma característica pode ser implementada por uma ou mais classes do framework. No Quadro 4.1 são apresentadas as características do framework H e as classes que as representam. A característica B é independente, ou seja, não utiliza classes de outras características do modelo; as características C e D dependem da característica B porque as classes C e D possuem um relacionamento com uma classe B herdado da classe A. A característica C também depende da característica D porque a classe C possui um relacionamento com a classe D.

Quadro 4.1. Lista das características identificadas no framework H.

#	Característica	Classes Representantes
1	B	B
2	D	D, A, B
3	C	C, A, B, D

4.2.1.2 Análise das Características

Nesta etapa são analisados os detalhes de cada característica com o intuito de identificar as informações necessárias para a reutilização do framework. Esses detalhes podem ser identificados respondendo-se as seguintes perguntas:

- Qual o nome, os atributos e as operações da classe na aplicação que reutiliza a característica?
- Quais classes na aplicação reutilizam classes do framework das quais a característica é dependente?
- Quais informações são requeridas pelos argumentos ou pelo código interno das operações da classe que representa a característica e de suas superclasses que, opcionalmente ou obrigatoriamente, devem ser invocadas/sobrescritas na classe na aplicação que reutiliza a característica?

Essas perguntas podem ser respondidas por meio de uma análise do código-fonte do framework ou de aplicações que reutilizam esse framework. Por exemplo, no código-fonte do framework existe uma operação abstrata `opA` que deve ser sobrescrita no código das aplicações. Ao observar o código dessa operação em uma classe de uma aplicação, é possível identificar quais informações da característica analisada a DSL deve conter. Isso possibilita a geração do código dessa operação nas aplicações modeladas com essa DSL.

No Quadro 4.2 é mostrada a lista detalhada das características do framework H com a inclusão das informações necessárias para sua reutilização. Todas as características necessitam das informações nome, atributos e operações específicas das classes da aplicação (linha zero). Além disso, para a reutilização do framework H, a classe `C` necessita de informação de qual classe da aplicação que reutiliza a classe `D`. Essa informação é necessária para que a operação abstrata `getDClass`, da classe `C`, seja sobrescrita nas classes das aplicações que reutilizam `C`, retornando qual classe reutiliza a classe `D`.

Quadro 4.2. Lista detalhada das características do framework H.

#	Característica	Classes Representantes	Informações para Reutilização
0	Todas	-	Nome da classe na aplicação que a reutiliza, atributos e operações específicos
1	B	B	-
2	D	D, A, B	Qual classe na aplicação reutiliza B?
3	C	C, A, B, D	Quais classes na aplicação reutilizam B e D?

4.2.1.3 Projeto da DSL

Nesta etapa as sintaxes abstrata e concreta da DSL são modeladas a partir da lista de características do framework. Na sintaxe abstrata são definidos os elementos que representam as características do domínio do framework e armazenam as informações necessárias para a geração do código das aplicações que reutilizam o framework. Na Figura 4.4 é mostrado o metamodelo da DSL (sintaxe abstrata) do framework H. Cinco metaclasses desse metamodelo são auxiliares: 1) `Application`, que é o elemento raiz dos modelos criados com a DSL; 2) `Feature`, que é abstrata e deve ser estendida pelas metaclasses que representam as características do framework, pois fornece as informações comuns a todas as características (linha 0 do Quadro 4.2); 3) `Attribute`, que representa os atributos específicos das classes das aplicações modeladas com a DSL; 4) `Operation`, que representa as operações específicas das classes das aplicações modeladas com a DSL; e 5) `Parameter`, que representa os parâmetros de entrada das operações. As metaclasses `B`, `C` e `D` correspondem às características do domínio do framework H. As informações necessárias para a reutilização dessas características (linhas 1 a 3 do Quadro 4.2) são obtidas por meio dos relacionamentos existentes entre as suas metaclasses.

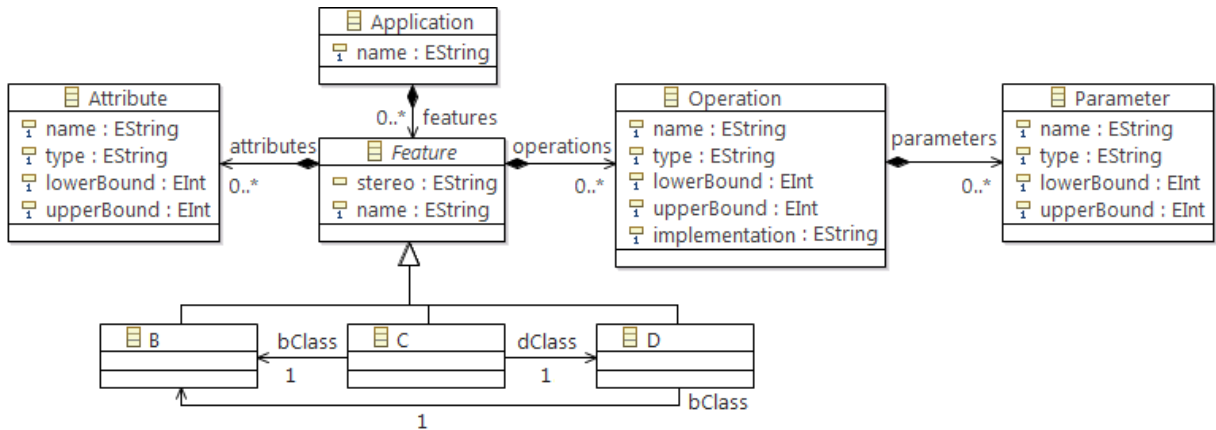


Figura 4.4. Metamodelo da DSL do framework H.

A sintaxe concreta é composta por elementos que permitem a visualização e a edição das informações definidas na sintaxe abstrata. A construção da sintaxe concreta de uma DSL é dependente da ferramenta adotada para esse fim. Por exemplo, o GMF (THE ECLIPSE FOUNDATION, 2013b; GRONBACK, 2009) utiliza três tipos de modelos para definir a sintaxe concreta das DSLs: *Gmfgraph*, *Gmftool* e *Gmfmap*.

No modelo *Gmfgraph* são definidos os componentes da notação gráfica da DSL que correspondem às características do framework e seus relacionamentos. Cada componente gráfico pode ser formado pela junção de vários componentes de forma aninhada. Por exemplo, na Figura 4.5 é mostrado que uma característica do framework é representada graficamente por um retângulo (*Rounded Rectangle FeatureFigure*) composto de outros retângulos que contêm os rótulos estereótipo (*Rectangle FeatureStereoArea*), do nome (*Rectangle FeatureNameArea*), dos atributos (*Rectangle PropertiesCompartment*) e das operações (*Rectangle OperationsCompartment*), respectivamente. Os relacionamentos de associação são representados por linhas contínuas (*Polyline Connection AssociationLink*).

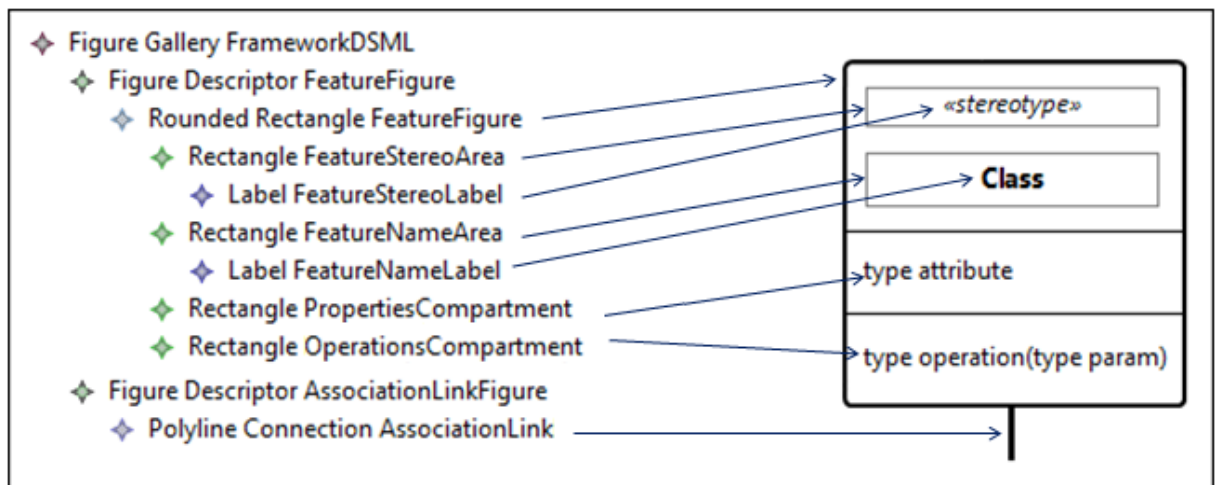


Figura 4.5. Definição da notação gráfica da DSL do framework H.

No modelo *Gmftool* são definidos os itens de menu dos elementos e dos relacionamentos que podem ser utilizados nos modelos das aplicações. Na Figura 4.6 é

mostrado o modelo *Gmftool* do framework H, no qual são definidos os itens que compõem a caixa de menu da DSL desse framework. Foi criado um item para cada um dos elementos que fazem parte do modelo gráfico da DSL: atributo, operação, parâmetro, as características B, C e D e as ligações entre as características.

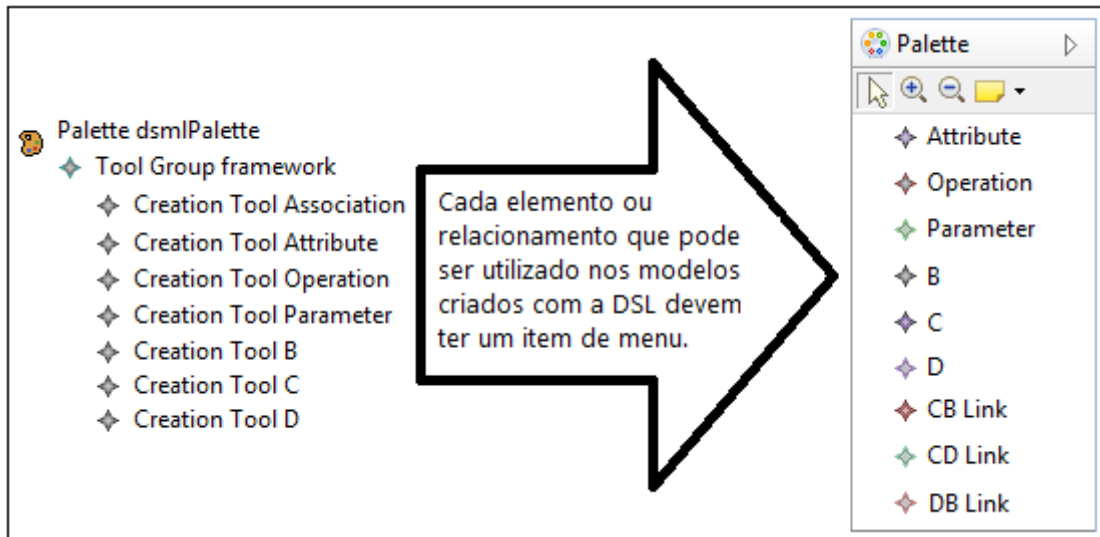


Figura 4.6. Definição dos itens do menu da DSL do framework H.

No modelo *Gmfmap* os elementos do metamodelo são combinados com os seus respectivos componentes da notação gráfica e itens de menu. Na Figura 4.7 é ilustrado um exemplo de modelo *Gmfmap* que combina as metaclasses e os relacionamentos do metamodelo (Figura 4.4) com a notação gráfica definida no modelo *Gmfgraph* (Figura 4.5) e com os itens de menu definidos no modelo *Gmftool* (Figura 4.6). Nesse modelo *Gmfmap* cada característica é definida com a inclusão de um *Top Node Reference* contendo um *Node Mapping*. Os *Node Mappings* dos elementos internos às características, como atributos e operações, são incluídos por meio de uma *Child Reference* e de um *Compartment Mapping*. De forma semelhante, o *Node Mapping* dos parâmetros é incluso nas operações. Por fim, *Feature Labels* são acrescentados para permitir a visualização e a edição dos nomes e dos tipos das características, dos atributos, das operações e dos parâmetros.

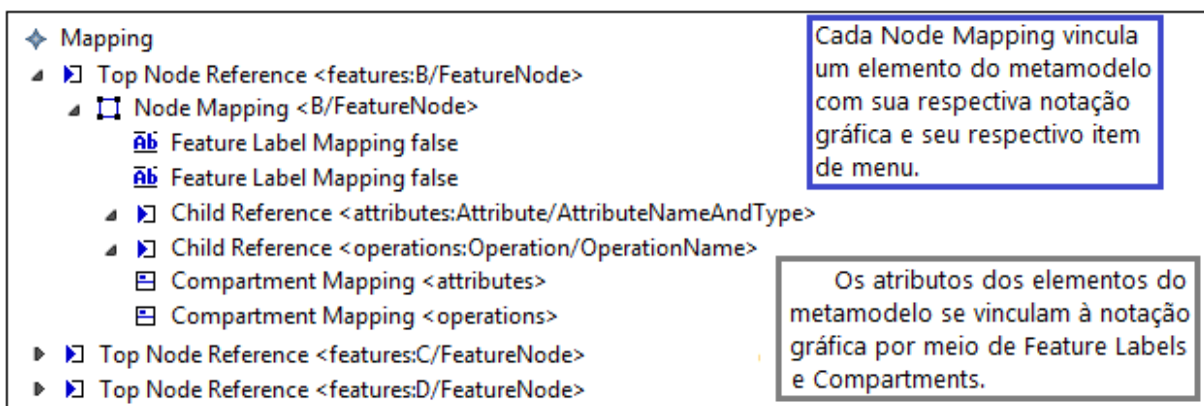


Figura 4.7. Modelo *Gmfmap* da DSL do framework H.

4.2.1.4 Construção da DSL

Normalmente, as ferramentas de desenvolvimento de DSL realizam a geração do código do editor da DSL a partir dos modelos. No GMF o código é gerado na forma de plug-ins para o Eclipse IDE a partir do metamodelo e dos modelos *Gmfgraph*, *Gmftool* e *Gmfmap*. Um mecanismo de validação também é criado a partir das propriedades e dos relacionamentos dos elementos definidos na sintaxe abstrata da DSL. Com isso, é possível verificar, por exemplo, se todas as características obrigatórias do framework foram incluídas no modelo de uma aplicação.

4.2.1.5 Construção do Gerador

Nesta etapa é realizada a construção do gerador de código das aplicações modeladas com a DSL do framework. Para garantir maior flexibilidade a esse gerador, a estrutura do código a ser gerado é definida por meio de templates implementados com o uso de uma linguagem de transformação. O gerador combina os seus templates com as informações provenientes dos modelos criados com a DSL para gerar o código-fonte das aplicações. Normalmente, é implementado um template específico para cada característica incluída na DSL do framework. Além disso, também é criado o template principal, que é responsável por invocar os templates específicos das características à medida que são encontrados elementos nos modelos das aplicações.

```
1 <c:iterate select="/Application/features" var="feature">
2   <c:choose select="$feature">
3
4     <c:when test="self::B">
5       <java:class name="{\$feature/@name}" template="templates/B.jet"/>
6     </c:when>
7
8     <c:when test="self::C">
9       <java:class name="{\$feature/@name}" template="templates/C.jet"/>
10    </c:when>
11
12    <c:when test="self::D">
13      <java:class name="{\$feature/@name}" template="templates/D.jet"/>
14    </c:when>
15
16  </c:choose>
17 </c:iterate>
```

Figura 4.8. Exemplo de template principal para o framework mostrado na Figura 4.3.

Na Figura 4.8 é apresentado o código JET do template principal do gerador da DSL do framework H. Nesse template foi implementado um laço (linhas 1 a 17) sobre todos os elementos dos modelos das aplicações, com o intuito de identificar as características às

quais eles correspondem (linhas 4, 8 e 12) e invocar o template específico de cada uma (linhas 5, 9 e 13).

Na Figura 4.9 é apresentado um exemplo de um template para geração das classes Java que reutilizam, por meio de herança (caixa branca), a classe `D` do framework `H`. Na linha 1 as partes fixas desse template são identificadas por palavras-chave da linguagem Java para a declaração da classe e a parte variável corresponde a uma *tag* JET que acessa o nome da classe (`$feature`) no modelo da aplicação. Nas linhas 2 a 5 é criado um laço que gera os atributos definidos para a classe no modelo da aplicação e nas linhas 6 a 9 é gerada a operação `getBClass`.

```

1 public class <c:get select="$feature/@name"/> extends D {
2   <c:iterate select="$feature/attributes" var="att">
3     private <c:get select="$att/@type"/>
4       <c:get select="$att/@name"/>;
5   </c:iterate>
6   public Class<? extends B> getBClass() {
7     return
8       <c:get select="$feature/bClass/@name"/>.class;
9   }

```

Figura 4.9. Trecho do template da característica `D` do framework `H`.

Além dos templates para geração das classes das aplicações, podem ser implementados templates para geração de testes que verificam as funções da aplicação implementadas com o reúso do framework. Além disso, também podem ser implementados templates que geram o *script* do banco de dados e copiam o código do framework, entre outros detalhes.

4.2.2 Engenharia da Aplicação

Na Figura 4.10 é mostrado o modelo da fase de Engenharia da Aplicação com reutilização do framework, com as seguintes etapas: 1) Modelagem da Aplicação; e 2) Construção da Aplicação.

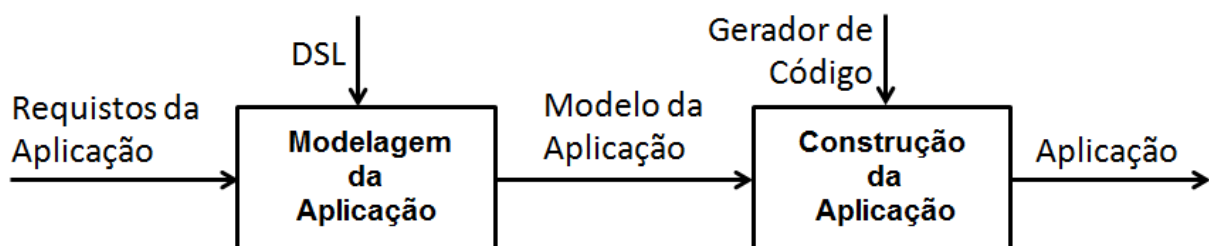


Figura 4.10. Etapas da fase de Engenharia da Aplicação.

4.2.2.1 Modelagem da Aplicação

Na etapa de Modelagem da Aplicação, a DSL do framework desenvolvida na fase de Engenharia da DSL é usada para a criação de um modelo considerando os requisitos da aplicação a ser gerada. Em vez de classes genéricas, os elementos dessa DSL correspondem às características do domínio do framework, facilitando o trabalho de associação desses elementos aos requisitos da aplicação. Para exemplificar esta etapa, na Figura 4.11 é mostrado o uso da DSL do framework H para modelar os requisitos de uma aplicação que o reutiliza. Os estereótipos dos elementos («B», «C», «D») indicam as características do framework utilizadas na aplicação. Por fim, as linhas entre os elementos são inseridas com base nos relacionamentos existentes no metamodelo da DSL (Figura 4.4). Por exemplo, a classe `Dapp` reconhece que `Bapp` é a classe da aplicação que estende a classe `B` do framework por meio da ligação entre as classes `Dapp` e `Bapp`.

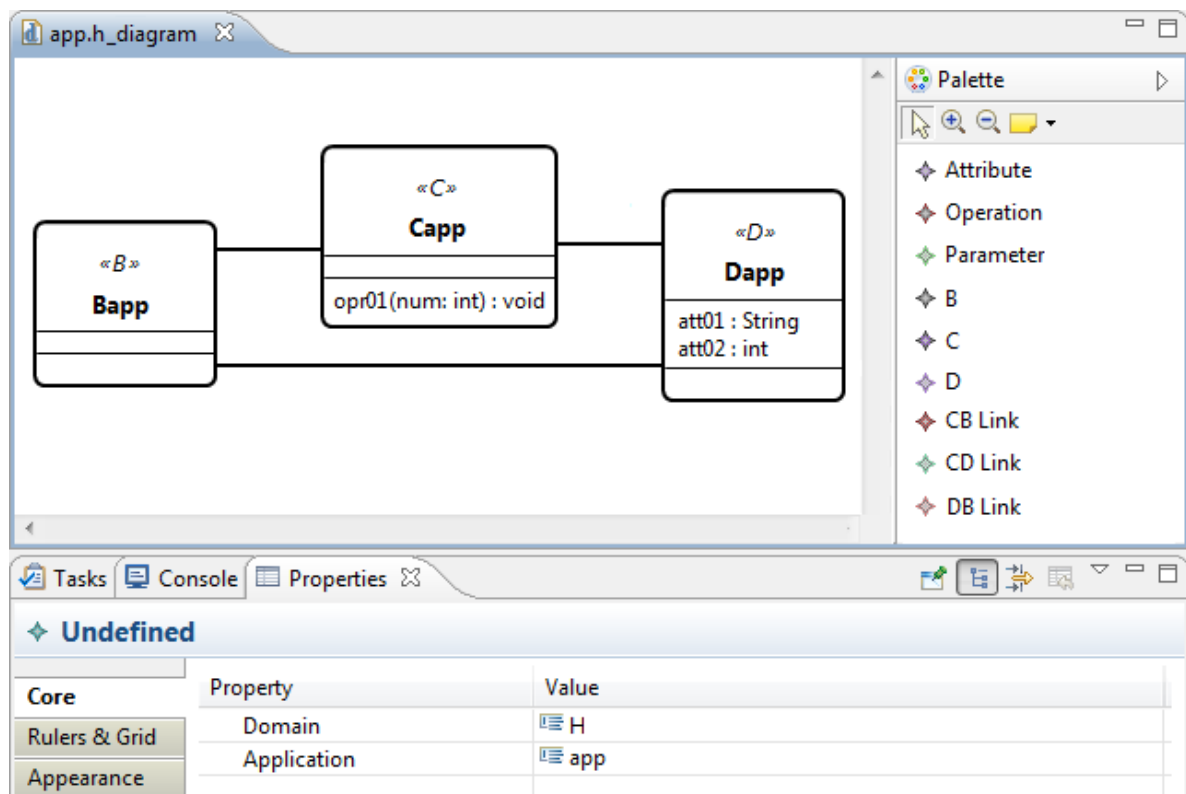


Figura 4.11. Exemplo de utilização da DSL do framework H.

Uma das vantagens das DSLs sobre as linguagens de propósito geral é que as DSLs contém as regras específicas do seu domínio. Um mecanismo de validação dos modelos pode ser usado sobre os modelos das aplicações com o intuito de verificar a presença de características obrigatórias e relacionamentos inválidos. Na Figura 4.12.a é mostrado um modelo de uma aplicação sem a característica B, obrigatória de acordo com a DSL do framework H. Na Figura 4.12.b é mostrado o erro retornado pelo mecanismo de validação.

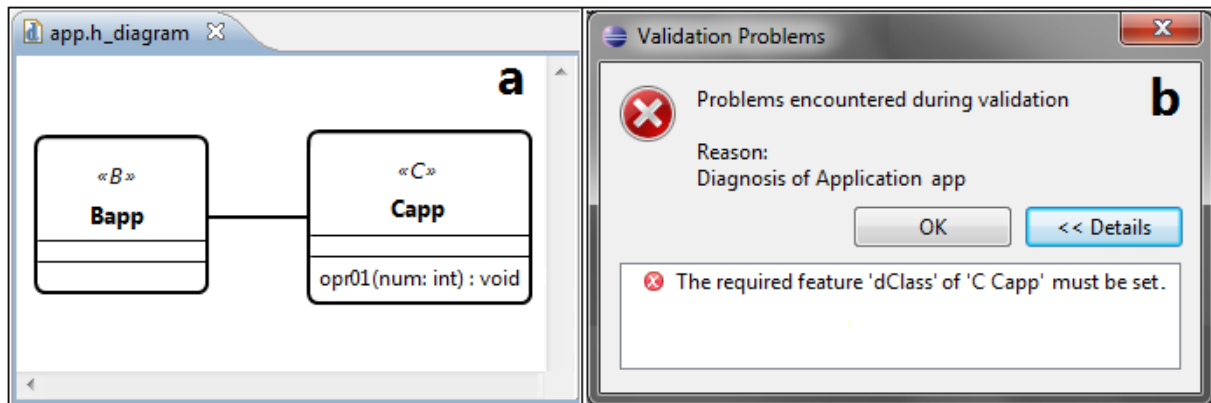


Figura 4.12. Mecanismo de validação das DSLs construídas com o GMF.

4.2.2.2 Construção da Aplicação

Nesta etapa o código-fonte da aplicação é gerado pelo gerador de código da DSL a partir do modelo dessa aplicação. Na Figura 4.13 é mostrado o código-fonte gerado a partir da combinação do template (Figura 4.9) com o modelo apresentado na Figura 4.11.

```
public class Dapp extends D {
    private String att01;
    private int att02;

    public Class<? extends B> getBClass() {
        return Bapp;
    }
}
```

Figura 4.13. Código gerado para a classe Dapp.

Na etapa de Construção da Aplicação também podem ser executados testes para verificar se o código da aplicação atende aos requisitos solicitados. Caso tenham sido criados templates para a geração de classes de teste, esses podem ser executados tão logo o código seja gerado. Caso o código tenha sido modificado manualmente para atender algum requisito não previsto pelo framework, esses testes gerados podem ser utilizados como testes de regressão para verificar se a funcionalidade da aplicação apresenta alguma falha. Também podem ser manualmente implementados e executados testes que verificam a funcionalidade das operações adicionadas no modelo da aplicação.

4.3 Exemplos de Utilização da Abordagem de DSLs de Frameworks

Com o intuito de melhorar o entendimento a respeito da abordagem de construção de DSLs que facilitam a instanciação de frameworks, nesta seção são apresentados dois

exemplos. O primeiro, apresentado na Subseção 4.3.1, refere-se à construção de uma DSL para o framework GRENJ (DURELLI et al., 2010; VIANA, 2009). O segundo, apresentado na Subseção 4.3.2, refere-se à construção de uma DSL para o framework Hibernate (JBOSS COMMUNITY, 2013).

4.3.1 DSL do Framework GRENJ

Nas subseções 4.3.1.1 a 4.3.1.5 são descritos os procedimentos para a realização das etapas da fase de Engenharia da DSL do framework GRENJ. Em seguida, nas Subseções 4.3.1.6 e 4.3.1.7, são apresentadas as etapas da fase de Engenharia da Aplicação em que a DSL construída foi utilizada na instanciação do framework GRENJ em uma aplicação para uma locadora de carros.

4.3.1.1 Engenharia da DSL do framework GRENJ: Identificação das Características

Como foi comentando na Seção 2.3.1, o framework GRENJ foi desenvolvido a partir da linguagem de padrões GRN. Os modelos de classes dos padrões da GRN forneceram uma visão geral das características existentes no framework GRENJ e dos relacionamentos existentes entre essas características. Essa visão foi complementada com consultas ao código-fonte do framework para identificar as demais classes que implementam essas características.

Após a identificação das características, foi definida a sequência na qual elas deviam ser tratadas nas etapas seguintes. A sequência proposta pelos padrões da GRN não pode ser adotada por duas razões: 1) em alguns padrões foram identificadas mais de uma característica; e 2) foram encontradas dependências entre características identificadas em diferentes padrões. Por exemplo, a característica Recurso da Transação, identificada no primeiro padrão da GRN, depende da característica Quantificação do Recurso, identificada no segundo padrão. Nessas duas situações, a sequência de tratamento das características foi definida priorizando-se as características que não dependiam de outras, como descrito na Seção 4.2.1.1.

Na Figura 4.14 é mostrado o modelo das classes do framework GRENJ referente ao primeiro padrão da GRN, Identificar o Recurso. Nesse padrão foram identificadas as características Recurso da Transação, Tipo Simples do Recurso e Tipo Aninhado do Recurso, cujas classes principais são, respectivamente, `Resource`, `SimpleType` e `NestedType`. A característica Recurso da Transação foi tratada por último, pois a classe `Resource` referencia as classes `SimpleType` e `NestedType` por meio do relacionamento `types`, herdado da classe `QualifiableObject`, para identificar os tipos do recurso. A

classe `NestedType` também utiliza esse relacionamento para identificar os subtipos de um tipo aninhado do recurso, portanto, a característica Tipo Aninhado do Recurso foi tratada em segundo lugar. Apesar de também herdar o relacionamento `types`, a classe `SimpleType` não o utiliza, pois as operações `typeFieldInitialize` e `typeClasses` foram sobrescritas nessa classe de modo a impedir que os tipos simples possuam subtipos. Assim, a característica Tipo Simples do Recurso foi tratada em primeiro lugar.

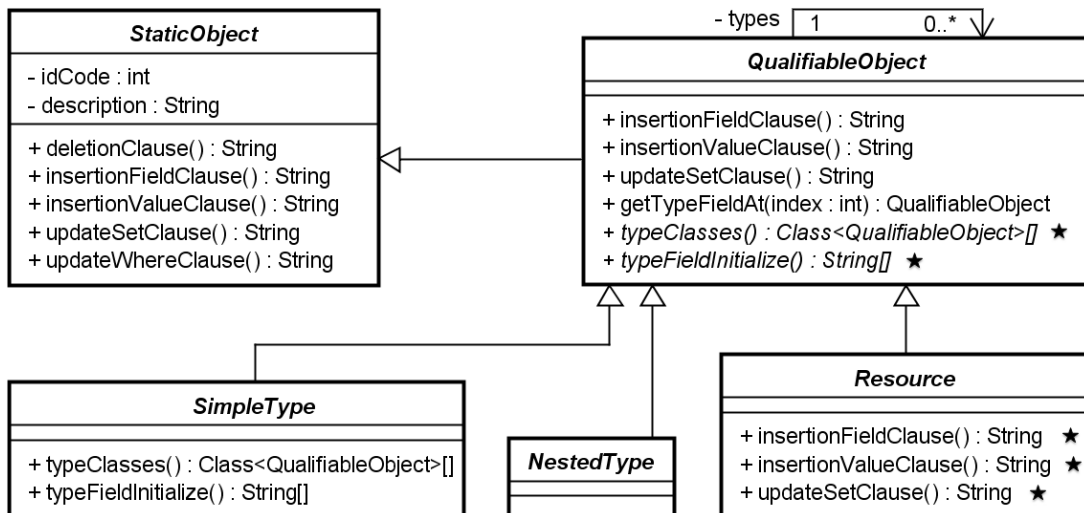


Figura 4.14. Classes do framework GRENJ que implementam o Padrão 01 da GRN.

No Quadro 4.3 é mostrada uma lista parcial de algumas características do framework GRENJ, que possibilitam o desenvolvimento de aplicações que tratam de transações de aluguel. Os números na primeira coluna indicam a sequência em que as características devem ser tratadas nas etapas seguintes.

Quadro 4.3. Algumas das características do framework GRENJ.

#	Nome da Característica	Classes no Código-Fonte
1	Tipo Simples do Recurso	SimpleType, QualifiableObject, StaticObject
2	Tipo Aninhado do Recurso	NestedType, QualifiableObject, StaticObject
3	Quantificação do Recurso	SingleResource, ResourceInstance, InstantiableResource, QuantificationStrategy
4	Recurso da Transação	Resource, QualifiableObject, StaticObject, SimpleType, NestedType
5	Destino do Recurso na Transação	DestinationParty, StaticObject
6	Transação de Aluguel	ResourceRental, BusinessResourceTransaction, Resource, DestinationParty

4.3.1.2 Engenharia da DSL do framework GRENJ: Análise das Características

Nesta etapa foi realizada uma análise no código-fonte do framework GRENJ e de aplicações que reutilizam esse framework com o objetivo de obter as informações

necessárias para a instanciação de cada característica identificada na etapa anterior. Como o framework GRENJ é caixa branca, essa análise identifica as informações necessárias para que suas operações sejam sobrescritas. Por exemplo, durante a análise da característica Recurso da Transação, analisou-se o código da classe `Resource`, de suas superclasses e de classes de aplicações que a estendem para identificar quais informações são necessárias para sobrescrever suas operações.

Na Figura 4.15 é mostrado o código de duas operações da classe `Resource`: `typeClasses`, que é abstrata e indica as classes que representam os tipos dos recursos nas aplicações; e `insertFieldClause`, que indica os nomes dos campos na base de dados que armazenam os dados dos atributos específicos da classe que estende `Resource`. Essas operações devem ser sobrescritas pelas classes que estendem `Resource` nas aplicações.

```
/**
 * Implementations of this method return a list with the
 * classes (.class) that represents the types, or null
 * if the object is an instance of {@link SimpleType}.
 */
public abstract Class[] typeClasses();

/**
 * Returns the field names of the insert clauses.
 */
@Override
public String insertFieldClause() {
    StringBuilder str = new StringBuilder( super.insertFieldClause() );
    String clause = quantification.insertFieldClause();
    if ( clause != null )
        str.append( ", " + clause );
    return strBuilder.toString();
}
```

Figura 4.15. Operações da classe `Resource` que devem ser sobrescritas.

Considere uma aplicação para locação de filmes que reutiliza o framework GRENJ. Na Figura 4.16 é mostrado o código da classe `Movie`, que estende a classe `Resource` e sobrescreve as operações mostradas na Figura 4.15. É possível observar que a operação `typeClasses` foi sobrescrita para informar para o framework quais classes implementam os tipos do recurso (`Category` e `Genre`) nessa aplicação. Constatou-se, então, que as classes que representam os tipos do recurso nas aplicações são informações necessárias para estender a classe `Resource`. De forma semelhante, a operação `insertFieldClause` foi sobrescrita para que o framework reconhecesse o atributo `year`

da classe `Movie` durante a gravação de dados no banco de dados da aplicação. Portanto, os atributos das classes da aplicação também são informações necessárias para a instanciação do framework GRENJ.

```
public class Movie extends Resource {
    private int year;

    @Override
    public Class<? extends QualifiableObject>[] typeClasses() {
        return new Class[] { Category.class, Genre.class };
    }

    @Override
    public String insertionFieldClause() {
        StringBuilder newInsertionFieldClause =
            new StringBuilder( super.insertionFieldClause() );
        newInsertionFieldClause.append( ", year" );
        return newInsertionFieldClause.toString();
    }
}
```

Figura 4.16. Trecho do código da classe `Movie`.

No Quadro 4.4 estão listadas as características do framework GRENJ, incluindo as informações necessárias para a reutilização das suas classes principais.

Quadro 4.4. Informações sobre sete características do framework GRENJ.

#	Característica	Classes Representantes	Informações para Instanciação
0	Todas as características	-	Nome da classe que representa a característica na aplicação, seus atributos, suas operações e os parâmetros dessas operações.
1	Tipo Simples do Recurso	<code>SimpleType</code> , <code>QualifiableObject</code> , <code>StaticObject</code>	-
2	Tipo Aninhado do Recurso	<code>NestedType</code> , <code>QualifiableObject</code> , <code>StaticObject</code>	As classes que representam os seus subtipos.
3	Quantificação do Recurso	<code>SingleResource</code> , <code>ResourceInstance</code> , <code>InstantiableResource</code> , <code>QuantificationStrategy</code>	O recurso é simples ou instanciável?
4	Recurso da Transação	<code>Resource</code> , <code>QualifiableObject</code> , <code>StaticObject</code> , <code>SimpleType</code> , <code>NestedType</code>	As classes que representam os tipos do recurso e a sua quantificação.
5	Destino do Recurso na Transação	<code>DestinationParty</code> , <code>StaticObject</code>	-
6	Transação de Aluguel	<code>ResourceRental</code> , <code>BusinessResourceTransaction</code> , <code>Resource</code> , <code>DestinationParty</code>	As classes que representam o recurso e o destino das transações.

4.3.1.3 Engenharia da DSL do framework GRENJ: Modelagem da DSL

Nas Figuras 4.17.a até 4.17.f é mostrado como foi realizada a inclusão das características do framework GRENJ no metamodelo de sua DSL.

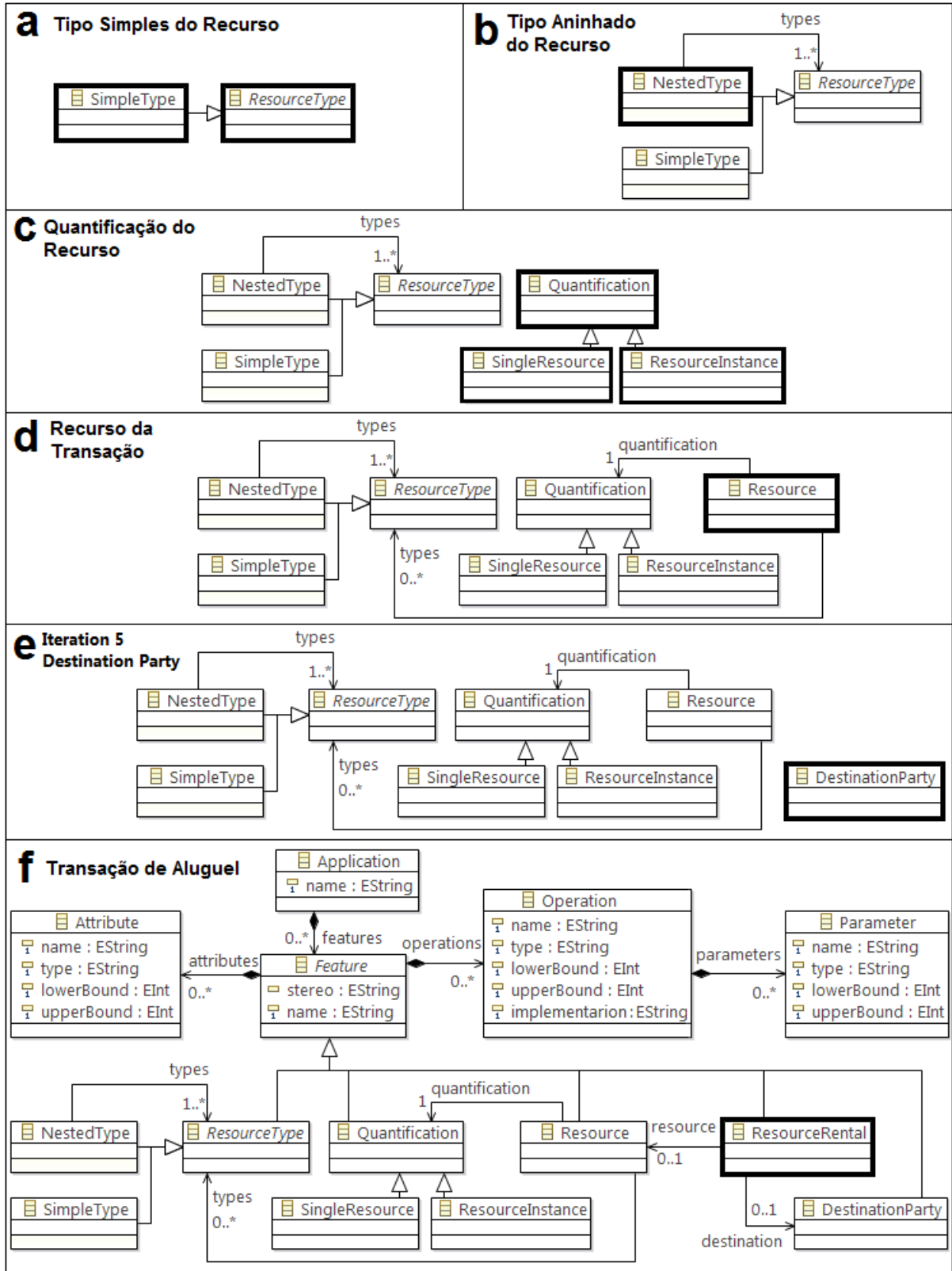


Figura 4.17. Sequência de criação do metamodelo da DSL do framework GRENJ.

As classes em destaque nas Figuras 4.17.a até 4.17.f indicam as metaclasses que foram incluídas no metamodelo em cada passo da sequência, de acordo com o Quadro 4.3. As informações necessárias para instanciação do framework são definidas no metamodelo por meio de atributos ou relacionamentos entre as metaclasses. Por exemplo, o relacionamento `quantification` entre `Resource` e `Quantification` possui multiplicidade igual a 1, indicando que é obrigatório informar a quantificação do recurso nos modelos criados com a DSL do framework GRENJ. As metaclasses que permitem a inclusão de atributos, operações e parâmetros nas classes das aplicações são mostradas na Figura 4.17.f.

O modelo *Gmfgraph* que define a notação gráfica da DSL do framework GRENJ é semelhante ao mostrado na Figura 4.5 (Seção 4.2.1.3). Nesse modelo foi definido que as características do framework são representadas graficamente por um *Rounded Rectangle*, que é composto por outros retângulos com os rótulos do estereótipo, do nome, dos atributos e das operações. Os relacionamentos de associação entre as características foram definidos por linhas contínuas.

Na Figura 4.18.a é apresentado o modelo *Gmftool*, no qual foram definidos os itens de menu da DSL do framework GRENJ. Na Figura 4.18.b é mostrado o modelo de *Gmfmap*, no qual foi feita a junção das características incluídas no metamodelo com a notação gráfica e com os itens de menu da DSL do framework GRENJ.

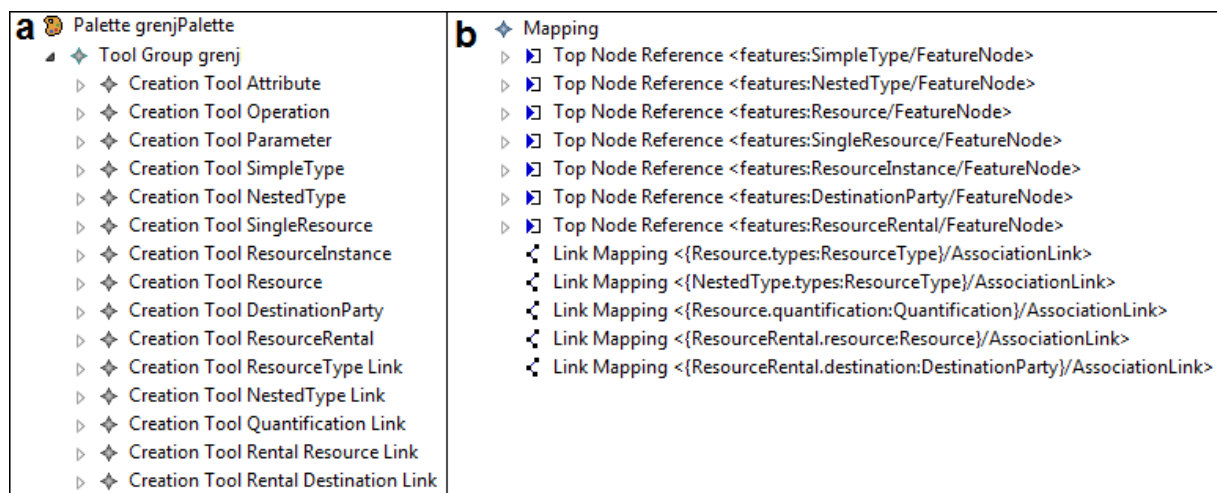


Figura 4.18. Conteúdo dos modelos (a) Gmftool e (b) Gmfmap do framework GRENJ.

4.3.1.4 Engenharia da DSL do framework GRENJ: Construção da DSL

Como o GMF foi utilizado durante o projeto da DSL, a implementação da DSL do framework GRENJ foi realizada com a geração do código dos *plug-ins* para o Eclipse IDE que contêm a funcionalidade das sintaxes abstrata e concreta dessa DSL.

4.3.1.5 Engenharia da DSL do framework GRENJ: Construção do Gerador

A implementação dos templates para a geração do código-fonte das aplicações modeladas com a DSL do framework GRENJ foi realizada com o uso do JET. As partes fixas e variantes desses templates foram identificadas a partir da análise do código-fonte de aplicações que o framework GRENJ instancia. Na Figura 4.19 é mostrada uma parte do template das subclasses de `ResourceRental`. As partes fixas desse template são formadas por comandos Java, por exemplo, as palavras-chave `public class`. As partes variantes são formadas por tags, como, por exemplo, o nome da classe, os atributos, a implementação das operações das aplicações e outras declarações. O conteúdo da operação `getResourceClass` é gerado a partir do nome da classe que é referenciada pelo relacionamento `resource` da metaclassa `ResourceRental` no metamodelo da Figura 4.17.

```
public class <c:get select="$feature/@name"/> extends ResourceRental {
    public Class<? extends Resource> getResourceClass() {
        return <c:get select="$feature/resource/@name"/>.class;
    }
    <c:iterate select="$feature/operations" var="opr">
        public <c:get select="$opr/@type"/> <c:get select="$opr/@name"/>{
            <c:iterate select="$opr/parameters" var="prm" delimiter=", ">
                <c:get select="$prm/@type"/> <c:get select="$prm/@name"/>
            </c:iterate> } {
                <c:get select="$opr/@implementation"/>
            }
        }
    </c:iterate>
}
```

Figura 4.19. Parte do template para subclasses de `ResourceRental`.

4.3.1.6 Engenharia da Aplicação com o Framework GRENJ: Modelagem da Aplicação de Aluguel de Carros

Para exemplificar o uso da DSL do framework GRENJ, foi desenvolvida a aplicação de Aluguel de Carros, considerando os requisitos funcionais apresentados no Quadro 4.5.

Quadro 4.5. Requisitos funcionais da aplicação de Aluguel de Carros.

#	Descrição
1	A loja aluga carros para clientes cadastrados. Os atributos de um carro são: código, descrição e número de portas.
2	Pode haver mais de um veículo de cada carro. Um veículo possui código, localização, placa, ano, cor e status.
3	Os carros são classificados por categoria, com código, nome e valor do aluguel (VA).
4	O cadastro dos clientes possui código, nome, endereço e telefone.
5	As informações sobre o alugueis de carros são: número, código, data do aluguel (DA), data de retorno esperada (DE), data de retorno efetiva (DR), cliente, veículo e valor total (VT). Esse valor total é calculado da seguinte forma: $VT = VA(DR - DA) + VA(DR - DE)$.

Na Figura 4.20 é mostrado o modelo da aplicação de Aluguel de Carros criado com o uso da DSL do framework GRENJ a partir dos requisitos apresentados no Quadro 4.5. Parte dos atributos especificados nesses requisitos não é acrescentada nesse modelo, pois o framework GRENJ os fornece, como mostrado na aba *Properties*. Os estereótipos dos elementos referenciam as características do framework utilizadas na aplicação.

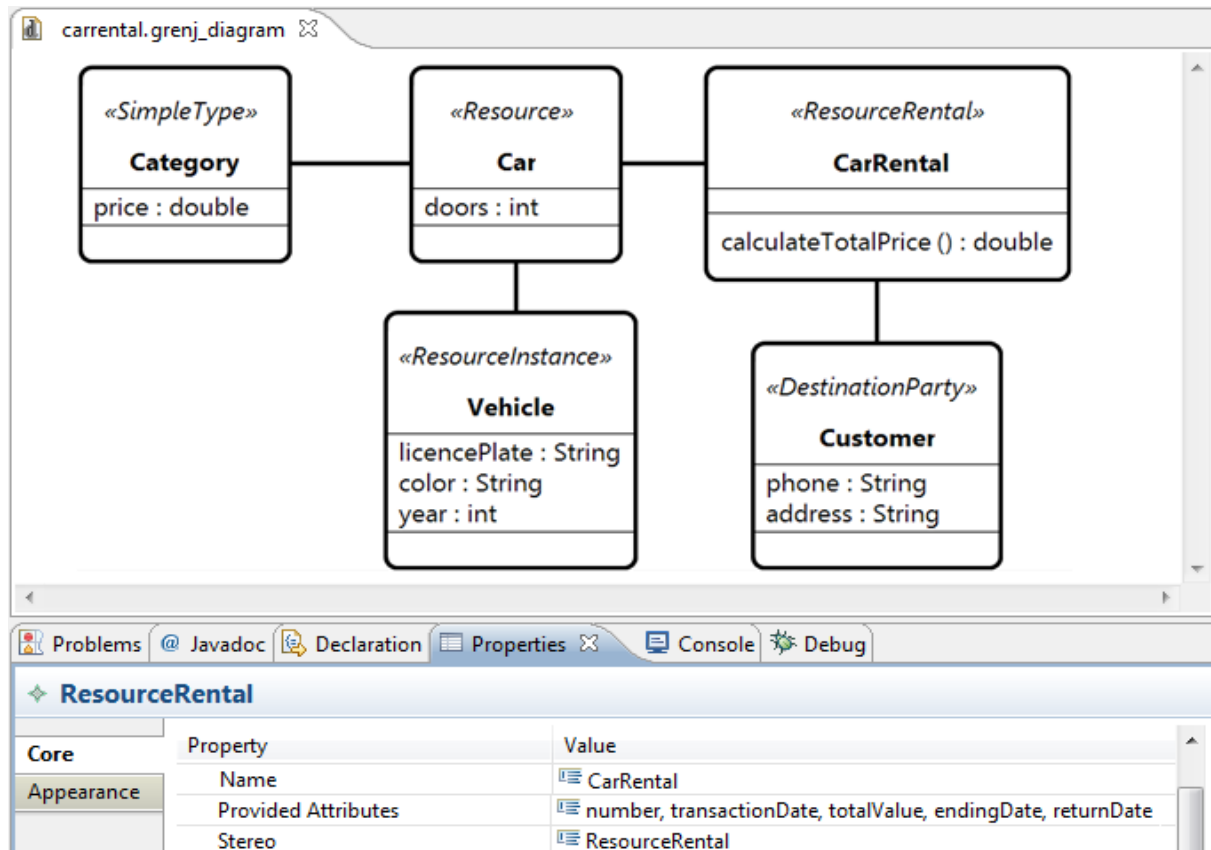


Figura 4.20. Modelo da aplicação de Aluguel de Carros.

4.3.1.7 Engenharia da Aplicação com o Framework GRENJ: Construção da Aplicação de Aluguel de Carros

O código-fonte da aplicação de Aluguel de Carros foi gerado a partir do modelo da Figura 4.20 por meio do gerador de aplicações da DSL do framework GRENJ. Cada elemento do modelo originou uma classe que estende a classe do framework, indicada pelos estereótipos no modelo. Por exemplo, a classe `CarRental` da aplicação de Aluguel de Carros estende a classe `ResourceRental` do framework GRENJ. Parte do código da classe `CarRental` é mostrada na Figura 4.21. Esse código foi gerado a partir do modelo da Figura 4.20 e do template da Figura 4.19. O código das operações definidas nos modelos das aplicações, como `calculateTotalPrice`, pode ser digitado na propriedade `implementation` de `Operation`. Essa propriedade não é graficamente visualizada nos modelos das aplicações, mas está presente na aba *Properties* do editor da DSL.

```

public class CarRental extends ResourceRental {

    public Class<? extends Resource> getResourceClass() {
        return Car.class;
    }

    public double calculateTotalPrice() {
        Category category = (Category) getResource().getTypes().get(0);
        int drda = Period.numberofDays(getDate(), getReturningDate());
        int drde = Period.numberofDays(getFinishDate(), getReturningDate());
        return category.getPrice() * drda + category.getPrice() * drde;
    }
}

```

Figura 4.21. Parte do código gerado para a classe CarRental.

4.3.2 DSL do Framework Hibernate

Nas subseções 4.3.2.1 a 4.3.2.5 encontra-se a descrição de como foram realizadas as etapas da fase de Engenharia da DSL da funcionalidade de persistência do framework Hibernate. Nas Subseções 4.3.2.6 e 4.3.2.7 são apresentados detalhes das etapas da realização da fase de Engenharia da Aplicação em que a DSL do framework Hibernate foi utilizada no desenvolvimento da aplicação para Aluguel de Carros.

4.3.2.1 Engenharia da DSL do Framework Hibernate: Identificação das Características

O framework Hibernate é caixa preta e sua instanciação é feita por meio de anotações, a partir das quais a identificação das características foi realizada. Com relação à funcionalidade de persistência desse framework, foram identificadas duas características: Entity (Entidade), que indica uma classe, cujos dados dos objetos devem ser persistidos no banco de dados; e Named Query, que são consultas que podem ser definidas nas classes para a recuperação dos dados no banco de dados. A lista de características do framework Hibernate é apresentada no Quadro 4.6. Como uma Named Query não existe sem a sua Entity, essa última foi priorizada na sequência de criação da DSL do framework Hibernate.

Quadro 4.6. Características de persistência do framework Hibernate.

#	Característica	Anotações Representantes
1	Entity	@Entity
2	Named Query	@NamedQuery

4.3.2.2 Engenharia da DSL do Framework Hibernate: Análise das Características

A partir da análise da documentação e de exemplos de utilização do framework Hibernate tem-se que uma Entity está vinculada a uma tabela no banco de dados e pode estar associada ou estender outras Entities. Uma Entity pode ter uma ou mais Named

Queries para recuperação de dados. No Quadro 4.7 é apresentada a lista de características do framework Hibernate com as informações necessárias para a sua instanciação. As Entities não podem ter outras operações além daquelas exigidas pelo framework.

Quadro 4.7. Características de persistência do framework Hibernate após análise.

#	Característica	Anotações Representantes	Informações para Instanciação
1	Entity	Entity	Nome, tabela, atributos e associações. URL, esquema, usuário e senha do BD. Possui uma super entidade?
2	Named Query	NamedQuery	Nome e consulta.

4.3.2.3 Engenharia da DSL do Framework Hibernate: Projeto da DSL

O projeto da DSL do framework Hibernate também foi desenvolvido com o uso do GMF. Na Figura 4.22 é mostrado o metamodelo da DSL do framework Hibernate, criado a partir da lista das suas características. As metaclasses `Entity` e `NamedQuery` armazenam as informações dessas características. Como os atributos e as associações das Entidades possuem diversas informações, as metaclasses auxiliares `Attribute` e `Association` foram criadas, respectivamente, para representá-los. Como o GMF requer um elemento raiz para conter todos os elementos dos modelos de aplicações, foi criada a metaclasses `Application`. Como todas as entidades de uma aplicação acessam o mesmo banco de dados, as informações referentes à conexão com o SGBD foram inseridas nessa metaclasses. As enumerações `GeneratedValueStrategy` e `InheritanceStrategy` contêm os valores válidos para as estratégias de tratamento dos atributos identificadores e dos relacionamentos de herança dos drivers dos SGBDs.

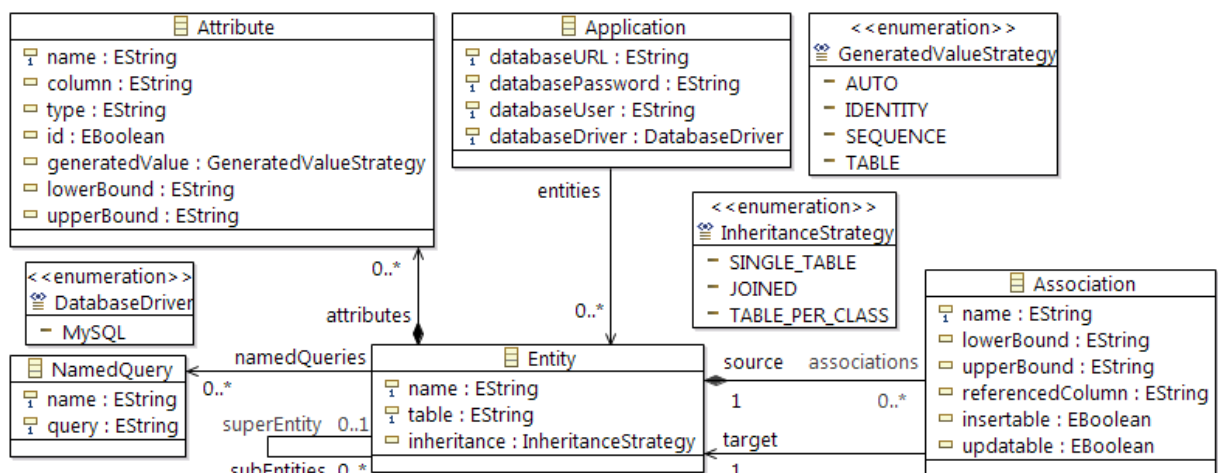


Figura 4.22. Metamodelo da DSL do framework Hibernate.

O modelo *Gmfgraph* que define a notação gráfica da DSL do framework Hibernate é mostrado na Figura 4.23. Nesse modelo a característica Entity é representada

graficamente por um retângulo (*Rectangle EntityFigure*) composto de: um retângulo (*Rectangle NameArea*) com o rótulo do nome da Entitiy; um retângulo (*Rectangle TableArea*) com o rótulo da tabela de banco de dados da Entitiy; um retângulo (*Rectangle NamedQueryCompartment*) para as Named Queries; e um retângulo (*Rectangle AttributeCompartment*) para os atributos. Os relacionamentos de herança são formados por uma linha contínua com um polígono triangular na sua extremidade final (*Polyline Connection InheritanceFigure*). Por fim, os relacionamentos de associação são formados por uma linha contínua com uma seta na sua extremidade final (*Polyline Connection AssociationFigure*).

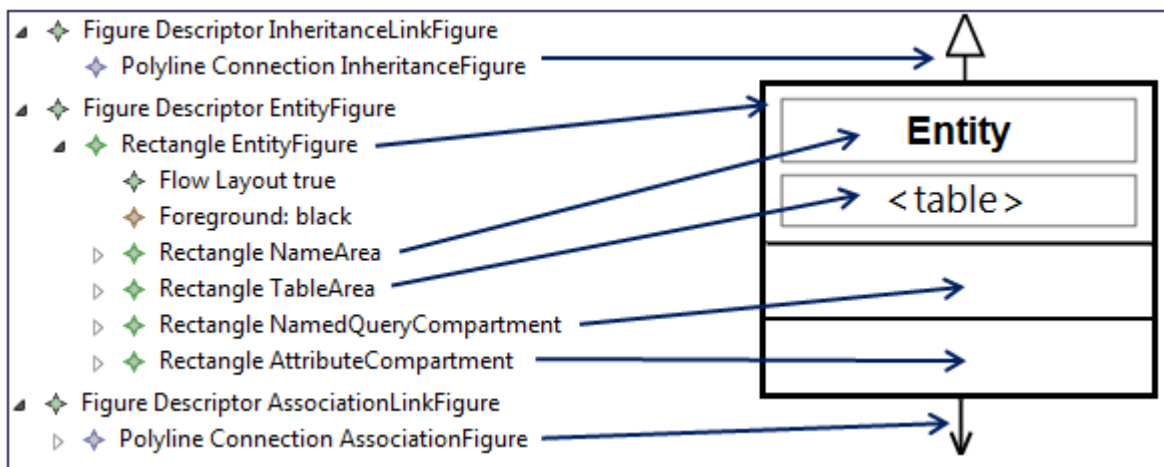


Figura 4.23. Modelo Gmfigraph da DSL do framework Hibernate.

Na Figura 4.24.a é mostrado o modelo *Gmftool* da DSL do framework Hibernate, na qual foram definidos os itens de menu das características Entity e Named Query, dos atributos e dos relacionamento de herança e de associação. Na Figura 4.24.b é mostrado o modelo *Gmfmap* que faz a ligação entre os elementos do metamodelo, os elementos da notação gráfica e os itens de menu da DSL do framework Hibernate.

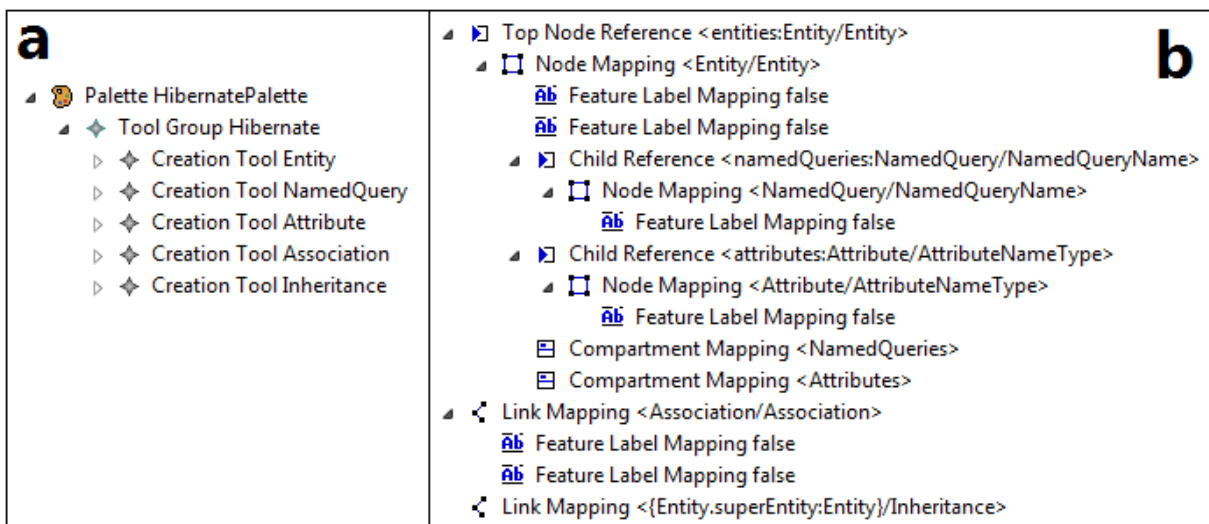


Figura 4.24. Modelos (a) Gmftool e (b) Gmfmap da DSL do framework Hibernate.

4.3.2.4 Engenharia da DSL do Framework Hibernate: Construção da DSL

Como a DSL do framework Hibernate também foi projetada com o uso do GMF, o seu código foi gerado na forma de *plug-ins* para o Eclipse IDE. Esses *plug-ins* devem ser carregados na IDE para que a DSL possa ser utilizada para a modelagem de aplicações que reutilizam o framework Hibernate.

4.3.2.5 Engenharia da DSL do framework Hibernate: Construção do Gerador

Três templates foram implementados para a geração do código das aplicações que reutilizam o framework Hibernate: 1) o template das entidades, que gera as classes de entidade da aplicação; 2) o template de persistência, que gera o arquivo XML utilizado pelo Hibernate para ter acesso ao banco de dados e identificar as classes de entidade da aplicação; e 3) o template principal, que realiza um incremento sobre os elementos incluídos nos modelos criados com a DSL, invocando o template das entidades para cada um deles.

As classes geradas a partir do template das entidades possuem as anotações que o Hibernate utiliza para estabelecer a comunicação entre os objetos dessas classes e o banco de dados, como mostrado na Figura 4.25. Esse template foi criado a partir de exemplos de aplicações que reutilizam o framework Hibernate.

```

1 @Entity
2 @Table(name = "<c:get select=\"$entity/@table\"/>",
3       catalog = "<c:get select=\"/Application/@databaseName\"/>", schema = "")
4 @Inheritance(strategy= <c:get select="$entity/@inheritance\"/>)
5 @XmlElement
6 @NamedQueries({<c:iterate select="$entity/namedQueries" var="nQuery" delimiter=",">
7     @NamedQuery(name = "<c:get select="$nQuery/@name\"/>",
8     query = "<c:get select="$nQuery/@query\"/>")</c:iterate>})
9 public class <c:get select="$entity/@name\"/> <c:if test="$entity/superEntity">
10 extends <c:get select="$entity/superEntity/@name\"/> </c:if>
11 implements Serializable {
12
13     private static final long serialVersionUID = 1L;
14
15     <c:iterate select="$entity/attributes" var="attribute">
16     <c:if test="$attribute/[@id = 'true']">@Id
17     @GeneratedValue(strategy = <c:get select="$attribute/@generatedValue\"/>)</c:if>
18     @Column(name = "<c:get select="$attribute/@column\"/>")
19     private <c:get select="$attribute/@type\"/> <c:get select="$attribute/@name\"/>;
20     </c:iterate>

```

Figura 4.25. Parte do template que gera as classes de entidade.

4.3.2.6 Engenharia da Aplicação com o framework Hibernate: Modelagem da Aplicação de Aluguel de Carros

A DSL do framework Hibernate foi usada para desenvolver uma outra versão da aplicação de Aluguel de Carros, com os mesmos requisitos apresentados no Quadro 4.5. Nesse caso, *Car*, *Category*, *Vehicle*, *Customer* e *CarRental* foram identificadas como entidades da aplicação que devem ser persistidas no banco de dados.

O modelo da aplicação de Aluguel de Carros criado com a DSL do framework Hibernate é mostrado na Figura 4.26. Cada *Entity* nesse modelo está dividida em três partes: 1) na parte superior encontra-se o nome da *Entity* escrito em negrito e o nome da tabela correspondente no banco de dados escrito entre < e >; 2) na parte central encontram-se as *Named Queries*, que possuem o prefixo *findBy*; e 3) na parte inferior, encontram-se os atributos.

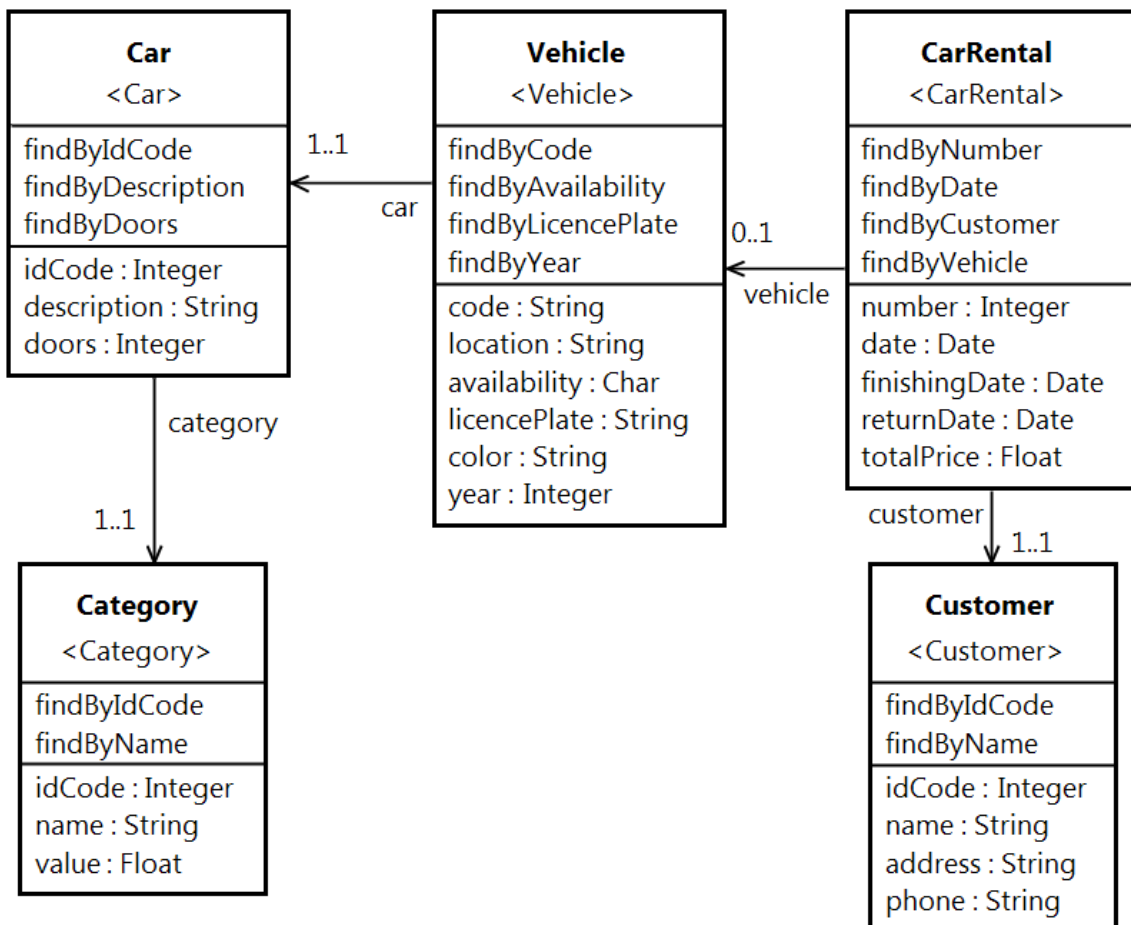


Figura 4.26. Modelo da aplicação de Aluguel de Carros criado com a DSL do framework Hibernate.

4.3.2.7 Engenharia da Aplicação com o framework Hibernate: Construção da Aplicação de Aluguel de Carros

O gerador de aplicações da DSL do framework Hibernate foi acionado a partir do modelo mostrado na Figura 4.26 para gerar o código da aplicação de Aluguel de Carros. Cada entidade do modelo originou uma classe que contém anotações referentes ao uso do framework Hibernate.

Parte do código da classe `Car`, após ter sido gerada com base nas informações do modelo da Figura 4.26 e do template da Figura 4.25, é mostrado na Figura 4.27. As anotações existentes nessa classe configuram o reuso do framework Hibernate. Por exemplo, a anotação `@Entity` indica que os dados dos objetos da classe `Car` devem ser persistidos no banco de dados. Cada anotação `@NamedQuery` define um tipo de consulta no banco de dados. Os atributos da classe possuem a anotação `@Column` para indicar que correspondem a colunas da tabela `Car` no banco de dados.

```
@Entity
@Table(name = "Car", catalog = "carrental", schema = "")
@XmlRootElement
@NamedQueries({
    @NamedQuery(name = "Car.findByIdCode",
        query = "SELECT c FROM Car c WHERE c.idCode = :idCode"),
    @NamedQuery(name = "Car.findByDescription",
        query = "SELECT c FROM Car c WHERE c.description = :description"),
    @NamedQuery(name = "Produto.findByDoors",
        query = "SELECT c FROM Car c WHERE c.doors = :doors"))
public class Car implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "idCode")
    private Integer idCode;

    @Column(name = "description")
    private String description;

    @Column(name = "doors")
    private Integer doors;
```

Figura 4.27. Parte do código da classe `Car` com o reuso do framework Hibernate.

4.4 Considerações Finais

Neste capítulo foi apresentada uma abordagem para a construção de DSLs com o intuito de facilitar a instanciação de frameworks para o desenvolvimento de aplicações.

Uma ferramenta CASE é elaborada com a construção de uma DSL para a modelagem das aplicações com o uso dessa DSL. Além disso, templates são construídos com o uso de uma linguagem de transformação *Model-to-Text* (M2T) com o intuito de prover a funcionalidade de geração do código-fonte das aplicações modeladas com essa ferramenta CASE.

A existência de elementos significativos para o domínio, em vez de classes genéricas, facilita a modelagem das aplicações. Além disso, as DSLs reduzem o risco do desenvolvedor configurar incorretamente os pontos variáveis do framework. Isso se justifica pelo fato de nas DSLs terem sido implementadas as regras e os relacionamentos do domínio, o que impossibilita a existência de relacionamentos inválidos nos modelos das aplicações. Conseqüentemente, a inserção de defeitos no código das aplicações é reduzida.

Outra vantagem proporcionada pelo uso da DSL do framework é a eficiência obtida na instanciação das aplicações, pois o desenvolvedor não necessita implementar manualmente o código-fonte dessas aplicações. Entretanto, a ferramenta CASE possibilita ao desenvolvedor realizar, manualmente, modificações no código gerado, como por exemplo, a implementação do código interno dos métodos específicos das aplicações. Essas modificações podem ser mantidas mesmo que o código das aplicações seja gerado novamente a partir dos modelos.

Com os exemplos apresentados neste capítulo, percebeu-se que a abordagem de construção de DSLs se adapta melhor a frameworks específicos de domínio, como o framework GRENJ, pois as características são identificadas mais facilmente nesses frameworks. Contudo, também pode se aplicar a outros tipos de frameworks, como o Hibernate. Assim, a DSL pode facilitar a implementação de uma funcionalidade, como a persistência de dados, ou servir como meio de configuração de produtos de uma linha de produtos de software com base em um framework.

A construção da DSL de um framework requer que seu desenvolvedor tenha conhecimento sobre o domínio, as classes e os pontos fixos e variantes desse framework. A documentação do framework auxilia nesse processo, mas o seu código deve ser sempre analisado para confirmar a real implementação e como ocorre o seu funcionamento.

No Capítulo 5 é apresentada uma ferramenta desenvolvida para apoiar o uso das abordagens F3 e de construção de DSLs de frameworks. Dessa forma, auxiliando tanto no desenvolvimento quanto no reúso de frameworks. No Capítulo 6 são discutidos alguns experimentos que foram realizados para avaliar essas abordagens e sua ferramenta.

Capítulo 5

A FERRAMENTA F3T

5.1 Considerações Iniciais

Abordagens de desenvolvimento de software fornecem diretrizes para que os desenvolvedores construam o produto final de forma facilitada e com qualidade. Essas abordagens podem considerar, dentre outros, o reuso de recursos, como frameworks e padrões, para apoiar esse desenvolvimento.

No Capítulo 3 foi apresentada a abordagem F3 que facilita o desenvolvimento de frameworks a partir da modelagem de um domínio e do uso de padrões que indicam quais unidades de código devem ser construídas para compor um framework para esse domínio. Essa abordagem não faz uso de recursos específicos, porém, apesar de facilitar o trabalho do desenvolvedor indicando como o framework deve ser construído, ainda é manual.

No Capítulo 4 foi apresentada uma abordagem de construção de DSLs para facilitar o reuso de frameworks. Essa abordagem faz uso de metalinguagens e linguagens de transformação para a construção de editores para as DSLs dos frameworks e de geradores do código-fonte das aplicações que reutilizam esses frameworks. A facilidade na reutilização dos frameworks, obtida com essa abordagem, possui um custo inicial para a construção da DSL e do gerador de aplicações. Além disso, é necessário conhecimento sobre a metalinguagem e a linguagem de transformações escolhidas para a construção da DSLs e seu gerador de aplicações.

Neste capítulo é apresentada a *From Features to Frameworks Tool (F3T)*, uma ferramenta que apoia o uso da abordagem F3 no desenvolvimento de frameworks e da abordagem de construção de DSLs para o reuso desses frameworks (VIANA et al., 2013d). Desse modo, além das vantagens oferecidas pelas abordagens descritas nos Capítulos 3 e 4, a F3T proporciona maior facilidade e eficiência uma vez que automatiza algumas etapas dessas abordagens.

Os detalhes da F3T estão apresentados nas seções que compõem este capítulo: a forma como a F3T foi construída é descrita na Seção 5.2; um exemplo de utilização dessa ferramenta para a construção e reuso de um framework é apresentado na Seção 5.3; e na Seção 5.4 são comentadas as considerações finais deste capítulo.

5.2 Construção da F3T

A F3T foi construída para ser utilizada em conjunto com os demais recursos oferecidos pelo ambiente de desenvolvimento Eclipse IDE (THE ECLIPSE FOUNDATION, 2013a). Os plug-ins da F3T são organizados em três módulos de acordo com a sua funcionalidade:

- 1) **Módulo de Domínios**, que é responsável pela modelagem dos domínios dos frameworks;
- 2) **Módulo de Frameworks**, que é responsável por gerar o código-fonte e a DSL dos frameworks; e
- 3) **Módulo de Aplicações**, que é responsável por gerar o código-fonte das aplicações.

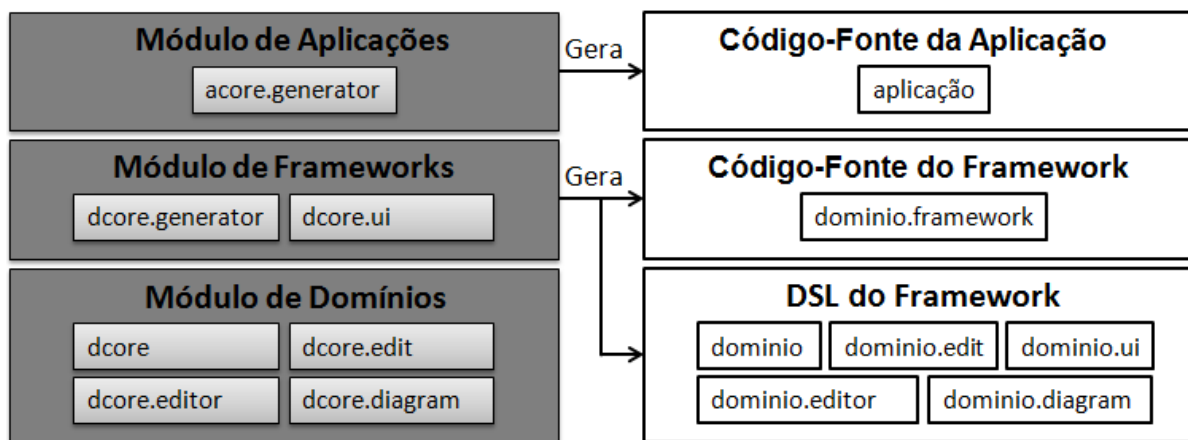


Figura 5.1. Módulos da F3T.

5.2.1 Módulo de Domínios

O Módulo de Domínios constitui um editor gráfico para modelos F3. Esse módulo foi desenvolvido com o apoio do *Eclipse Modeling Framework* (EMF) e do *Graphical Modeling Framework* (GMF) (THE ECLIPSE FOUNDATION, 2013b; GRONBACK, 2009). O EMF foi utilizado para criar a sintaxe abstrata dos modelos F3 definida no metamodelo da Figura 5.2. Os *plug-ins* *dcore*, *dcore.edit* e *dcore.editor* da F3T (Figura 5.1) foram gerados a partir desse metamodelo.

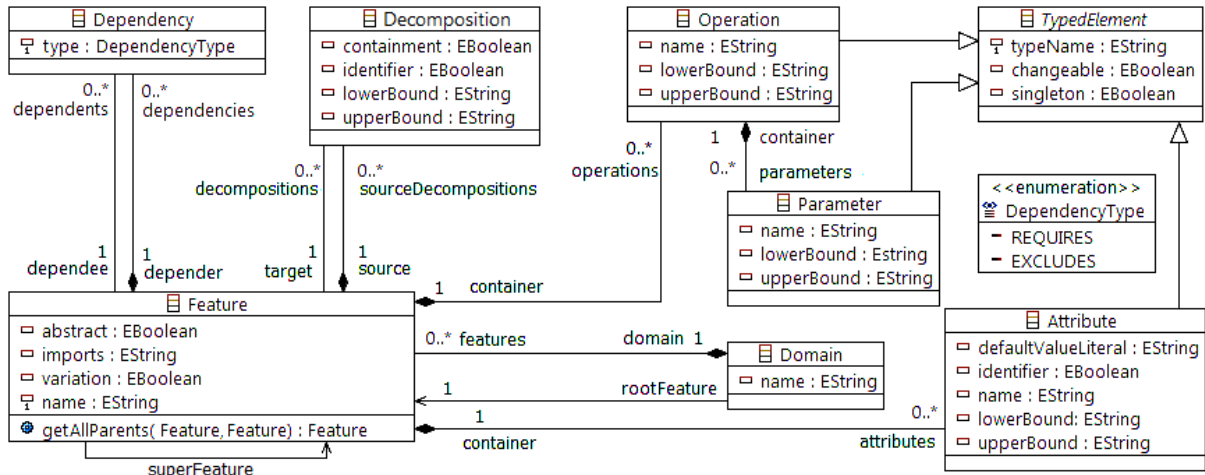


Figura 5.2. Metamodelo do editor pra modelo F3.

As metaclasses do metamodelo mostrado na Figura 5.2 representam os elementos e relacionamentos dos modelos F3, como explicado na Seção 3.2.1, e outros elementos necessários para a construção do editor. Essas metaclasses são:

- **Domain:** é o elemento raiz dos modelos F3 que armazena o nome e todas as características do domínio;
- **Feature:** representa as características e suas variantes e pode conter atributos e operações. A referência `superFeature` representa o relacionamento de generalização entre características e suas variantes;
- **Attribute:** representa os atributos das características e variantes;
- **Operation:** representa as operações das características e variantes e pode conter parâmetros;
- **Parameter:** representa os parâmetros das operações;
- **Decomposition:** representa os relacionamentos de decomposição;
- **Dependency:** representa os relacionamentos de dependência, cujos tipos estão definidos no *enumeration* `DependencyType`.

A sintaxe concreta dos modelos F3 foi definida por meio de três tipos de modelos providos pelo GMF:

1. Modelo *Gmfgraph*, em que foi definida a notação gráfica do editor;
2. Modelo *Gmftool*, em que é criada a barra de menu do editor;
3. Modelo *Gmfmap*, em que os elementos dos dois modelos anteriores são ligados aos elementos do metamodelo (sintaxe abstrata).

Na Figura 5.3 é mostrado um exemplo de parte desses modelos para ilustrar como o Modelo *Gmfmap* (a) faz a ligação entre os elementos `Feature` do Metamodelo (b) com um retângulo de bordas arredondadas do Modelo *Gmfgraph* (c) e com o item de menu `Feature` no Modelo *Gmftool* (d).

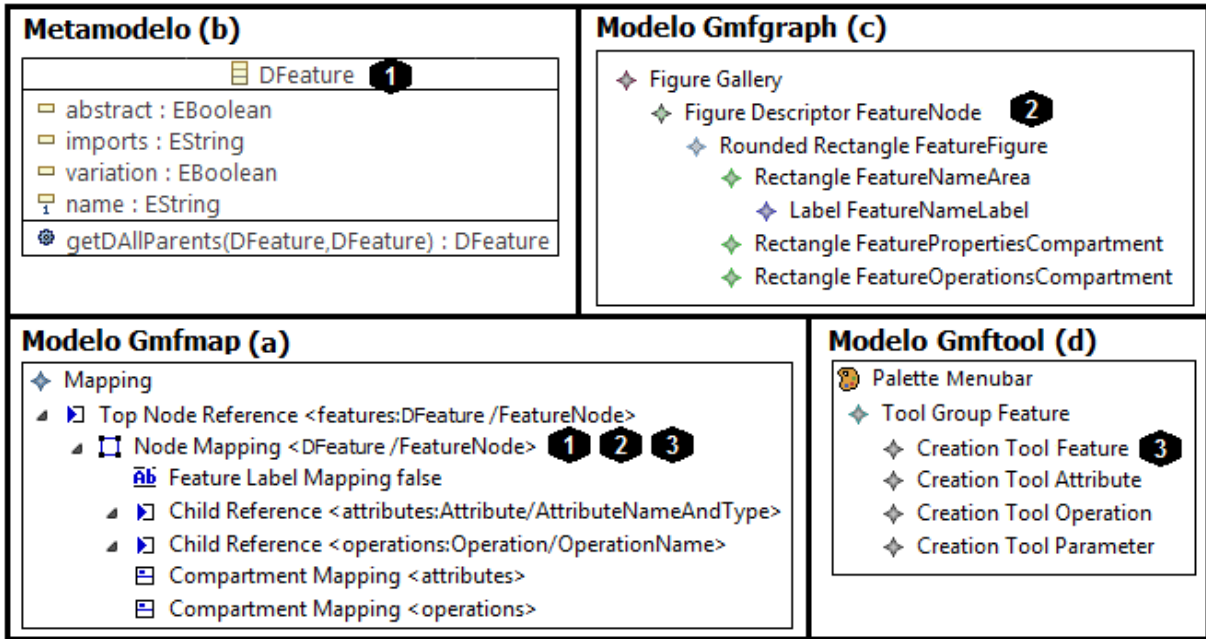


Figura 5.3. Ligação dos elementos das sintaxes abstrata e concreta dos modelos F3.

Na Figura 5.4 é mostrado o editor de modelos F3 da F3T, que é composto de três partes: 1) o espaço de modelagem; 2) o painel de propriedades; e 3) a barra de menu.

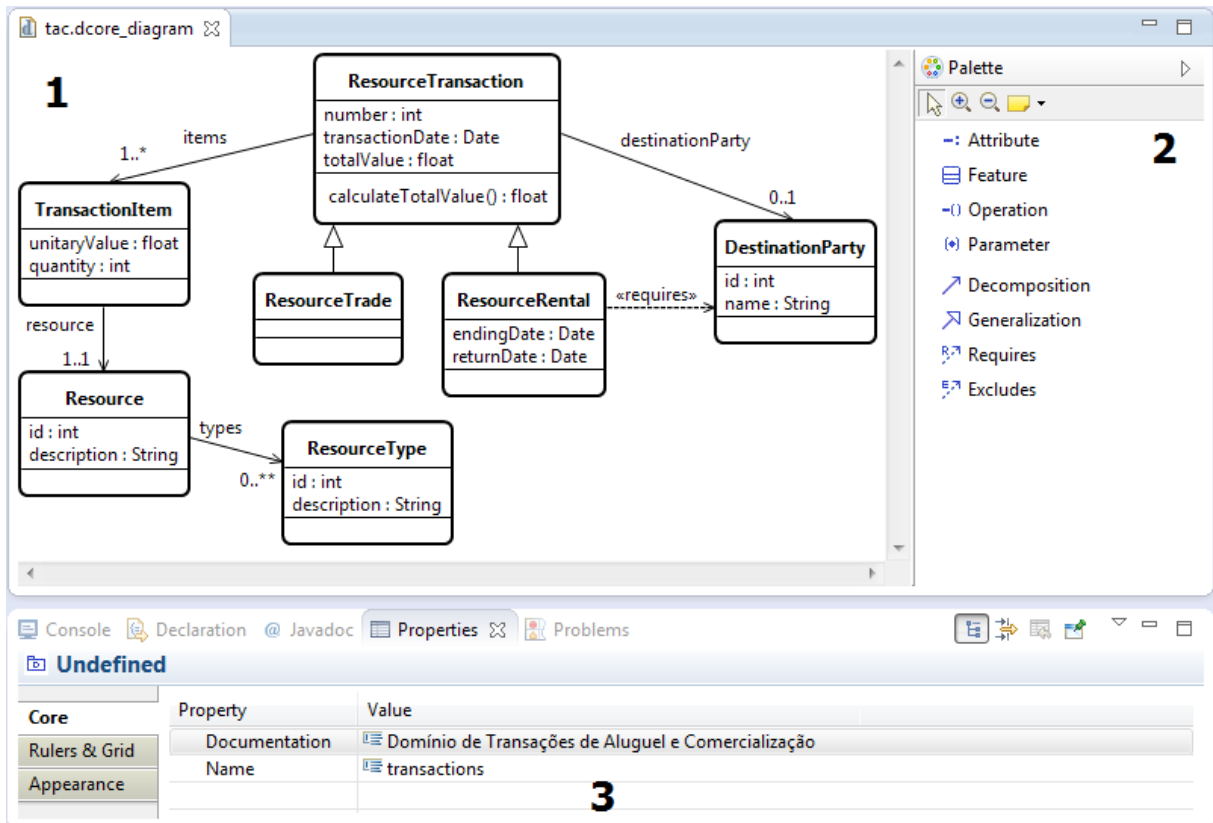


Figura 5.4. Editor gráfico da F3T para modelos F3.

Nos Quadros 5.1 a 5.5 são apresentadas as principais propriedades das características, atributos, operações, parâmetros e decomposições. As propriedades com o nome escrito em *itálico* são opcionais. A coluna Valores Válidos indica os valores aceitos por

essa propriedade, sendo que o primeiro valor mostrado nessa coluna é o *default*. Por exemplo no Quadro 5.1 a propriedade *Abstract* tem como valores válidos *true* e *false*; sendo que *true* é o valor *default*.

Quadro 5.1. Propriedades das características.

Propriedade	Descrição	Valores Válidos
<i>Abstract</i>	Define se a característica é ou não abstrata.	true, false.
<i>Documentation</i>	Descrição da característica.	Texto (String)
Name	Nome da característica. Deve ser único, sem espaços em branco e iniciar com Letra maiúscula.	Texto (String)

Quadro 5.2. Propriedades dos atributos.

Propriedade	Descrição	Valores Válidos
<i>Changeable</i>	Define se o atributo não é ou é constante.	true, false.
<i>Default Value Literal</i>	Define um valor padrão (default) para o atributo.	Texto (String)
<i>Documentation</i>	Descrição do atributo.	Texto (String)
<i>Identifier</i>	Define se o atributo é ou não uma chave primária.	false, true.
<i>Lower Bound</i>	Define se o atributo é obrigatório ou não.	0, 1
Name	Nome do atributo. Deve ser único dentro da característica que o contém e sem espaços em branco.	Texto (String)
<i>Static</i>	Define se o atributo é ou não estático.	false, true.
<i>Type Name</i>	Define o tipo do atributo. Deve ser um tipo válido da linguagem Java.	Texto (String)
<i>Upper Bound</i>	Define se o atributo aceita um único valor ou vários (lista).	1, *

Quadro 5.3. Propriedades das operações.

Propriedade	Descrição	Valores Válidos
<i>Changeable</i>	Define se a operação não é ou é constante.	true, false.
<i>Documentation</i>	Descrição da operação.	Texto (String)
<i>Lower Bound</i>	Define se a operação é obrigatória ou não.	0, 1
Name	Nome da operação. Deve ser único dentro da característica que o contém e sem espaços em branco.	Texto (String)
<i>Static</i>	Define se a operação é ou não estática.	false, true.
<i>Type Name</i>	Define o tipo e retorno da operação. Deve ser um tipo válido da linguagem Java.	Texto (String)
<i>Upper Bound</i>	Define se a operação retorna um único valor ou vários (lista).	1, *

Quadro 5.4. Propriedades dos parâmetros das operações.

Propriedade	Descrição	Valores Válidos
<i>Changeable</i>	Define se o parâmetro não é ou é constante.	true, false.
<i>Documentation</i>	Descrição do parâmetro.	Texto (String)
<i>Lower Bound</i>	Define se o parâmetro é obrigatório ou não.	0, 1
Name	Nome do parâmetro. Deve ser único dentro da operação que o contém e sem espaços em branco.	Texto (String)
<i>Static</i>	Define se o parâmetro é ou não estático.	false, true.
<i>Type Name</i>	Define o tipo do parâmetro. Deve ser um tipo válido da linguagem Java.	Texto (String)
<i>Upper Bound</i>	Define se o parâmetro retorna um único valor ou vários (lista).	1, *

Quadro 5.5. Propriedades das decomposições.

Propriedade	Descrição	Valores Válidos
Containment	Define se é ou não uma composição.	false, true
Identifier	Define se a característica alvo é uma chave primária da característica de origem da associação.	false, true
Lower Bound	Define se a característica alvo é obrigatória ou não.	0, 1
Multiple	Define se a característica de origem da associação aceita ou não mais de uma subclasse da característica alvo. O valor true exige que o valor da propriedade Upper Bound seja *.	false, true
Name	Nome do atributo na característica de origem da associação que é do tipo da característica alvo. Deve ser único dentro da característica de origem e sem espaços em branco.	Texto (String)
Upper Bound	Define se o atributo na característica de origem que é resultante da associação possui único valor (1) ou vários do mesmo tipo (*) o vários variantes (**).	1, *, **

A F3T possui um mecanismo de validação dos modelos F3, que verifica, por exemplo: o uso de nomes inválidos (*Invalid Name*) para o domínio, as características, as decomposições, os atributos e as operações; a existência de valores inválidos nas multiplicidades mínima e máxima (*Upper Bound Invalid Name*) das decomposições; a existência de mais de uma característica raiz (*No Multiple Roots*); e a ausência de atributo identificador (*Missing Identifier*) em uma característica. Na Figura 5.5. é mostrada a janela que aparece quando o mecanismo de validação retorna mensagens de erro.

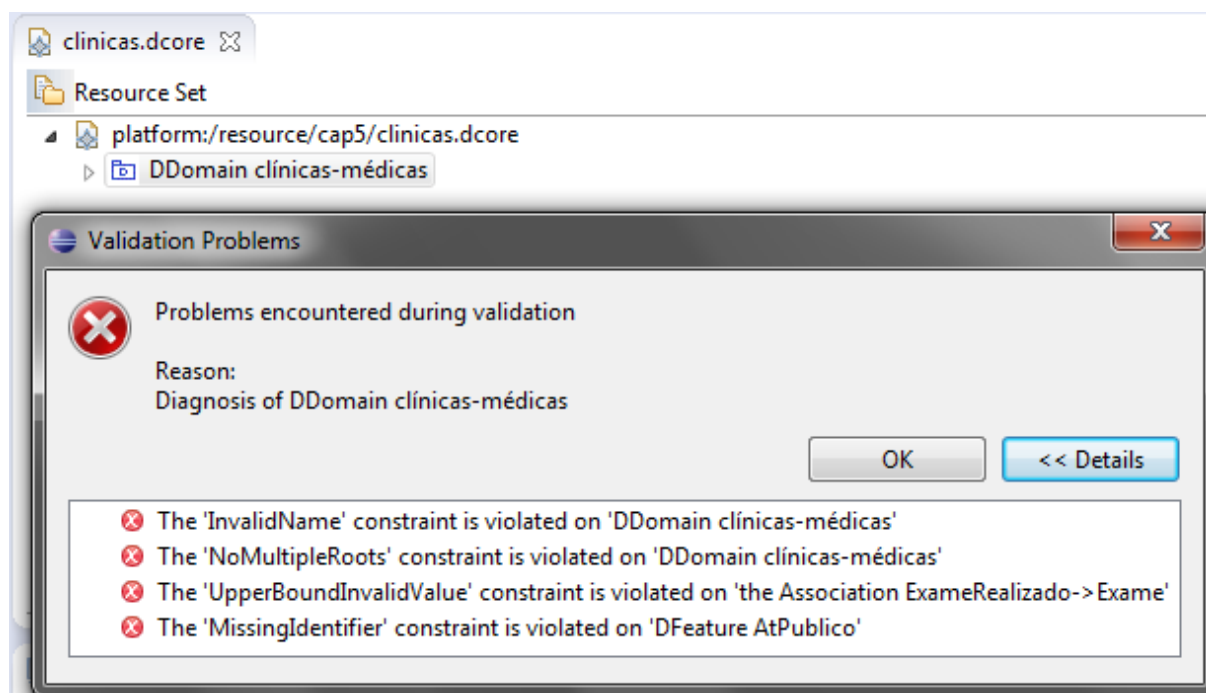


Figura 5.5. Mecanismo de validação dos modelos F3 na F3T.

5.2.2 Módulo de Frameworks

Cada modelo F3 criado com a F3T é salvo em dois arquivos XML: um com extensão *dcore*, que contém os dados do domínio; e outro com extensão *dcore_diagram*, que contém as informações gráficas do modelo. O Módulo de Frameworks foi projetado para ser um gerador *Model-to-Text* (M2T) que acessa o conteúdo do arquivo com extensão *dcore* e gera o código-fonte e a DSL do framework.

O Módulo de Frameworks foi desenvolvido com o apoio do *Java Emitter Templates* (JET) (GRONBACK, 2009), com o qual é possível construir um gerador M2T a partir de um conjunto de templates (Seção 2.4.2). Os templates do Módulo de Frameworks estão organizados em dois grupos na Figura 5.6. O grupo iniciado pelo template *DSC* está relacionado com a geração do código-fonte dos frameworks. O grupo iniciado pelo template *DSL* está relacionado com a geração dos modelos EMF/GMF das DSLs dos frameworks. Ambos os grupos são invocados pelo Módulo de Frameworks a partir do template principal (*Main*). As setas na Figura 5.6 indicam os templates que invocam outros templates.

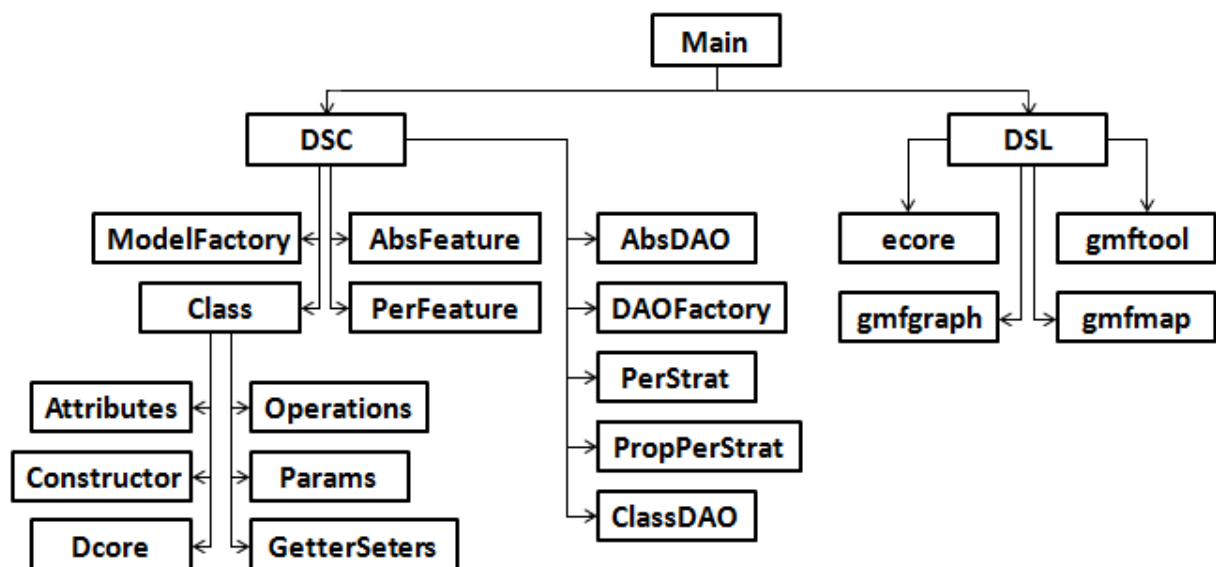


Figura 5.6. Organização dos templates do Módulo de Frameworks da F3T.

O código-fonte do framework é gerado pelo Módulo de Frameworks com base nos padrões da LPF3 (Seção 4.2.2). Desse modo, é gerada uma classe para cada característica encontrada nos modelos F3. Estas classes contêm os atributos e operações definidas em suas características. Relacionamentos de generalização resultam em heranças e os relacionamentos de decomposição resultam em associações entre as classes do framework. Operações adicionais estão incluídas nas classes do framework para tratar as variabilidades e limitações dos domínios definidos nos modelos F3.

Por exemplo, a partir dos relacionamentos de decomposição encontrados nos modelos F3, o Módulo de Frameworks gera nas classes do framework as operações que identificam as classes das aplicações de acordo com os padrões Decomposição Obrigatória e Decomposição Opcional da LPF3. Nesse caso, o valor da multiplicidade mínima (*lowerBound*) é verificado para determinar se o relacionamento de decomposição é opcional (0) ou obrigatório (1). Essa funcionalidade está implementada no template *Dcore* do Módulo de Frameworks, conforme é mostrado na Figura 5.7.a. Considerando o exemplo de domínio mostrado na Figura 5.4, o código de template mostrado na Figura 5.7.a gera as operações mostradas na Figura 5.7.b.

<pre>public <c:if test="\$decomposition/@lowerBound = '1'">abstract </c:if> Class<? extends <c:get select="\$decomposition/target/@name"/>> get<c:get select="uppercaseFirst(\$decomposition/target/@name)"/>Class() <c:choose select="\$decomposition/@lowerBound"> <c:when test="'1'">;</c:when> <c:otherwise> { return null; }</c:otherwise> </c:choose></pre>	a
<pre>public abstract Class<? extends TransactionItem> getTransactionItemClass(); public Class<? extends DestinationParty> getDestinationPartyClass() { return null; }</pre>	b

Figura 5.7. Código (a) de template para as operações definidas pelos padrões Decomposição Opcional e Decomposição Obrigatória da LPF3 e (b) de exemplos de operações geradas a partir desse template.

Além do código-fonte, o Módulo de Frameworks também gera os modelos *Ecore* (metamodelo), *Gmfgraph*, *Gmftool* e *Gmfmap* da DSL do framework gerado como descrito na abordagem apresentada no Capítulo 3. A partir desses modelos, o desenvolvedor usa o EMF e GMF para gerar os *plug-ins* da DSL e instalá-la no Eclipse IDE.

As DSLs geradas pela F3T possuem apenas um único tipo de relacionamento que pode ser utilizado entre seus elementos, ao invés de uma associação específica para cada elemento, como foi proposto na abordagem do Capítulo 3. Essa modificação foi necessária para facilitar a geração da DSL. Ter um único tipo de relacionamento entre os elementos da DSL torna a modelagem de uma aplicação mais fácil, porém não permite que algumas restrições sejam acionadas automaticamente durante a modelagem. Para compensar isso, a F3T implementa essas restrições no mecanismo de validação da DSL.

5.2.3 Módulo de Aplicações

O Módulo de Aplicações é responsável pela geração do código-fonte das aplicações a partir dos modelos criados com as DSLs dos frameworks. Esse módulo também foi desenvolvido com o apoio do JET. Seus templates geram classes que estendem as classes das camadas de modelo e de persistência dos frameworks e sobrescrevem as operações (*hot spots*) definidas pelos padrões da LPF3. Além disso foram criados templates para gerar o *script* SQL do banco de dados e o arquivo de propriedades das aplicações. Na Figura 5.8 é mostrada a organização dos templates do Módulo de Aplicações.

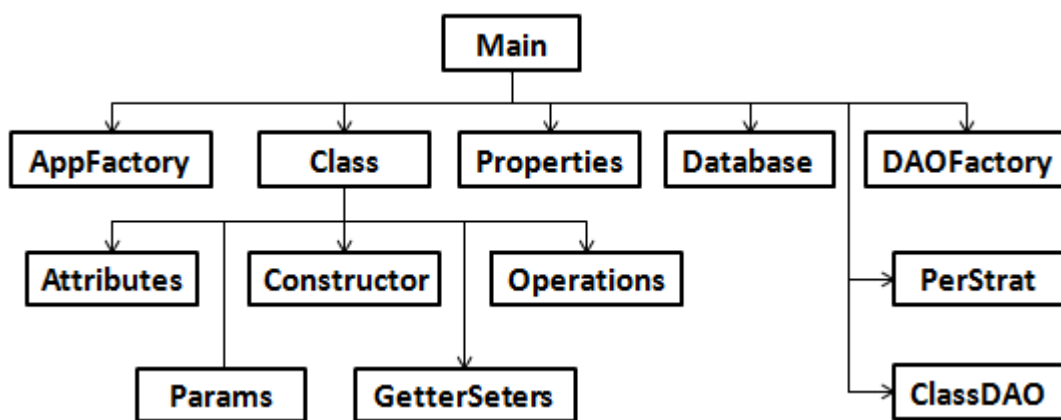


Figura 5.8. Organização dos templates do Módulo de Aplicações.

Na Figura 5.9.a é mostrado o trecho de código do template do Módulo de Aplicações que gera as operações que sobrescrevem as operações de identificação das classes das aplicações. Na Figura 5.9.b é mostrado um exemplo de código de aplicação gerado a partir desse template em que `VendaItem` e `Cliente` são classes da aplicação que, respectivamente, estendem as classes `TransactionItem` e `DestinationParty` do framework.

```

public Class<? extends <c:get select="$association/@stereo"/>>
    get<c:get select="$association/@stereo"/>Class() {
    return <c:get select="$association/@name"/>.class;
}
a

```

```

public abstract Class<? extends TransactionItem> getTransactionItemClass() {
    return VendaItem.class;
}

public Class<? extends DestinationParty> getDestinationPartyClass() {
    return Cliente.class;
}
b

```

Figura 5.9. Código de template que gera as operações que identificam as classes da aplicação e exemplo de código gerado por esse template.

5.3 Exemplo de Utilização da F3T

Nesta Seção é apresentado um exemplo de uso da F3T em que foi desenvolvido um framework para o domínio de Clínicas Médicas e uma aplicação para uma Clínica Veterinária. O uso da F3T é mostrado na Seção 5.3.1 para a fase de Engenharia do Domínio e na Seção 5.3.2 para a fase de Engenharia da Aplicação.

5.3.1 Engenharia do Domínio: Framework de Clínicas Médicas

A fase de Engenharia de Domínio corresponde ao desenvolvimento do código-fonte e da DSL do framework. Seguindo a abordagem F3, primeiramente, é realizada a etapa de Modelagem do Domínio utilizando o editor gráfico de modelos F3, fornecido pela F3T. Isso é feito utilizando a opção **F3 Model Diagram** do menu **From Features to Framework** e fornecendo um nome para o modelo F3 (*clinicas.dcore_diagram*), como mostrado na Figura 5.10.

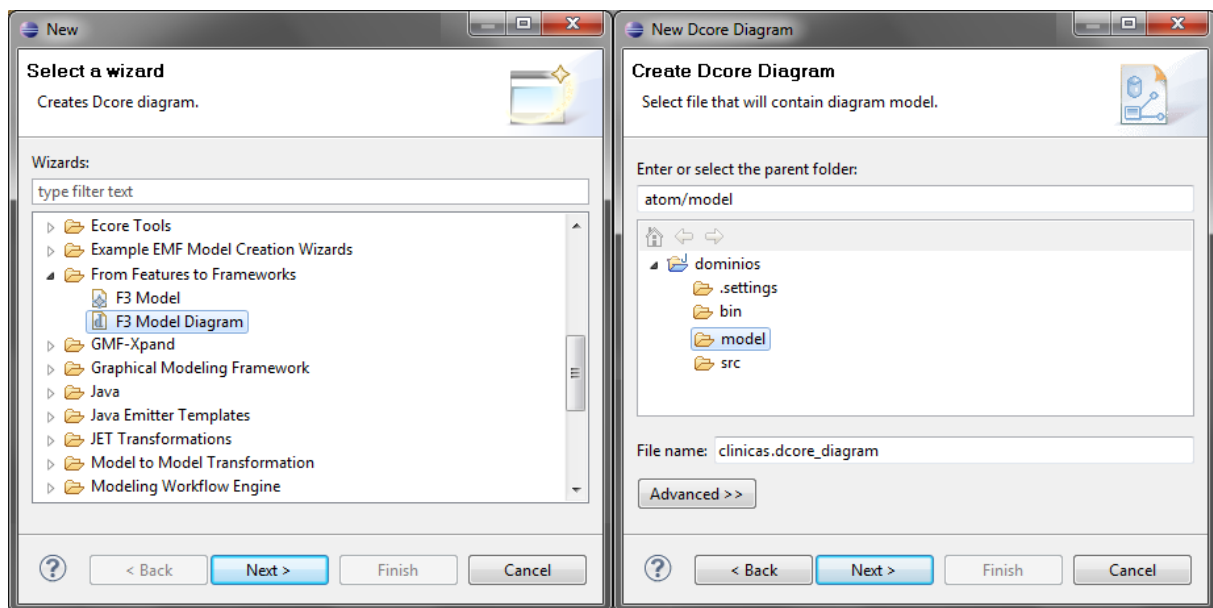


Figura 5.10. Criação de um modelo F3 na F3T.

Na Figura 5.11 é mostrado o modelo F3 do domínio de clínicas médicas, que possui os seguintes requisitos:

- Um **atendimento** possui código, data, hora, paciente, exames realizados e tratamentos. Os atendimentos podem ser **particulares** ou **públicos**. Além dos atributos previamente citados, atendimentos privados possuem também valor do atendimento e valor total a ser pago (atendimento + exames).

- Cada atendimento é realizado por um **especialista**, que possui código, nome e registro.
- **Exames** podem ser realizados para auxiliar no diagnóstico do paciente. A clínica possui um registro dos exames que podem ser realizados, com código, nome e descrição. Os exames solicitados pelo especialista que realizou o atendimento devem ser registrados, incluindo o valor e o resultado. Assim como os atendimentos, os exames podem ser realizados em instituições **públicas** ou **particulares**, sendo que nessas últimas os exames possuem um valor de custo.
- **Tratamentos** também podem ser prescritos para auxiliar no atendimento ao paciente, com código, nome e descrição.
- Os atendimentos são solicitados por **pacientes** identificados por um código e que também possuem nome e data de nascimento.
- Alguns pacientes podem ser incapazes de responder por si próprios, sendo necessário um **responsável** que possui código e nome.
- Para os atendimentos particulares, o paciente pode indicar um **plano de saúde** para arcar com todos os custos. Um plano de saúde possui código, nome e registro.

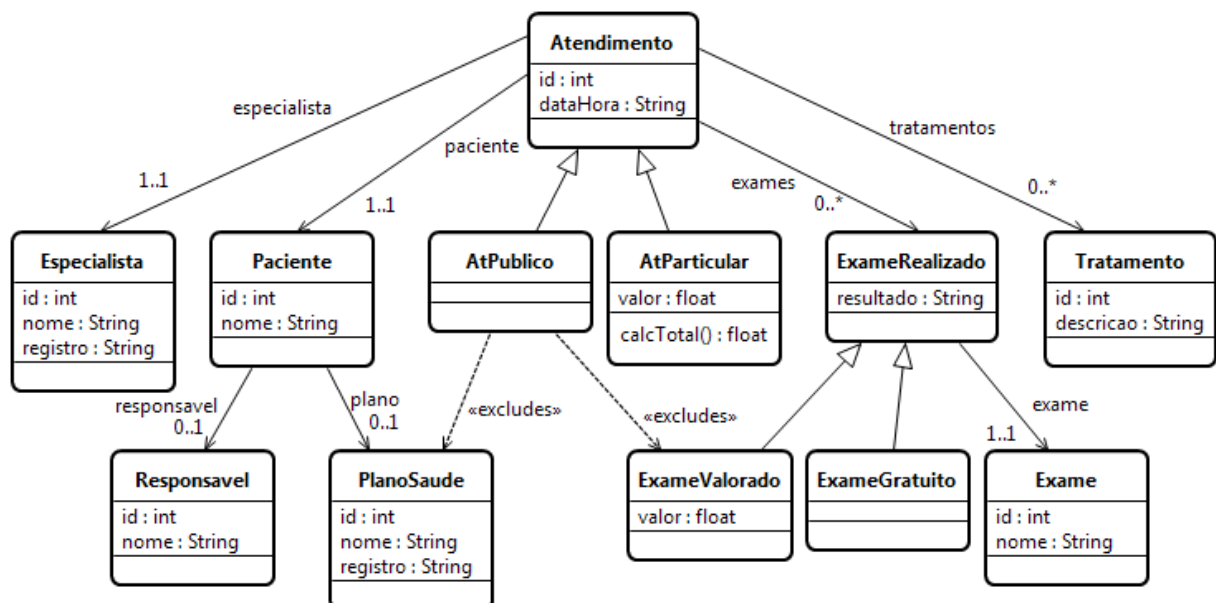


Figura 5.11. Modelo F3 do domínio de clínicas médicas.

Relacionamentos de dependência do tipo `excludes` foram inseridos no modelo para que a DSL possa invalidar que uma aplicação que trata de atendimentos públicos registre informações sobre planos de saúde e exames com valor. Contudo, o domínio prevê que atendimentos particulares podem ter exames sem custo, por isso, não há uma dependência do tipo `excludes` entre as características `AtParticular` e `ExameGratuito`.

O modelo F3 do domínio de Clínicas Médicas foi validado sem erros (Figura 5.12.a). O código-fonte e a DSL do framework são gerados a partir da opção *Generate framework and DSL* ao clicar sobre o arquivo do modelo F3 do domínio (Figura 5.12.b). Os projetos do código-fonte e dos *plug-ins* da DSL do framework aparecem no *workspace* do Eclipse IDE identificados pelo nome do domínio. O código interno das operações não é gerado, porém, é possível implementá-lo utilizando o editor Java do Eclipse IDE.

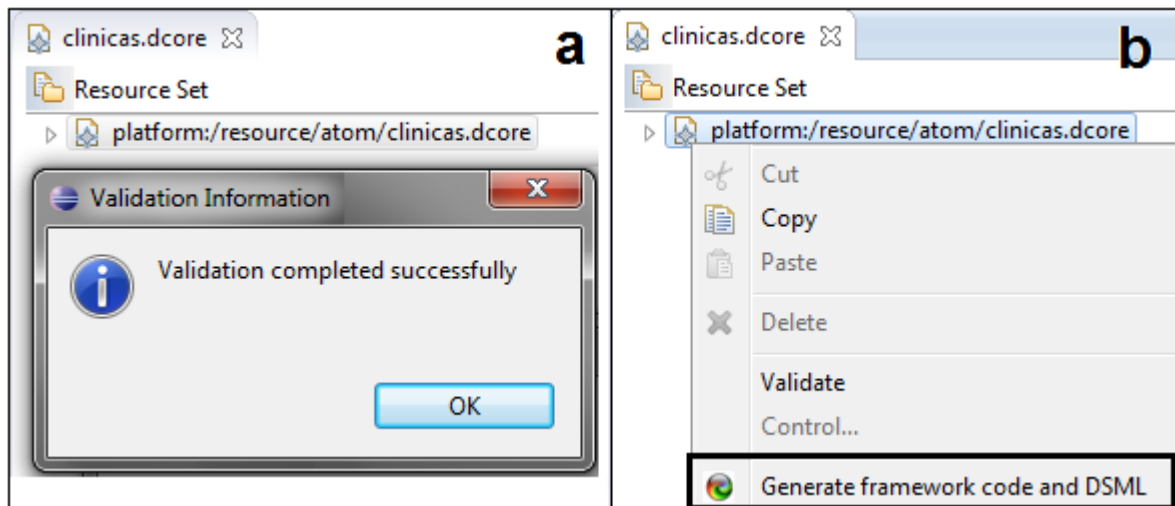


Figura 5.12. Função de geração do código-fonte e da DSL na F3T.

O código do framework é gerado em um projeto Java que possui o nome do domínio mais o sufixo *framework*. A DSL é composta por *plug-ins* do Eclipse IDE. O *plug-in* principal é identificado pelo nome do domínio e contém os modelos EMF/GMF, conforme explicado na Seção 5.2.1. Os demais *plug-ins* correspondem a implementação da notação gráfica da DSL e são identificados pelo nome do domínio e os sufixos *edit*, *editor*, *diagram* e *ui*.

A F3T não gera todo o conteúdo dos *plug-ins* da DSL do framework, pois esse conteúdo é gerado a partir dos modelos EMF/GMF ao serem executados os seguintes passos (Figura 5.13):

1. Selecionar as opções *Generate Model Code*, *Edit Code* e *Editor Code* do modelo *Genmodel* no *plug-in* principal;
2. Selecionar a opção *Create generator model* ao clicar no modelo *Gmfmap*;
3. Selecionar a opção *Generate diagram code* ao clicar no modelo *Gmfgen*.

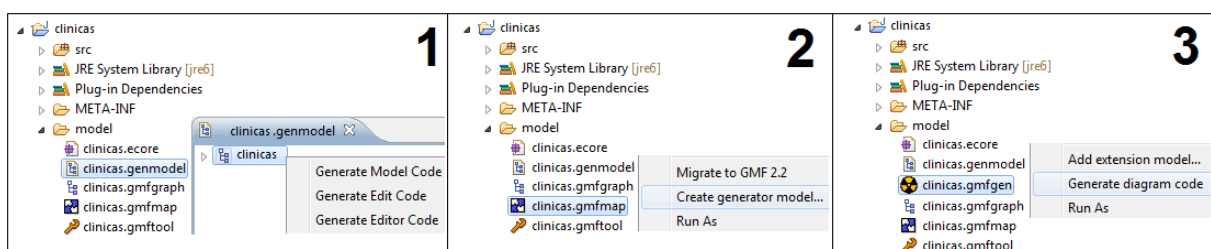


Figura 5.13. Geração do restante do conteúdo dos *plug-ins* da DSL do framework.

O código-fonte do framework é organizado em dois pacotes: 1) *clinicas* (nome do domínio) que possui as classes geradas para o domínio modelado e 2) *dcore*, que contém classes independentes de domínio. Na Figura 5.14 são mostrados *plug-ins* da DSL (*clinicas*, *clinicas.diagram*, *clinicas.edit*, *clinicas.editor* e *clinicas.ui*) gerados pela F3T para o domínio de Clínicas Médicas. Em destaque nessa figura, está o projeto *clinicas.framework*, que contém o código-fonte do framework, também gerado pela F3T.

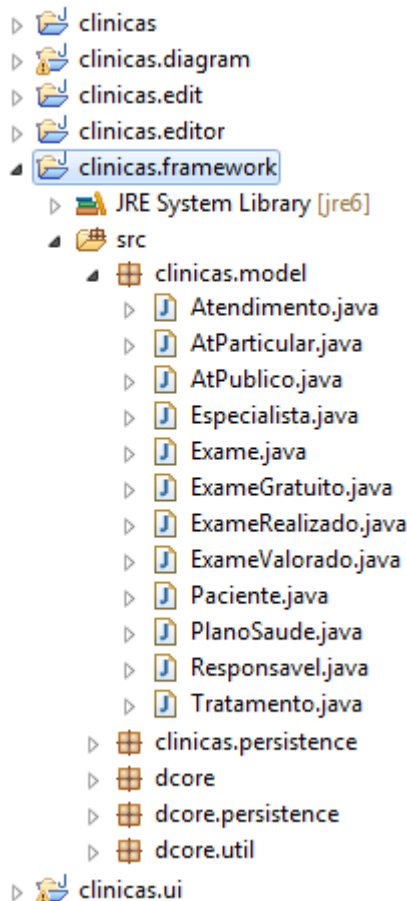


Figura 5.14. Projetos da DSL e do código-fonte do framework gerados pela F3T.

O código interno das operações das características definidas nos modelos F3 é gerado pela F3T com uma exceção de operação não implementada (`UnsupportedOperationException()`). Contudo, o código interno dessa operação pode ser modificado pelo desenvolvedor utilizando o editor do Eclipse IDE. As unidades de código geradas pela F3T contém uma anotação `generated`, que indica, para o módulo de geração da ferramenta, as unidades do código que foram geradas por ela. No caso de uma unidade do código ser modificada, por exemplo, uma operação, é necessário remover essa anotação ou modificá-la para `generated NOT`, para indicar para a F3T que aquela unidade de código foi modificada manualmente. Na Figura 5.15 é mostrado código da operação `calcTotal` da classe `AtParticular` (a) antes e (b) depois de ter sido modificada.

```

a
/**
 * @generated
 */
public float calcTotal() {
    // TODO: implement this method
    // Ensure that you remove @generated or mark it @generated NOT
    throw new UnsupportedOperationException();
}

b
/**
 * @generated NOT
 */
public float calcTotal() {
    float total = valor;
    for ( ExameRealizado exame : getExames() ) {
        total += ((ExameValorado)exame).getValor();
    }
    return total;
}
    
```

Figura 5.15. Código da operação calcTotal (a) antes e (b) depois da alteração.

5.3.2 Engenharia da Aplicação: Clínica Veterinária

A fase de Engenharia de Aplicação corresponde ao reuso do framework a partir da DSL para a instanciação de uma aplicação. Após a instalação dos plug-ins da DSL, o modelo de uma aplicação pode ser criado no Eclipse IDE. Na Figura 5.16 é mostrada a opção de criação de um modelo de uma aplicação para o domínio de Clínicas Médicas.

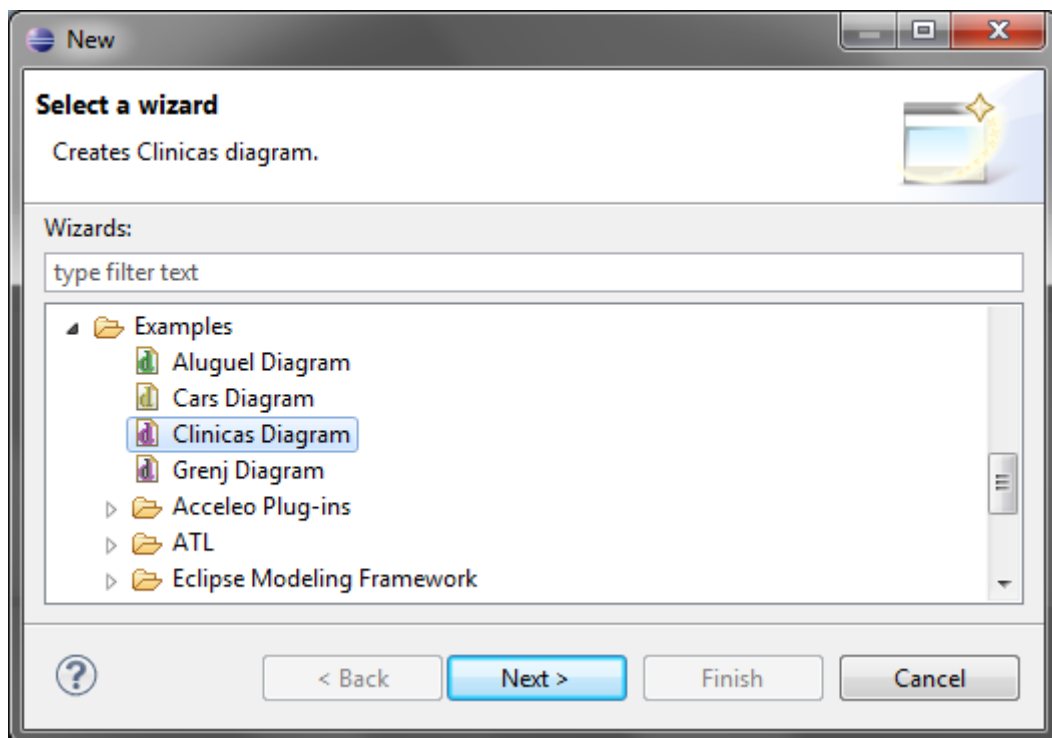


Figura 5.16. Criação de um modelo com a DSL do framework de Clínicas Médicas.

Nesta seção é apresentado o exemplo da instanciação do framework de Clínicas Médicas para uma aplicação de Clínica Veterinária com os seguintes requisitos:

- Na Clínica Veterinária, animais de pequeno porte como cães e gatos são atendidos em **consultas**. Uma consulta possui código, data, hora, animal, exames realizados, tratamentos, valor da consulta e valor total.
- Cada atendimento é realizado por um **veterinário**, que possui código, nome, registro e telefone.
- **Exames veterinários** podem ser realizados para auxiliar no diagnóstico do animal. A clínica possui um registro dos exames que podem ser realizados, com código, nome e descrição. Os exames solicitados em uma consulta devem ser registrados, incluindo o valor e o resultado.
- **Tratamentos veterinários** também podem ser sugeridos no atendimento ao animal, com código, nome e descrição.
- Os atendimentos são solicitados pelo **dono** do animal identificados por um código e que também possuem nome, endereço, CPF e telefone.
- Os **animais** possuem código, nome, raça, espécie e data de nascimento.

Na Figura 5.17 é mostrado o modelo da aplicação para uma Clínica Veterinária criado com a DSL do framework para o domínio de Clínicas Médicas. Do lado direito dessa figura (Palette) são mostradas as características do domínio, definidas no modelo da Figura 5.11. As características utilizadas na aplicação são identificadas pelos estereótipos das classes. O elemento *Veterinaria* define o nome da aplicação e outras propriedades, tais como a URL, o usuário e a senha do banco de dados.

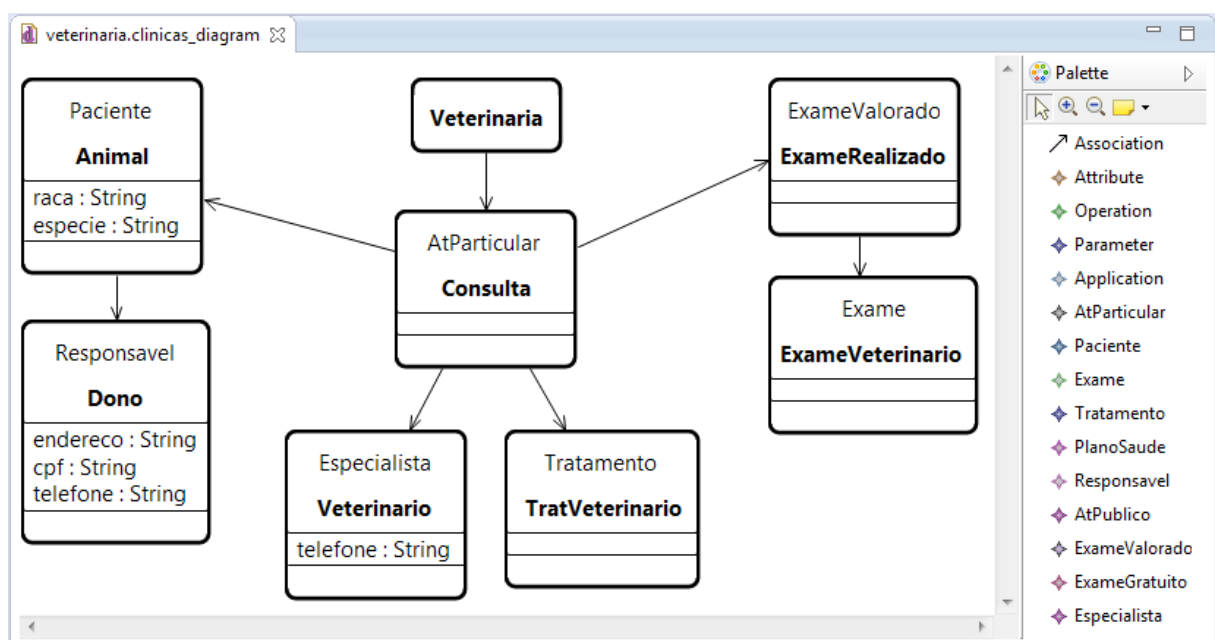


Figura 5.17. Modelo da aplicação para uma Clínica Veterinária.

Quando uma característica do domínio é selecionada no modelo da aplicação, é possível ver no Painel de Propriedades da F3T algumas propriedades, que fornecem informações sobre o framework, que auxiliam o desenvolvedor da aplicação. São elas: *Primary Key*, que indica quais atributos são chaves primárias; *Provided Attributes*, que informa ao desenvolvedor quais atributos essa característica possui no framework; e *Stereo*, que indica o nome da classe/característica no framework. Na Figura 5.18 é mostrado o Painel de Propriedades da F3T quando a característica *AtParticular* é selecionada.

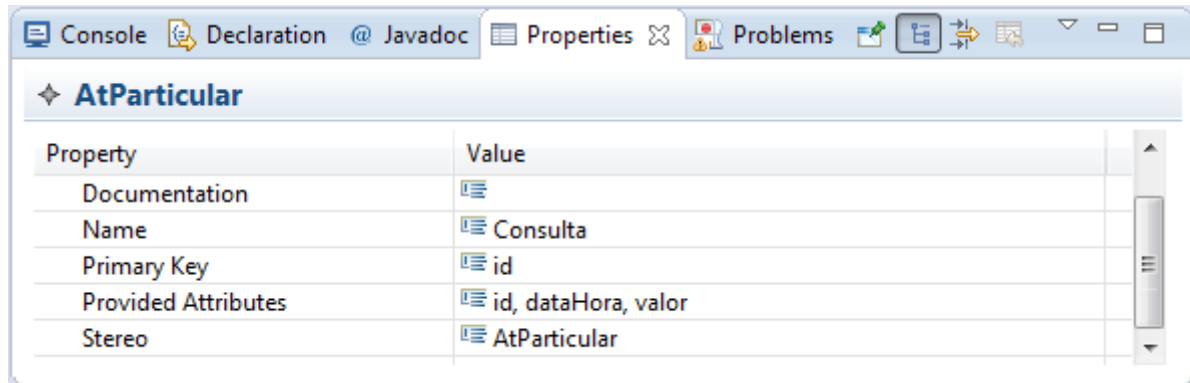


Figura 5.18. Painel de Propriedades da F3T com a característica *AtParticular*.

O código-fonte da aplicação é gerado selecionando a opção *Generate application code* do arquivo do modelo da aplicação (Figura 5.19.a). Além das classes, também são gerados um arquivo *sql*, que contém o script de criação das tabelas do banco de dados, e um arquivo *properties* com as informações de acesso ao banco de dados. O pacote *veterinaria.model* contém as classes da aplicação definidas no modelo e o pacote *veterinaria.persistence* contém as classes DAO. Na Figura 5.19 são mostrados os pacotes da aplicação para uma clínica veterinária mais os pacotes do framework para clínicas médicas.

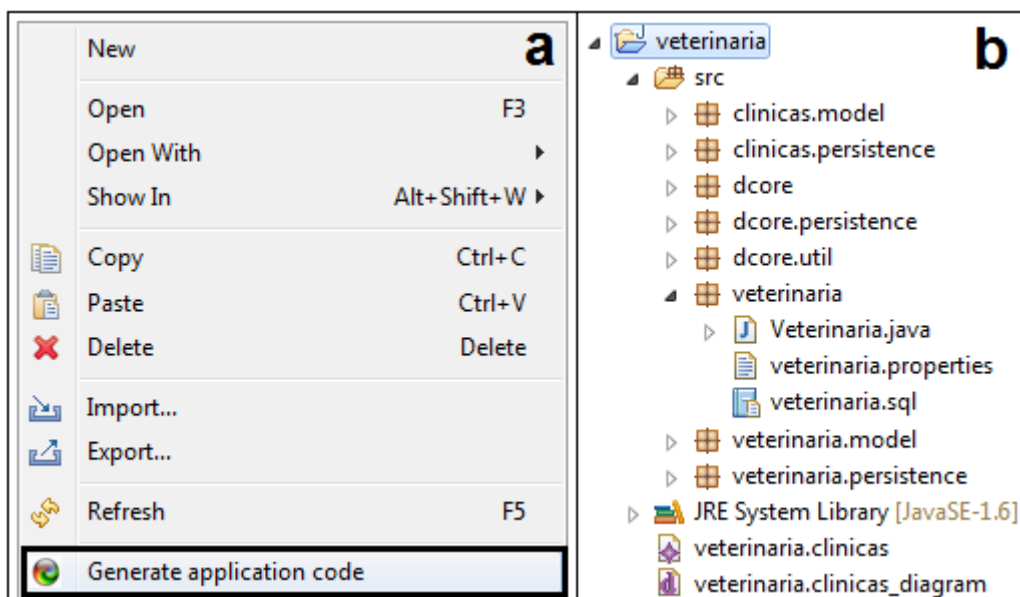


Figura 5.19. Geração da aplicação para uma Clínica Veterinária.

5.4 Considerações Finais

O fato das abordagens F3 e de construção de DSLs serem realizadas de forma sistematizada, possuindo atividades e regras bem definidas, permitiu a construção de uma ferramenta que automatiza as etapas dessas abordagens. A ferramenta F3T pôde ser desenvolvida a partir de tecnologias relacionadas com MDE e dão apoio aos processos de desenvolvimento e o reúso de frameworks.

Com a F3T os desenvolvedores direcionam seus esforços para a criação dos modelos de domínios e de aplicações. O código dos frameworks e das aplicações é gerado a partir desses modelos. Desse modo, esses processos ocorrem de forma mais fácil e eficiente, pois o desenvolvedor tem a preocupação na elaboração dos modelos e a ferramenta na geração do código.

A F3T é utilizada em duas fases, Engenharia do Domínio e Engenharia da Aplicação, da mesma forma que as abordagens de construção de DSLs e de frameworks, apresentadas nos Capítulos 3 e 4, respectivamente. Na Engenharia do Domínio é realizada a criação do modelo F3 para um determinado domínio e a geração do código-fonte e da DSL do framework. Na Engenharia da Aplicação, o desenvolvedor realiza a modelagem da aplicação utilizando a DSL do framework e a F3T gera o código-fonte dessa aplicação. Os frameworks e as aplicações desenvolvidos com o apoio da F3T são implementados em linguagem Java.

Com base no que foi apresentando neste capítulo, pode-se afirmar que a F3T possui as seguintes vantagens:

- Eficiência e facilidade no desenvolvimento dos frameworks, da DSL e das aplicações, pois o desenvolvedor não constrói manualmente o código desses artefatos;
- Confiabilidade nos artefatos produzidos, pois o código-fonte gerado pela F3T representa as características modeladas. Os resultados indesejados ocorrem somente quando a modelagem não for condizente com o domínio desejado;
- Facilidade de correção de problemas, pois é necessário corrigir somente os modelos criados pelo desenvolvedor;
- Validação da instanciação do framework, uma vez que os modelos das aplicações são validados de acordo com as regras definidas no modelo do domínio;
- Reúso do domínio, uma vez que o framework pode ser instanciado para diversas aplicações usando a sua DSL.

Com relação às desvantagens da F3T, podem ser feitas as seguintes afirmações:

- Dependência do ambiente de desenvolvimento, uma vez que a F3T funciona somente no Eclipse IDE;
- Dependência da linguagem de implementação, pois todo código gerado pela F3T é implementado somente em linguagem Java;
- Em sua versão atual, a F3T gera somente as classes das camadas de persistência e de modelo dos framework e das aplicações.

No capítulo seguinte são apresentados os experimentos realizados durante esta tese de doutorado para avaliar o reuso de frameworks por meio de DSLs, a abordagem F3 e a ferramenta F3T.

Capítulo 6

EXPERIMENTOS

6.1 Considerações Iniciais

Nos Capítulos 3 e 4 foram apresentadas duas abordagens para facilitar o reúso e o desenvolvimento de frameworks. Essas abordagens propõem atividades, com regras bem definidas, com o objetivo de guiar os desenvolvedores na construção dos artefatos de código. Como essas abordagens são manuais, no Capítulo 5 foi apresentada uma ferramenta, F3T, que automatiza a construção das unidades de código, deixando sob responsabilidade dos desenvolvedores somente a definição das características dos domínios e das aplicações. Desse modo, com o uso da ferramenta minimiza-se a probabilidade de inserção de erros e, se esses ocorrem, o desenvolvedor deve verificar os modelos criados. As vantagens em usar a F3T são a facilidade e eficiência no reúso de frameworks e na instanciação desses para a geração de aplicações de um domínio.

Com o intuito de verificar se essas abordagens realmente proporcionam essas vantagens (facilidade e eficiência) na prática, alguns experimentos foram realizados ao longo desta tese de doutorado. Esses experimentos foram planejados e executados seguindo a abordagem definida por Wohlin et al. (2000), que é composta por três fases: 1) definição e planejamento, em que são especificados o contexto, as hipóteses, as variáveis, os participantes, os instrumentos e o modelo do experimento; 2) operação, em que ocorre a preparação e a execução do experimento com os participantes; e 3) análise dos dados, em que os dados são organizados e analisados por meio de técnicas estatísticas.

Na Seção 6.2 há uma descrição dos testes estatísticos aplicados nos experimentos realizados durante esta tese de doutorado. Na Seção 6.3 é descrito o experimento que avaliou o reúso de frameworks por meio de DSLs; na Seção 6.4 é apresentado, o experimento que avaliou a abordagem F3; na Seção 6.5 é apresentado o experimento que avaliou a F3T. Na Seção 6.6 são comentadas as considerações finais deste capítulo.

6.2 Testes Estatísticos

Wohlin et al. (2000) afirmaram que os experimentos têm como objetivo responder questões a respeito de um objeto de estudo. Para cada questão são definidas uma ou mais métricas a partir da qual os dados são coletados e um conjunto de hipóteses sobre os possíveis resultados. Esses conjunto de hipóteses é formado por:

- Uma **hipótese nula**, que considera que não há uma diferença significativa entre os dados obtidos ao se aplicarem os diferentes tratamentos sobre o objeto de estudo;
- Uma ou mais **hipóteses alternativas**, que consideram os demais possíveis resultados. Por exemplo, a hipótese alternativa 1 considera que a abordagem X é mais eficiente do que a abordagem Y e a hipótese alternativa 2 considera que X é menos eficiente que Y.

Alguns cálculos podem ser realizados sobre os dados coletados para se obter um resultado e uma das hipóteses alternativas ser aceita. Por exemplo, cálculos de média e de porcentagem podem indicar que, em determinado experimento, o tempo gasto pelos participantes para desenvolver uma aplicação foi menor quando a abordagem X foi utilizada do que quando a abordagem Y foi utilizada. Contudo são necessários testes estatísticos para comprovar que esse resultado é significativo, refutando a hipótese nula.

Nos experimentos realizados ao longo desta tese de doutorado, foram aplicados os testes estatísticos sugeridos por Wohlin et al. (2000): **Shapiro-Wilk**, **Paried T-Test** e **Wilcoxon Signed Rank**. Esses testes são explicados nas Subseções 6.2.1 a 6.2.3 e foram aplicados com o apoio da ferramenta **Action** (ESTATCAMP, 2013), que é um software para análise estatísticas integrado ao MS Excel. Uma explicação detalhada sobre esses testes foge do escopo desta tese de doutorado, porém pode ser encontrada no Portal Action⁴.

6.2.1 Shapiro-Wilk

O teste *Shapiro-Wilk* é aplicado para verificar se um conjunto de dados segue, ou não, uma distribuição normal, que possui graficamente o formato de um sino simétrico em relação à sua média (Figura 6.1). Se o P-valor (resultado) do teste *Shapiro-Wilk* sobre um conjunto de dados for menor que 0,05, significa que a chance dos dados seguirem uma distribuição normal é menor do que 5%. Quando esse resultado ocorre, considera-se, estatisticamente, que os dados não seguem uma distribuição normal (ESTATCAMP, 2013).

⁴ <http://www.portalaction.com.br/>

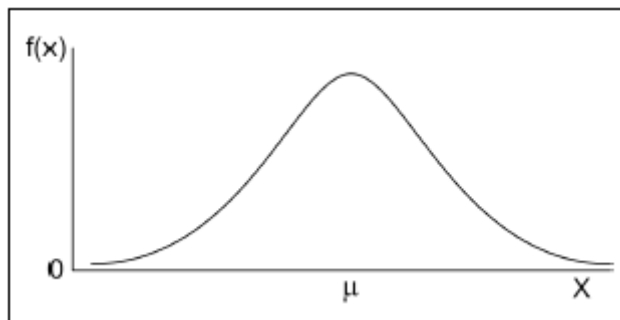


Figura 6.1. Representação gráfica de uma distribuição normal de dados.

Em geral, as ferramentas de estatística utilizam um gráfico de probabilidade (Q-Q Plot) (ESTATCAMP, 2013) para representar graficamente a distribuição de um conjunto de dados. Nesse gráfico, quando os dados se posicionam ao redor da linha diagonal, considera-se que os dados seguem uma distribuição normal. Na Figura 6.2 são mostrados dois exemplos de gráficos de probabilidade, um com distribuição normal e outro não normal.

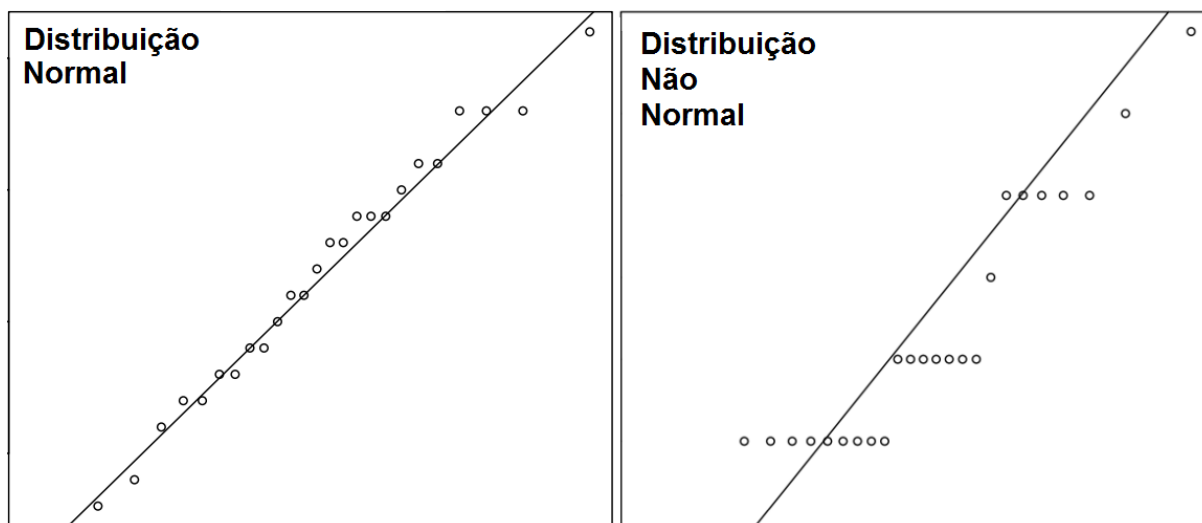


Figura 6.2. Exemplos de gráficos de probabilidade.

6.2.2 Paried T-Test

O *Paried T-Test* é utilizado para confrontar dois conjuntos de dados. Esse teste pode ser aplicado somente quando ambos os conjuntos de dados seguem uma distribuição normal. Se o P-valor (resultado) do *Paried T-Test* for menor que 0,05, significa que a chance dos dois conjuntos de dados serem estatisticamente semelhantes é menor do que 5%. Portanto, nesse caso, a hipótese nula deve ser rejeitada (ESTATCAMP, 2013).

6.2.3 Wilcoxon Signed Rank

O teste *Wilcoxon Signed Rank* é utilizado para confrontar dois conjuntos de dados quando ao menos um desses conjuntos não segue uma distribuição normal. Assim como no *Paired T-Test*, se o P-valor (resultado) do teste *Wilcoxon Signed Rank* for menor que 0,05, significa que a chance dos dois conjuntos de dados serem estatisticamente semelhantes é menor do que 5%. Portanto, a hipótese nula deve ser rejeitada (ESTATCAMP, 2013).

6.3 Experimento 1: Reúso de Frameworks por Meio de DSLs

Nesta seção é apresentado um experimento conduzido com o objetivo de analisar as vantagens propiciadas pelas DSLs na instanciação de frameworks. Para esse experimento foi utilizado o framework GRENJ (VIANA, 2009; DURELLI, 2008), apresentado na Seção 2.3.1, e a sua DSL, cuja construção foi descrita na Seção 3.3.1. O uso do framework GRENJ nesse experimento ocorreu em razão do: 1) acesso total ao seu código-fonte; e 2) conhecimento detalhado desse framework por parte do autor desta tese, de modo a permitir que os dados resultantes do experimento pudessem ser melhor examinados.

6.3.1 Definição e Planejamento do Experimento 1

Esse experimento foi definido como apresentado no Quadro 6.1. As etapas da fase de planejamento do experimento estão descritas nas Subseções 6.3.1.1 a 6.3.1.5.

Quadro 6.1. Definição do Experimento 1.

Análise da	instanciação de frameworks por meio de DSLs
Com o propósito de	avaliação
A partir do ponto de vista dos	desenvolvedores
Com respeito à	eficiência (tempo) e à dificuldade (número de problemas)
No contexto de	estudantes de graduação dos cursos de Ciência da Computação e de Engenharia da Computação da Universidade Federal de São Carlos (UFSCar)

6.3.1.1 Contexto e Participantes

O contexto desse experimento foi **multi-teste dentro de um objeto de estudo** (WOHLIN et al., 2000), pois consistiu em testes experimentais executados por um grupo de indivíduos para estudar uma única abordagem, que é o reúso de frameworks por meio de DSLs. O experimento foi realizado em um laboratório de computação no ambiente universitário. Ao todo, participaram do experimento 18 alunos de graduação.

6.3.1.2 Formulação das Hipóteses

As questões, métricas e hipóteses definidas para esses experimentos foram:

Questão 1 (Q1): A DSL torna a instanciação do framework GRENJ mais eficiente quando comparada com a instanciação realizada com o wizard?

Métrica 1 (M1): tempo (t) gasto pelos participantes na instanciação do framework GRENJ em uma aplicação.

Hipótese Nula (H1₀): Não há diferença significativa entre a DSL e o wizard quanto ao tempo gasto para a instanciação do framework GRENJ em uma aplicação. Isso pode ser formalizado como $t_{DSL} = t_{Wizard}$.

Hipótese Alternativa (H1₁): Com a DSL o tempo gasto pelos participantes na instanciação do framework GRENJ em uma aplicação é menor do que com o wizard. Isso pode ser formalizado como $t_{DSL} < t_{Wizard}$.

Hipótese Alternativa (H1₂): Com a DSL o tempo gasto pelos participantes na instanciação do framework GRENJ em uma aplicação é maior do que com o wizard. Isso pode ser formalizado como $t_{DSL} > t_{Wizard}$.

Questão 2 (Q2): A DSL reduz a chance dos participantes cometerem erros durante a instanciação do framework GRENJ em uma aplicação? O termo “problemas”, ao invés de “defeitos” ou “falhas”, porque também foram consideradas situações em que o código-fonte não continha defeitos nem apresentava falhas, porém a aplicação não estava correta por falta de alguma característica ou função.

Métrica 2 (M2): número de problemas (p) no código-fonte das aplicações por causa de erros cometidos pelos participantes durante o reuso do framework GRENJ.

Hipótese Nula (H2₀): Não há diferença significativa no número de problemas encontrados nas aplicações desenvolvidas pelos participantes com o reuso do framework GRENJ com a DSL e com o wizard. Isso pode ser formalizado como $p_{DSL} = p_{Wizard}$.

Hipótese Alternativa (H2₁): Com a DSL o número de problemas encontrados nas aplicações desenvolvidas com o reuso do framework GRENJ é menor do que com o wizard. Isso pode ser formalizado como $p_{DSL} < p_{Wizard}$.

Hipótese Alternativa (H2₂): Com a DSL o número de problemas encontrados nas aplicações desenvolvidas com o reuso do framework GRENJ é maior do que com o wizard. Isso pode ser formalizado como $p_{DSL} > p_{Wizard}$.

6.3.1.3 Variáveis

Esse experimento teve as seguintes variáveis independentes: 1) o framework GRENJ; 2) o Eclipse IDE versão 4.2.1; 3) o wizard do framework GRENJ; 4) a linguagem de

programação Java, e 5) as aplicações desenvolvidas pelos participantes: 5.a) Sistema de Biblioteca e 5.b) Sistema de Hotel. As duas aplicações tratam de transações de aluguel e têm o mesmo nível de complexidade.

As variáveis dependentes são as seguintes: 1) eficiência, que está relacionada com o tempo gasto com a instanciação do framework GRENJ para as aplicações do Sistema de Biblioteca e do Sistema de Hotel, e 2) o número de problemas encontrados nas aplicações por decorrência de erros cometidos pelos participantes durante a instanciação do framework.

6.3.1.4 Modelo

O modelo de experimento utilizado foi **um fator com dois tratamentos pareados** (WOHLIN et al., 2000). Nesse experimento, o **fator** foi o uso de uma abordagem para instanciar o framework em uma aplicação, enquanto que **os tratamentos** foram as abordagens aplicadas: DSL e wizard.

O experimento seguiu o formato em que os participantes são alocados em grupos homogêneos para que o nível de experiência deles não impactasse nos resultados. Um Formulário de Caracterização de Participante (Apêndice C) foi elaborado e distribuído entre os participantes para determinar o nível de experiência de cada um. Nesse formulário os participantes responderam questões de múltipla escolha a respeito do seu conhecimento sobre: 1) programação Java, 2) Eclipse IDE, 3) padrões de software e 4) frameworks. Cada questão possuía a seguinte pontuação:

- 0 – quando o participante não tinha conhecimento algum;
- 1 – quando o participante tinha apenas conhecimento teórico;
- 2 – quando o participante tinha conhecimento teórico e prático.

Nessa fase de planejamento do experimento, foi decidido que os participantes seriam divididos em dois grupos (G1 e G2), sendo que cada grupo deveria ser composto com o mesmo número (ou aproximado) de participantes que obtiveram pontuação baixa (0-2), média (3-5) e alta (6-8) no Formulário de Caracterização de Participante.

Para que pudesse realizar o experimento, os participantes foram treinados na instanciação do framework GRENJ, tanto utilizando a DSL quanto usando o wizard. Para desenvolver uma aplicação reutilizando o framework GRENJ com o wizard, os participantes tinham que selecionar e preencher os formulários do wizard de acordo com os requisitos da aplicação e gerar o código dessa aplicação. Para desenvolver uma aplicação reutilizando o framework GRENJ com a DSL, os participantes tinham que criar um modelo da aplicação utilizando a DSL e gerar o código-fonte dessa aplicação a partir desse modelo.

No quadro 6.2 é apresentado como foram definidas as atividades de desenvolvimento das aplicações reutilizando o framework GRENJ para cada um dos participantes do G1 e do G2.

Quadro 6.2. Atividades do Experimento 1.

Atividade	Aplicação	Forma de reúso do framework GRENJ	
		Grupo 1 (G1)	Grupo 2 (G2)
1	Hotel	Wizard	DSL
2	Biblioteca	DSL	Wizard

6.3.1.5 Instrumentação

Os participantes receberam os seguintes materiais para a execução do experimento: documento de requisitos das aplicações; unidades de teste para verificar a existência de problemas nas aplicações; e o Formulário de Coleta de Dados (Apêndice C), no qual era preenchido o tempo gasto no desenvolvimento das aplicações, as mensagens de erro retornadas pelas unidades de testes e opiniões e sugestões a respeito da DSL e do wizard.

6.3.2 Operação do Experimento 1

Depois de definir e planejar o experimento, sua fase de operação foi realizada em duas etapas: Preparação e Execução.

6.3.2.1 Preparação

Alguns dias antes da realização do experimento, os participantes preencheram o Formulário de Caracterização de Participante (Apêndice C), reportando sua experiência com relação aos conceitos e tecnologias utilizados no experimento. Os participantes também foram treinados no desenvolvimento de aplicações reutilizando o framework GRENJ tanto com a DSL quanto com o wizard.

6.3.2.2 Execução

Primeiramente, os participantes foram posicionados nos grupos com base na sua pontuação no Formulário de Caracterização de Participante. Cada grupo ficou com 9 participantes. O G1 ficou com 4 participantes com pontuação baixa (0-2), 3 com pontuação média (3-5) e 2 com pontuação alta (6-8). O G2 ficou com 5 participantes com pontuação baixa (0-2), 2 com pontuação média (3-5) e 2 com pontuação alta (6-8). Após tomarem conhecimento de qual grupo pertenciam, os participantes receberam o material para executarem a Atividade 1. O tempo limite era de 60 minutos para cada atividade.

Na Atividade 1, os participantes do G1 desenvolveram a aplicação Sistema de Hotel reutilizando o framework GRENJ com o wizard e os participantes do G2 desenvolveram a mesma aplicação, porém reutilizando o framework GRENJ com a DSL. Cada participante desenvolveu sua aplicação individualmente e registrou o tempo gasto nessa atividade desde a modelagem (DSL) ou preenchimento dos formulários (wizard) até a geração do código-fonte das aplicações. Depois disso, com o relógio parado, os participantes executavam as unidades de teste para verificar se o framework foi instanciado corretamente ou não. Quando as unidades de testes retornavam mensagens de erro, os participantes escreviam essas mensagens no Formulário de Coleta de Dados (Apêndice C) e, então, mediam o tempo gasto para corrigir os problemas na aplicação e executavam novamente as unidades de teste. Quando ocorriam interrupções não havia pausa na contagem do tempo, porém o tempo gasto com essas interrupções também foi anotado pelos participantes para que fosse abatido do tempo total de desenvolvimento da aplicação. A execução das unidades de teste não era considerada como uma interrupção.

A Atividade 2 foi executada da mesma forma que a Atividade 1. Contudo, nessa segunda atividade, os participantes do G1 desenvolveram a aplicação do Sistema de Biblioteca reutilizando o framework GRENJ com a DSL e os participantes do G2 desenvolveram a mesma aplicação reutilizando o framework GRENJ com o wizard.

6.3.3 Análise dos Dados do Experimento 1

Os dados do experimento estão apresentados no Quadro 6.3. Os participantes desenvolveram as atividades de acordo com o planejado. A análise dos dados é apresentada nas subseções seguintes.

6.3.3.1 Análise Descritiva dos Dados

No Quadro 6.3 pode ser visto que houve uma redução no tempo gasto pelos participantes no desenvolvimento das aplicações quando o reuso do framework GRENJ foi realizado com a DSL, em comparação com o wizard. Aproximadamente, 56,4% do tempo total do experimento foi gasto com o reuso do framework com o wizard e 43,6% com o reuso do framework com a DSL. Um dos motivos para isso é que os participantes cometeram mais erros durante o reuso do framework com o wizard e, desse modo, gastaram mais tempo corrigindo os problemas encontrados nas aplicações pelas unidades de teste. Outro motivo é que os participantes estão mais acostumados com o uso de modelos gráficos do que o uso de wizards para o desenvolvimento de aplicações. Infere-se, então, que os participantes possuíam um habilidade maior no manuseio da DSL.

Quadro 6.3. Dados coletados do experimento 1.

Aplicação	Wizard			DSL		
	Participante	Tempo (min)	Problemas	Participante	Tempo (min)	Problemas
Hotel	S1	24	2	S10	22	3
	S2	26	3	S11	18	0
	S3	31	6	S12	23	4
	S4	34	8	S13	21	1
	S5	25	2	S14	17	0
	S6	23	0	S15	19	0
	S7	29	4	S16	20	2
	S8	28	3	S17	21	4
	S9	22	0	S18	17	0
	Média	26,89	3,11	-	19,78	1,56
Biblioteca	S10	32	6	S1	18	0
	S11	27	4	S2	23	2
	S12	33	5	S3	22	1
	S13	25	3	S4	25	4
	S14	22	0	S5	21	1
	S15	24	1	S6	19	0
	S16	23	4	S7	23	2
	S17	30	6	S8	22	5
	S18	20	0	S9	18	0
	Média	26,22	3,22	-	21,22	1,67
Média Geral	26,56	3,17	-	20,5	1,61	
Mediana	25,5	3	-	21	1	
Desvio Padrão	4,16	2,43	-	2,36	1,72	
Porcentagem	56,43	66,28	-	43,57	33,72	

Com relação aos problemas, aproximadamente, 66,3% deles foram encontrados nas aplicações desenvolvidas com o wizard e 33,7% nas aplicações desenvolvidas com a DSL. Os principais problemas encontrados foram o reuso indevido de uma classe do framework e a inclusão de atributos já fornecidos pelo framework. Ambos relacionados com o conhecimento que os participantes possuem sobre o domínio e as classes do framework GRENJ. De acordo com o *feedback* dos participantes, a principal razão para essa diferença é que a interface gráfica da DSL é mais intuitiva e mais fácil de ser utilizada do que os formulários do wizard. Os modelos criados com a DSL proporcionam uma visão geral de todas as características, enquanto que no wizard somente é possível visualizar um formulário por vez, sendo que cada formulário contém as informações de uma única classe da aplicação. Além disso, o mecanismo de validação dos modelos criados com a DSL se mostrou mais eficaz do que as verificações realizadas nos campos preenchidos nos formulários do wizard.

Na Figura 6.3 é mostrado o gráfico *boxplot* que representa a distribuição dos dados do experimento. Nesse gráfico, as caixas representam 50% dos dados e a linha em destaque dentro dessas caixas representa a mediana. As delimitações na forma de um T acima e abaixo das caixas indicam, respectivamente, o maior e o menor valor dos dados considerados válidos. Para serem considerados válidos, os dados devem estar a uma distância de, no máximo, 1,5 vezes a altura da caixa em relação aos limites inferiores e superiores da própria caixa. Dados além desse limite são considerados atípicos (*outliers*) e devem ser desconiderados nos cálculos estatísticos para validação das hipóteses do experimento. No gráfico da Figura 6.3 é possível perceber que nenhum dos dados foi considerado atípico, pois não há pontos além das delimitações em forma de T.

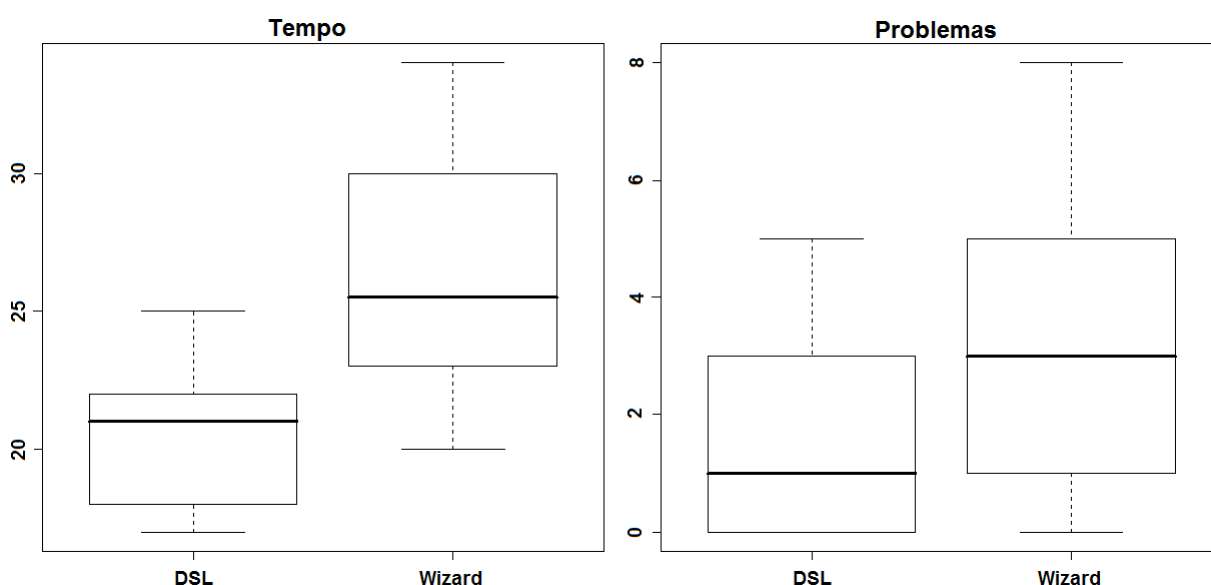


Figura 6.3. Boxplot criado a partir dos dados do experimento 1.

6.3.3.2 Teste das Hipóteses

Nesta seção são apresentados os resultados dos testes estatísticos aplicados sobre os dados coletados com o experimento. Para cada medida foi aplicado um conjunto de testes explicado a seguir.

Tempo

O P-valor resultante do teste *Shapiro-Wilk* sobre os dados do tempo gasto no desenvolvimento das aplicações foi 0,4746 para o reúso do framework GRENJ com o wizard e 0,3383 para o reúso com a DSL. Portanto, como o P-valor dos dois testes foi superior a 0,05, pode-se afirmar, com nível de confiança de 95%, que os dados do tempo gasto no desenvolvimento das aplicações seguem uma distribuição normal. Isso pode ser verificado nos gráficos mostrados na Figura 6.4, em que dados estão distribuídos sobre a reta.

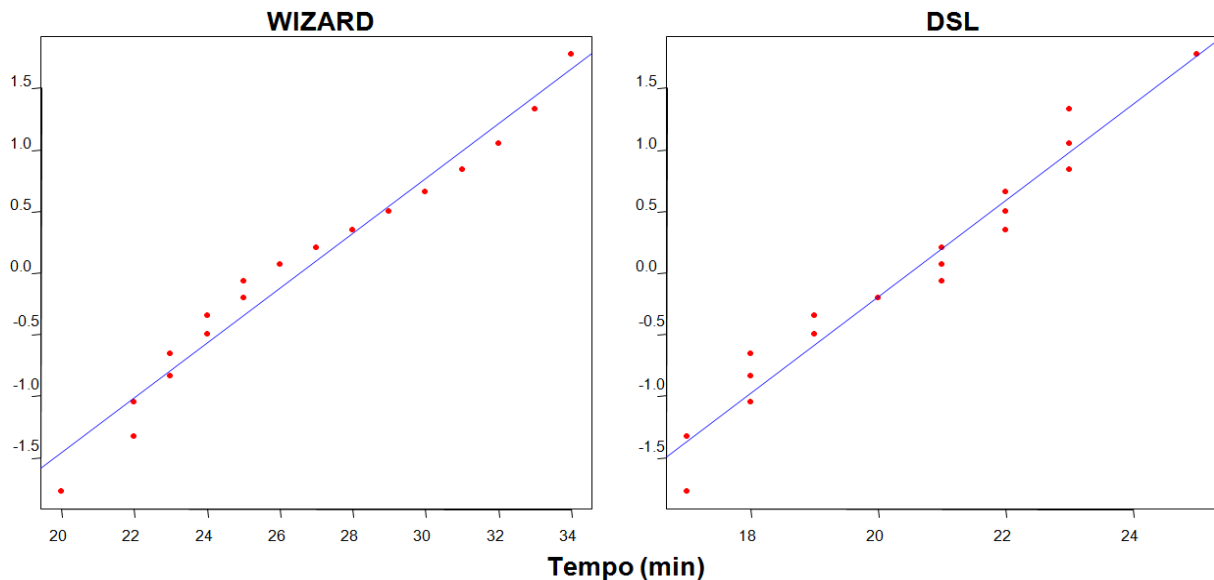


Figura 6.4. Gráficos resultantes do teste de normalidade sobre os dados do tempo gasto no desenvolvimento das aplicações.

Como os dados estão normalizados, o *Paired T-Test* foi aplicado sobre os dados do tempo gasto no desenvolvimento das aplicações para verificar as hipóteses da Q1 do experimento (Seção 6.3.1.2). O resultado desse teste foi um P-valor = $1,21E-05 < 0,05$, então, com nível de confiança de 95%, existem evidências de diferença entre o tempo gasto com o reuso do framework GRENJ com a DSL e com o wizard. Portanto, a hipótese nula (H_{10}) foi refutada e a H_{11} foi aceita, pois $t_{DSL} < t_{Wizard}$.

Número de Problemas

O P-valor resultante do teste *Shapiro-Wilk* sobre os dados do número de problemas no código-fonte das aplicações foi 0,2459 para o reuso do framework GRENJ com o wizard e 0,0064 para o reuso por meio da DSL. Portanto, como o P-valor para o reuso com a DSL foi inferior a 0,05, pode-se afirmar, com nível de confiança de 95%, que os dados obtidos relacionados ao número de problemas no código-fonte das aplicações não seguem uma distribuição normal. Isso também pode ser verificado nos gráficos da Figura 6.5, pois os dados obtidos relacionados ao número de problemas no reuso com a DSL não estão distribuídos sobre a reta.

Como os dados não estão normalizados, o teste *Wilcoxon Signed-Rank* foi aplicado sobre esses dados para verificar as hipóteses da Q2 do experimento (Seção 6.3.1.2). O resultado desse teste foi um P-valor = $0,0115 < 0,05$, então, com nível de confiança de 95%, existem evidências de diferença entre o número de problemas nas aplicações desenvolvidas em que ocorreu o reuso do framework GRENJ com o wizard e com a DSL. Portanto, a hipótese nula (H_{20}) foi refutada e a H_{21} foi aceita, pois $p_{DSL} < p_{Wizard}$.

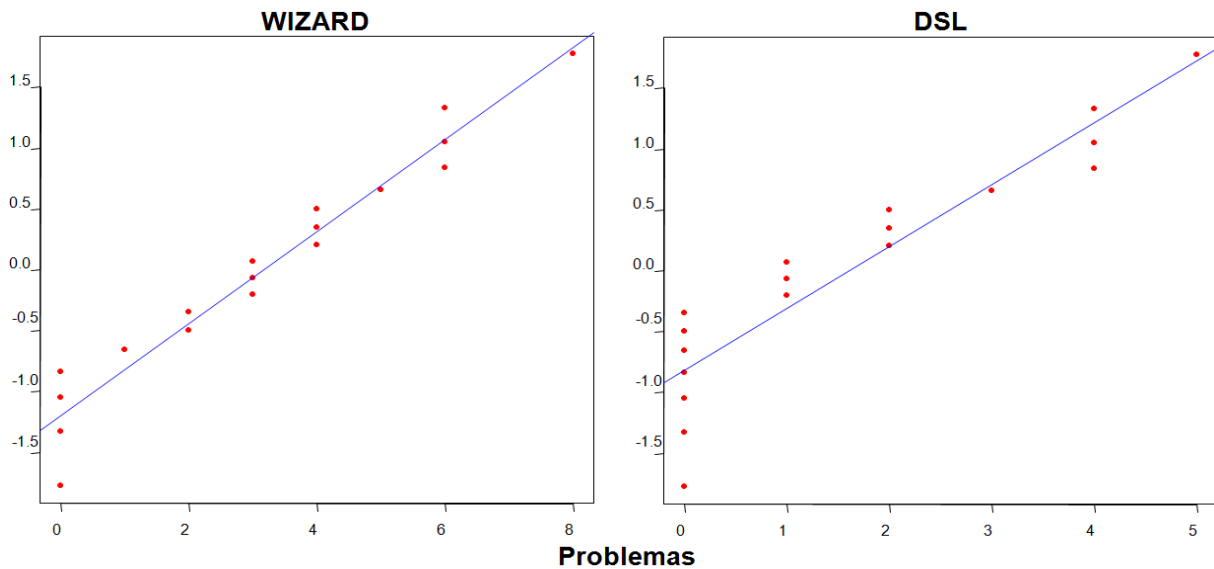


Figura 6.5. Gráficos resultantes do teste de normalidade sobre os dados do número de problemas no código-fonte das aplicações.

6.3.4 Ameaças à Validade do Experimento 1

Validade Interna:

- Requisitos das aplicações: a adição de funcionalidade não prevista pelos frameworks no código-fonte das aplicações não foi verificada neste experimento, pois essa atividade é realizada sem o apoio da DSL ou do wizard;
- Nível de experiência dos participantes: os participantes possuíam diferentes níveis de conhecimento e isso poderia afetar os dados coletados. Para mitigar essa ameaça, os participantes foram divididos em dois blocos equilibrados, considerando o seu nível de conhecimento com base no Formulário de Caracterização de Participante (Apêndice C). Além disso, todos os participantes possuíam experiência prévia no desenvolvimento de aplicações e foram previamente treinados para que soubessem reutilizar o framework GRENJ tanto pelo wizard quanto pela DSL;
- Produtividade sob avaliação: o resultado do experimento poderia ser influenciado, caso os participantes pensassem que estavam sendo avaliados e, por isso, precisavam desenvolver as aplicações com rapidez e sem erros. A fim de atenuar isso, foi explicado para os participantes que ninguém estava sendo avaliado e sua participação foi considerada anônima;
- Recursos utilizados durante o experimento: diferentes computadores e instalações poderiam afetar os tempos registrados. Assim, os indivíduos usaram a mesma configuração de hardware e de sistema operacional, bem como as mesmas ferramentas de desenvolvimento.

Validade Externa:

- Interação entre configuração e tratamento: é possível que as atividades realizadas no experimento não representem o desenvolvimento de aplicações do mundo real. Para atenuar essa ameaça, os exercícios foram projetados considerando o framework GRENJ, que implementa um domínio comumente retratado no mundo real, o de transações de aluguel, vendas e manutenção de bens e recursos.

Validade da Construção:

- Expectativas da hipótese: os participantes poderiam saber previamente que uma das abordagens deveria ser melhor que a outra. Estas questões poderiam ter afetado os dados e ter feito com que a experiência tenha sido menos imparcial. A fim de manter a imparcialidade, foi informado aos participantes que o mais importante era obter um resultado que refletisse a realidade, independente da hipótese que fosse concretizada.

Validade da Conclusão:

- Confiabilidade das métricas: refere-se às métricas usadas para medir o esforço de desenvolvimento. Para atenuar essa ameaça, foram consideradas métricas que refletem pontos importantes, como o tempo gasto no desenvolvimento e os problemas encontrados nas aplicações desenvolvidas;
- Baixa potência estatística: a capacidade de um teste estatístico em revelar dados confiáveis. Para atenuar essa ameaça, foram aplicados os testes *Paired T-Test* e *Wilcoxon Signed-Rank Test*, para analisar estatisticamente o tempo gasto para desenvolver as aplicações e o número de problemas encontrados nessas aplicações.

6.4 Experimento 2: Abordagem F3

Nesta seção é apresentado um experimento conduzido com o objetivo de analisar as vantagens propiciadas pelo uso da abordagem F3 (Capítulo 4) no desenvolvimento de frameworks (VIANA et al., 2013c). Para atingir esse objetivo, foi realizada uma comparação entre a abordagem F3 e uma abordagem *ad hoc*, adaptada da abordagem de Amatriain e Arumi (2011).

6.4.1 Definição e Planejamento do Experimento 2

O Experimento 2 foi definido como apresentado no Quadro 6.4. As etapas da fase de planejamento do experimento estão descritas nas Subseções 6.4.1.1 a 6.4.1.5.

Quadro 6.4. Definição do Experimento 2.

Análise da	abordagem F3
Com o propósito de	avaliação
Com respeito à	eficiência (tempo) e à dificuldade (número de problemas)
A partir do ponto de vista dos	desenvolvedores
No contexto de	de estudantes dos cursos de Mestrado em Ciência da Computação da Universidade Federal de São Carlos (UFSCar).

6.4.1.1 Contexto e Participantes

O contexto desse experimento foi **multi-teste dentro de um objeto de estudo** (WOHLIN et al., 2000), pois o experimento consistiu em testes experimentais executados por um grupo de indivíduos para estudar uma única abordagem, a abordagem F3. O experimento foi realizado em um laboratório de computação no ambiente universitário. Ao todo, participaram do experimento 26 alunos dos cursos de mestrado em Ciência da Computação da UFSCar. Todos tinham experiência prévia em desenvolvimento de software utilizando a linguagem Java e alguns deles já haviam trabalhado com alguns padrões de software e frameworks.

6.4.1.2 Formulação das Hipóteses

As questões, métricas e hipóteses definidas para esses experimento foram:

Questão 1 (Q1): Qual abordagem propicia o desenvolvimento de frameworks mais eficiente em termos de tempo?

Métrica 1 (M1): tempo (t) gasto no desenvolvimento dos frameworks.

Hipótese Nula (H1₀): Não há diferença significativa no tempo gasto pelos participantes no desenvolvimento dos frameworks, seja utilizando a abordagem F3 ou a *ad hoc*. Isso pode ser formalizado como $t_{F3} = t_{ad\ hoc}$.

Hipótese Alternativa (H1₁): O desenvolvimento de frameworks é mais eficiente quando a abordagem F3 é utilizada em vez da abordagem *ad hoc*. Isso pode ser formalizado como $t_{F3} < t_{Adhoc}$.

Hipótese Alternativa (H1₂): O desenvolvimento de frameworks é menos eficiente quando a abordagem F3 é utilizada em vez da abordagem *ad hoc*. Isso pode ser formalizado como $t_{F3} > t_{Adhoc}$.

Questão 2 (Q2): Em qual abordagem os participantes desenvolveram frameworks com o menor número de problemas?

Métrica 2 (M2): números de problemas (p) encontrados no código-fonte dos frameworks desenvolvidos pelos participantes.

Hipótese Nula ($H2_0$): Não há diferença significativa no número de problemas encontrados nos frameworks desenvolvidos pelos participantes, seja com a abordagem F3 ou com a *ad hoc*. Isso pode ser formalizado como $p_{F3} = p_{ad hoc}$.

Hipótese Alternativa ($H2_1$): Os frameworks desenvolvidos com a abordagem F3 possuem um menor número de problemas do que os desenvolvidos com a abordagem *ad hoc*. Isso pode ser formalizado como $p_{F3} < p_{ad hoc}$.

Hipótese Alternativa ($H2_2$): Os frameworks desenvolvidos com a abordagem F3 possuem um maior número de problemas do que os desenvolvidos com a abordagem *ad hoc*. Isso pode ser formalizado como $p_{F3} > p_{ad hoc}$.

6.4.1.3 Variáveis

Esse experimento teve as seguintes variáveis independentes: 1) o Astah Community 6.4; 2) o Eclipse IDE versão 4.2.1; 3) a linguagem de programação Java, e 4) os frameworks desenvolvidos pelos participantes: 4.a) um para o domínio de transações de aluguel e comercialização (Fw1) e 4.b) outro para o domínio de veículos autônomos (Fw2), adaptado do Lego Mindstorms⁵. Os dois possuíam 10 características e complexidade semelhante.

As variáveis dependentes foram: 1) eficiência, que está relacionada com o tempo gasto para o desenvolvimento dos frameworks, e 2) o número de problemas encontrados nos frameworks desenvolvidos.

6.4.1.4 Modelo

O modelo de experimento utilizado foi **um fator com dois tratamentos pareados** (WOHLIN et al., 2000). O **fator** foi a abordagem de desenvolvimento de frameworks, enquanto que o **tratamento** foram as abordagens aplicadas: F3 e *ad hoc*.

O experimento seguiu o formato em que os participantes são alocados em grupos homogêneos para que o nível de experiência deles não impactasse nos resultados. Um Formulário de Caracterização de Participante (Apêndice C), semelhante ao utilizado no Experimento 1, foi distribuído para determinar o nível de experiência de cada participante. Cada questão possuía a seguinte pontuação: 0, quando o participante não tinha

⁵ <http://mindstorms.lego.com/en-us/products/default.aspx\#t>

conhecimento algum; 1, quando o participante tinha apenas conhecimento teórico, e 2, quando o participante tinha conhecimento teórico e prático.

Nessa fase de planejamento do experimento, foi decidido que os participantes seriam divididos em dois grupos (G1 e G2), sendo que cada grupo ficariam com o mesmo número de participantes que obtiveram pontuação baixa (0-2), média (3-5) e alta (6-8) no Formulário de Caracterização de Participante.

No quadro 6.5 é apresentado como foram definidas as atividades de desenvolvimento dos frameworks para cada um dos participantes do G1 e do G2. Nesse experimento optou-se que, em cada atividade, os dois grupos aplicariam a mesma abordagem, porém em frameworks diferentes. Esse formato foi escolhido para que a abordagem F3 fosse aplicada somente na segunda atividade e nenhum dos dois grupos utilizasse o conhecimento aprendido com o uso dos padrões da LPF3 durante o desenvolvimento com a abordagem *ad hoc*.

Quadro 6.5. Atividades do Experimento 2.

Atividade	Abordagem	Framework Desenvolvido	
		Grupo 1 (G1)	Grupo 2 (G2)
1	<i>Ad hoc</i>	Fw1	Fw2
2	F3	Fw2	Fw1

Para realizar o experimento os participantes receberam um tutorial sobre frameworks e seus mecanismos mais comuns e também foram treinados no uso das abordagens *ad hoc* e F3 (Capítulo 4). Nesse experimento, para desenvolver os frameworks aplicando a abordagem *ad hoc* os participantes tiveram que criar o modelo de classes e implementar o código-fonte. Já para desenvolver um framework aplicando a abordagem F3 os participantes tiveram que criar o modelo F3 do domínio e implementar o código-fonte do framework com base na LPF3.

6.4.1.5 Instrumentação

Os participantes receberam os seguintes materiais para a execução do experimento: descrição dos domínios dos frameworks; documentação da LPF3; unidades de teste para verificar a existência de problemas nos frameworks desenvolvidos; e o Formulário de Coleta de Dados (Apêndice C), no qual eram preenchidos o tempo gasto no desenvolvimento dos frameworks e as mensagens de erro retornadas pelas unidades de testes.

6.4.2 Operação do Experimento 2

Depois de definir e planejar o experimento, sua fase de operação foi realizada em duas etapas: 1) Preparação e 2) Execução.

6.4.2.1 Preparação

Alguns dias antes da realização do experimento, os participantes preencheram o Formulário de Caracterização de Participante (Apêndice C), reportando sua experiência com relação aos conceitos e tecnologias utilizados no experimento. Os participantes também foram treinados no desenvolvimento de frameworks utilizando as abordagens F3 e *ad hoc* para que aprendessem a aplicá-las corretamente durante o experimento.

6.4.2.2 Execução

Os participantes foram inseridos nos grupos com base na sua pontuação no Formulário de Caracterização de Participante. O G1 ficou com 6 participantes com pontuação baixa (0-2), 4 com pontuação média (3-5) e 3 com pontuação alta (6-8). O G2 ficou com 7 participantes com pontuação baixa (0-2), 3 com pontuação média (3-5) e 3 com pontuação alta (6-8). Cada grupo ficou com 13 participantes no total.

Na Atividade 1, os participantes do G1 desenvolveram o Fw1 (domínio de transações de aluguel e comercialização) e os do G2 desenvolveram o Fw2 (domínio de veículos autônomos), ambos aplicando a abordagem *ad hoc*. Cada participante desenvolveu seu framework individualmente e cronometrou o tempo gasto nessa atividade desde a modelagem até a implementação do código-fonte. Depois disso, com o cronômetro parado, os participantes executavam as unidades de teste para verificar se o framework foi implementado corretamente ou não. Quando as unidades de testes retornavam mensagens de erro, os participantes escreviam essas mensagens no Formulário de Coleta de Dados (Apêndice C), mediam o tempo gasto para corrigir os problemas e executavam novamente as unidades de teste. O tempo gasto com interrupções também foi anotado pelos participantes para que fosse abatido do tempo total de desenvolvimento dos frameworks.

A Atividade 2 foi executada da mesma forma que a Atividade 1. Contudo, nessa segunda atividade a abordagem F3 foi utilizada, sendo que os participantes do G1 desenvolveram o Fw2 e os participantes do G2 desenvolveram o Fw1.

6.4.3 Análise dos Dados do Experimento 2

Os dados do experimento estão apresentados no Quadro 6.6. Os participantes desenvolveram as atividades de acordo com o planejado. A análise dos dados é apresentada nas subseções seguintes.

Quadro 6.6. Dados coletados do experimento 2.

Par.	Tempo (minutos)		Número de Problemas									
			Incoerência		Estrutura		Bad Smells		Interface		Total	
	Ad hoc	F3	Ad hoc	F3	Ad hoc	F3	Ad hoc	F3	Ad hoc	F3	Ad hoc	F3
S1	108	72	5	1	2	0	2	2	7	0	16	3
S2	113	74	7	1	4	1	2	0	4	1	17	3
S3	139	83	9	3	11	1	3	1	12	2	35	7
S4	124	78	7	1	5	2	2	1	7	2	21	6
S5	101	67	4	0	3	0	1	0	3	0	11	0
S6	133	81	8	4	7	3	3	3	9	3	27	13
S7	131	79	5	3	3	1	2	1	6	2	16	7
S8	116	73	6	1	5	0	3	0	5	1	19	2
S9	109	79	7	1	4	2	2	1	7	2	20	6
S10	106	69	4	2	3	0	1	0	3	1	11	3
S11	119	71	4	1	4	1	2	0	7	0	17	2
S12	148	83	8	3	6	1	3	3	11	4	28	11
S13	110	74	4	1	2	1	3	1	5	0	14	3
S14	107	72	2	1	3	0	3	0	6	1	14	2
S15	117	76	5	3	5	2	2	1	4	2	16	8
S16	97	68	3	1	1	0	2	0	3	0	9	1
S17	137	80	8	5	9	4	3	3	10	3	30	15
S18	121	75	4	1	6	2	2	2	2	2	14	7
S19	115	73	3	0	4	1	2	0	4	0	13	1
S20	134	81	7	2	6	3	3	1	9	3	25	9
S21	144	86	9	3	7	3	3	3	12	6	31	15
S22	111	76	3	2	4	1	1	1	5	1	13	5
S23	129	83	7	4	8	3	3	2	11	3	29	12
S24	123	79	5	2	5	1	3	1	7	2	20	6
S25	127	77	5	3	3	1	1	0	4	1	13	5
S26	131	78	6	2	4	1	2	2	6	2	18	7
Média	121,15	76,42	5,58	1,96	4,77	1,35	2,27	1,11	6,5	1,69	19,11	6,11
Mediana	120	76,5	5	2	4	1	2	1	6	2	17	6
Desvio Padrão	13,47	4,99	1,98	1,28	2,28	1,13	0,72	1,07	2,94	1,43	7,08	4,29
%	61,32	38,68	73,98	26,02	77,99	22,01	67,05	32,95	79,34	20,66	75,76	24,24

6.4.3.1 Análise Descritiva dos Dados

No Quadro 6.6 pode ser visto que com a abordagem F3 os participantes gastaram menos tempo para desenvolver os frameworks do que com a abordagem *ad hoc*. Considerando o tempo total de execução do experimento, cerca de 38,7% foi gasto com a abordagem F3 e 61,3 % com a abordagem *ad hoc*. De acordo com o *feedback* fornecido

pelos participantes no Formulário de Coleta de Dados, esse resultado se deve ao fato de que, com a abordagem F3, os padrões da LPF3 auxiliaram na construção das classes do framework. Embora parte do tempo gasto na atividade foi com a identificação desses padrões, houve ganho uma vez que eles indicam as classes, atributos e operações que deveriam ser implementados. Por outro lado, quando os participantes estavam desenvolvendo os frameworks aplicando a abordagem *ad hoc*, grande parte do tempo foi gasto tentando identificar quais as unidades de código deveriam ser implementadas. Além disso, a maioria dos participantes relatou que passou a maior parte do tempo corrigindo o código implementado, porque as unidades do teste retornaram muitas mensagens de erro.

No Quadro 6.6 também é possível visualizar os quatro tipos de problemas que foram analisados nos frameworks desenvolvidos pelos participantes: incoerência, estrutura, *bad smells* e interface incompleta.

O problema de incoerência indica que os participantes não desenvolveram os frameworks com as características e restrições (recursos obrigatórios, opcionais e alternativas) esperadas. Em outras palavras, eles não projetaram e implementaram todas as classes, atributos e operações necessárias para que o framework se comportasse como determina o seu domínio. Como pode ser visto no Quadro 6.6, de todos os problemas de incoerência encontrados, 26% ocorreram em frameworks desenvolvidos com a abordagem F3 e 74% em frameworks desenvolvidos com a abordagem *ad hoc*.

O problema da estrutura indica que os participantes não implementaram os frameworks com todos os elementos necessários. Por exemplo, a implementação de classes sem construtor. No Quadro 6.6 observa-se que de todos os problemas de estrutura encontrados, 22% ocorreram em frameworks desenvolvidos com a abordagem F3 e 78% em frameworks desenvolvidos com a abordagem *ad hoc*.

O problema de *bad smells* indica pontos fracos do projeto dos frameworks desenvolvidos que não afetam sua funcionalidade, mas os tornam mais difíceis de serem mantidos (Fowler et al., 1999). Esse problema não foi detectado pelas unidades de teste, portanto, os participantes não os corrigiram. Porém, esse problema foi identificado a partir de análises no código-fonte dos frameworks, nos procurou-se identificar três tipos de *bad smells*: classes que misturam métodos com propósitos diferentes; métodos grandes demais; e atributos e métodos repetidos em várias classes. De todos os problemas de *bad smells* encontrados, 33% deles ocorreram em frameworks desenvolvidos com a abordagem F3 e 67% em frameworks desenvolvidos com a abordagem *ad hoc*, pelos dados apresentados no Quadro 6.6.

O problema de interface incompleta indica ausência de operações *getter/setter* e de operações que permitem ao framework identificar as classes das aplicações. Normalmente, esse tipo de problema é consequência dos problemas de estrutura, portanto, o número de

problemas com esses dois tipos é bastante semelhante. Conforme pode ser visto no Quadro 6.6, dos problemas encontrados relacionados à interface incompleta, 21% deles ocorreram em frameworks desenvolvidos com a abordagem F3 e 79% em frameworks desenvolvidos com a abordagem *ad hoc*.

Na Figura 6.6 é mostrado o gráfico *boxplot* que representa a distribuição dos dados do experimento. Nesse gráfico, as caixas representam 50% dos dados e a linha em destaque dentro dessas caixas representa a mediana. As delimitações na forma de um T acima e abaixo das caixas indicam, respectivamente, o maior e o menor valor dos dados considerados válidos. É possível perceber que nenhum dos dados foi considerado atípico, pois não há pontos além das delimitações em forma de T.

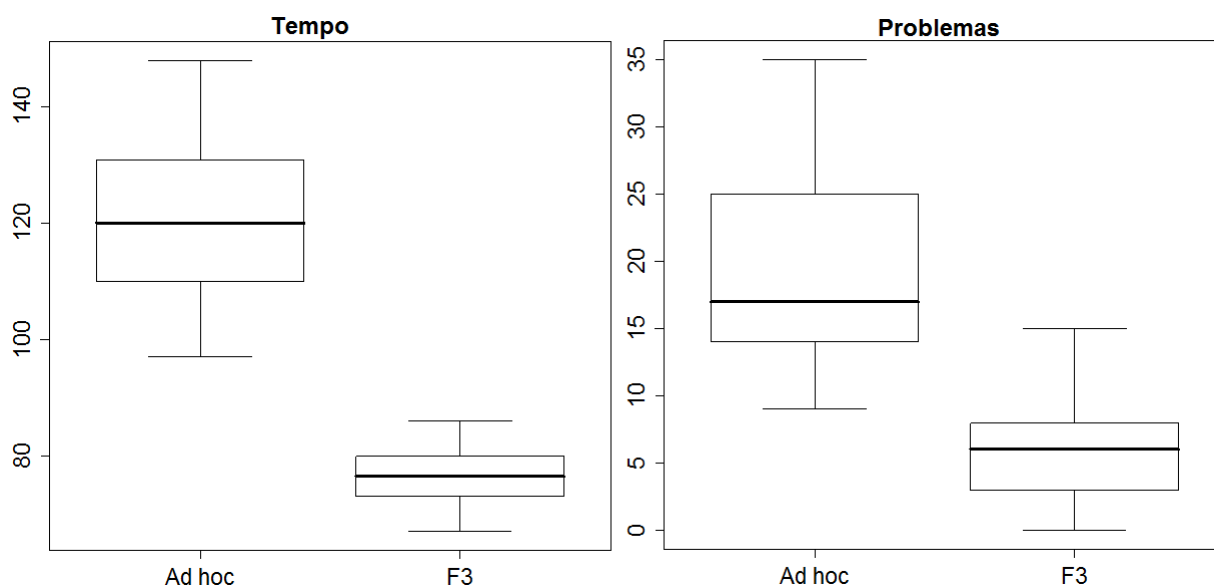


Figura 6.6. Boxplot criado a partir dos dados do experimento 2.

6.4.3.2 Teste das Hipóteses

Nesta seção são apresentados os resultados dos testes estatísticos aplicados sobre os dados coletados com o experimento realizado. Para cada medida foi aplicado um conjunto de testes, conforme explicado a seguir.

Tempo

O P-valor resultante do teste *Shapiro-Wilk* sobre os dados obtidos com o tempo gasto no desenvolvimento dos frameworks foi 0,8589 com a abordagem *ad hoc* e 0,878 com a abordagem F3. Portanto, como nos dois casos o P-valor foi superior a 0,05, pode-se afirmar, com nível de confiança de 95%, que esses dados seguem uma distribuição normal. Isso também pode ser verificado nos gráficos mostrados na Figura 6.7, uma vez que os dados estão distribuídos sobre a reta.

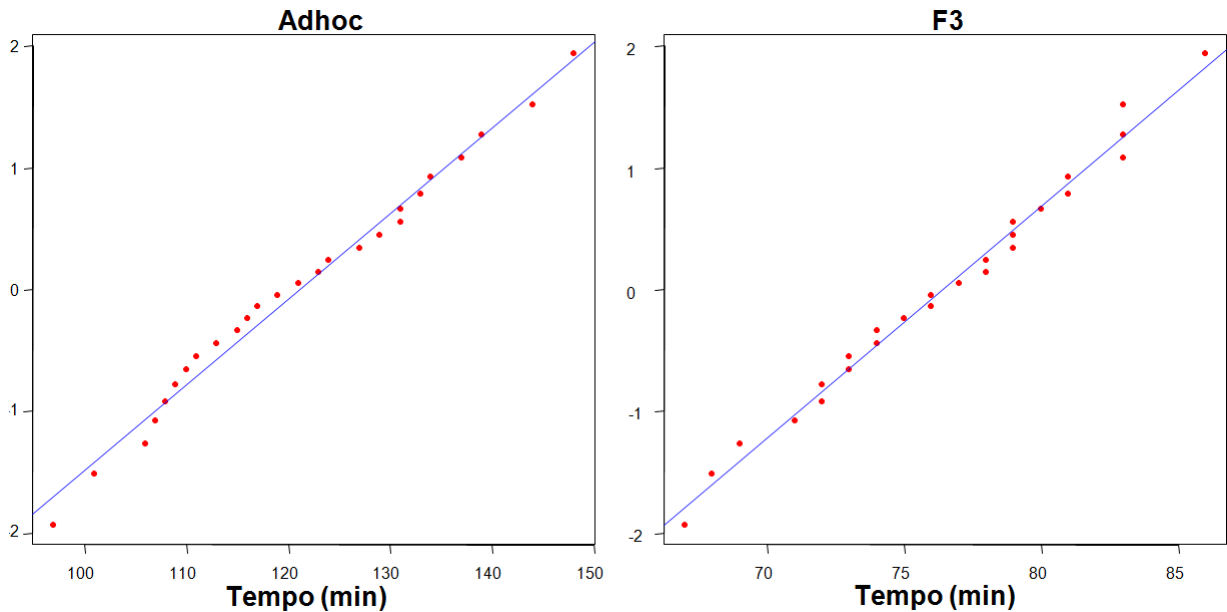


Figura 6.7. Gráficos resultantes do teste de normalidade sobre os dados do tempo gasto no desenvolvimento dos frameworks.

Como os dados estão normalizados, o *Paired T-Test* foi aplicado sobre os dados do tempo gasto no desenvolvimento dos frameworks para verificar as hipóteses da Q1 do experimento. O resultado desse teste foi um P-valor = $6,08E-19 < 0,05$, então, com nível de confiança de 95%, existem evidências de diferença entre o tempo gasto no desenvolvimento dos frameworks com a abordagem F3 e a *ad hoc*. Portanto, a hipótese nula ($H1_0$) foi refutada e a $H1_1$ foi aceita, pois $t_{F3} < t_{ad hoc}$.

Número de Problemas

O P-valor resultante do teste *Shapiro-Wilk* sobre os dados referentes ao número de problemas nos frameworks desenvolvidos foi 0,0438 com a abordagem *ad hoc* e 0,669 com a abordagem F3. Portanto, como um dos P-valores dos testes foi inferior a 0,05, pode-se afirmar, com nível de confiança de 95%, que os dados do número de problemas nos frameworks desenvolvidos não seguem uma distribuição normal. Isso também pode ser verificado nos gráficos mostrados na Figura 6.8, pois os dados não estão distribuídos sobre as retas.

Como os dados não estão normalizados, o teste *Wilcoxon Signed-Rank* foi aplicado sobre os dados referentes ao número de problemas nos frameworks desenvolvidos para verificar as hipóteses da Q2 do experimento. O resultado desse teste foi um P-valor = $8,49E-06 < 0,05$, então, com nível de confiança de 95%, existem evidências de diferença o número de problemas nos frameworks desenvolvidos com a abordagem F3 e a *ad hoc*. Portanto, a hipótese nula ($H2_0$) foi refutada e a $H2_1$ foi aceita, pois $p_{F3} < p_{ad hoc}$.

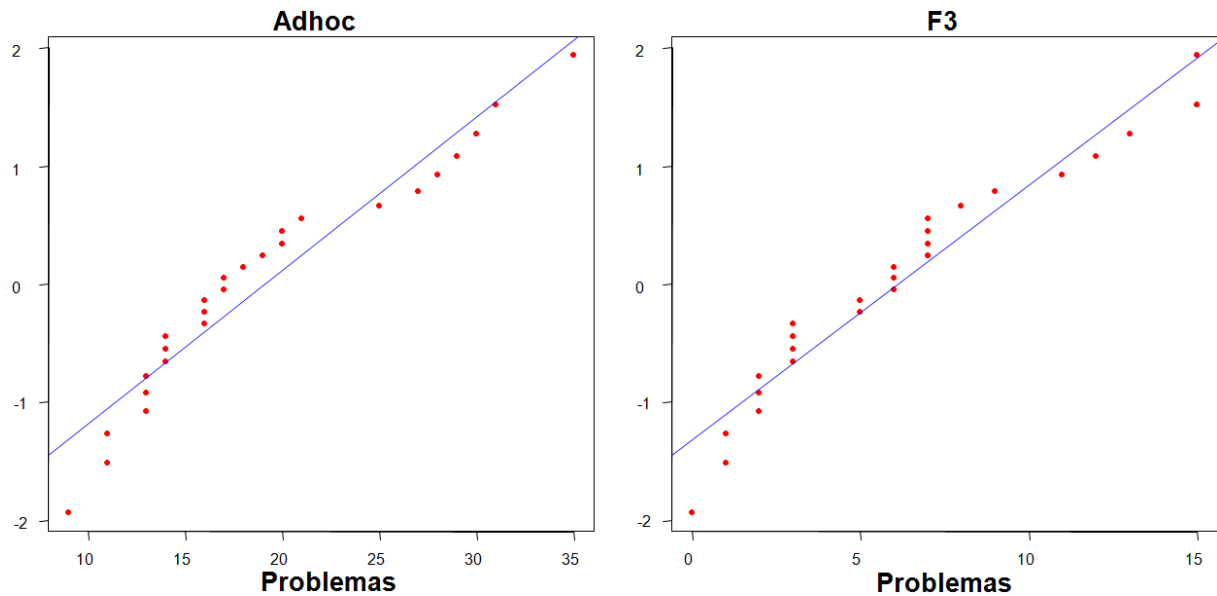


Figura 6.8. Gráficos resultantes do teste de normalidade sobre os dados do número de problemas nos frameworks desenvolvidos pelos participantes.

6.4.4 Ameaças à Validade do Experimento 2

Validade Interna:

- Influência entre abordagens: os participantes poderiam ter aprendido com o treinamento realizado antes do experimento e aplicado regras da abordagem F3 durante o uso da abordagem *ad hoc*. Para evitar isso, procurou-se não repetir as regras a serem aplicadas nos treinamentos e durante a execução do experimento. Além disso, a abordagem *ad hoc* sempre foi aplicada antes da abordagem F3;
- Nível de experiência dos participantes: os participantes possuíam diferentes níveis de conhecimento, podendo afetar os dados coletados. Para mitigar essa ameaça, os participantes foram divididos em dois blocos equilibrados, considerando o seu nível de conhecimento com base no Formulário de Caracterização de Participante (Apêndice C). Além disso, todos os participantes foram treinados a respeito de frameworks e das abordagens aplicadas;
- Produtividade sob avaliação: o resultado do experimento poderia ser influenciado, caso os participantes pensassem que estavam sendo avaliados e, por isso, precisavam desenvolver as aplicações com rapidez e sem erros. A fim de atenuar isso, foi explicado para os participantes que ninguém estava sendo avaliado e sua participação foi considerada anônima;
- Recursos utilizados durante o experimento: diferentes computadores e instalações poderiam afetar os tempos registrados. Assim, os indivíduos usaram a mesma configuração de hardware e de sistema operacional.

Validade Externa:

- Interação entre configuração e tratamento: Pode ser argumentado que a abordagem foi testada apenas no meio acadêmico, pois os participantes do experimento são estudantes. Contudo, o experimento foi realizado atendendo a todos os pontos evidenciados por seus autores, como ocorre em um ambiente industrial.

Validade da Construção:

- Expectativas da hipótese: os participantes poderiam saber previamente que uma das abordagens deveria ser melhor que a outra. Estas questões poderiam ter afetado os dados e ter feito com que a experiência tenha sido menos imparcial. A fim de manter a imparcialidade, foi informado aos participantes que o mais importante era obter um resultado que refletisse a realidade, independente da hipótese que fosse concretizada.

Validade da Conclusão:

- Confiabilidade das métricas: refere-se às métricas usadas para medir o esforço de desenvolvimento. Para atenuar essa ameaça, foram consideradas métricas que refletem pontos importantes, como o tempo gasto no desenvolvimento e os problemas encontrados nas aplicações desenvolvidas;
- Baixa potência estatística: a capacidade de um teste estatístico em revelar dados confiáveis. Para atenuar essa ameaça, foram aplicados os testes *Paired T-Test* e *Wilcoxon Signed-Rank Test*, para analisar estatisticamente o tempo gasto para desenvolver as aplicações e o número de problemas encontrados nessas aplicações.

6.5 Experimento 3: Ferramenta F3T

Nesta seção é apresentado um experimento conduzido com o objetivo de analisar o uso da ferramenta F3T (Capítulo 5) nas fases de Engenharia de Domínio e Engenharia da Aplicação. Para atingir esse objetivo, foi realizada uma comparação entre F3T e a Pure::variants (PURE SYSTEMS, 2013) (Seção 2.5.1).

6.5.1 Definição e Planejamento do Experimento 3

Esse experimento foi definido como apresentado no Quadro 6.7. As etapas da fase de planejamento do experimento estão descritas nas Subseções 6.5.1.1 a 6.5.1.5.

Quadro 6.7. Definição do Experimento 3.

Análise da	ferramenta F3T
Com o propósito de	avaliação
Com respeito à	eficiência (tempo)
A partir do ponto de vista dos	desenvolvedores
No contexto de	de estudantes de mestrado e de graduação dos cursos da área de Computação da Universidade Federal de São Carlos (UFSCar).

6.5.1.1 Contexto e Participantes

O contexto desse experimento foi **multi-teste dentro de um objeto de estudo** (WOHLIN et al., 2000), uma vez que foi constituído de testes experimentais executados por um grupo de indivíduos para estudar uma única abordagem, que é a ferramenta F3T. O experimento foi realizado em um laboratório de computação no ambiente universitário. Ao todo, participaram do experimento 32 alunos dos cursos de mestrado em computação e de graduação em computação, Bacharelado em Ciência da Computação e Engenharia de Computação, da UFSCar. Todos tinham experiência prévia em desenvolvimento de software utilizando a linguagem Java e alguns deles já haviam realizado implementação de sistemas de software utilizando alguns padrões de software e frameworks.

6.5.1.2 Formulação das Hipóteses

As seguintes questões, métricas e hipóteses foram definidas para esse experimento:

Questão 1 (Q1): Qual ferramenta permite que os artefatos de Engenharia do Domínio e da Aplicação sejam desenvolvidos com maior eficiência?

Métrica 1 (M1): tempo (t) gasto pelos participantes no desenvolvimento dos artefatos de Engenharia do Domínio e da Aplicação.

Hipótese Nula (H₁₀): Não há diferença significativa no tempo gasto pelos participantes no desenvolvimento dos artefatos de Engenharia do Domínio e da Aplicação, seja utilizando a ferramenta F3T ou a Pure::variants. Isso pode ser formalizado como $t_{F3T} = t_{pure}$.

Hipótese Alternativa (H₁₁): O desenvolvimento dos artefatos de Engenharia do Domínio e da Aplicação é mais eficiente quando a F3T é utilizada em vez da Pure::variants. Isso pode ser formalizado como $t_{F3T} < t_{pure}$.

Hipótese Alternativa (H₁₂): O desenvolvimento dos artefatos de Engenharia do Domínio e da Aplicação é menos eficiente quando a F3T é utilizada em vez da Pure::variants. Isso pode ser formalizado como $t_{F3T} > t_{pure}$.

6.5.1.3 Variáveis

Esse experimento teve as seguintes variáveis independentes: 1) o Eclipse IDE versão 4.2.1 com a F3T instalada; 2) a ferramenta Pure::variants versão de avaliação 3.2; 3) a linguagem de programação Java; 4) os domínios desenvolvidos pelos participantes: 4.a) um para transações de compra e aluguel (ED1) (semelhante ao apresentado na Seção 4.3) com uma aplicação uma biblioteca (EA1) e 4.b) um domínio para clínicas médicas (ED2) (semelhante ao apresentado na Seção 5.3) com uma aplicação para uma clínica veterinária (EA2).

A variável dependente é a eficiência, que está relacionada ao tempo gasto para o desenvolvimento dos frameworks e das aplicações.

6.5.1.4 Modelo

O modelo de experimento utilizado foi **um fator com dois tratamentos pareados** (WOHLIN et al., 2000). Nesse experimento, o **fator** foi a ferramenta utilizada no desenvolvimento dos artefatos de Engenharia do Domínio e da Aplicação, enquanto que os **tratamentos** foram as ferramentas utilizadas: F3T e Pure::variants.

O experimento seguiu o formato em que os participantes são alocados em grupos homogêneos para que o nível de experiência deles não impactasse nos resultados. Um Formulário de Caracterização de Participante (Apêndice C) foi distribuído para determinar o nível de experiência de cada participante. Nesse formulário os participantes tinham de responder questões de múltipla escolha a respeito do conhecimento deles sobre: 1) programação Java, 2) Eclipse IDE, 3) padrões de software; 4) frameworks; 5) MDE; 6) a ferramenta F3T e 7) a ferramenta Pure::variants. Cada questão possuía a seguinte pontuação: 0, quando o participante não tinha conhecimento algum; 1, quando o participante tinha apenas conhecimento teórico, e 2, quando o participante tinha conhecimento teórico e prático.

Os participantes foram treinados no uso das duas ferramentas utilizadas no experimento. Para esse experimento não foi encontrada uma outra ferramenta que gerasse o código-fonte de um framework e apoiasse o desenvolvimento de aplicações com o reuso desse framework, como a F3T. A Pure::variants foi selecionada porque, assim como a F3T, pode ser utilizada no contexto de linhas de produtos de software e possui as fases de Engenharia de Domínio e de Aplicação bem delimitadas.

Foi decidido que os participantes seriam divididos em dois grupos (G1 e G2), sendo que cada grupo ficaria com o mesmo número de participantes que obtiveram pontuação baixa (0-2), média (3-5) e alta (6-8) no Formulário de Caracterização de Participante. No

quadro 6.8 é apresentado como foram definidas as atividades de desenvolvimento dos framework para cada um dos participantes do G1 e do G2.

Quadro 6.8. Atividades do Experimento 3.

Atividade	Domínio/Aplicação	Ferramenta Utilizada	
		Grupo 1 (G1)	Grupo 2 (G2)
1	ED1/EA1	F3T	Pure::Variants
2	ED2/EA2	Pure::Variants	F3T

Quando fossem utilizar a F3T os participantes tinham que: 1) criar o modelo F3 do domínio; 2) gerar o código-fonte e a DSL do framework; 3) instalar a DSL; 4) modelar a aplicação com a DSL do framework; e 5) gerar o código-fonte da aplicação. Quando fossem utilizar a Pure::variants os participantes tinham que: 1) implementar a aplicação a partir do modelo de classes; 2) criar os modelos de arquitetura e de características do domínio a partir do código-fonte da aplicação; 3) definir o espaço de configuração; 4) selecionar as características da variante da aplicação; e 5) gerar o código-fonte da variante da aplicação.

6.5.1.5 Instrumentação

Os participantes receberam os seguintes materiais para a execução do experimento: documentação sobre os domínios e as aplicações; manuais da Pure:variants (Apêndice B) e da F3T (Apêndice C); um documento com a descrição das atividades a serem realizadas com cada ferramenta; e o Formulário de Coleta de Dados (Apêndice C), no qual os participantes preenchem o tempo gasto nas etapas de Engenharia do domínio e Engenharia da Aplicação.

6.5.2 Operação do Experimento 3

Depois de definir e planejar o experimento, sua fase de operação foi realizada em duas etapas: 1) Preparação e 2) Execução.

6.5.2.1 Preparação

Uma semana antes da execução do experimento os participantes foram treinados com o uso da F3T e da Pure:variants. Eles também preencheram o Formulário de Caracterização de Participante (Apêndice C), reportando sua experiência com relação aos conceitos utilizados no experimento. Um treinamento também foi realizado que os participantes obtivessem experiência com a realização das atividades do experimento.

6.5.2.2 Execução

Para a execução do experimento, primeiramente, os participantes foram colocados nos grupos de acordo com a sua pontuação obtida a partir do Formulário de Caracterização de Participante (Apêndice C). Cada grupo ficou com 16 participantes no total. O G1 ficou com 6 participantes com pontuação baixa (0-2), 4 com pontuação média (3-5) e 3 com pontuação alta (6-8). O G2 ficou com 7 participantes com pontuação baixa (0-2), 3 com pontuação média (3-5) e 3 com pontuação alta (6-8). Após a formação dos grupos os participantes receberam o material para executarem a Atividade 1. O tempo limite foi de 70 minutos para cada uma das atividades.

Na Atividade 1, os participantes desenvolveram a ED1 e a EA1, sendo que no G1 foi utilizada a F3T e no G2 foi utilizada a Pure::variants. Cada participante realizou as suas atividades individualmente e cronometrou o tempo gasto para as fases de Engenharia do Domínio e de Engenharia da Aplicação, separadamente.

A Atividade 2 foi executada da mesma forma que a Atividade 1. Contudo, na Atividade 2 os participantes desenvolveram o ED2 e a EA2, sendo que os participantes do G1 utilizaram a Pure::Variants e os do G2 utilizaram a F3T.

6.5.3 Análise dos Dados do Experimento 3

Os participantes desenvolveram as atividades de acordo com o esperado e a análise dos dados é apresentada nas subseções seguintes. Os dados referentes ao tempo gasto pelos participantes do experimento estão apresentados no Quadro 6.9, na qual foram usadas as seguintes siglas e abreviações: Participantes (Part.); Implementação da Aplicação Base (IAB), Engenharia do Domínio (ED) e Engenharia da Aplicação (EA).

6.5.3.1 Análise Descritiva dos Dados

No Quadro 6.9 pode ser visto que os participantes gastaram 45,27% do tempo total de execução do experimento utilizando a Pure:variants e 54,73 % utilizando a F3T para desenvolver os artefatos das fases ED e EA. O menor tempo ter ocorrido com o uso da Pure:variants se deve ao fato de que a maioria dos modelos utilizados, por essa ferramenta, é gerada a partir do código-fonte da aplicação base. Somente na EA, que era necessário que os participantes tomassem decisões sobre quais características da aplicação base seriam inclusos na variante a ser gerada. Enquanto que na F3T, os participantes tiveram que interpretar os requisitos do domínio e criar o seu modelo F3 na ED e criar o modelo da aplicação utilizando a DSL do framework.

Quadro 6.9. Dados coletados do experimento 3.

Domínio	Pure:variants				F3T			
	Part.	Tempo (min)			Part.	Tempo (min)		
		ED	EA	ED+EA		ED	EA	ED+EA
Clínicas Médicas	S1	19	10	29	S17	39	19	58
	S2	32	16	48	S18	20	8	28
	S3	30	8	38	S19	26	21	47
	S4	18	14	32	S20	18	10	28
	S5	15	9	24	S21	31	19	50
	S6	20	12	32	S22	36	26	62
	S7	31	11	42	S23	23	14	37
	S8	27	14	41	S24	35	22	57
	S9	26	18	44	S25	36	24	60
	S10	30	9	39	S26	32	18	50
	S11	23	13	36	S27	28	12	40
	S12	18	11	29	S28	23	16	39
	S13	22	10	32	S29	26	13	39
	S14	25	11	36	S30	31	22	53
	S15	21	15	36	S31	21	15	36
	S16	29	13	42	S32	24	19	43
Transações	S17	30	10	40	S1	33	18	51
	S18	14	11	25	S2	25	20	45
	S19	24	12	36	S3	38	18	56
	S20	33	10	43	S4	30	12	42
	S21	20	8	28	S5	30	9	39
	S22	28	15	43	S6	26	13	39
	S23	24	12	36	S7	31	20	51
	S24	33	16	49	S8	28	17	45
	S25	18	10	28	S9	20	11	31
	S26	32	17	49	S10	21	10	31
	S27	30	9	39	S11	24	13	37
	S28	29	15	44	S12	24	14	38
	S29	23	10	33	S13	27	16	43
	S30	25	12	37	S14	35	22	57
	S31	19	13	32	S15	29	17	46
	S32	22	14	36	S16	32	14	46
Média		24,69	12,13	36,81	-	28,19	16,31	44,5
Mediana		24,5	12	36		28	16,5	44
Desvio Padrão		5,5	2,67	6,69		5,64	4,57	9,31
Porcentagem		67,03	32,94	45,27		63,34	36,66	54,73

A desvantagem da Pure:variants em relação à F3T é que precisa existir uma aplicação base (conforme explicado na Seção 2.5.1) para a criação dos artefatos das fases de Engenharia de Domínio e de Engenharia de Aplicação. A implementação da aplicação base foi realizada manualmente com o apoio parcial de recursos de geração de código, como, por exemplo, a geração dos construtores e operações *getter/setter* a partir da declaração dos atributos das classes.

Na Figura 6.9 é mostrado o gráfico *boxplot* que representa a distribuição dos dados (ED+EA) do experimento. Nesse gráfico, as caixas representam 50% dos dados e a linha em destaque dentro dessas caixas representa a mediana. As delimitações na forma de um T acima e abaixo das caixas indicam, respectivamente, o maior e o menor valor dos dados considerados válidos. É possível perceber que nenhum dos dados foi considerado atípico, pois não há pontos além das delimitações em forma de T.

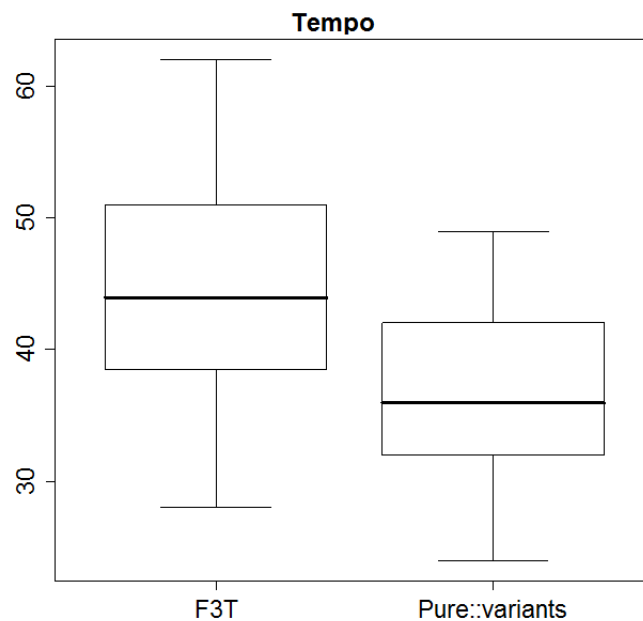


Figura 6.9. Boxplot criado a partir dos dados do experimento 3.

6.5.3.2 Teste das Hipóteses

Nesta seção são apresentados os resultados dos testes estatísticos aplicados sobre os dados coletados com o experimento.

Tempo

O P-valor resultante do teste *Shapiro-Wilk* referente aos dados do tempo gasto (ED+EA) foi 0,9532 com o uso da Pure:variants e 0,499 com o uso da F3T. Portanto, como nos dois casos o P-valor foi superior a 0,05, pode-se afirmar, com nível de confiança de 95%, que os dados do tempo gasto com o uso das duas ferramentas seguem uma distribuição normal. Isso também pode ser verificado nos gráficos mostrados Figura 6.10, pois os dados estão distribuídos sobre a reta.

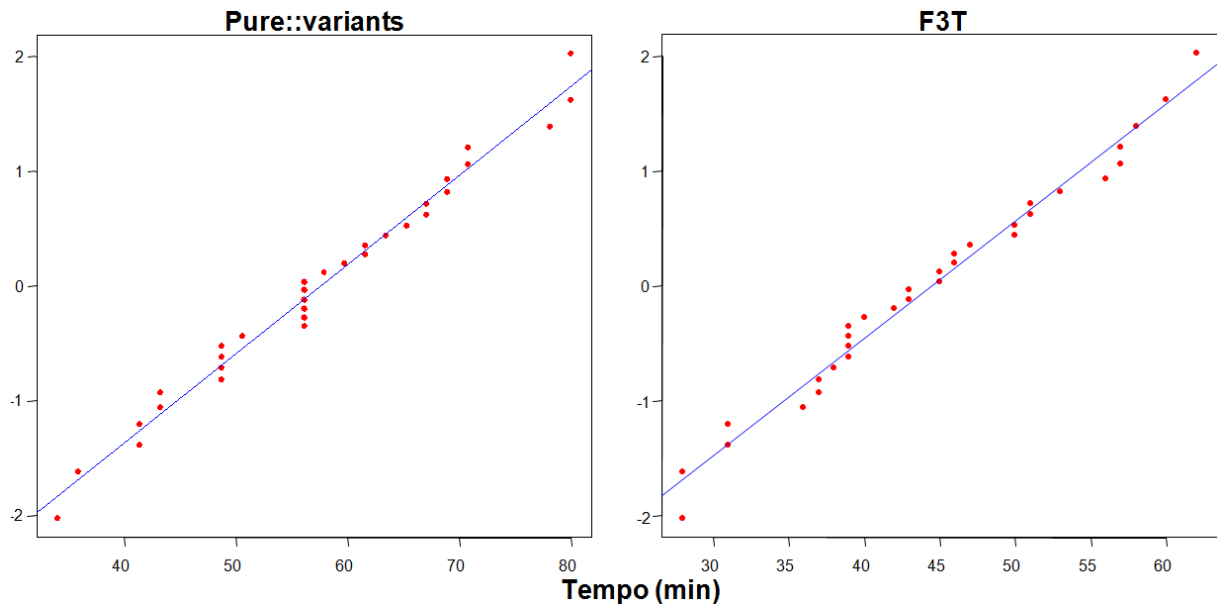


Figura 6.10. Gráficos resultantes do teste de normalidade sobre o tempo gasto com as ferramentas Pure::Variants e F3T.

Como os dados estão normalizados, o *Paired T-Test* foi aplicado sobre os dados do tempo gasto com a construção dos artefatos de ED e EA nas duas ferramentas. O resultado desse teste foi um P-valor = $1,19E-3 < 0,05$. Assim, com nível de confiança de 95%, existem evidências de diferença entre o tempo gasto na construção dos artefatos de ED e EA da Pure::variants e da F3T. Portanto, a hipótese nula (H_{10}) foi refutada e a H_{12} foi aceita, pois $t_{F3T} > t_{Pure}$.

Comentários dos Participantes sobre Facilidades e Dificuldades de Cada Ferramenta

No final de cada atividade do experimento, os participantes também preencheram no Formulário de Coleta de Dados (Apêndice C) explicitando quais foram as facilidades e dificuldades no uso das ferramentas utilizadas. A dificuldade mais citada pelos participantes sobre as duas ferramentas foi que a criação dos modelos possui muitos passos para serem executados. A justificativa para as duas ferramentas terem esse problema é que ambas têm como base o Eclipse IDE, cuja criação de arquivos é baseada no preenchimento de formulários.

Sobre a Pure:variants, os participantes comentaram que a principal facilidade é que os modelos na fase de ED são gerados, enquanto que na F3T o modelo F3 é criado manualmente pelo desenvolvedor. Contudo, cada um desses modelos da ED da Pure:variants é gerado a partir de uma sequência de 4 a 5 ações e, por isso, os participantes argumentaram que tiveram a necessidade de consultar o manual da ferramenta constantemente para saber como proceder. Outra dificuldade citada pelos participantes é o fato da Pure:variants não fornecer a opção de criação de um projeto Java

e criação de novas classes. Por causa disso, a aplicação base teve que ser implementada em um outro Eclipse IDE e, em seguida, ser importada para a Pure::variants.

Um fato importante não comentado pelos participantes é que a Pure::variants não é uma ferramenta de uso livre. No experimento foi utilizada uma versão de avaliação por tempo limitado. Ela utiliza como base o Eclipse IDE, mas não é um *plug-in* que pode ser instalado em um Eclipse IDE já existente no computador, como a F3T.

Sobre a F3T, os participantes citaram as seguintes facilidades: não é necessária nenhuma implementação prévia para desenvolver os modelos da ED; o número menor de modelos na ED (apenas 1 modelo) em comparação com a Pure::variants (4 modelos); a geração do framework do domínio; e a maior facilidade na customização das aplicações geradas.

Alguns participantes citaram que sentiram dificuldade na criação do modelo F3 na F3T, contudo, isso pode ser atribuído a dois fatores: a pouca experiência dos desenvolvedores na modelagem de domínios e a dificuldade de interpretação dos requisitos dos domínios utilizados no experimento.

Outro ponto em que os participantes sentiram dificuldade na F3T foi com relação à DSL do framework, sendo necessário consultar diversas vezes o manual da ferramenta. Como explicado no Capítulo 5, a F3T gera somente os modelos EMF/GMF e a partir desses são gerados os *plug-ins* da DSL. Para usar a DSL na fase de EA, esses *plug-ins* devem ser exportados para o diretório de *plug-ins* do Eclipse IDE e esse deve ser reinicializado. Caso os participantes tivessem maior experiência com o uso do Eclipse IDE e o GMF, eles não teriam dificuldades em realizar essas ações.

6.5.4 Ameaças à Validade do Experimento 3

Validade Interna:

- Diferenças entre as ferramentas: As ferramentas utilizadas no experimento geram artefatos diferentes. A Pure::variants gera um conjunto de modelos a partir do código-fonte de uma aplicação base e depois gera o código-fonte das variantes, que possuem um subconjunto das classes da aplicação base. Já a F3T gera o código-fonte e a DSL de um framework e depois gera o código-fonte das aplicações a partir de modelos criados com a DSL do framework. Apesar dessas diferenças, o objetivo final das duas ferramentas o desenvolvimento de aplicações com reuso;
- Nível de experiência dos participantes: os participantes possuíam diferentes níveis de conhecimento e isso poderia afetar os dados coletados. Para mitigar essa ameaça, os participantes foram divididos em dois blocos equilibrados, considerando o seu nível de conhecimento com base no Formulário de Caracterização de Participante

(Apêndice C). Além disso, todos os participantes possuíam experiência prévia no desenvolvimento de aplicações e foram previamente treinados para que soubessem utilizar a F3T e a Pure::variants;

- Produtividade sob avaliação: o resultado do experimento poderia ser influenciado, caso os participantes pensassem que estavam sendo avaliados e, por isso, precisavam desenvolver as aplicações com rapidez e sem erros. A fim de atenuar isso, foi explicado para os participantes que ninguém estava sendo avaliado e sua participação foi considerada anônima;
- Recursos utilizados durante o experimento: diferentes computadores e instalações poderiam afetar os tempos registrados. Assim, os indivíduos usaram a mesma configuração de hardware e de sistema operacional.

Validade Externa:

- Interação entre configuração e tratamento: Pode ser argumentado que a abordagem foi testada apenas no meio acadêmico, pois os participantes do experimento são estudantes. Contudo, o experimento foi realizado atendendo a todos os pontos evidenciados por seus autores, como ocorre em um ambiente industrial.

Validade da Construção:

- Expectativas da hipótese: os participantes poderiam saber previamente que uma das abordagens deveria ser melhor que a outra. Estas questões poderiam ter afetado os dados e ter feito com que a experiência tenha sido menos imparcial. A fim de manter a imparcialidade, foi informado aos participantes que o mais importante era obter um resultado que refletisse a realidade, independente da hipótese que fosse concretizada.

Validade da Conclusão:

- Confiabilidade das métricas: refere-se às métricas usadas para medir o esforço de desenvolvimento. Para atenuar essa ameaça, foi considerada uma métrica que reflete um ponto importante, como é o caso do tempo gasto no uso das ferramentas;
- Baixa potência estatística: a capacidade de um teste estatístico em revelar dados confiáveis. Para atenuar essa ameaça, foram aplicados os testes *Paired T-Test* e *Wilcoxon Signed-Rank Test*, para analisar estatisticamente o tempo gasto para desenvolver as aplicações e o número de problemas encontrados nessas aplicações.

6.6 Considerações Finais

Neste capítulo foram apresentados os experimentos realizados para avaliar as abordagens e a ferramenta desenvolvidas durante este doutorado. Esses experimentos foram previamente planejados com base nos objetivos pretendidos, nos recursos a serem utilizados, nos participantes e nas variáveis. Além disso, testes estatísticos foram aplicados para obter maior confiabilidade nos resultados obtidos. Um resumo dos resultados obtidos com os experimentos apresentados neste capítulo é mostrado no Quadro 6.10.

Quadro 6.10. Resultados dos experimentos.

Experimento	Objetivo	Resultado
1	Analisar o uso de DSLs para a instanciação de frameworks.	O uso de DSLs torna o reúso de frameworks mais eficiente e resulta em aplicações com menor número de problemas do que os wizards.
2	Analisar o uso da abordagem F3 para o processo de desenvolvimento de frameworks.	O uso da abordagem F3 torna o reúso de frameworks mais eficiente e reduz o número de problemas encontrados nos frameworks desenvolvidos quando comparada ao uso de uma abordagem com modelagem e implementações tradicionais (ad hoc).
3	Analisar o uso da F3T nas fases de Engenharia de Domínio e Engenharia de Aplicação.	O uso da F3T é menos eficiente do que o uso da Pure::variants em relação a construção dos artefatos das fases de Engenharia de Domínio e Engenharia de Aplicação.

Os experimentos corroboraram com as vantagens preconizadas pelos conceitos aplicados nas abordagens desenvolvidas neste doutorado. No Experimento 1 foi visto que a DSL torna a instanciação de frameworks mais fácil e eficiente, pois, com a DSL, as informações requeridas pelo framework são definidas e validadas durante a modelagem das aplicações. Além disso, o uso de modelos gráficos para a modelagem de aplicações é mais vantajoso do que o uso de wizards. Contudo, o experimento também mostrou que, tanto com o uso de wizards quanto de DSLs, é necessário que o desenvolvedor possua conhecimento sobre os domínios e as estruturas fornecidas pelos frameworks para que possa relacionar corretamente os requisitos das aplicações às classes dos frameworks.

No Experimento 2 foi visto que a abordagem DSL facilita o desenvolvimento de frameworks, uma vez que os padrões que foram propostos quando da criação dessa abordagem guiam os desenvolvedores durante a implementação das unidades de código. Desse modo, o desenvolvimento dos frameworks também se torna mais eficiente, uma vez que os desenvolvedores precisam somente seguir as sugestões dos padrões da abordagem. Contudo, a responsabilidade sobre o entendimento e a modelagem do domínio ainda dependem das habilidades e da experiência dos desenvolvedores.

No Experimento 3 foi visto que a F3T e a Pure::variants são ferramentas distintas, mas que apoiam o desenvolvimento dos artefatos das fases de Engenharia do Domínio e

Engenharia de Aplicação. A Pure::variants mostrou-se mais eficiente e mais indicada quando existe uma aplicação base a partir da qual as aplicações desenvolvidas serão derivadas. Por outro lado, a F3T é mais indicada quando o desenvolvimento se inicia a partir dos requisitos do domínio e quando se deseja obter um software intermediário (um framework) que é reutilizado pelas aplicações.

Capítulo 7

CONCLUSÕES

7.1 Considerações Finais da Tese de Doutorado

Nesta tese de doutorado o objetivo foi buscar soluções que facilitam o desenvolvimento e o reúso de frameworks. Para isso, foram criadas a abordagem *From Features to Frameworks* (F3) e a abordagem para Construção de DSLs de Frameworks a partir de conceitos relacionados com padrões de software, Engenharia Dirigida por Modelos e Linhas de Produtos de Software.

A abordagem F3 foi criada com base em dois princípios: 1) a definição dos domínios dos frameworks por meio de modelos de características; e 2) a construção das unidades de código dos frameworks com base em padrões encontrados nos modelos dos domínios.

A abordagem para Construção de DSLs de Frameworks fez uso das vantagens proporcionadas pelo poder de abstração dos modelos e pela geração de código para esconder as complexidades de instanciação dos frameworks e aumentar a eficiência do desenvolvimento de aplicações.

Além disso, nesta tese de doutorado também foi construída uma ferramenta que apoia o uso dessas duas abordagens, com editores gráficos para a modelagem dos domínios e das aplicações e automatiza as etapas relacionadas com a implementação de código dos frameworks, das DSLs e das aplicações.

As contribuições e as limitações desta tese, os trabalhos futuros e os artigos publicados são apresentados nas Seções 7.2 a 7.5.

7.2 Contribuições

As principais contribuições desta tese de doutorado são:

1. **Abordagem de construção de DSLs:** criada para auxiliar aos desenvolvedores de aplicações durante a instanciação de frameworks. Essa abordagem é independente de ferramenta e de linguagem de programação e pode ser aplicada tanto em frameworks de domínios de sistemas de informação (*Enterprise Application Frameworks*) quanto em frameworks com domínios não funcionais (*Middleware Integration Frameworks*). A partir dessa abordagem obteve-se:
 - a. **Maior facilidade de construção de DSLs:** a abordagem de Construção de DSLs fornece regras para a construção das sintaxes abstrata e concreta de DSLs a partir da identificação e da análise das características dos frameworks;
 - b. **Maior facilidade no reuso de frameworks:** os modelos gráficos das DSLs escondem as complexidades de implementação de código no reuso de frameworks;
 - c. **Maior eficiência nos processos de desenvolvimento de aplicações com reuso de frameworks:** o código-fonte das aplicações é gerado a partir dos modelos criados com a DSLs, economizando tempo que seria gasto na implementação manual;
 - d. **Validação dos modelos das aplicações:** os modelos das aplicações construídos com DSLs podem ser validados de acordo com as regras e restrições do domínio do framework;
 - e. **Maior confiabilidade nas aplicações desenvolvidas:** a possibilidade de existirem defeitos no código-fonte das aplicações geradas com DSLs é menor do que na implementação manual, pois os templates do gerador foram previamente testados;
2. **Abordagem *From Features To Frameworks* (F3):** criada para auxiliar aos desenvolvedores durante o desenvolvimento de frameworks. Essa abordagem é independente de linguagem de programação e pode ser aplicada manualmente ou com o apoio da ferramenta F3T. A partir dessa abordagem obteve-se:
 - a. **Maior facilidade na modelagem do domínio de frameworks:** na abordagem F3 os domínios são definidos em modelos F3, uma versão estendida dos modelos de características que inclui algumas propriedades dos metamodelos, como atributos e operações. Por meio dos modelos F3 os desenvolvedores podem definir as características e as regras do domínio do framework sem se preocupar com as complexidades de implementação;
 - b. **Maior facilidade na construção de frameworks:** a abordagem F3 não somente define um conjunto de etapas para o desenvolvimento de frameworks, mas também fornece uma linguagem de padrões que indica aos

desenvolvedores quais unidades de código devem ser criadas para que o framework implemente as características e variabilidades do domínio definido em um modelo F3. Os padrões dessa linguagem propõem soluções relacionadas com a estrutura, a interface e a persistência do framework;

3. **From Features to Framework Tool (F3T)**: ferramenta que serve de apoio para as abordagens F3 e de Construção de DSLs de Frameworks, fornecendo os seguintes recursos: editor gráfico para modelos F3 (modelagem do domínio); gerador do código-fonte e da DSL de frameworks; editor gráfico para a criação de modelos de aplicações com as DSLs; gerador de código-fonte das aplicações. Com essa ferramenta obteve-se:
 - a. **Maior eficiência no desenvolvimento do código-fonte e da DSL dos frameworks**: o código-fonte dos frameworks é gerado a partir dos modelos F3, economizando tempo que seria gasto no uso da linguagem de padrões da abordagem F3 e na implementação manual;
 - b. **Maior confiabilidade nos frameworks desenvolvidos**: a possibilidade de existirem defeitos no código-fonte dos frameworks é menor do que na implementação manual, pois o gerador da F3T aplica automaticamente os padrões da abordagem F3, evitando que algum deles não seja aplicado ou seja aplicado incorretamente;

7.3 Limitações

As abordagens e a ferramenta desenvolvidos nesta tese de doutorado possuem as seguintes limitações:

1. A abordagem de Construção de DSLs de frameworks descreve como criar o metamodelo que define a sintaxe abstrata das DSLs. Contudo, a forma como a sintaxe concreta é criada depende da ferramenta utilizada nesse processo e do conhecimento que o desenvolvedor possui sobre essa ferramenta. Nesta tese de doutorado foram realizados exemplos práticos com o GMF no Eclipse IDE;
2. As DSLs aumentam o nível de abstração do processo de instanciação dos frameworks, escondendo as complexidades relacionadas com a configuração dos *hot-spots*. Contudo, o mapeamento dos requisitos das aplicações sobre as características do domínio continua sendo responsabilidade dos desenvolvedores;
3. Assim como em outras abordagens encontradas na literatura, na abordagem F3 a definição das características do domínio depende do conhecimento e da habilidade dos desenvolvedores;

4. A abordagem F3 auxilia o desenvolvimento de frameworks, mas nesta tese de doutorado foram consideradas somente as camadas de persistência e de modelo. Dessa forma, os frameworks construídos com essa abordagem tratam principalmente das operações de cadastro, remoção, atualização e recuperação de dados. Requisitos não funcionais não são abordados;
5. Tanto a abordagem F3 quanto a abordagem de construção de DSLs de frameworks estão mais relacionadas com frameworks específicos de domínio (EAF). Para outros tipos de frameworks, pode ser exigido um esforço maior por parte do desenvolvedor;
6. A abordagem F3 auxilia na construção da estrutura dos frameworks. Apesar dos modelos F3 permitirem a definição de métodos, a implementação do código interno desses métodos e das características comportamentais do framework é de responsabilidade dos desenvolvedores;
7. A ferramenta F3T gera o código-fonte dos frameworks e das aplicações somente em linguagem Java;
8. A ferramenta F3T foi desenvolvida para ser utilizada no Eclipse IDE, portanto, o conhecimento sobre esse IDE é um pré-requisito necessário para o manuseio dessa ferramenta.

7.4 Trabalhos Futuros

Esta tese de doutorado resultou em contribuições que auxiliam aos desenvolvedores no desenvolvimento e reúso de frameworks. Porém, alguns trabalhos podem ser realizados futuramente para que essas contribuições sejam ampliadas:

1. Novos estudos podem ser realizados com a construção de DSLs com outras ferramentas além do GMF. A construção da sintaxe abstrata não é afetada significativamente pela ferramenta utilizada, porém, a sintaxe concreta pode resultar em uma interface gráfica para DSL com um número maior ou menor de recursos úteis para o desenvolvedor;
2. A linguagem de padrões da abordagem F3 pode ser expandida para incluir padrões que auxiliam na construção das camadas de controle e de visão dos frameworks. As classes dessas camadas seriam reutilizadas pelas aplicações assim como ocorre atualmente com as classes das camadas de persistência e de modelo;
3. Outra expansão da abordagem F3 pode incluir a construção de testes para verificar o funcionamento dos frameworks desenvolvidos e das aplicações que reutilizam esses frameworks;

4. Com a inclusão de padrões para as camadas de controle e visão na abordagem F3, poderia ser incluído na F3T formas de customização das interfaces gráficas das aplicações geradas;
5. Pode ser criada uma versão alternativa dos padrões de persistência da linguagem de padrões da abordagem F3 que considera o reúso de frameworks de persistência, como, por exemplo, o Hibernate. Com isso, a F3T permitiria ao desenvolvedor escolher qual o formato de persistência seria aplicado no framework em desenvolvimento;
6. A geração do código das classes de persistência dos frameworks e das aplicações feita pela F3T poderia ser alterada para permitir o uso de diferentes Sistemas Gerenciadores de Banco de Dados e não somente o MySQL;
7. A usabilidade da F3T pode ser melhorada no sentido de reduzir os passos para a geração da DSL e também para que alguns recursos de validação sejam executados automaticamente durante a modelagem dos domínios e das aplicações.

7.5 Artigos Publicados

Durante esta tese de doutorado foram publicados os seguintes artigos científicos:

1. Matheus Viana, Rosângela Penteado, Antônio Francisco do Prado. **Domain-Specific Modeling Languages to Improve Framework Instantiation**. Journal of Systems and Software, v. 86, n. 12, dezembro de 2013, p. 3123-3139. Editora Elsevier. Qualis: A2.
2. Matheus Viana, Rosângela Penteado, Antônio Francisco do Prado. **Developing Frameworks from Extended Feature Models**. Capítulo de livro: Enterprise Information Systems, Lecture Notes – Advances in Intelligent and Soft Computing, Springer Berlin Heidelberg (em processo de edição).
3. Matheus Viana, Rosângela Penteado, Antônio Francisco do Prado. **Building Domain-Specific Modeling Languages for Frameworks**. Capítulo de livro: Enterprise Information Systems, Lecture Notes – Business Information Processing, v. 141. p 191-206. Springer Berlin Heidelberg, Berlin, Alemanha, 2013.
4. Matheus Viana, Rosângela Penteado, Antônio Francisco do Prado, Rafael Durelli. **F3T: From Features to Frameworks Tool**. XXVII Simpósio Brasileiro de Engenharia de Software (SBES). Brasília, Brasil, 29 de Setembro a 04 de Outubro de 2013. Qualis: B2.

5. Matheus Viana, Rosângela Penteadó, Antônio Francisco do Prado, Rafael Durelli. **An Approach to Develop Frameworks from Feature Models**. XIV IEEE International Conference on Information Reuse and Integration (IRI). São Francisco, EUA, 14-16 de Agosto de 2013. Qualis: B2. Esse artigo foi selecionado para compor um capítulo em um volume da série Lecture Notes – Advances in Intelligent and Soft Computing (ver item 2).
6. Matheus Viana, Rafael Durelli, Rosângela Penteadó, Antônio Francisco do Prado. **F3: From Features to Frameworks** (*Short Paper*). XV International Conference on Enterprise Information Systems (ICEIS). Angers, França. 04-07 de Julho de 2013. Qualis: B1.
7. Matheus Viana, Rosângela Penteadó, Antônio Francisco do Prado. **Generating Applications: Framework Reuse Supported by Domain-Specific Modeling Languages**. XIV International Conference on Enterprise Information Systems (ICEIS). Wrocław (Breslávia), Polônia, 28 de Junho a 01 de Julho de 2012. Qualis: B1. Esse artigo foi selecionado para compor um capítulo em um volume da série Lecture Notes – Business Information Processing (ver item 3).
8. Matheus Viana, Rosângela Penteadó, Carolina Nhoque, Antônio Francisco do Prado. **Utilizando o GMF para a Construção de Linguagens de Modelagem Específicas do Domínio de Frameworks de Aplicação**. XXXVII Conferência Latinoamericana de Informática (CLEI), Quito, Equador, 10-14 de Outubro de 2011. Qualis: B4.

REFERÊNCIAS

ABI-ANTOUN, M. Making Frameworks Work: A Project Retrospective. In: CONFERENCE ON OBJECT ORIENTED PROGRAMMING SYSTEMS LANGUAGES AND APPLICATIONS – OOPSLA. Montreal, Quebec, Canada, 2007. **Proceedings...** New York: ACM 2007. p. 1004-1018.

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

AMATRIAIN, X.: CLAM: A Framework for Audio and Music Application Development. **IEEE Software**, v. 24, n. 1, p. 82-85, Janeiro 2007.

AMATRIAIN, X.; ARUMI, P. Frameworks Generate Domain-Specific Languages: A Case Study in the Multimedia Domain. **IEEE Transactions on Software Engineering**, v. 37, n. 4, p. 544–558, Julho 2011.

ANGUSWAMY, R.; FRAKES, W.B. A Study of Reusability, Complexity, and Reuse Design Principles. In: ACM-IEEE INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT - ESEM. Lund, Suécia, 2012. **Proceedings...** New York: ACM 2012. p. 161-164.

ANTKIEWICZ, M.; CZARNECKI, K.; STEPHAN, M. Engineering of Framework-Specific Modeling Languages. **IEEE Transactions on Software Engineering**, v. 35, n. 6, p. 795–823, Dezembro 2009.

BAUER, V. Facts and Fallacies of Reuse in Practice. In: 17th EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING - CSMR. Genova, Itália, 2013. **Proceedings...** Washington: IEEE Computer Society 2013. p. 431-434.

BAYER, J.; FLEGE, O.; KNAUBER, P.; LAQUA, R.; MUTHIG, D.; SCHMID, K.; WIDEN, T.; DEBAUD, J. M. PuLSE: A Methodology to Develop Software Product Lines. In: SYMPOSIUM ON SOFTWARE REUSABILITY - SSR. Los Angeles, USA, 1999. **Proceedings...** New York: ACM 1999. p. 122-131.

BEN-AMMAR, L.; MAHFOUDHI, A. Usability Driven Model Transformation. In: 6th INTERNATIONAL CONFERENCE ON HUMAN SYSTEM INTERACTION - HSI. Gdansk, Polônia, 2013. **Proceedings...** Washington: IEEE Computer Society 2013. p. 110-116.

BENAVIDES, D.; SEGURA, S.; CORTÉS, A. R. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, v. 35, n. 6, p. 615-636, Setembro, 2010.

BRAGA, R. T. V. **Um Processo para Construção e Instanciação de Frameworks Baseados em uma Linguagem de Padrões para um Domínio Específico**. 2002. 232f. Tese (Doutorado em Ciência da Computação), Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, 2002.

BRAGA, R. T. V.; GERMANO, F. S. R.; MASIERO, P. C. **GRN: Uma Linguagem de Padrões para Gerenciamento de Recursos de Negócio**. 1999. Disponível em: <http://www.icmc.usp.br/~rtvb/>. Acessado em 06 de Novembro de 2007.

BRAGA, R.; MASIERO, P. C. Building a Wizard for Framework Instantiation Based on a Pattern Language. In: 9th INTERNATIONAL CONFERENCE ON OBJECT-ORIENTED INFORMATION SYSTEMS – OOIS. Genebra, Suíça, 2003. **Proceedings...** Germany: Springer 2003, p. 95–106.

BRAGANÇA, A.; MACHADO, R. J. Model Driven Development of Software Product Lines. In: 6th INTERNATIONAL CONFERENCE ON THE QUALITY OF INFORMATION AND COMMUNICATIONS TECHNOLOGY – QUATIC. Lisboa, Portugal, 2007. **Proceedings...** Washington: IEEE Computer Society 2007. p. 199-203.

BRUCH, M.; MEZINI, M.; MONPERRUS, M. Mining Subclassing Directives to Improve Framework Reuse. In: 7th IEEE Working Conference on Mining Software Repositories - MSR. Cape Town, África do Sul, 2010. **Proceedings...** Washington: IEEE Computer Society 2010. p. 141-150.

BRUGALI, D.; SYCARA, K. Frameworks and Pattern Languages: an Intriguing Relationship. **ACM Computing Surveys**, v. 32, n. 1, p. 2-7, Março 2000.

BUSHMANN, F.; HENNEY, K.; SCHMIDT, D. C. Past, Present and Future Trends in Software Patterns. **IEEE Software**, v. 24, n. 4, p. 31-37, Julho 2007.

CLEMENTS P.; NORTHROP, L. **Software Product Lines: Practices and Patterns**. New York: Addison-Wesley, 2001. 608p.

CHAVES, A. P.; LEAL, G. C. L.; HUZITA, E. H. M. An Experimental Study of the FIB Framework Driven by the PDCA Cycle. In: INTERNATIONAL CONFERENCE OF THE CHILEAN COMPUTER SCIENCE SOCIETY - SCCC. Punta Arenas, Chile, 2008. **Proceedings...** Washington: IEEE Computer Society 2008. p. 23-31.

COPLIEN, J. **Software Patterns**. SIGS Books, 1996. Disponível em: <http://users.rcn.com/jcoplien/Patterns/WhitePaper/SoftwarePatterns.pdf>. Acesso em: 08 de Setembro de 2013.

CUADRADO, J.; MOLINA, J. A Model-Based Approach to Families of Embedded Domain-Specific Languages. **IEEE Transactions on Software Engineering**, v. 35, n. 6, p. 825–840, Dezembro 2009.

CZARNECKI, K. Overview of Generative Software Development. In: REINHARTZ-BERGER, I.; STURM, A.; CLARK, T.; COHEN, S.; BETTIN, J. *Lecture Notes in Computer Science: Unconventional Programming Paradigms*, v. 3566. Berlim, Alemanha: Springer Verlag, 2004. p. 326-341.

CZARNECKI, K.; HELSEN, S. Feature-Based Survey of Model Transformation Approaches. **IBM Systems Journal**, v. 45, n. 3, p. 621-645, Março 2006.

CZARNECKI, K.; WASOWSKI, A. Feature Diagrams and Logics: There and Back Again. In: 11th INTERNATIONAL SOFTWARE PRODUCT LINE CONFERENCE. Kyoto, Japão, 2007. **Proceedings...** Washington: IEEE Computer Society 2007. p. 23-34.

CZARNECKI, K.; GRÜNBACHER, P.; RABISER, R.; SCHMID, K.; WASOWSKI, A. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In: 6th INTERNATIONAL WORKSHOP ON VARIABILITY MODELING OF SOFTWARE-INTENSIVE SYSTEMS. Leipzig, Alemanha, 2012. **Proceedings...** New York: ACM 2012. p. 173-182.

DJUKIC, V.; LUKOVIC, I.; POPOVIC, A. Domain-Specific Modeling in Document Engineering. In: Federated Conference on Computer Science and Information Systems – FedCSIS. Szczecin, Polônia, 2011. **Proceedings...** Washington: IEEE Computer Society 2010. p. 817-824.

DRUCKENMILLER, D. A.; JENKINS, J.; MITTLEMAN, D.; BOOTSMAN, P. A Pattern Language Approach to the Design of a Facilitation Reporting Database. In: 43rd HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES – HICSS. Koloa, Kauai, HI, USA, 2010. **Proceedings...** Washington: IEEE Computer Society 2010. p. 1-10.

DURELLI, V. H. S., Reengenharia Iterativa do Framework GREN. 2008. 142f. Dissertação (Mestrado em Ciência da Computação) – Centro de Ciências Exatas e de Tecnologia, Universidade Federal de São Carlos, São Carlos, 2008.

DURELLI, V. H. S.; VIANA, M. C.; PENTEADO, R. A. D. Uma Proposta de Reúso de Interface Gráfica com o Usuário Baseada no Padrão Arquitetural MVC. In: IV SIMPÓSIO BRASILEIRO DE SISTEMAS DE INFORMAÇÃO – SBSI. Rio de Janeiro, Brasil, 2008. **Anais...** Rio de Janeiro: UNIRIO 2008. p. 48-59.

ESTATCAMP (2013). **Portal Action**. Disponível em: <http://www.portalaction.com.br>. Acesso em: 27 de Dezembro de 2013.

FAYAD, M. E.; JOHNSON, R. E. **Domain Specific Application Frameworks: Frameworks Experience by Industry**. New York: Wiley, 1999, 704p.

FAYAD, M. E.; SCHMIDT, D. C. Object-Oriented Application Frameworks. **Communications of the ACM**, v. 40, n. 10, p. 32-38, Outubro 1997.

FOWLER, M. **Analysis Patterns: Reusable Object Models**. New York: Addison-Wesley, 1997. 384p.

FOWLER, M.; Beck, K.; Brant, J.; Opdyke, W.; Roberts, D. **Refactoring: Improving the Design of Existing Code**. New York: Addison-Wesley, 1999, 464p.

FRAKES, W.; KANG, K. Software Reuse Research: Status and Future. **IEEE Transactions on Software Engineering**, v. 31, n. 7, p. 529–536, Julho 2005.

FRANCE, R.; RUMPE, B. Model-Driven Development of Complex Software: A Research Roadmap. In: 29th INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING - FUTURE OF SOFTWARE ENGINEERING – ICSE. Minneapolis, USA, 2007. **Proceedings...** Washington: IEEE Computer Society 2007. p. 37–54.

FREEMAN, E. T; FREEMAN, E. R; SIERRA, K. e BATES, B. **Head First Design Patterns**. Sebastopol: O'Reilly Media, 2004, 688p.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design Patterns: Elements of Reusable Object-Oriented Software**. New York: Addison-Wesley, 1995, 416p.

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

GRAND, M. **Java Enterprise Design Patterns**. New York: John Wiley & Sons, 2001, 416p.

GRONBACK, R. C. **Eclipse Modeling Project: a Domain-Specific Language (DSL) Toolkit**. New York: Addison-Wesley, 2009, 736p.

HARRISON, N. B.; AVGERIOU, P.; ZDUN, U. Using Patterns to Capture Architectural Decisions. **IEEE Software**, v. 24, n. 4, p. 38-45, Julho 2007.

HOU, D. Investigating the Effects of Framework Design Knowledge in Example-Based Framework Learning. In: IEEE International Conference on Software Maintenance - ICSM. Pequim, China, 2008. **Proceedings...** Washington: IEEE Computer Society 2008. p. 37–46.

HSUEH, N. L.; CHU, P. H.; HSIUNG, P. A.; CHUANG, M. J.; CHU, W.; CHANG, C. H.; KOONG, C. S.; SHIH, C. H. Supporting Design Enhancement by Pattern-based Transformation. In: 34th ANNUAL COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE – COMPSAC. Seul, Coréia do Sul, 2010. **Proceedings...** Washington: IEEE Computer Society 2010. p. 462–467.

HUTCHINSON, J; WHITTLE, J.; ROUNCFIELD, M.; KRISTOFFERSEN, S. Empirical Assessment of MDE in Industry. In: 33RD INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING – ICSE. Honolulu, HI, EUA, 2011. **Proceedings...** New York: ACM. p. 471–480.

ISIS. **Generic Modeling Environment**. Disponível em: <http://www.isis.vanderbilt.edu/Projects/gme/>. Acesso em: 15 de Setembro de 2011.

JBOSS COMMUNITY. **Hibernate**. Disponível em: <http://www.hibernate.org/>. Acesso em: 05 de Setembro de 2013.

JEZEQUEL, J. M. Model-Driven Engineering for Software Product Lines. **ISRN Software Engineering**, v. 2012, 24p, Outubro 2012.

JHOTDRAW.ORG. **JHotDraw as Open-Source Project**. Disponível em: <http://www.jhotdraw.org/>. Acesso em: 03 de novembro de 2013.

JOHNSON, R. E. Frameworks = (components + patterns). **Communications of the ACM**, v.40, n. 10, p. 39–42, Outubro 1997.

KANG, K. C.; COHEN, S. G.; HESS, J. A.; NOVAK, W. E.; PETERSON, A. S. **Feature-Oriented Domain Analysis (FODA) Feasibility Study**. 1990. Technical Report CMU/SEI-90-TR-21 - Carnegie Mellon University/Software Engineering Institute.

KEEPENCE, B.; MANNION, M. Using Patterns to Model Variability in Product Families. **IEEE Software**, vol. 16, no. 4, p. 102-108, Julho 1999.

KELLY, S.; TOLVANEN, J. P. **Domain-Specific Modeling: Enabling Full Code Generation**. New York: Wiley-IEEE Computer Society Press, 2008, 448p.

KIM, D. K.; YANG, Y. J.; JUNG, H. T. Development of an Object-Oriented Framework for Intranet-Based Groupware Systems. In: IEEE INTERNATIONAL CONFERENCE ON SYSTEMS, MAN, AND CYBERNETICS. Tucson, EUA, 2001. **Proceedings...** Washington: IEEE Computer Society 2001. V. 3, p. 1982-1987.

KIM, S. D.; CHANG, S. H.; CHANG, C. W. A Systematic Method to Instantiate Core Assets in Product Line Engineering. In: 11th ASIA-PACIFIC CONFERENCE ON SOFTWARE ENGINEERING – APSEC. Busan, Coreia do Sul, 2004. **Proceedings...** Washington: IEEE Computer Society 2004. p. 92-98.

KIRCHER, M.; VÖLTER, M. Guest Editors' Introduction: Software Patterns. **IEEE Software**, v. 24, n. 4, p. 28-30, Julho 2007.

KIRK, D.; ROPER, M.; WOOD, M. Identifying and Addressing Problems in Object-Oriented Framework Reuse. **Empirical Software Engineering**, v. 12, n. 3, p. 243-274, Junho 2007.

KELLY, S.; TOLVANEN, J. P. **Domain-Specific Modeling: Enabling Full Code Generation**. New York: Wiley-IEEE Computer Society Press, 2008, 448p.

KRAMER, O.; STURM, A. Bridging Programming Productivity, Expressiveness, and Applicability: a Domain Engineering Approach. In: INTERNATIONAL WORKSHOP ON DOMAIN ENGINEERING – DE@CAISE. Hammamet, Tunisia, 2010. **Proceedings...** Alemanha: Sun SITE Central Europe 2010. p. 47-60.

LARMAN, C. **Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development**. 3. ed., New York: Prentice Hall, 2004, 736p.

LEE, K.; KANG, K. C. Feature Dependency Analysis for Product Line Component Design. **Lecture Notes on Computer Science**, v. 3107, p. 69–85, 2004.

LETHBRIDGE, T. C. What Knowledge Is Important to a Software Professional? **IEEE Computer**, v. 33, n. 5, p. 44-50, Maio 2000.

LOO, K. N.; LEE, S. P. Representing Design Pattern Interaction Roles and Variants. In: 2nd INTERNATIONAL CONFERENCE ON COMPUTER ENGINEERING AND TECHNOLOGY – ICCET. Chengdu, China, 2010. **Proceedings...** Washington: IEEE Computer Society 2010. v. 6, p. 470-474.

LOPES, S. F.; TAVARES, A. C.; SILVA, C. A.; MONTEIRO, J. L. Application Development by Reusing Object-Oriented Frameworks. In: THE INTERNATIONAL CONFERENCE ON COMPUTER AS A TOOL - EUROCON. Belgrado, Sérvia, 2005. **Proceedings...** Washington: IEEE Computer Society 2005. p. 583-586.

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

LUCRÉDIO, D; FORTES, R. P. M.; ALMEIDA, E. S.; MEIRA, S. L. Designing Domain Architectures for Model-Driven Engineering. In: IV SIMPÓSIO BRASILEIRO DE COMPONENTES, ARQUITETURAS E REÚSO DE SOFTWARE – SBCARS. Salvador, Brasil, 2010. **Anais...** Salvador: UFBA 2010. p. 100-109.

MAGALHÃES, A. P.; ANDRADE, A.; MACIEL, R. P. MTP: Model Transformation Profile. In: VII SIMPÓSIO BRASILEIRO DE COMPONENTES, ARQUITETURAS E REUTILIZAÇÃO DE SOFTWARE - SBCARS. Brasília, Brasil, 2013. **Anais...** Brasília. UNB 2013. p. 125-134.

MAHMOOD, S.; AHMED, M.; ALSHAYEB, M. Reuse Environments for Software Artifacts: Analysis Framework. In: 12th IEEE/ACIS INTERNATIONAL CONFERENCE ON COMPUTER AND INFORMATION SCIENCE - ICIS. Toki Messe, Niigata, Japão, 2013. **Proceedings...** Washington: IEEE Computer Society 2013. p. 35-40.

MANOLESCU, D.; KOZACZYNSKI, W.; MILLER, A.; HOGG, J. The Growing Divide in the Patterns World. **IEEE Software**, v. 24, n. 4, p. 61-67, Julho 2007.

MARCA, D. A.; McGOWAN, C.L. SADT: **Structured Analysis and Design Technique**. McGraw-Hill Book Co., Inc.: New York, NY, 1988.

MARKIEWICZ, M. E.; LUCENA, C. Object Oriented Framework Development. **Crossroads** v. 7, n. 4, Junho 2001.

MCLAUGHLIN, B. D.; POLLICE G.; WEST, D. **Head First: Head First Object-Oriented Analysis and Design**. Sebastopol: O'Reilly Media, 2006, 636p.

MELLOR, S. J.; CLARK, A. N.; FUTAGAMI, T. Model-Driven Development. **IEEE Software**, v. 20, n. 5, p. 14–18, Setembro 2003.

MOHAMED-ALI, M.; MOAWAD, R. An approach for Requirements Based Software Product Line Testing. In: 7th INTERNATIONAL CONFERENCE ON INFORMATICS AND SYSTEMS. Cairo, Egito, 2010. **Proceedings...** Washington: IEEE Computer Society 2010. p. 1-10.

MUKHTAR, M. A. O.; HASSAN, M. F. B.; JAAFAR, J. B.; RAHIM, L. A. Enhanced Approach for Developing Web Applications Using Model Driven Architecture. INTERNATIONAL CONFERENCE ON RESEARCH AND INNOVATION IN INFORMATION SYSTEMS – ICRIIS. Kuala Lumpur, Malásia, 2013. **Proceedings...** Washington: IEEE Computer Society 2013. p. 145-150.

NIWE, M.; STIRNA, J. Pattern Approach to Business-to-Business Transactions. In: INTERNATIONAL CONFERENCE FOR INTERNET TECHNOLOGY AND SECURED TRANSACTIONS – ICITST. Londres, Inglaterra, 2009. **Proceedings...** Washington: IEEE Computer Society 2009. p. 1-6.

OKANOVIC, V.; MATELJAN, T. Designing a Web Application Framework. In: 18th INTERNATIONAL CONFERENCE ON SYSTEMS, SIGNALS AND IMAGE PROCESSING - IWSSIP. Sarajevo, Bosnia e Herzegovina, 2011. **Proceedings...** Washington: IEEE Computer Society 2011. p. 1-4.

OLIVEIRA, T. C.; ALENCAR, P.; COWAN, D. ReuseTool - An Extensible Tool Support for Object-Oriented Framework Reuse. **The Journal of Systems and Software**, v.84, n. 12, p. 2234–2252, Dezembro 2011.

OMG (2013). **Object Management Group**. Disponível em: <http://www.omg.org>. Acesso em: 15 de Setembro de 2013.

ORACLE (2013a). **Java**. Disponível em: <http://www.java.com>. Acesso em: 08 de Setembro de 2013.

ORACLE (2013b). **MYSQL: The World's Most Popular Open Source Database**. Disponível em: <http://www.mysql.com/>. Acesso em: 08 de Setembro de 2013.

ORACLE (2013c). **Java Persistence API**. Disponível em: <http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html>. Acesso em: 12 de Setembro de 2013.

ORACLE (2013d). **JavaServer Faces Technology**. Disponível em: <http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html>. Acesso em: 13 de Setembro de 2013.

PEREIRA, J. A.; SOUZA, C.; FIGUEIREDO, E.; ABILIO, R.; VALE, G.; COSTA, H. A. X. Software Variability Management: An Exploratory Study with Two Feature Modeling Tools. In: VII SIMPÓSIO BRASILEIRO DE COMPONENTES, ARQUITETURAS E REUTILIZAÇÃO DE SOFTWARE - SBCARS. Brasília, Brasil, 2013. **Anais...** Brasília. UNB 2013. p. 36-45.

PEROVICH, D.; ROSSEL, P. O.; BASTARRICA, M. C. Feature Model to Product Architectures: Applying MDE to Software Product Lines. In: EUROPEAN CONFERENCE ON SOFTWARE ARCHITECTURE – ECSA. Cambridge, Inglaterra, 2009. **Proceedings...** Washington: IEEE Computer Society 2009. p. 201-210.

POHL, K.; BÖCKLE, G.; VAN DER LINDEN, F. **Software Product Line Engineering Foundations, Principles, and Techniques**. Springer, 2005, 468p.

PRESSMAN, R. S. **Engenharia de Software**. 6. ed. São Paulo: McGraw-Hill, 2006, 720p.

PURE SYSTEMS (2013). **Pure::variants**. Disponível em: http://www.pure-systems.com/pure_variants.49.0.html. Acesso em: 20 de Setembro de 2013.

RAMOS, M.; BRAGA, R.; MASIERO, P.; PENTEADO, R. From Software Product Lines to System of Systems. In: 14th IEEE INTERNATIONAL CONFERENCE ON Information Reuse and Integration – IRI. San Francisco, CA, EUA, 2013. **Proceedings...** Washington: IEEE Computer Society 2013. p. 394-401.

RASOOL, G.; MÄDER, P. Flexible Design Pattern Detection Based on Feature Types. In: 26th IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING – ASE. Lawrence, KS, EUA, 2011. **Proceedings...** Washington: IEEE Computer Society 2011. p. 243-252.

ROBILLARD, M. P. What Makes APIs Hard to Learn? Answers from Developers. **IEEE Software**, v. 26, n. 6, p. 27–34, Novembro 2009.

ROUILLE, E.; COMBEMALE, B.; BARAIS, O.; TOUZET, D.; JEZEQUEL, J. M. Leveraging CVL to Manage Variability in Software Process Lines. In: 19th ASIA-PACIFIC SOFTWARE ENGINEERING CONFERENCE – APSEC. Hong Kong, China, 2012. **Proceedings...** Washington: IEEE Computer Society 2012, p. 148–157.

RUMPE, B.; SCHINDLER, M.; VÖLKEL, S.; WEISEMÖLLER, I. Generative Software Development. In: 32nd ACM/IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING. Cape Town, África do Sul, 2010. **Proceedings...** Washington: IEEE Computer Society 2010. p. 473-474.

SACERDOTIANU, G.; ILIE, S.; BADICA, C. Software Framework for Agent-Based Games and Simulations. In: 13th INTERNATIONAL SYMPOSIUM ON SYMBOLIC AND NUMERIC ALGORITHMS FOR SCIENTIFIC COMPUTING - SYNASC. Timisoara, Romênia, 2011. **Proceedings...** Washington: IEEE Computer Society 2011. p. 381-388.

SANTOS, A. L.; KOSKIMIES, K.; LOPES, A. Automated Domain-Specific Modeling Languages for Generating Framework-Based Applications. In: 12th INTERNATIONAL SOFTWARE PRODUCT LINE CONFERENCE – SPLC. Limerick, Irlanda, 2008. **Proceedings...** Washington: IEEE Computer Society 2008. p. 149-158.

SARASA-CABEZUELO, A.; TEMPRADO-BATTAD, B.; CEREZO, D. R.; SIERRA, J. Building XML-Driven Application Generators with Compiler Construction Tools. **Computer Science and Information Systems**, v. 9, n. 2, p. 485-504, Junho 2012.

SCHMID, K.; ALMEIDA, E. S. Product Line Engineering. **IEEE Software**, v. 30, n. 4, p. 24-30, Julho 2013.

SCHMIDT, D. Guest Editor's Introduction: Model-Driven Engineering. **IEEE Computer**, v. 39, n. 6, p. 25–31, Fevereiro 2006.

SEPULVEDA, S.; CARES, C.; CACHERO, C. Feature Modeling Languages: Denotations and Semantic Differences. In: 7th Iberian Conference on Information Systems and Technologies – CISTI. Madrid, Espanha, 2012. **Proceedings...** Washington: IEEE Computer Society 2012. p. 1-6.

SHALLOWAY, A.; TROTT, J. **Design Patterns Explained: A New Perspective On Object-Oriented Design**. 2. ed. New York: Addison-Wesley, 2004, 480p.

SHIVA, S. G.; SHALA, L. A. Software Reuse: Research and Practice. In: 4TH INTERNATIONAL CONFERENCE ON INFORMATION TECHNOLOGY – ITNG. Las Vegas, USA, 2007. **Proceedings...** Washington: IEEE Computer Society 2007. p. 603–609.

SOBEL, J. M.; FRIEDMAN, D. P. **An Introduction to Reflection-Oriented Programming**. Disponível em: <http://www.cs.indiana.edu/~jsobel/rop.html>. Acesso em: 21 de agosto de 2011.

SRINIVASAN, S. Design Patterns in Object-Oriented Frameworks. **ACM Computer**, v. 32, n. 2, p. 24-32, Fevereiro 1999.

STANOJEVIC, V.; VLAJIC, S.; MILIC, M.; OGNJANOVIC, M. Guidelines for Framework Development Process. In: 7TH CENTRAL AND EASTERN EUROPEAN SOFTWARE ENGINEERING CONFERENCE. Moscou, Rússia, 2011. **Proceedings...** Washington: IEEE Computer Society 2011. p. 1-9.

STURM, A.; KRAMER, O. Utilizing Application Frameworks: a Domain Engineering Approach. In: REINHARTZ-BERGER, I; STURM, A; CLARK, T.; COHEN, S.; BETTIN, J. *Domain Engineering*. Berlim, Alemanha: Springer Berlin Heidelberg, 2013. p. 113-130.

THE ECLIPSE FOUNDATION (2013a). **The Eclipse Foundation Open Source Community Website**. Disponível em: <http://eclipse.org/>. Acesso em: 25 de Agosto de 2013.

THE ECLIPSE FOUNDATION (2013b). **Graphical Modeling Project**. Disponível em: <http://eclipse.org/modeling/>. Acesso em: 25 de Agosto de 2013.

THE ECLIPSE FOUNDATION (2013c). **Eclipse Modeling Model To Text (M2T)**. Disponível em: <http://www.eclipse.org/modeling/m2t/>. Acesso em: 25 de Agosto de 2011.

THE HILLSIDE GROUP (2013). **Design Patterns Library**. Disponível em: <http://hillside.net/patterns>. Acesso em: 23 de Setembro de 2013.

VIANA, M. C. **Construção da Camada de Interface Gráfica e do Wizard para o Framework GRENJ**. 2009. 135f. Dissertação (Mestrado em Ciência da Computação) – Centro de Ciências Exatas e de Tecnologia, Universidade Federal de São Carlos, São Carlos, 2009.

VIANA, M. C.; PENTEADO, R. A. D.; NHOQUE, C. R.; PRADO, A. F. Utilizando o GMF para a Construção de Linguagens de Modelagem Específicas do Domínio de Frameworks de Aplicação. In: 37ª CONFERÊNCIA LATINOAMERICANA DE INFORMÁTICA – CLEI. Quito, Equador, 2011. **Anais...** Quito: PUCE 2011.

VIANA, M.; PENTEADO, R.; PRADO, A. F. Generating Applications: Framework Reuse Supported by Domain-Specific Modeling Languages. In: XIV International Conference on Enterprise Information Systems - ICEIS. Wrocław, Polônia, 2012. **Proceedings...** Lisboa: Scitepress, 2012. p. 5-14.

VIANA, M.; PENTEADO, R.; PRADO, A. F. Domain-Specific Modeling Languages to Improve Framework Instantiation. *The Journal of Systems and Software*, v. 86, n. 12, p. 3123-3139, Dezembro, 2013a.

VIANA, M.; PENTEADO, R.; PRADO, A. F. F3: From Features to Frameworks. In: XV INTERNATIONAL CONFERENCE ON ENTERPRISE INFORMATION SYSTEMS - ICEIS. Angers, França, 2013. **Proceedings...** Lisboa: Scitepress, 2013b. p. 110-117.

VIANA, M.; PENTEADO, R.; PRADO, A. F. An Approach to Develop Frameworks From Features. In: XIV IEEE INTERNATIONAL CONFERENCE ON INFORMATION REUSE AND INTEGRATION - IRI. São Francisco, EUA, 2013. **Proceedings...** Washington: IEEE Computer Society, 2013c. p. 594-601.

VIANA, M.; PENTEADO, R.; PRADO, A. F. F3T: From Features to Frameworks Tool. In: XXVII Simpósio Brasileiro de Engenharia de Software - SBES. Brasília, Brasil, 2013. **Proceedings...** Washington: IEEE Computer Society, 2013d. p. 104-113.

VIANA, M.; PENTEADO, R.; PRADO, A. F. Building Domain-Specific Modeling Languages for Frameworks. In: CORDEIRO, J.; MACIASZEK, L. A.; FILIPE, J. *Lecture Notes in Computer Science: Business Information Processing*, v. 141. Berlim, Alemanha: Springer Berlin Heidelberg, 2013e. p. 191-206.

W3C. **W3Schools Online Web Tutorials**. Disponível em: <http://www.w3schools.com/>. Acesso em: 15 de Setembro de 2013.

WEISS, D. M.; LAI, C. T. R. **Software Product Line Engineering: A Family-Based Software Development Process**. New York: Addison-Wesley, 1999, 448p.

WOHLIN, C.; RUNESON, P; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. **Experimentation in Software Engineering: An Introduction**. Norwell: Kluwer Academic Publishers, 2000, 204p.

WU-DONG, L.; KE-QING, H.; YINGSHI; HUI, X.; YI-XING, J. A Pattern Language Model for Framework Development. In: 27th Annual International Computer Software and Applications Conference - COMPSAC. Dallas, EUA, 2003. **Proceedings...** Washington: IEEE Computer Society 2003. p. 669-673.

XU, L.; BUTLER, G. Cascaded Refactoring for Framework Development and Evolution. In: AUSTRALIAN SOFTWARE ENGINEERING CONFERENCE – ASWEC. Sydney, Australia, 2006. **Proceedings...** Washington: IEEE Computer Society 2006. p. 1-10.

YE, H.; LIU, H. Approach to Modeling Feature Variability and Dependencies in Software Product Lines. **IEEE Proceedings - Software**, v. 132, n. 3, p. 101-109, Julho 2005.

YODER, J. W.; JOHNSON, R. E.; WILSON, Q. D.; Douglas, M. Connecting Business Objects to Relational Databases. In: FIFTH CONFERENCE ON PATTERNS LANGUAGES OF PROGRAMS – PLoP. Monticello, Illinois, USA, 1998. **Proceedings...** Disponível em: <http://www.joeyoder.com/PDFs/Persista.pdf>. Acesso em: 04 de Setembro de 2011.

ZAMANI, B.; BUTLER, G. Describing Pattern Languages for Checking Design Models. In: 16TH ASIA-PACIFIC SOFTWARE ENGINEERING CONFERENCE – APSEC. Penang, Malaysia, 2009. **Proceedings...** Washington: IEEE Computer Society 2009, p. 197–204.

ZHANG, T.; XIAO, X.; WANG, H.; QIAN, L. A Feature-Oriented Framework Model for Object-Oriented Framework: A MDA Approach. In: 9th INTERNATIONAL CONFERENCE ON COMPUTER AND INFORMATION TECHNOLOGY - CIT. Xiamen, China, 2009. **Proceedings...** Washington: IEEE Computer Society 2009, v. 2, p. 199–204.

Apêndice A

INTRODUÇÃO AO GMF

A.1 Considerações Iniciais

O *Graphical Modeling Framework* (GMF) faz parte do pacote *Modeling Project* do Eclipse IDE e permite a construção de linguagens de modelagem com notação gráfica. O GMF é o resultado da união de dois outros *frameworks*: 1) o *Eclipse Modeling Framework* (EMF), utilizado para a criação de metamodelos de linguagens; e 2) o *Graphical Editing Framework* (GEF), utilizado para a criação de editores gráficos e *views* para o Eclipse IDE. Na prática, é considerado que o GMF permite a construção de uma sintaxe concreta (notação gráfica) para a sintaxe abstrata definida por um metamodelo criado com o EMF.

Quando se utiliza o GMF para criar uma linguagem de modelagem, é obtido um conjunto de plug-ins para o Eclipse IDE que compõem uma ferramenta CASE dessa linguagem. É importante ressaltar, também, que o GMF utiliza o termo “diagrama” para se referir à linguagem de modelagem que está sendo criada, portanto, esse será o termo utilizado neste apêndice.

Maiores informações sobre o GMF, o EMF e o GEF podem ser adquiridas em: <http://www.eclipse.org/modeling/>. Como pré-requisito para a realização deste apêndice é necessário possuir o Eclipse IDE, que pode ser obtido em: <http://www.eclipse.org/download>.

A.2 Utilizando o GMF

A criação de um diagrama com o uso do GMF é iniciada com a criação de um projeto GMF no Eclipse IDE. Na Figura A.2 é mostrado o *wizard* de criação de projetos do Eclipse IDE em que é destacada a criação de um projeto GMF.

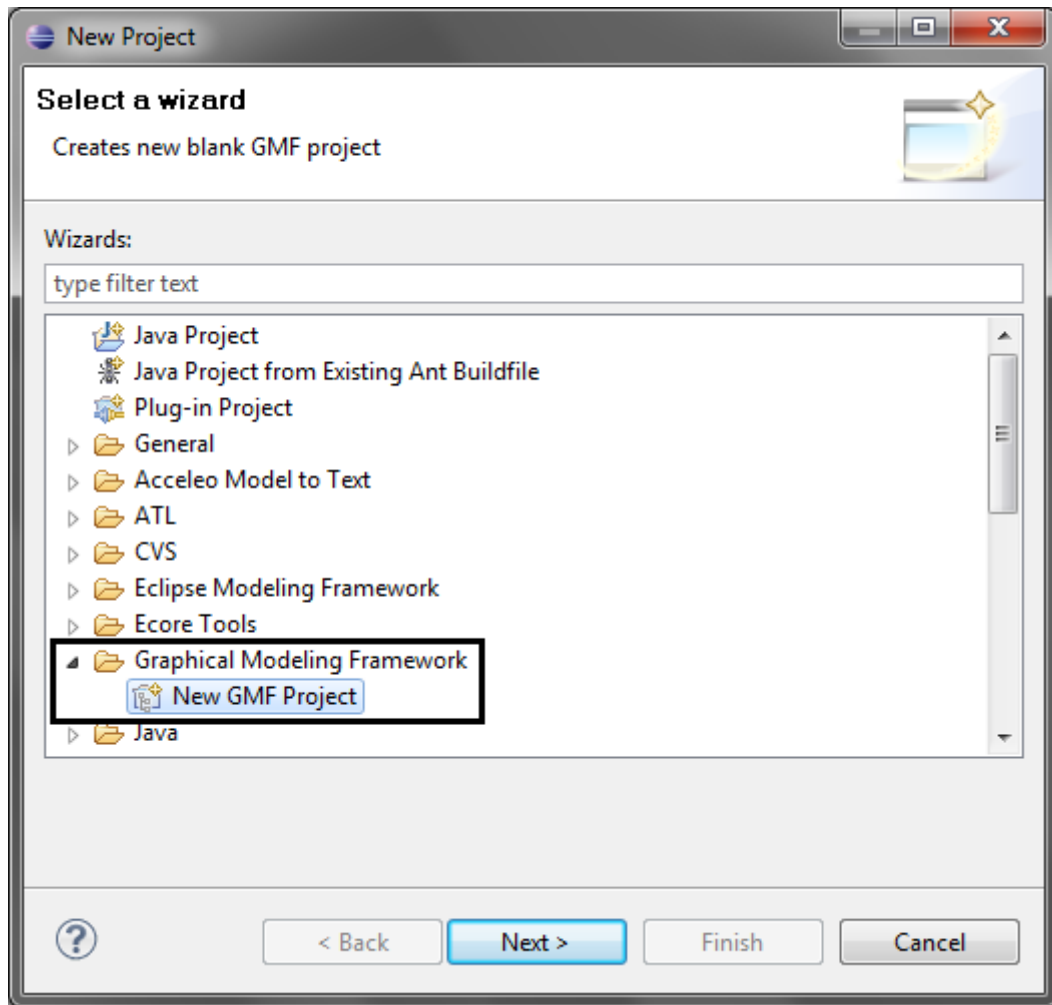


Figura A.1. Passos para a criação de um Projeto GMF.

O GMF define um conjunto de arquivos para a criação de um diagrama e fornece um quadro, chamado *GMF Dashboard*, que define a sequência em que esses arquivos devem ser criados, conforme é mostrado na Figura A.2.

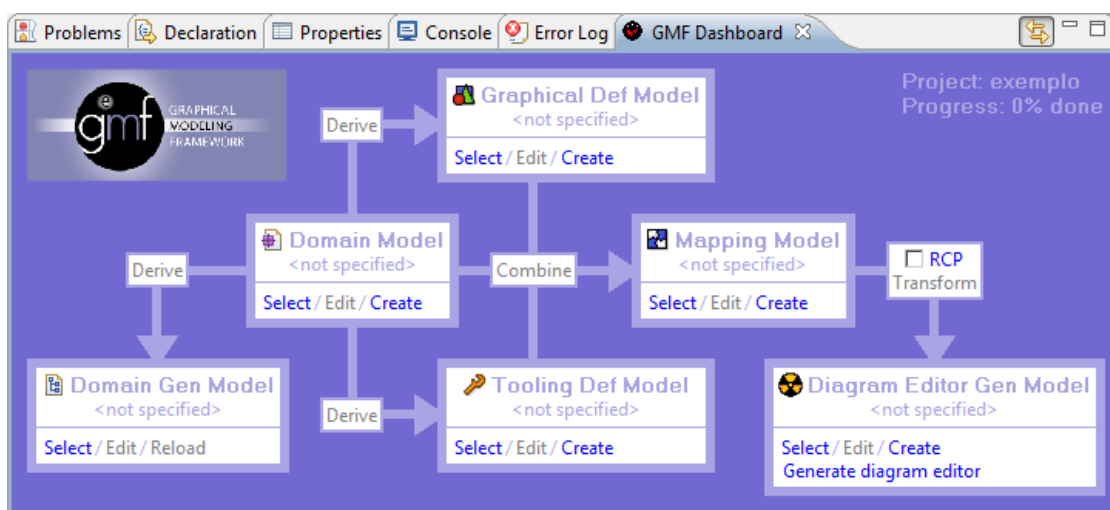


Figura A.2. GMF Dashboard.

O primeiro arquivo a ser criado é o *Domain Model* (arquivo com extensão *.ecore*), que representa o metamodelo do diagrama. Na Figura A.3 é apresentado o metamodelo do diagrama criado neste apêndice. O GMF determina que deve haver uma metaclassa que representa as instâncias do diagrama (modelos) e armazena os elementos inseridos nessas instâncias, portanto, a metaclassa *Domain* foi criada para esse propósito. A metaclassa *Node* corresponde aos elementos do diagrama e a metaclassa *Edge* representa as relações entre os elementos, que possui um *Node* como origem (*source*) e outro como alvo (*target*). A referência *relations* indica as relações que cada *Node* possui.

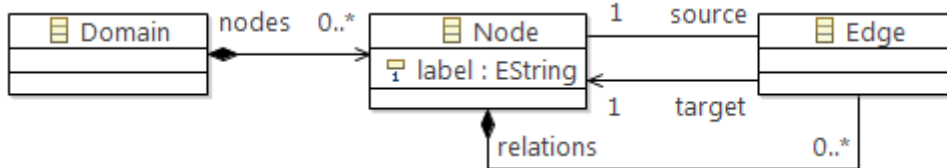


Figura A.3. Metamodelo do diagrama.

Após criar o metamodelo do diagrama é necessário gerar o *Domain Gen Model* (arquivo com extensão *.genmodel*) do diagrama. Isso é feito por meio de um *wizard* que é aberto a partir da opção *derive* ao lado do *Domain Model* no *GMF Dashboard*, conforme é mostrado na Figura A.4.

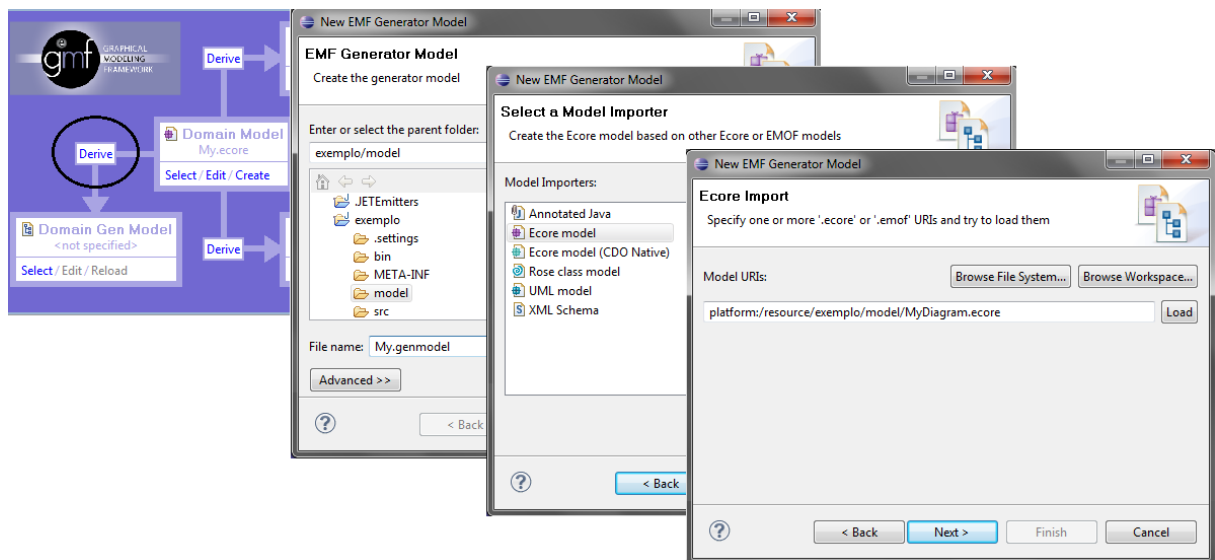


Figura A.4. Passos para a criação do arquivo com extensão *genmodel*.

O *Domain Gen Model* é responsável por gerar o código dos plug-ins do metamodelo e do painel de propriedades e do editor do diagrama. Para gerar esses plug-ins é necessário clicar com o botão direito no *Domain Gen Model* e selecionar as opções *Generate Model Code*, *Generate Edit Code* e *Generate Editor Code*, conforme é apresentado na Figura A.5.

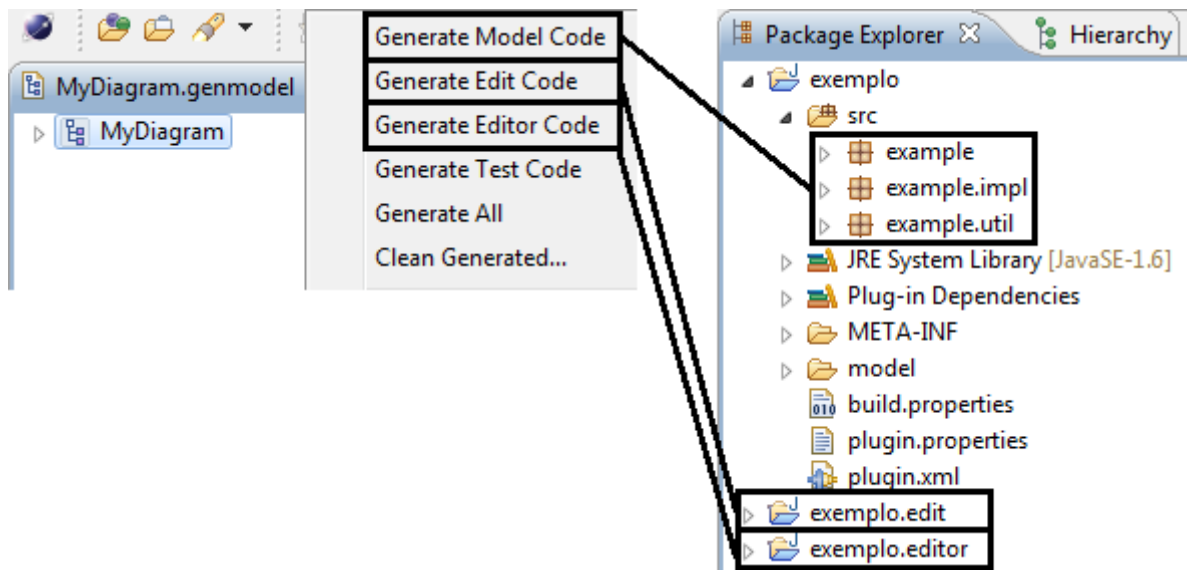


Figura A.5. Metamodelo do diagrama.

Após gerar os códigos dos plug-ins do metamodelo e do painel de propriedades, deve-se informar como os elementos do metamodelo (Figura A.3) serão representados graficamente. Para realizar esse processo é necessário criar o *Graphical Def Model* (modelo *Gmfgraph*), conforme é mostrado na Figura A.6.

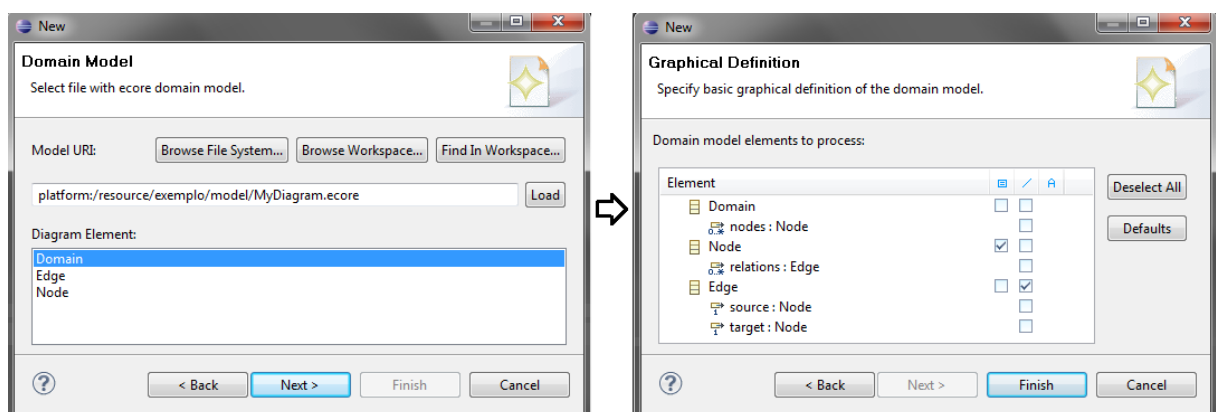


Figura A.6. Criação do modelo Gmfgraph.

Na Figura A.7 é apresentada a criação do conteúdo do *Graphical Def Model* de acordo com os seguintes passos: a) uma elipse, denominada *NodeEllipse*, foi inserida para representar os elementos *Node*; b) um *Label*, denominado *NodeLabelFigure*, foi inserido nessa elipse para indicar que essa imagem possui um rótulo; c) o *Child Access* *getNodeLabelFigure* foi inserido para fornecer acesso ao conteúdo desse rótulo; d) uma linha contínua (*Polyline Connection*) foi inserida para representar o relacionamento *Edge*; e) um *Diagram Label* foi inserido para representar o rótulo do *Node* e vinculá-lo ao rótulo da elipse. Os demais elementos do diagrama, *Node Node* e *Connection Edge*, foram inseridos automaticamente durante a criação do arquivo.

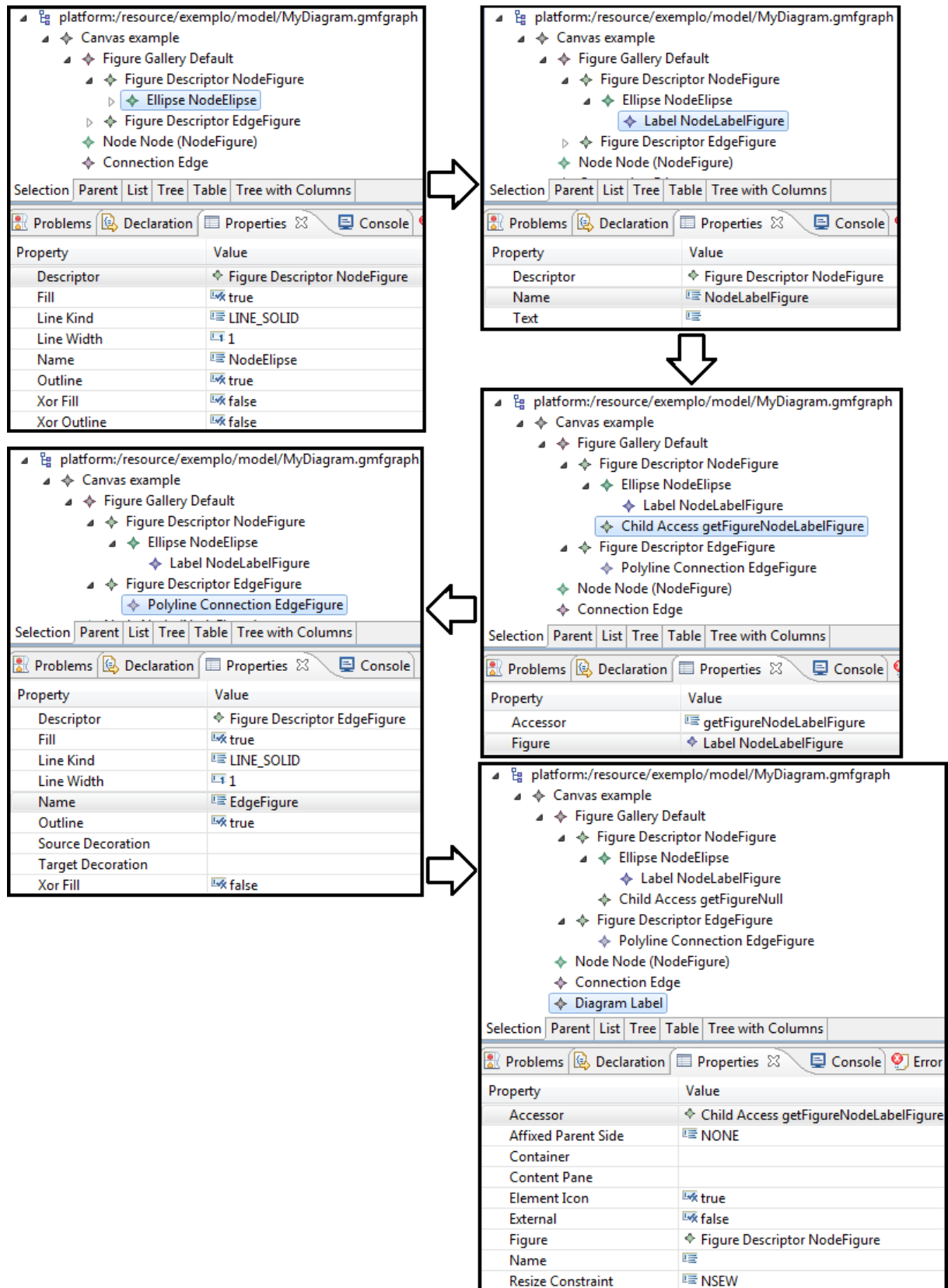


Figura A.7. Passos para criar a notação gráfica do diagrama.

Após a criação da notação gráfica do diagrama, deve-se criar a caixa de menu da ferramenta CASE do diagrama por meio do *Tooling Def Model* (modelo *Gmftool*). Esse arquivo é o mais simples de ser criado, pois é necessário somente seguir os passos do seu

wizard de forma semelhante ao que foi feito com o *Graphical Def Model*. A regra geral é que deve haver um item para cada elemento ou relacionamento do diagrama. Na Figura A.8 é apresentado o *Tooling Def Model* do diagrama criado neste Apêndice.

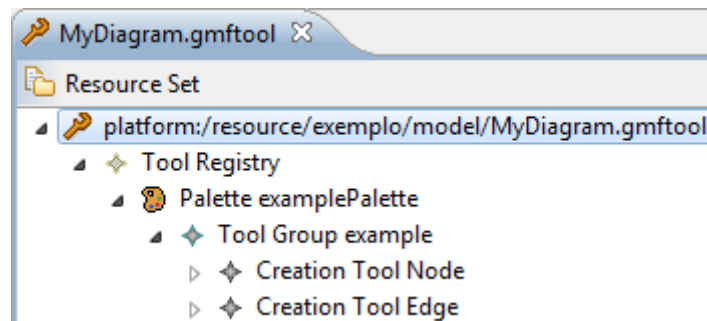


Figura A.8. Modelo Gmftool.

O *Mapping Model* (modelo *Gmfmap*) combina os elementos do *Domain Model* com as respectivas imagens do *Graphical Def Model* e com os respectivos itens da caixa de menu do *Tooling Def Model*. Na Figura A.9 é apresentado o *wizard* para criação do *Mapping Model*.

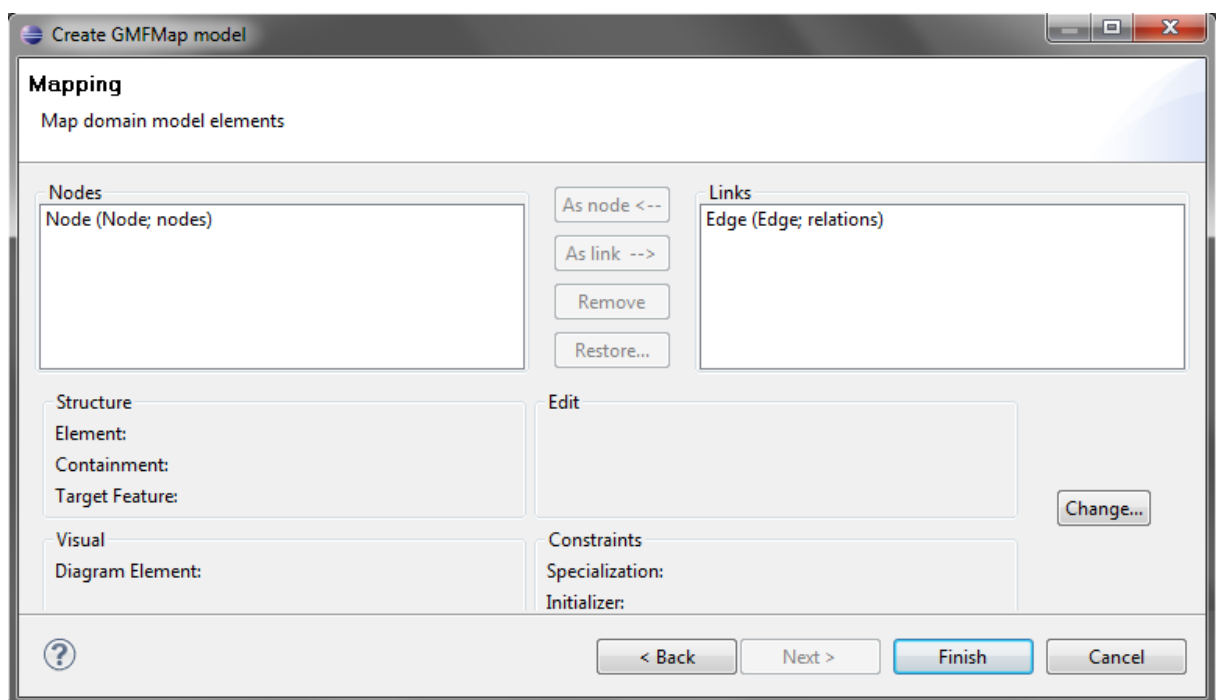


Figura A.9. Passos para a criação modelo Gmfmap.

No *Mapping Model* cada elemento do diagrama é representado por um *Top Node Reference* e cada relacionamento por um *Link Mapping*. Além disso, um *Feature Label Mapping* devem ser inseridos para associar os rótulos da notação gráfica com os atributos dos elementos do metamodelo.

Na Figura A.10 é apresentado o *Mapping Model* do diagrama apresentado neste apêndice. O *Top Node Reference* contém um *Node Mapping*, que realiza o mapeamento

entre o metamodelo, a notação gráfica e a caixa de menu por meio das seguintes propriedades: *Element*, que indica a metaclassa que deve ser instanciada no modelo; *Diagram Node*, que indica a imagem gráfica que deve ser visualizada no modelo; e *Tool*, que indica o item de menu que insere o elemento no modelo. O *Feature Label Mapping* e o *Link Mapping* possuem propriedades semelhantes.

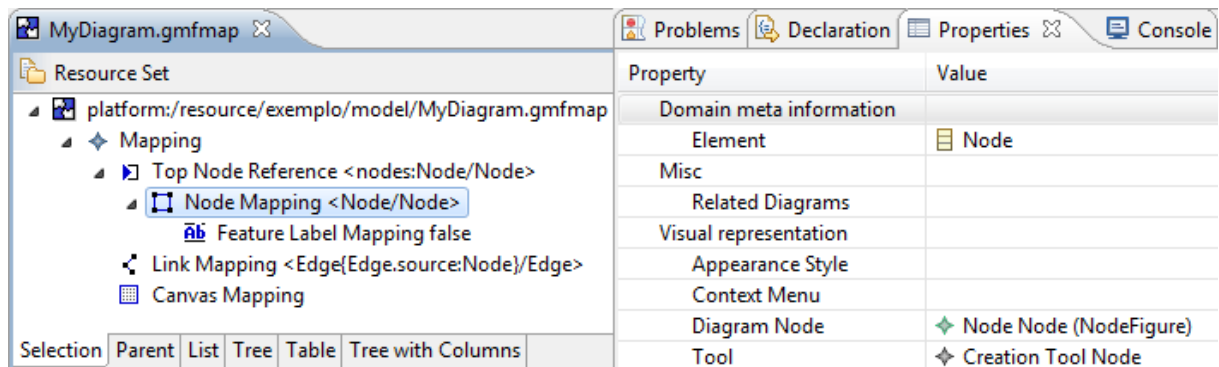


Figura A.10. Modelo Gmfmap.

Após finalizar a configuração do *Mapping Model*, o próximo passo é gerar o *Generator Model* (arquivo com extensão *.gmfgen*), responsável por gerar o *plug-in* do diagrama. Para gerar esse arquivo, é necessário clicar com o botão direito no *Mapping Model* e selecionar a opção *Create Generator Model*, conforme é ilustrado na Figura A.11.

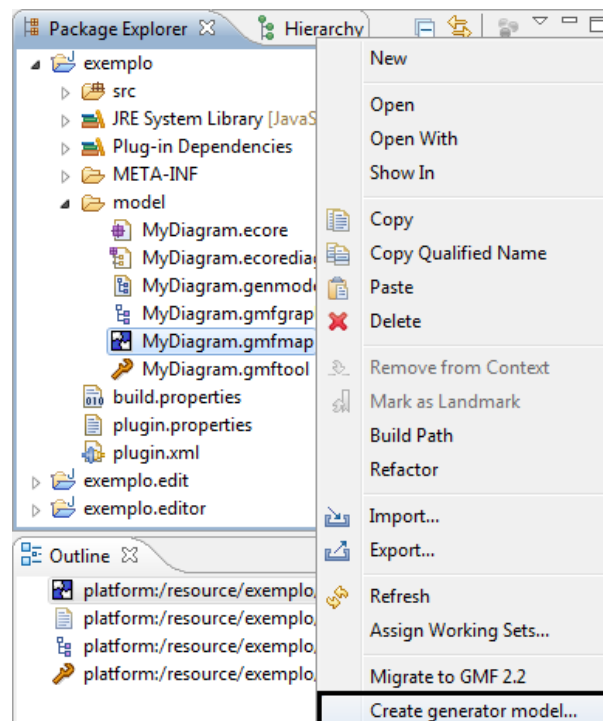


Figura A.11. Geração do arquivo Gmfgen.

Por fim, é necessário clicar com o botão direito no *Generator Model* e selecionar a opção *Generate diagram code* para gerar o *plug-in* do diagrama, conforme é ilustrado na Figura A.12.

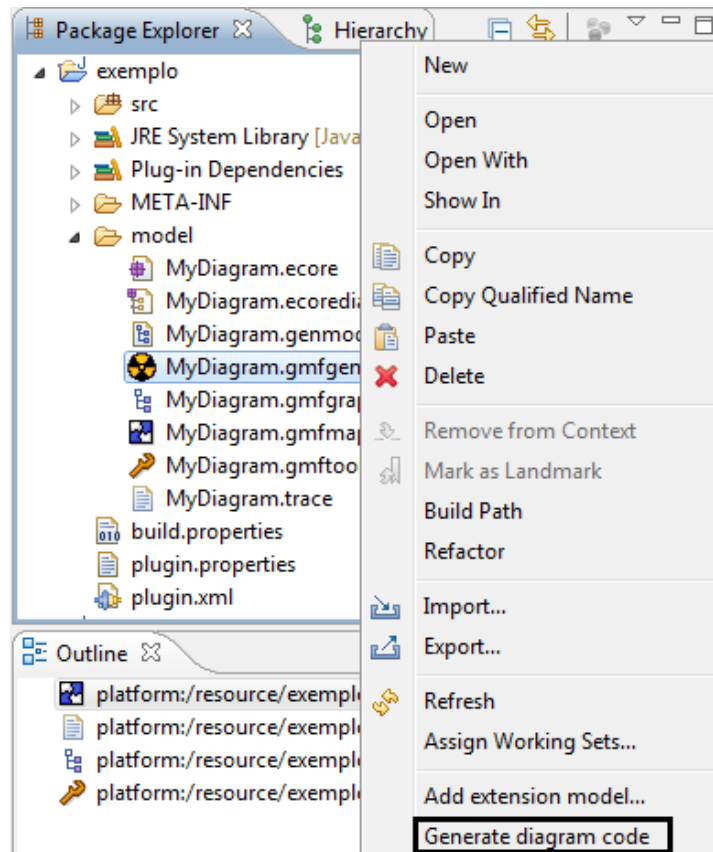


Figura A.12. Geração do plug-in do diagrama.

Para utilizar essa ferramenta CASE do diagrama é necessário inserir os seus plug-ins no Eclipse IDE, criar um projeto e criar um modelo do diagrama. Na Figura A.13 é apresentado como esse processo deve ser realizado.

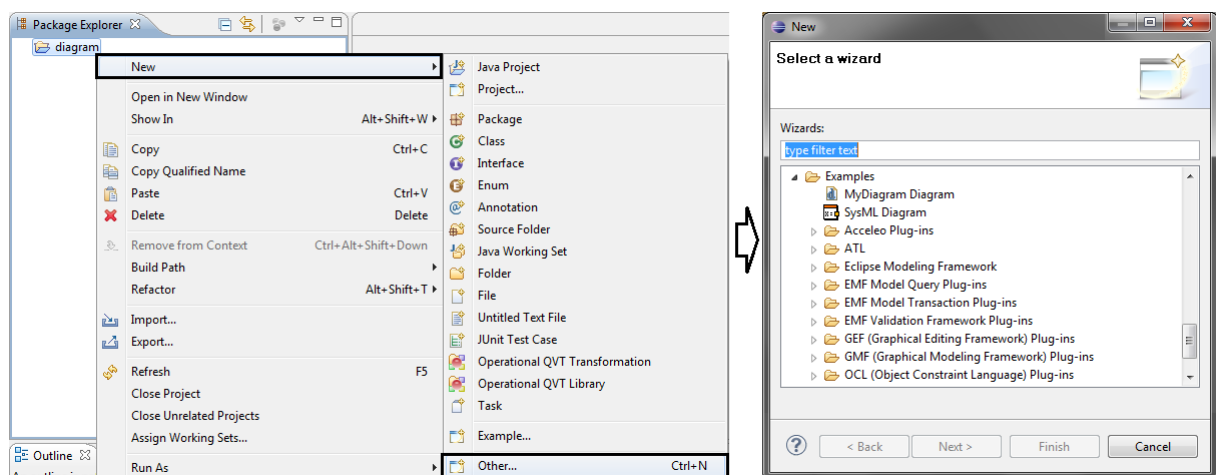


Figura A.13. Passos para a criação de um modelo.

Na Figura A.14 é apresentada uma visão geral da ferramenta CASE do diagrama. Normalmente, um modelo criado de um diagrama construído com o GMF é organizado em dois arquivos com formato XML:

1. O arquivo dos dados, que armazena as informações do modelo relacionadas com o domínio definido no metamodelo do diagrama e é identificado por uma extensão formada pelo nome desse domínio;
2. O arquivo do diagrama, que armazena as dimensões (posição, tamanho, cor, etc.) dos elementos contidos no modelo e é identificado por uma extensão formada pelo nome do domínio do diagrama concatenado com a *string* “_diagram”.

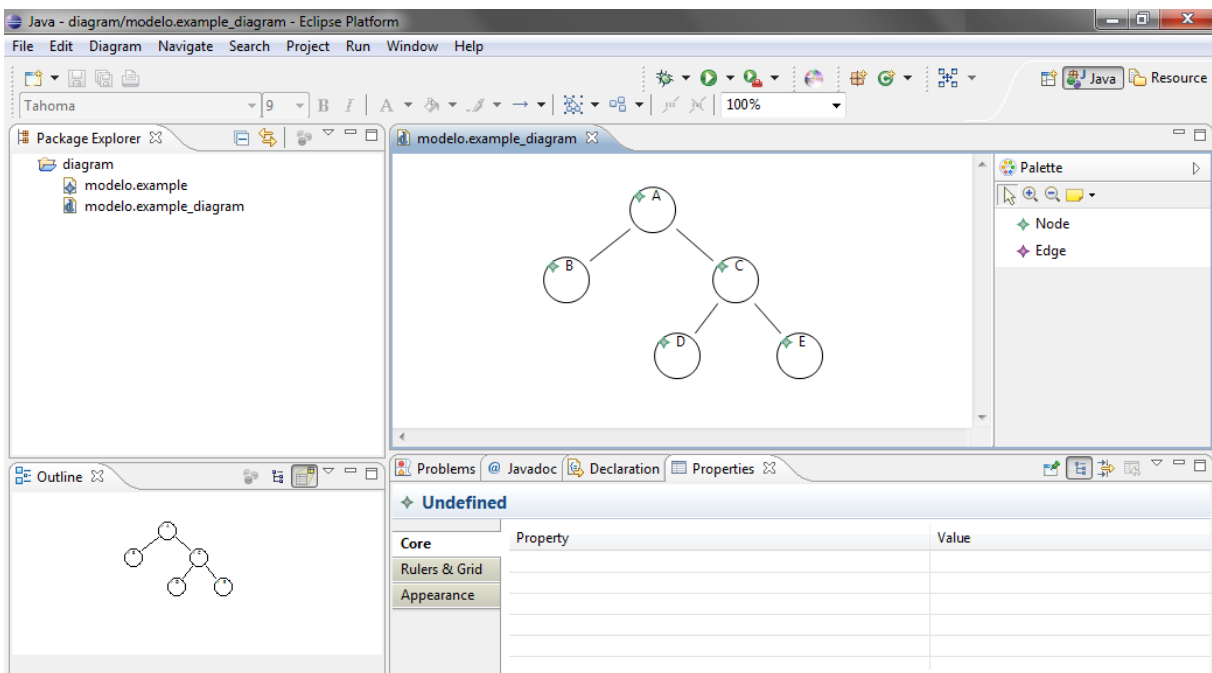


Figura A.14. Visão geral da ferramenta CASE do diagrama.

A.3 Considerações Finais

O GMF é uma ferramenta que pode ser utilizada para a criação de diagramas (linguagens de modelagem) com notação gráfica. Diversos tipos de imagens podem ser utilizadas para representar graficamente os elementos e os relacionamentos existentes nos modelos criados com os diagramas. Uma das grandes vantagens do GMF é estar inserido no Eclipse IDE, de modo que os demais recursos dessa IDE, como, por exemplo, transformações *Model-to-Model*, geração de código, compiladores, *frameworks* de testes, entre outros, podem ser utilizados para aumentar a funcionalidade da ferramenta CASE dos diagramas e utilizá-los no processo de desenvolvimento de software.

Apêndice B

INTRODUÇÃO À PURE::VARIANTS

B.1 Introdução

Pure::variants⁶ é uma ferramenta de uso proprietário para o desenvolvimento de Linhas de Produto de Software (LPS). Essa ferramenta foi criada com base no Eclipse IDE e pode manipular aplicações desenvolvidas em diferentes linguagens de programação e para diferentes plataformas. Na Figura B.1 é mostrada a interface da ferramenta pure::variants.

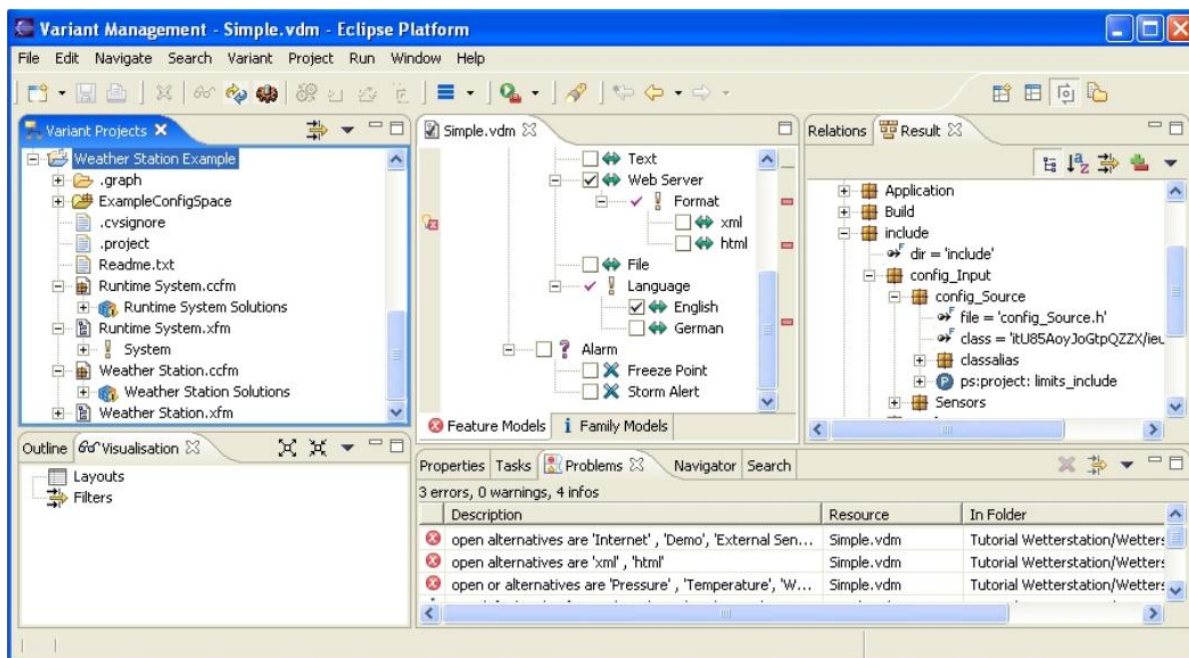


Figura B.1. Interface da ferramenta Pure::variants.

O desenvolvimento de uma LPS com a ferramenta pure::variants segue o modelo de processo com as fases de Engenharia de Domínio e Engenharia de Aplicação. Porém, essa

⁶ <http://www.pure-systems.com>

ferramenta permite o desenvolvimento de uma LPS a partir do código-fonte de uma aplicação. Neste manual são apresentados os passos para realizar esse processo. Como exemplo, é utilizada uma aplicação que controla os dispositivos de um carro.

B.2 Aplicação para o Domínio de Carros

Os requisitos da aplicação que controla os dispositivos de um carro são:

- Freios (Breaks): todo carro tem um sistema de freios;
- Motor (Engine): todo carro possui um motor;
- Caixa de Marcha (Gearbox): todo carro possui um dos dois tipos de caixa de marcha:
 - Manual: o motorista é responsável por selecionar a marcha manualmente;
 - Automático (Automatic): o carro seleciona a marcha automaticamente;
- Acessório (Accessory): item opcional que pode ser adicionado a qualquer carro:
 - ABS: sistema que controla a forma de funcionamento dos freios;
 - Ar Condicionado (Air Conditioner): controla a temperatura da cabine.

O modelo de classes da aplicação é mostrado na Figura B.2. O valor mínimo da multiplicidade dos relacionamentos indica a obrigatoriedade das características: 0, para opcionais; 1 para obrigatório. A multiplicidade * indica que mais de uma das variantes de uma característica pode ser escolhida para um único carro.

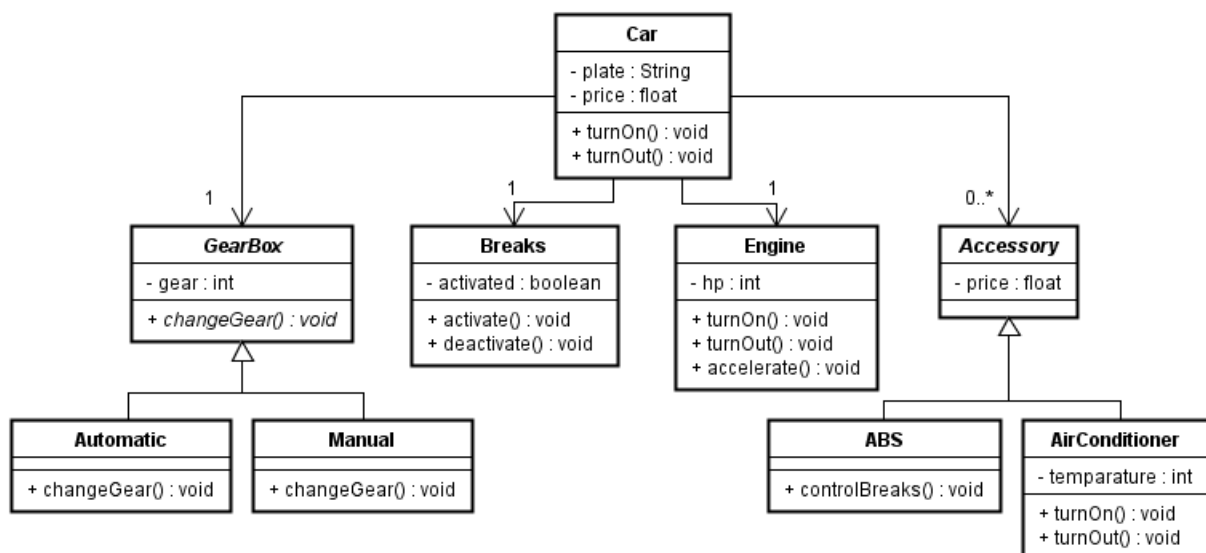


Figura B.2. Modelo de classes do domínio de carros.

Nessa aplicação podem ser identificadas características opcionais (acessórios) e alternativas (caixa de marcha). Portanto, podem ser criadas outras variantes dessa aplicação a partir das possíveis combinações dessas características. A ferramenta pure::variants gera essas variantes, que são produtos de uma LPS, com base no código-fonte da aplicação original.

B.3 Engenharia de Domínio

Para realizar o desenvolvimento da LPS é necessário que o projeto da aplicação original esteja aberto no *workspace* da ferramenta. Na Figura B.3 é mostrada a interface da pure::variants com o projeto Java da aplicação de carros. A fase de engenharia de domínio com a pure::variants inicia com a criação de projeto de variantes.

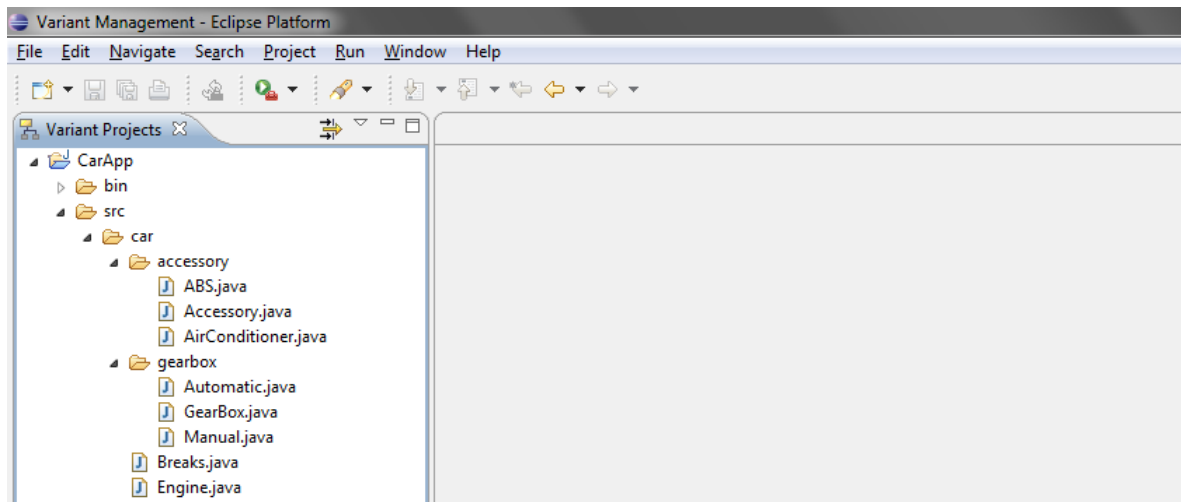


Figura B.3. Interface da pure::variants com o projeto Java da aplicação de carros.

B.3.1 Criando o projeto da LPS

Na Figura B.4 são mostrados os passos para a criação de um projeto de LPS vazio: A) com um clique com o botão direito do mouse no painel de projetos, selecione a opção *New >> Variant Project*; B) na janela que abre, digite o nome do projeto, selecione o tipo de projeto vazio (*Project Type Empty*) e clique no botão *Finish*.

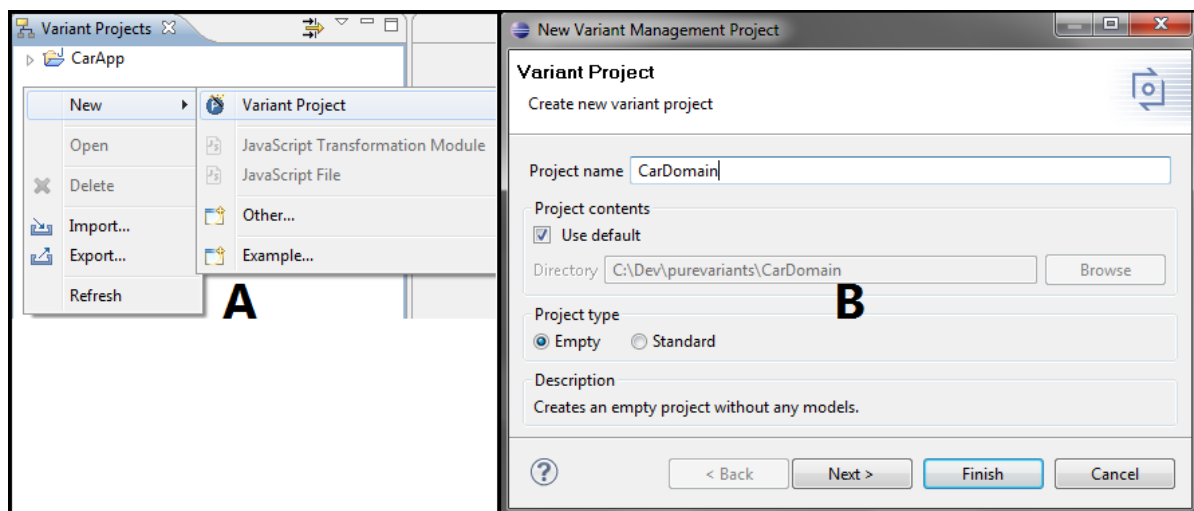


Figura B.4. Criação de um projeto de LPS (Variant Project) com a pure::variants.

B.3.2 Criando o modelo da arquitetura da LPS

Com o projeto da LPS criado, é necessário identificar os componentes da aplicação que serve de base para a LPS, que nesse caso, é a aplicação de carros. Para isso, realize os seguintes passos mostrados na Figura B.5: 1) clique com o botão direito sobre a pasta do projeto da LPS e selecione a opção *import*; 2) na janela que abre, selecione a opção *Variant Models or Projects* e clique no botão *Next*; 3) selecione a opção *Import a Family Model from source folders* e clique novamente no botão *Next*; 4) selecione a opção *Java Project* e clique novamente no botão *Next*; 5) por fim, selecione a pasta do código-fonte da aplicação, selecione o projeto da LPS, indique um nome para o modelo do domínio e para o nome do arquivo desse modelo e clique no botão *Finish*. Ao final desse processo, é criado o modelo da arquitetura da LPS (arquivo de extensão *.ccfm*), mostrado na Figura B.6.

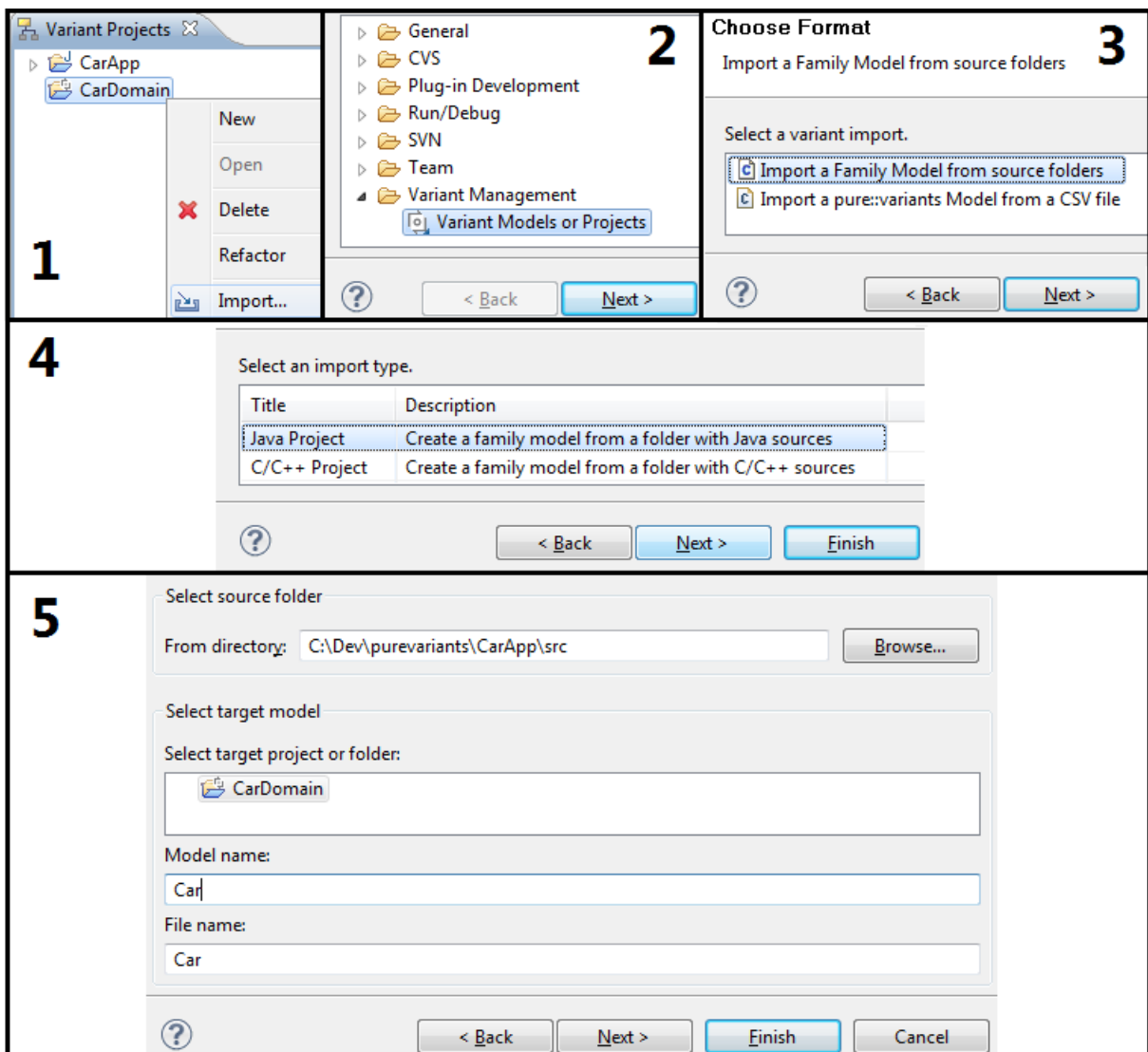


Figura B.5. Identificação dos componentes da aplicação para o projeto da LPS.

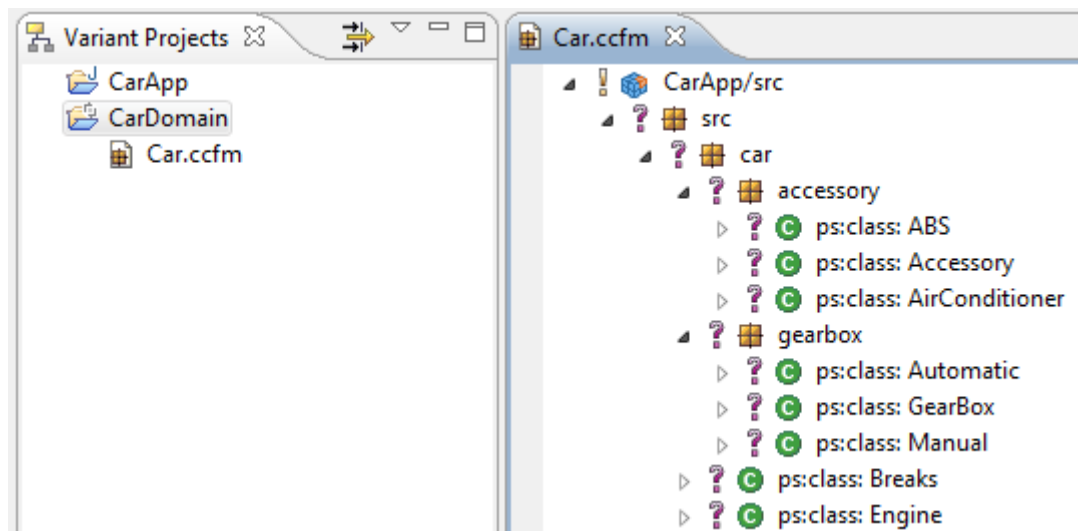


Figura B.6. Modelo da arquitetura da LPS.

B.3.3 Criando o modelo de características do domínio da LPS

O passo seguinte é criar o modelo de características (features) do domínio. Na Figura B.7 mostrado que para criar o arquivo desse modelo (arquivo de extensão *.xfm*) basta clicar com o botão direito do mouse sobre a pasta do projeto da LPS e selecionar a opção *New >> Feature Model*.

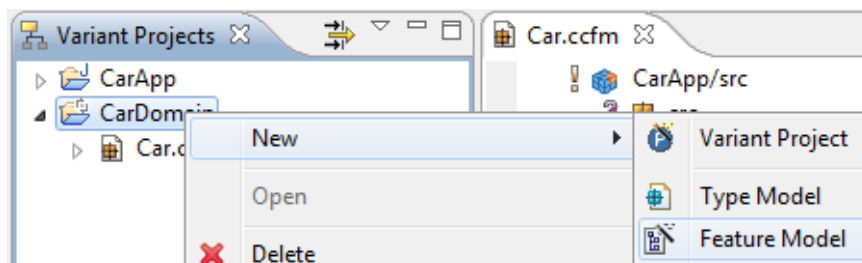


Figura B.7. Criação do arquivo do modelo de características da LPS.

Na Figura B.8 é mostrado o conteúdo do modelo de características do domínio de aplicações de carros. Esse modelo segue os requisitos definidos na Seção 2 deste manual.

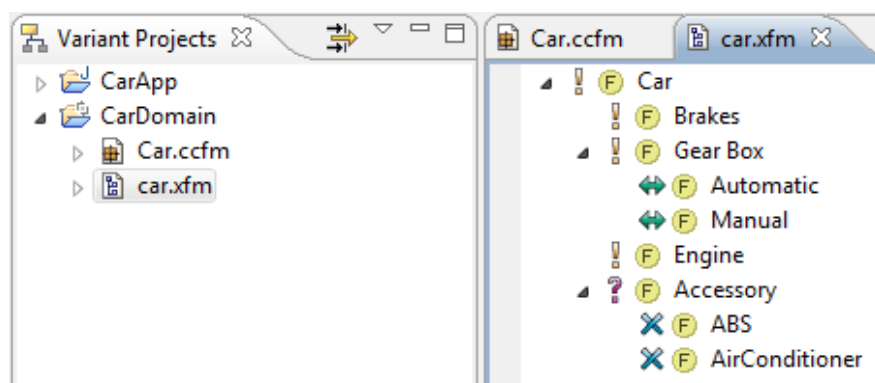


Figura B.8. Modelo de características da LPS de carros.

B.3.4 Criando o espaço de configuração dos produtos da LPS

O último passo da fase de engenharia de domínio da pure::variants é a criação do espaço de configuração dos produtos da LPS. Esse espaço é a pasta onde ficam armazenados os modelos das aplicações (variantes) criados na fase de engenharia de aplicação. Como mostrado na Figura B.9, o espaço de configuração é criado da seguinte forma: 1) clique com o botão direito do mouse sobre a pasta do projeto da LPS e selecione a opção NEW >> Configuration Space; e 2) na janela que abre, digite o nome do espaço de configuração, desmarque a opção *create default variant description* e clique no botão *Finish*.

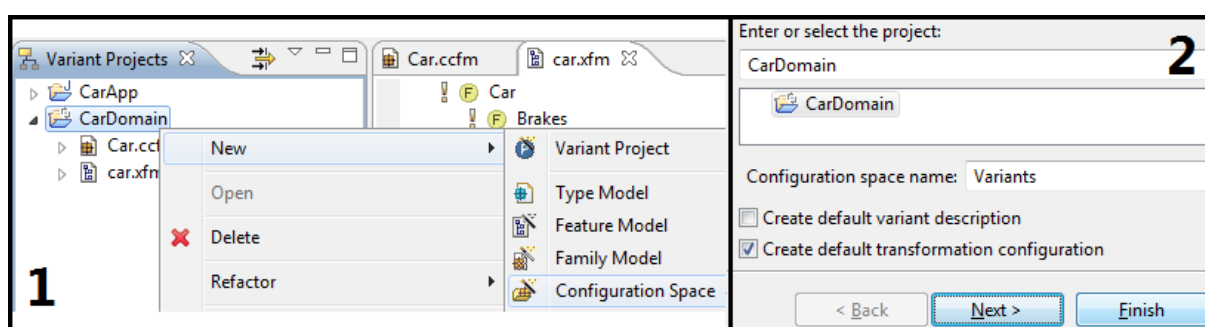


Figura B.9. Passos para criação do espaço de configuração da LPS.

B.4 Engenharia de Aplicação

A fase de engenharia de aplicação da pure::variants é realizada com a criação do modelo e a geração de uma aplicação (variante).

B.4.1 Criando o modelo de uma aplicação

Para criar o modelo de uma aplicação (variante da aplicação original) é necessário realizar os seguintes passos (Figura B.10): 1) Clique com o botão direito do mouse na pasta do espaço de configuração da LPS e selecione a opção *New >> Variant Model*; 2) informe o nome do modelo e clique no botão *Next*; 3) selecione os modelos da arquitetura e de características da LPS e clique no botão *next*; e 4) indique o workspace como diretório de entrada (*Input Path*), pois é onde se encontra o projeto da aplicação original, selecione a pasta de saída (*Output path*) onde será gerado o código das aplicações novas (no exemplo, a pasta *output* dentro do projeto da LPS) e clique no botão *Finish*;

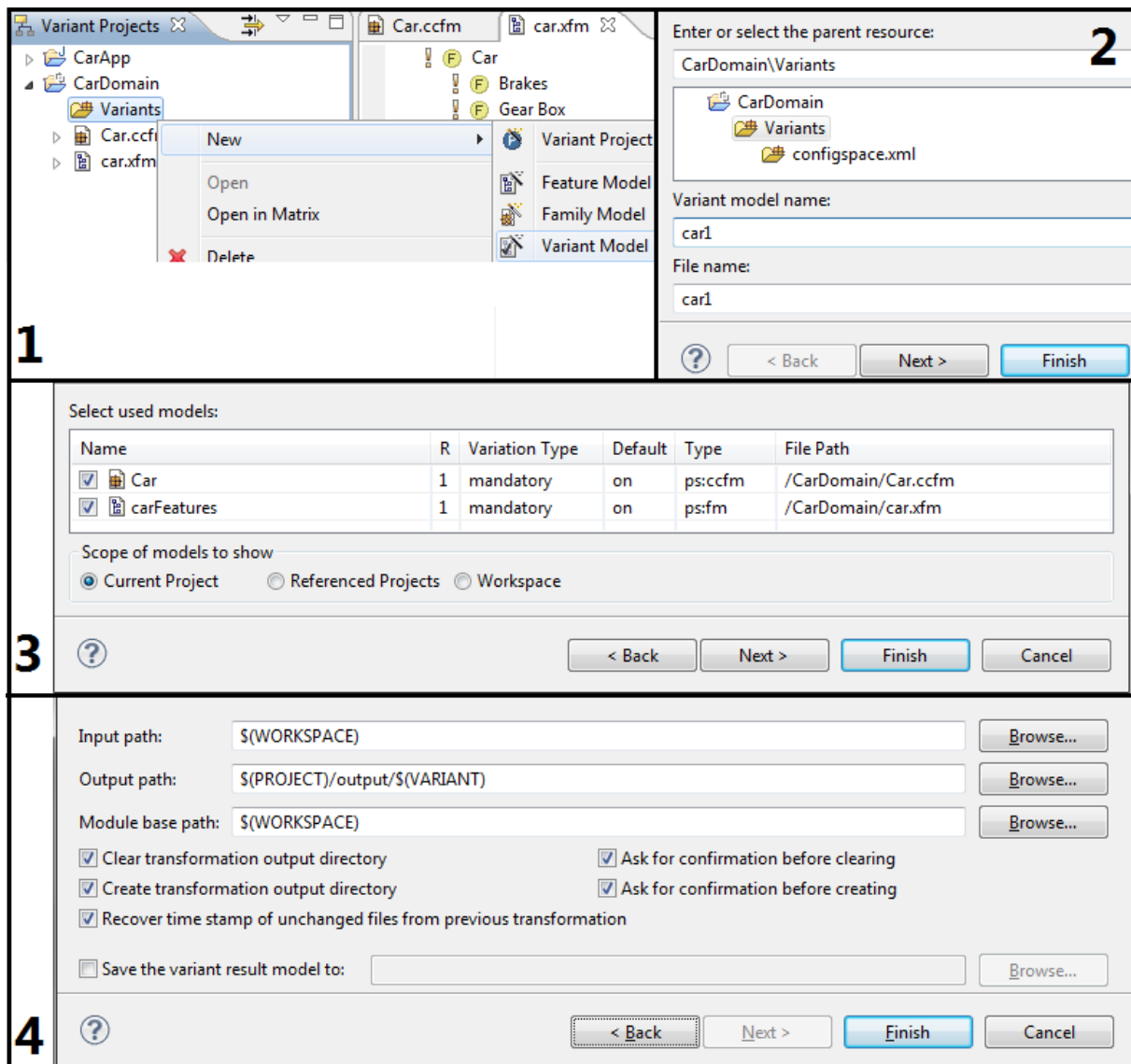


Figura B.10. Passos para criação do modelo de uma aplicação.

Na Figura B.11 é mostrado um exemplo de modelo de aplicação de um carro variante da aplicação original. Essa aplicação contém todas as características obrigatórias, freios, motor e caixa de marcha, porém não possui acessórios e utiliza o câmbio manual.

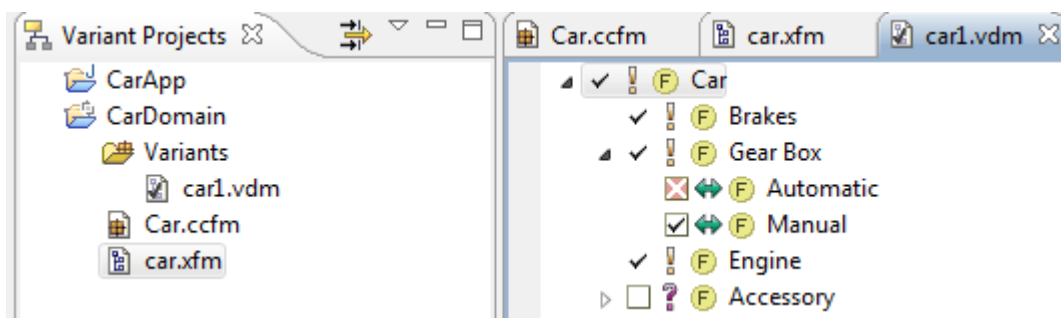


Figura B.11. Exemplo de modelo de uma aplicação de carros.

B.4.2 Gerando a aplicação

Para gerar a aplicação a partir do seu modelo, primeiramente é necessário abrir o arquivo da arquitetura da LPS e selecionar a visualização em tabela, como mostrado na Figura B.12. Nessa visualização, selecione os arquivos Java *Automatic*, *Accessory*, *ABS* e *AirConditioner* clicando com o botão direito do mouse e desmarque a opção *Refactor >> Default Selected*. Nesse passo, tome cuidado para selecionar os arquivos Java (ícones de pagina de texto) e não as classes (ícones verdes). Isso faz com que somente as características selecionadas no modelo da aplicação sejam inclusas no código gerado.

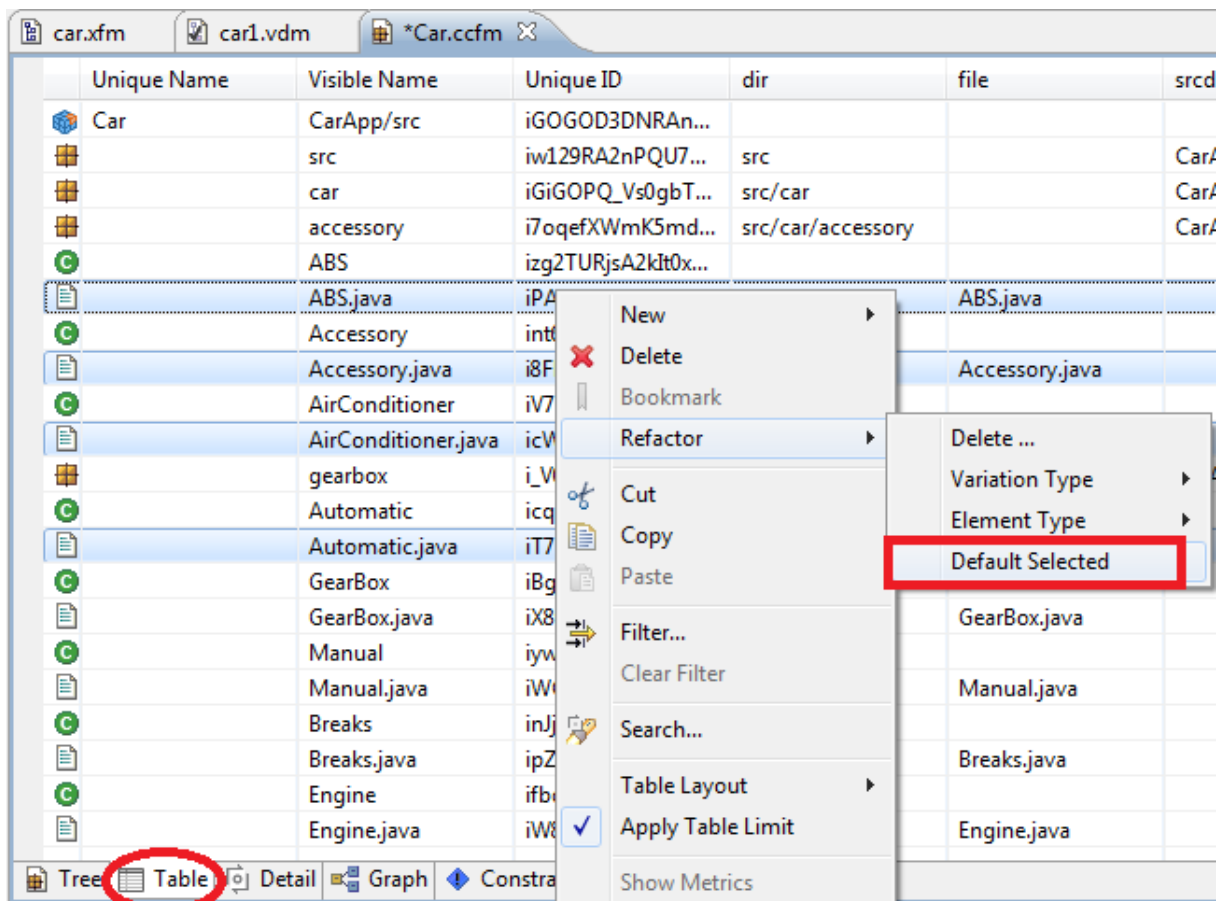


Figura B.12. Configurando o modelo de arquitetura para a geração da aplicação modelada.

Para gerar o código, volte para o modelo da aplicação e clique no botão *Transform Model* localizado no menu principal da ferramenta, como mostrado na Figura B.13. A variante da aplicação gerada possui somente as classes referentes às características selecionadas da aplicação base.

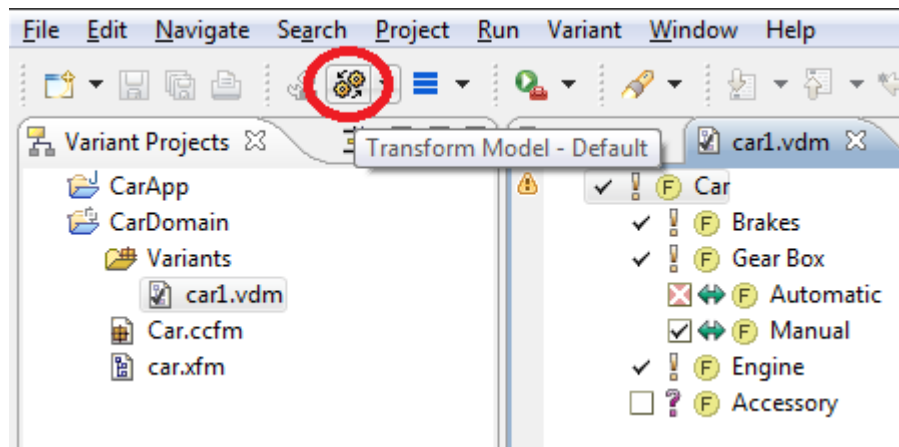


Figura B.13. Gerando o código da aplicação.

Na Figura B.14 são mostrados os arquivos que compõem a aplicação gerada. Note que as classes referentes às características não selecionadas no modelo não estão presentes na estrutura de arquivos da aplicação gerada.

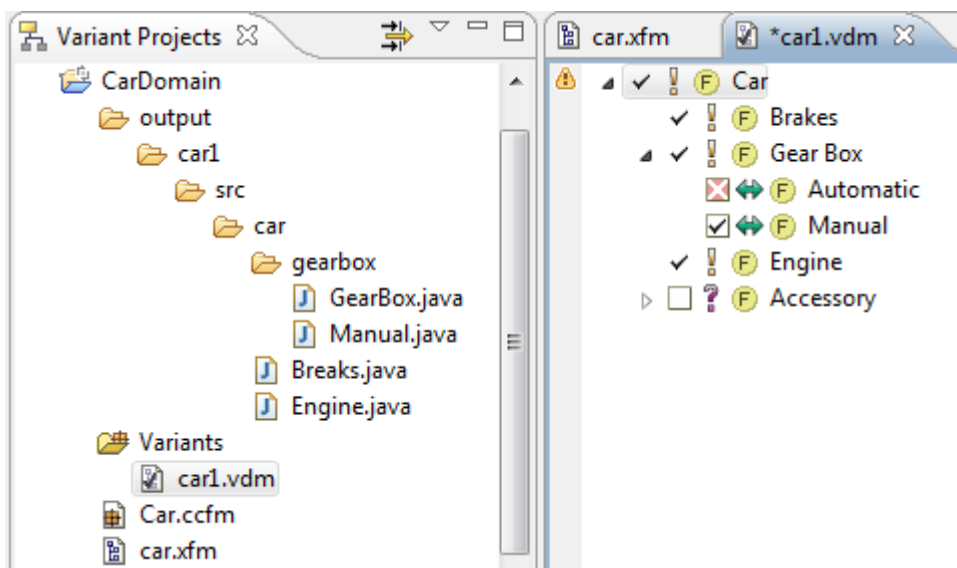


Figura B.14. Arquivos da aplicação.

Apêndice C

FORMULÁRIOS DOS EXPERIMENTOS

Formulário de Caracterização de Participante dos Experimentos 1 e 2

Nome:

1. Conhecimento sobre Linguagens de programação:

Já programou com quais linguagens de programação? _____

Você já trabalhou por pelo menos 1 ano com alguma dessas linguagens? () Sim () Não

2. Conhecimento sobre Java:

() Básico – uso de recursos comuns às linguagens de programação orientadas a objetos.

() Intermediário – Interface gráfica, estruturas de dados, genéricos.

() Avançado – programação web, reflection.

3. Conhecimento sobre Padrões de Software:

() Nunca tinha utilizado antes das aulas da disciplina.

() Já estudei vários padrões teoricamente, mas trabalhei pouco com eles na prática.

() Trabalho(ei) com padrões em aulas, projetos ou no desenvolvimento de sistemas.

4. Conhecimento sobre Frameworks:

() Nunca tinha utilizado antes das aulas da disciplina.

() Já estudei vários frameworks teoricamente, mas trabalhei pouco com eles na prática.

() Trabalho(ei) com frameworks em aulas, projetos ou no desenvolvimento de sistemas.

5. Conhecimento sobre MDD (marque uma opção em cada linha):

Entendi () pouco () bem sobre a teoria de MDD.

Entendi () pouco () bem sobre metamodelagem.

Entendi () pouco () bem sobre transformações.

Formulário de Coleta de Dados do Experimento 1

DADOS		
NOME		
ATIVIDADE	1 ()	2 ()
GRUPO	1 ()	1 ()
APLICAÇÃO	Hotel ()	Biblioteca ()
FERRAMENTA	DSL ()	Wizard ()

TEMPOS	Início	Fim

INTERRUPÇÕES	Início	Fim

OBSERVAÇÕES:

Problemas e dificuldades encontradas durante o experimento:

COMENTÁRIOS:

Pontos positivos e negativos da ferramenta utilizada (Ex: problemas na interface, facilidade de uso, funções difíceis de achar, eficiência, etc.):

Formulário de Coleta de Dados do Experimento 2

DADOS	
NOME	
ATIVIDADE	1 () 2 ()
GRUPO	1 () 1 ()
DOMÍNIO	Transações () Veículos ()
ABORDAGEM	F3 () Ad hoc ()

TEMPOS	Início	Fim

INTERRUPÇÕES	Início	Fim

OBSERVAÇÕES:

Problemas e dificuldades encontradas durante o experimento:

COMENTÁRIOS:

Pontos positivos e negativos da abordagem utilizada:

Formulário de Caracterização de Participante do Experimento 3

Nome:

1. Conhecimento sobre Linguagens de programação:

Já programou com quais linguagens de programação? _____

Você já trabalhou por pelo menos 1 ano com alguma dessas linguagens? () Sim () Não

2. Conhecimento sobre Java:

() Básico – uso de recursos comuns às linguagens de programação orientadas a objetos.

() Intermediário – Interface gráfica, estruturas de dados, genéricos.

() Avançado – programação web, reflection.

3. Conhecimento sobre Padrões de Software:

() Nunca tinha utilizado antes das aulas da disciplina.

() Já estudei vários padrões teoricamente, mas trabalhei pouco com eles na prática.

() Trabalho(ei) com padrões em aulas, projetos ou no desenvolvimento de sistemas.

4. Conhecimento sobre Frameworks:

() Nunca tinha utilizado antes das aulas da disciplina.

() Já estudei vários frameworks teoricamente, mas trabalhei pouco com eles na prática.

() Trabalho(ei) com frameworks em aulas, projetos ou no desenvolvimento de sistemas.

5. Conhecimento sobre MDD (marque uma opção em cada linha):

Entendi () pouco () bem sobre a teoria de MDD.

Entendi () pouco () bem sobre metamodelagem.

Entendi () pouco () bem sobre transformações.

6. Sobre a ferramenta F3T

() Não Sei () Sei criar um modelo F3 de um domínio e gerar um framework a partir dele.

() Não Sei () Sei criar os modelos GMF para a DSL do framework.

() Não Sei () Sei instalar e usar a DSL do framework no Eclipse.

() Não Sei () Sei gerar código a partir de um modelo de uma aplicação.

7. Sobre a ferramenta pure::variants

() Não Sei () Sei criar o modelo de arquitetura a partir do projeto de uma aplicação.

() Não Sei () Sei criar o modelo de features do domínio.

() Não Sei () Sei criar um espaço de configuração e um modelo de variante nele.

() Não Sei () Sei gerar código da aplicação a partir do seu modelo de variante.

Formulário de Coleta de Dados do Experimento 3

DADOS		
NOME		
ATIVIDADE	1 ()	2 ()
GRUPO	1 ()	1 ()
DOMINIO	Transações ()	Atendimento ()
FERRAMENTA	F3T ()	pure::variants ()

TEMPOS	Início	Fim
Implementação (Pure::variants)		
Engenharia do Domínio		
Engenharia da Aplicação		

INTERRUPÇÕES	Início	Fim

OBSERVAÇÕES:

Problemas e dificuldades encontradas durante o experimento:

COMENTÁRIOS:

Pontos positivos e negativos da ferramenta utilizada (Ex: problemas na interface, facilidade de uso, funções difíceis de achar, eficiência, etc.):