VICTOR TRAVASSOS SARINHO

OOFM - UMA TÉCNICA DE MODELAGEM DE FEATURES ORIENTADA A OBJETOS

Tese apresentada ao Programa Multiinstitucional de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia, Universidade Salvador e Universidade Estadual de Feira de Santana, como requisito parcial à obtenção do título de Doutor em Ciência da Computação.

Orientador: Prof. Dr. Antônio Lopes Apolinário Júnior

Salvador

2013

Sarinho, Victor Travassos.

OOFM - Uma Técnica de Modelagem de Features Orientada a Objetos / Victor Travassos Sarinho. – Salvador, 2013.

171f. : il.

Orientador: Prof. Dr. Antônio Lopes Apolinário Júnior.

Tese (doutorado) – Universidade Federal da Bahia, Instituto de Matemática, Doutorado Multiinstitucional em Ciência da Computação, 2013.

Referências bibliográficas.

1. Variabilidade de Software. 2. Linhas de Produto de Software. 3. Modelagem de Features.

I. Apolinário Jr., Antônio Lopes. II. Universidade Federal da Bahia, Instituto de Matemática. III. D.Sc.

CDU – 004.41

VICTOR TRAVASSOS SARINHO

ESTA PÁGINA DEVE SER ASSINADA PELA BANCA

OOFM - UMA TÉCNICA DE MODELAGEM DE FEATURES ORIENTADA A OBJETOS

Esta tese foi julgada adequada à obtenção do título de Doutor em Ciência da Computação e aprovada em sua forma final pelo Programa Multiinstitucional de Pós-Graduação em Ciência da Computação da UFBA-UNIFACS-UEFS.

Salvador, 18 de fevereiro de 2013.

____

Prof. Antônio Lopes Apolinário Júnior (orientador), D.Sc.
Universidade Federal da Bahia – UFBA

____

Prof. Vander Ramos Alves, D.Sc.
Universidade de Brasília - UnB

____

Prof. André Menezes Marques das Neves, D.Sc.
Universidade Federal de Pernambuco - UFPE

____

Prof. Manoel Gomes de Mendonça Neto, Ph.D.
Universidade Federal da Bahia – UFBA

____

Prof[a]. Eduardo de Almeida Santana, D.Sc.
Universidade Federal da Bahia – UFBA

RESUMO

Modelagem de Features é uma abordagem popular que descreve a comunalidade e variabilidade de famílias de softwares em termos de features. Variabilidade em Software Product Lines (SPLs) é geralmente descrita usando features, e instâncias de SPLs também são definidas pela seleção ou configuração de features requeridas. Entretanto, abordagens diversas e complexas de SPLs têm sido obtidas com o uso de features. Diferentes abordagens de modelagem de features também têm sido propostas nos últimos anos, abrindo novas perspectivas em aspectos de variabilidade a serem gerenciados.

A técnica de Modelagem de Features Orientada a Objetos (OOFM) foi proposta com o objetivo de cobrir tais aspectos de variabilidade, bem como fornecer uma solução padronizada de produção de SPLs baseadas em features. OOFM se baseia em abordagens OO representativas de features, operações comuns identificadas de manipulação de features, e recursos OO de programação e de herança existentes. Sua formalização é baseada em expressões OCL definidas e Perfil UML modelado. Ferramentas de suporte incrementam a sua usabilidade e compatibilidade com relação a técnicas de modelagem de features existentes. Finalmente, OOFM Framework e OOFM Process desenvolvidos garantem a produção padronizada de SPLs e de sistemas concretos com base na OOFM.

Foram realizadas importantes avaliações entre a proposta OOFM e técnicas de modelagem de features existentes. Como resultado, OOFM se apresentou como uma solução integrada de aspectos de variabilidade que permite a análise e o projeto da variabilidade de sistemas diversos em termos de features.

Palavras chave: Modelagem de Features, Linha de Produto de Software, Aspectos de Variabilidade, Modelagem de Features Orientada a Objetos.

ABSTRACT

Feature modeling is a popular approach that describes the commonality and variability of software families in terms of features. In general, Software Product Lines (SPLs) variability is described using features, and SPL instances are also defined by the selection or configuration of required features. However, different and complex SPL approaches have been provided using features. Different types of feature modeling approaches have also been proposed during the years, opening new perspectives in variability aspects that could be managed.

The Object Oriented Feature Modeling (OOFM) technique was proposed in order to cover these variability aspects, and provide a standardized solution for producing feature based SPLs. OOFM is based on OO representative approaches of features, identified operations for feature manipulation, and available resources of OO programming and inheritance. Its formalization is based on defined OCL expressions and modeled UML Profile. Support tools improve its usability and compatibility according to available feature modeling techniques. Finally, developed OOFM Framework and OOFM Process guarantee the standardized production of OOFM based SPLs and concrete systems.

Important evaluations were performed among OOFM and available feature modeling techniques. As a result, OOFM was presented as an integrated solution of variability aspects that allows variability analysis and design of systems in terms of features.

Keywords: Feature Modeling, Software Product Lines, Variability Aspects, Object Oriented Feature Modeling.

A minha Mãe Elizabeth, meu pai José Pessoa (in memoriam), minha esposa Poliany e meus filhos Vinícius, Júlia e Ícaro pelo simples fato de serem a razão da minha vida.

AGRADECIMENTOS

Quando eu decidi fazer o vestibular para Ciências da Computação em 1995 na UFPB, minha mãe Elizabeth, em um dado momento da minha vida depois que eu lhe informei sobre esta decisão, olhou para mim e disse: “que pena, parece que nenhum dos meus filhos vai querer ser doutor na vida”. De fato mãe, existem muitos tipos de “doutores” nesta vida, e é por isso que eu lhe dedico esta tese, para que você tenha certeza de que tem sim um filho para chamar de “doutor” na família.

Agradeço também ao meu pai José que, mesmo falecido, sempre me serviu de exemplo e referência de que tudo se consegue e é possível com estudo, honestidade, humildade, dedicação e perseverança. Como diria meu sogro Marinaldo e meu padastro Vicente, “sem sacrifício não há benefício” e “o céu é o limite”, respectivamente.

Gostaria também de agradecer ao meu orientador prof. Apolinário pela paciência e dedicação para com este orientando, uma vez que tenho ciência do trabalho que é ensinar o caminho do conhecimento aos nossos semelhantes. Também faço um agradecimento especial ao prof. Eduardo Santana (RiSE), pela sua contribuição direta e indireta em diversas etapas desta tese, e aos demais professores da UEFS, os quais permitiram o meu afastamento sem o qual não seria possível a conclusão desta etapa na minha vida.

Para minha esposa Poliany e meus filhos Vinícius, Júlia e Ícaro, que este trabalho sirva de exemplo e inspiração para que procurem sempre buscar o melhor de si nos seus objetivos de vida. Vocês sempre serão a razão da minha vida e tudo que eu faço é sempre pensando em vocês.

Finalmente, agradeço a todos que permitiram a criação e consolidação deste programa de doutoramento multiinstitucional UFBA/Unifacs/UEFS. Trata-se de um projeto único e de fundamental importância para todos que fazem a computação acontecer na Bahia.

Obrigado a todos!

Só sei que nada sei.

Sócrates

SUMÁRIO

1 INTRODUÇÃO ... 25
1.1 OBJETIVOS ... 28
1.2 ORGANIZAÇÃO DA TESE ... 29
2 A TÉCNICA DE MODELAGEM DE FEATURES ORIENTADA A OBJETOS ... 31
2.1 CONCEITOS CHAVE ... 32
2.2 DETALHANDO A OOFM ... 33
2.2.1 Restrições de Modelagem ... 38
2.3 ESPECIFICAÇÃO FORMAL ... 40
2.3.1 Expressões OCL para Estruturas OOFM ... 41
2.3.2 Expressões OCL para Operações OOFM ... 45
2.3.3 Um Perfil OOFM ... 47
2.4 FERRAMENTAS DE SUPORTE ... 49
2.5 CONCLUSÃO DO CAPÍTULO ... 52
3 LINHAS DE PRODUTO DE SOFTWARE BASEADAS NA OOFM ... 53
3.1 OOFM FRAMEWORK ... 54
3.2 ATIVIDADES DE SPLE PARA RECURSOS OOFM ... 58
3.3 OOFM PROCESS ... 60
3.4 CONCLUSÃO DO CAPÍTULO ... 62
4 IMPLEMENTANDO JOGOS DIGITAIS COM A OOFM ... 63
4.1 MODELOS NESI E GDS ... 64
4.2 FEATURE-BASED ENVIRONMENT FOR DIGITAL GAMES ... 67
4.3 DESENVOLVENDO O JOGO SIMPLIFIEDPACMAN ... 70
4.4 CONCLUSÃO DO CAPÍTULO ... 73
5 AVALIANDO A TÉCNICA OOFM ... 75
5.1 GERANDO DOCUMENTOS CORRETOS ... 76
5.2 MODELANDO RECURSOS COMPATÍVEIS ... 86
5.3 AVALIANDO ATRIBUTOS DE COMPLEXIDADE ... 92
5.4 AVALIANDO A USABILIDADE NA MODELAGEM ... 95

5.5 COMPARANDO ASPECTOS DE VARIABILIDADE 96
5.6 CONCLUSÃO DO CAPÍTULO 101
6 CONCLUSÕES... 103
REFERÊNCIAS.. 107
APÊNDICE A – QUESTIONÁRIO: AVALIAÇÃO DA ABORDAGEM OBJECT ORIENTED FEATURE MODELING (OOFM) NA MODELAGEM DE FEATURES...... 113

LISTA DE FIGURAS

FIGURA 1 IDÉIAS BASE DE TÉCNICAS DE MODELAGEM DE FEATURES USADAS NA OOFM...27
FIGURA 2 PRODUÇÃO DE SPL DE DOMÍNIO ESPECÍFICO COM A OOFM..27
FIGURA 3 EXEMPLO DE USO DOS CONCEITOS CHAVE DA OOFM........................33
FIGURA 4 O METAMODELO OOFM...34
FIGURA 5 OPERAÇÕES DE FEATURES DA CLASSE *FEATURECONFIGURATION*....37
FIGURA 6 EXPRESSÕES OCL PARA *FEATURETYPE E ATTRIBUTETYPE*.................42
FIGURA 7 EXPRESSÕES OCL PARA *GROUPTYPE*..43
FIGURA 8 EXPRESSÕES OCL PARA *FEATURE* E *ATTRIBUTE*.................................44
FIGURA 9 EXPRESSÕES OCL PARA *GROUP*..45
FIGURA 10 PRÉ E PÓS CONDIÇÕES DE ALGUMAS OPERAÇÕES DE *FEATUREMODEL*...46
FIGURA 11 PERFIL OOFM...48
FIGURA 12 MODELO OOFM PARA O DOMÍNIO DE AUTOMÓVEIS.........................50
FIGURA 13 EXEMPLO OOFM-XML PARA O DOMÍNIO DE AUTOMÓVEIS...51
FIGURA 14 *FEATURECLASS EDITOR* TRABALHANDO COM O EXEMPLO OOFM-XML PARA O DOMÍNIO DE AUTOMÓVEIS...52
FIGURA 15 DESCRIÇÃO PARCIAL DE OPERAÇÕES E VALIDAÇÕES DO OOFM FRAMEWORK EM JAVA...56
FIGURA 16 ARQUITETURA MVC DO OOFM FRAMEWORK......................................57
FIGURA 17 ATIVIDADES DE SPLE (À ESQUERDA) E SUAS ADAPTAÇÕES PARA RECURSOS OOFM (À DIREITA)..58
FIGURA 18 DIAGRAMA DE ATIVIDADES DO OOFM PROCESS...............................61

FIGURA 19 DIAGRAMA SIMPLIFICADO DO MODELO NESI.....64
FIGURA 20 MODELO DE FEATURES DA SUBFEATURE *ELEMENT*.....65
FIGURA 21 DIAGRAMA SIMPLIFICADO DO MODELO GDS.....66
FIGURA 22 MODELO DE FEATURES DA SUBFEATURE *SCENEVIEW*.....67
FIGURA 23 FENDIGA PROCESS.....68
FIGURA 24 VISÃO MODULAR DO FENDIGA FRAMEWORK.....69
FIGURA 25 DESCRIÇÃO PARCIAL DO MÉTODO *CREATESIMPLIFIEDPACMANGAME*.....71
FIGURA 26 DESCRIÇÃO PARCIAL DO MÉTODO *INITRESOURCES*.....72
FIGURA 27 ARQUITETURA E IMAGENS DO JOGO *SIMPLIFIEDPACMAN* VIA FENDIGA.....73
FIGURA 28 REPRESENTAÇÃO TEXTUAL DE ELEMENTOS DO PERFIL OOFM.....77
FIGURA 29 COMPILAÇÃO E CRIAÇÃO DE ELEMENTOS ESTRUTURAIS, COMPORTAMENTAIS E RESTRITIVOS DA OOFM VIA FERRAMENTO USE SYSTEM.....78
FIGURA 30 EXEMPLOS DE PROCEDIMENTOS ASSL.....80
FIGURA 31 SCRIPT ASSL PARA A GERAÇÃO DE ÁRVORES DE FEATURES OOFM.....83
FIGURA 32 SCRIPT ASSL PARA A GERAÇÃO DE HIERARQUIAS DE FEATURES OOFM.....84
FIGURA 33 DESCRIÇÃO PARCIAL DO SCRIPT ASSL *CREATINGTREETC.CMD*.....85
FIGURA 34 MODELO OOFM EQUIVALENTE AO DOMÍNIO DE E-SHOP VIA CBFM.....87
FIGURA 35 MODELO OOFM EQUIVALENTE AO DOMÍNIO DE EDITORES DE TEXTO VIA FORFAMEL.....88
FIGURA 36 DESCRIÇÃO PARCIAL DO TRANSFORMADOR XSL DE OOFM PARA SXFM.....89
FIGURA 37 MODELO SPLOT EQUIVALENTE AO MODELO OOFM DE AUTOMÓVEIS GERADO VIA TRANSFORMADOR XSL PROPOSTO.....90

LISTA DE TABELAS

TABELA 1 DESCRIÇÃO RESUMIDA DOS PROCEDIMENTOS ASSL IMPLEMENTADOS..81

TABELA 2 DESCRIÇÃO RESUMIDA DOS SCRIPTS DE TESTE DAS OPERAÇÕES DE FEATURES DA OOFM..84

TABELA 3 OPERAÇÕES OOFM EQUIVALENTES A OPERAÇÕES DE MODELAGEM DE FEATURES COMPARADAS..90

TABELA 4 MÉTRICAS DE MODELAGEM DE FEATURES PARA MODELOS OOFM EQUIVALENTES..93

TABELA 5 MÉDIA DOS RESULTADOS DO QUESTIONÁRIO DE USABILIDADE DA OOFM..96

TABELA 6 ASPECTOS DE VARIABILIDADE IDENTIFICADOS NA OOFM E DEMAIS TÉCNICAS DE MODELAGEM DE FEATURES...100

GLOSSÁRIO DE TERMOS

Termo	Definição
Modelagem de Features	Abordagem de representação de sistemas com base na identificação e associação de features representativas dos mesmos.
Árvore de Features	Estrutura organizacional usada para representar features identificadas de um sistema.
Aspectos de Variabilidade	Conjunto de preocupações de análise e de projeto encontradas em trabalhos estado-da-arte na modelagem de features.
Grafos de Features	Representações inválidas de árvores de features causadas por relacionamentos de dependência entre features derivadas de ramos distintos.
Backtracking	Algoritmo geral que busca encontrar soluções para um determinado problema, através da construção incremental de candidatos a solução e do abandono destes candidatos quando se verifica que os mesmos não podem produzir uma solução válida.
Middleware	Denominação aplicada a uma estrutura arquitetural intermediária definida numa arquitetura de um sistema qualquer.
Game Engine	Software ou conjunto de bibliotecas que simplificam e abstraem o desenvolvimento de jogos digitais.

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Program Interface*
ASSL	*A Snapshot Sequence Language*
CBFM	*Cardinality Based Feature Modeling*
CVM	*Context Variability Model*
DSL	*Domain Specific Language*
DSPL	*Dynamic Software Product Line*
EMOF	Essential Meta-Object Facility
FArM	*Feature Architectural Mapping*
FC	*Feature Configuration*
FEnDiGa	*Feature-based Environment for Digital Games*
FM	*Feature Model*
FODA	*Feature Oriented Domain Analisys*
GDS	*GameSystem-DecisionSupport-SceneView*
MPL	*Multi-Product Line*
MVC	*Model View Controller*
NESI	*Narrative-Entertainment-Simulation-Interaction*
OCL	*Object Constraint Language*
OMG	*Object Management Group*
OO	*Object Oriented*
OOFM	*Object Oriented Feature Modeling*
PLA	*Product Line Architecture*
SPL	*Software Product Line*
SPLE	*Software Product Line Engineering*
SPLOT	*SPL Online Tools*
SXFM	*Simple XML Feature Model*
UML	*Unified Modeling Language*
XML	*Extensible Markup Language*
XSL	*Extensible Stylesheet Language*

Capítulo

1

Este capítulo apresenta uma breve descrição dos conceitos de modelagem de features e de seu uso e limitações nas linhas de produto de software. Em seguida são descritos alguns aspectos de variabilidade juntamente com uma breve descrição de como eles serão trabalhados na produção de uma nova técnica de modelagem de features orientada a objetos. Por último, os objetivos deste trabalho são definidos e a sua organização é apresentada.

1 INTRODUÇÃO

Modelagem de features é uma abordagem popular que descreve a comunalidade e a variabilidade de famílias de software em termos de features (Babar, Chen e Shull, 2010; Chen, Babar e Ali, 2009). Com relação à features, elas são unidades lógicas de comportamento, as quais representam um conjunto de requisitos funcionais e de qualidade (Gurp, Svahnberg e Bosch, 2001) capazes de identificar a variabilidade em diferentes artefatos de software produzidos (Svahnberg, Gurp e Bosch, 2005). Software Product Line (SPL) (Clements e Northrop, 2001) é um conjunto de sistemas intensivos de software compartilhando um conjunto de features comuns e gerenciadas que satisfaz necessidades específicas de um mercado ou missão particular, e que são desenvolvidos de um conjunto comum de recursos base de uma maneira prescrita.

Segundo Rosenmüller e Siegmund (2010), SPLs podem ser descritas usando Feature Models (FMs), e instâncias de SPLs (programas concretos) podem ser definidas pela seleção de features requeridas.

FMs são descrições organizadas de features relevantes de uma família de sistemas (Kang et al., 1990). Elas são usualmente estruturadas como árvores hierárquicas de features e respectivas sub-features (Cechticky et al., 2004). Feature Configurations (FCs) são fornecidas pela seleção de features disponíveis num FM (Kang et al., 1990), caracterizando uma espécie

de “configuração direta” de features. FCs também podem ser obtidas pela criação de novas e equivalentes instâncias de features (Asikainen, Männistö e Soininen, 2006; Asikainen, Männistö e Soininen, 2007), caracterizando assim uma “configuração indireta” de features. Cada FC produzido define uma instância de software numa família de softwares representada por um FM.

Segundo Chen, Babar e Ali (2009) e Babar, Chen e Shull (2010), Feature Oriented Domain Analysis (FODA), um esforço inicial de modelagem para representar features numa família de sistemas (Kang et al., 1990), é a abordagem de modelagem de features mais utilizada na representação da variabilidade de SPLs. Entretanto, FODA é bastante simples, gerando estratégias diversas e complexas quando usada na produção de SPLs avançadas (Rosenmüller e Siegmund, 2010), tais como Multi-Product Lines (MPLs) (Rosenmüller e Siegmund, 2010) e Dynamic SPLs (DSPLs) (Trinidad et al., 2007).

Mais ainda, diferentes tipos de abordagens de modelagem features têm sido propostas nos últimos anos (Babar, Chen e Shull, 2010; Chen, Babar e Ali, 2009; Benavides et al., 2006), tais como Forfamel (Asikainen, Männistö e Soininen, 2006) e Multi-level Feature Tree (Reiser e Weber, 2006). Elas abriram novas perspectivas em aspectos de variabilidade não cobertos por FODA que podem ser gerenciados em técnicas de modelagem de features, tais como: documentação indireta, manipulação, encapsulamento, reuso, formalização, ferramentas de suporte, programação de features, produção de sistemas, entre outras.

Este trabalho propõe a técnica de Modelagem de Features Orientada a Objetos (OOFM) (Sarinho e Apolinário, 2010) como uma solução para cobrir tais aspectos de variabilidade de uma maneira unificada, bem como fornecer SPLs avançadas baseadas em features de uma maneira padronizada (Sarinho, Apolinário e Almeida, 2012).

Para tal, soluções de modelagem de features correspondente a cada aspecto de variabilidade identificado serão atendidas de uma maneira compatível na técnica OOFM proposta. A Figura 1 ilustra esta idéia de integração de soluções encontradas em técnicas de modelagem de features existentes. Ela mostra as principais inovações de cada técnica de modelagem de features utilizadas como inspiração na definição da técnica OOFM proposta.

Uma estratégia padronizada de produção de SPLs com base em atividades de SPL Engineering (SPLE) (Beuche e Dalgarno, 2006) também será aplicada na OOFM proposta.

Ela define um OOFM Framework e um OOFM Process abstratos capazes de gerar SPLs avançadas de domínios específicos quando devidamente especializados e configurados. A Figura 2 ilustra o uso destas estruturas abstratas OOFM Framework e OOFM Process na produção de SPLs de domínio específico.

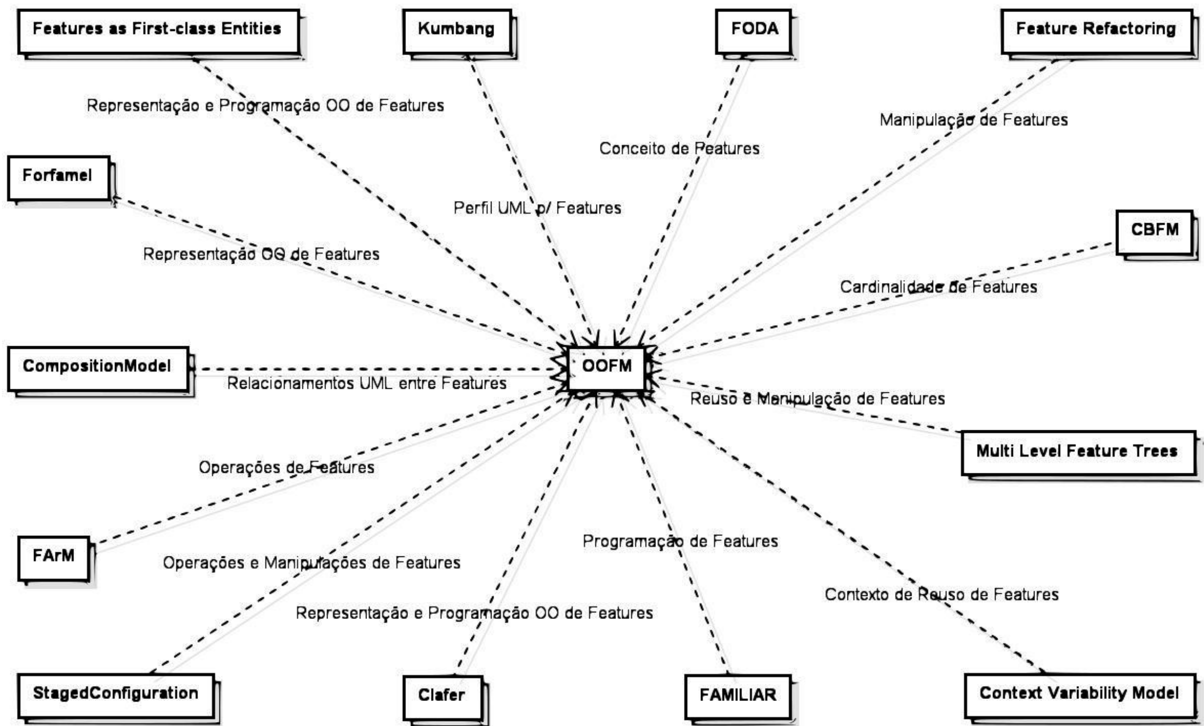


Figura 1 Idéias base de técnicas de modelagem de features usadas na OOFM.

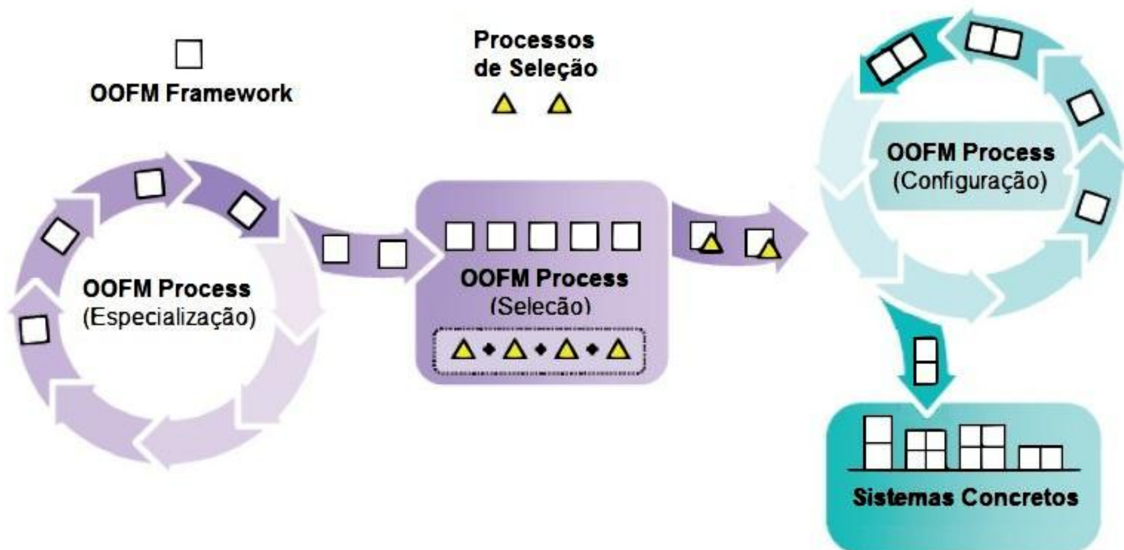


Figura 2 Produção de SPL de domínio específico com a OOFM.

Um exemplo de uso desta estratégia padronizada de produção de SPLs também será apresentado neste trabalho para o domínio dos jogos digitais. Trata-se, de uma especialização e configuração do OOFM Framework e do OOFM Process para trabalhar com recursos conceituais e de implementação de jogos digitais diversos. Ou seja, trata-se de uma validação importante da capacidade da OOFM em gerar SPLs e sistemas concretos para domínios de softwares específicos.

1.1 OBJETIVOS

O objetivo principal deste trabalho é mostrar a OOFM como uma nova solução de modelagem de features capaz de atender os aspectos de variabilidade identificados e de padronizar a produção de SPLs avançadas.

- Dentre os objetivos específicos deste trabalho, podemos destacar: Detalhamento dos conceitos-chave da técnica OOFM, baseados na integração de idéias estado-da-arte da modelagem de features com estruturas e regras consolidadas da OO;
- Formalização da técnica OOFM, buscando garantir a geração e manipulação de modelos OOFM bem-formados;
- Descrição do uso da OOFM na produção padronizada de SPLs avançadas e sistemas concretos para domínios de software específicos;
- Aplicação da técnica OOFM no desenvolvimento de um software de um domínio específico, confirmando assim a sua capacidade de gerar sistemas de software concretos baseados em features;
- Verificação e Validação da técnica OOFM, através da análise de questões voltadas a corretude, representatividade, complexidade e usabilidade de modelos OOFM gerados;
- Comparação dos aspectos de variabilidade da técnica OOFM com relação a demais técnicas de modelagem de features existentes; e

- Apresentação das possibilidades de análise e de projeto da variabilidade de sistemas com a técnica OOFM, tendo como base: a modelagem Unified Modeling Language (UML) (Booch, Rumbaugh e Jacobson, 1998) associada, scripts formais especificados, ferramenta de suporte desenvolvida e programação OO relacionada.

1.2 ORGANIZAÇÃO DA TESE

Esta tese está organizada conforme descrito a seguir. O Capítulo 2 apresenta o detalhamento dos conceitos chave da OOFM, a formalização aplicada na obtenção de um Perfil OOFM e as ferramentas de suporte desenvolvidas para representação e uso da OOFM na modelagem de features. O Capítulo 3 descreve a aplicação da OOFM na geração padronizada de SPLs avançadas e sistemas concretos para domínios de software específicos. O Capítulo 4 demostra o uso da OOFM na criação de um sistema de software concreto, sendo neste caso uma simplificação de um jogo digital conhecido. O Capítulo 5 discute importantes avaliações da OOFM com relação a técnicas de modelagem de features identificadas. Finalmente, o Capítulo 6 descreve algumas conclusões e trabalhos futuros relacionados à técnica OOFM proposta.

Capítulo

2

Este capítulo apresenta os princípios base da técnica OOFM. Para tal, a combinação de conceitos OO com conceitos de modelagem de features é apresentada e detalhada. A formalização dos aspectos estruturais e comportamentais da técnica OOFM também é descrita. E para finalizar, ferramentas de suporte desenvolvidas para representação e uso da OOFM também são apresentadas.

2 A TÉCNICA DE MODELAGEM DE FEATURES ORIENTADA A OBJETOS

Diferentes técnicas de modelagem de features propostas nos últimos anos. De fato, FODA (Kang et al., 1990) foi um esforço inicial para representar features de uma família de sistemas. Cardinality-Based Feature Modeling (CBFM) (Czarnecki, Helsen e Eisenecker, 2005) definiu cardinalidades e atributos para o projeto de FMs. Forfamel (Asikainen, Männistö e Soininen, 2006) apresentou um modelo de classes e uma linguagem para representar estruturas distintas de FMs e FCs. Multi-Level Feature Trees (Reiser e Weber, 2006) descreveu como fornecer e reusar FMs usando features referência e referenciadas. Composition Model (Rosenmüller e Siegmund, 2010) forneceu MPLs pela aplicação de relações OO entre FMs disponíveis. Entre outras.

No geral, tratam-se de técnicas que apresentam importantes aspectos de variabilidade capazes de serem gerenciados durante a produção de FMs e FCs, tais como: documentação, manipulação, encapsulamento, reuso, formalização, ferramentas de suporte, programação de features, produção de sistemas, etc. Elas também apresentam possíveis extensões de técnicas de modelagem de features disponíveis no intuito de integrar aspectos de variabilidade para features, tais como Composition Model especializando FODA e Multi-Level Feature Trees especializando CBFM.

OOFM é uma técnica que combina idéias propostas em abordagens distintas de modelagem de features de acordo com aspectos de variabilidade identificados. Ela fornece aspectos de documentação, manutenção, encapsulamento, reuso e programação de features de uma maneira integrada e parametrizada.

2.1 CONCEITOS CHAVE

OOFM surgiu a partir da idéia de combinar classes Forfamel com relacionamentos OO propostos pelo Composition Model para features. Tratam-se de técnicas importantes que apresentam abordagens distintas de aplicação de conceitos OO na representação de features. Enquanto Forfamel apresenta todo um conjunto de classes para representar FMs e FCs, Composition Model se preocupa apenas em indicar quais árvores de features encapsuladas em classes de SPLs podem ser associadas via relacionamentos OO. Assim, com a definição de uma abordagem capaz de integrar tais princípios de projeto, tem-se um novo conjunto de estruturas OO capaz de documentar, encapsular e reusar features.

FeatureClass é o elemento chave da técnica OOFM para realizar a combinação das técnicas Forfamel e Composition Model (Sarinho e Apolinário, 2010). Basicamente, ele trabalha como um contêiner de features modeladas, sendo estas representadas numa estrutura simplificada de classes Forfamel. Ele também trabalha como um integrador de FMs e FCs distintos, através da aplicação de relacionamentos OO entre *FeatureClasses* modeladas (contribuição do Composition Model).

Operações primitivas em features, tais como *add*, *exclude*, *merge*, entre outras, também podem ser aplicadas em instâncias *FeatureClass*, no intuito de cobrir o aspecto manutenção de features (Sarinho e Apolinário, 2010). Para tal, operações de features identificadas em técnicas de modelagem de features, a exemplo de Staged Configuration (Czarnecki, Helsen e Eisenecker, 2004), Multi-Level Feature Trees, FArM (Sochos, Riebisch e Philippow, 2006) e Feature Refactoring (Alves et al., 2006), foram sumarizadas numa Application Program Interface (API) comum para árvores de features. Esta API é usada por

métodos declarados na *FeatureClass* para executar manutenções desejadas em objetos *FeatureModel* e *FeatureConfiguration* instanciados (o aspecto de programação de features), de acordo com as intenções do projetista de features.

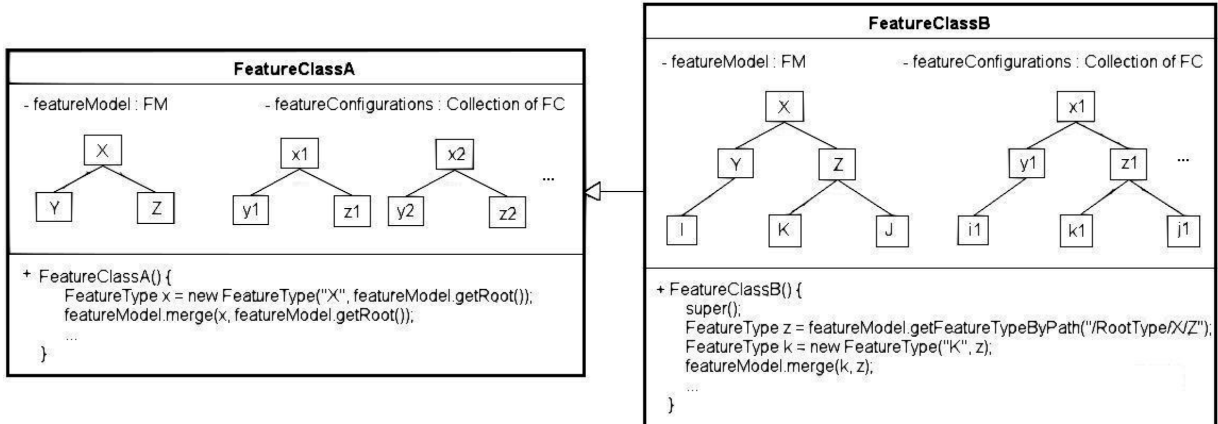


Figura 3 Exemplo de uso dos conceitos chave da OOFM.

A Figura 3 ilustra um exemplo de uso destes conceitos chave da OOFM, destacando FMs e FCs modelados numa hierarquia de *FeatureClasses* A e B, e operações de features capazes de manipular tais FMs e FCs encapsulados via programação de features.

2.2 DETALHANDO A OOFM

Numa visão mais detalhada da OOFM, cada *FeatureClass* é capaz de representar e manipular um *FeatureModel* e suas diversas instâncias *FeatureConfiguration* associadas. *FeatureModel*s e *FeatureConfigurations* modelam árvores de features diversas, representadas por instâncias *FeatureType* e *Feature* associadas. Operações de manipulação de features também estão associadas a instâncias *FeatureModel* e *FeatureConfiguration*, capazes de transformar valores e associações das *FeatureTypes* e *Features* definidas.

Esta subseção descreve com mais detalhes tais recursos de modelagem de features disponíveis. Para tal, a Figura 4 ilustra o metamodelo da OOFM (Sarinho e Apolinário, 2012),

contendo as estruturas necessárias para a representação de árvores de features, tais como: *FeatureType*, *Feature*, *GroupType*, *Attribute*, *Constraint*, entre outras.

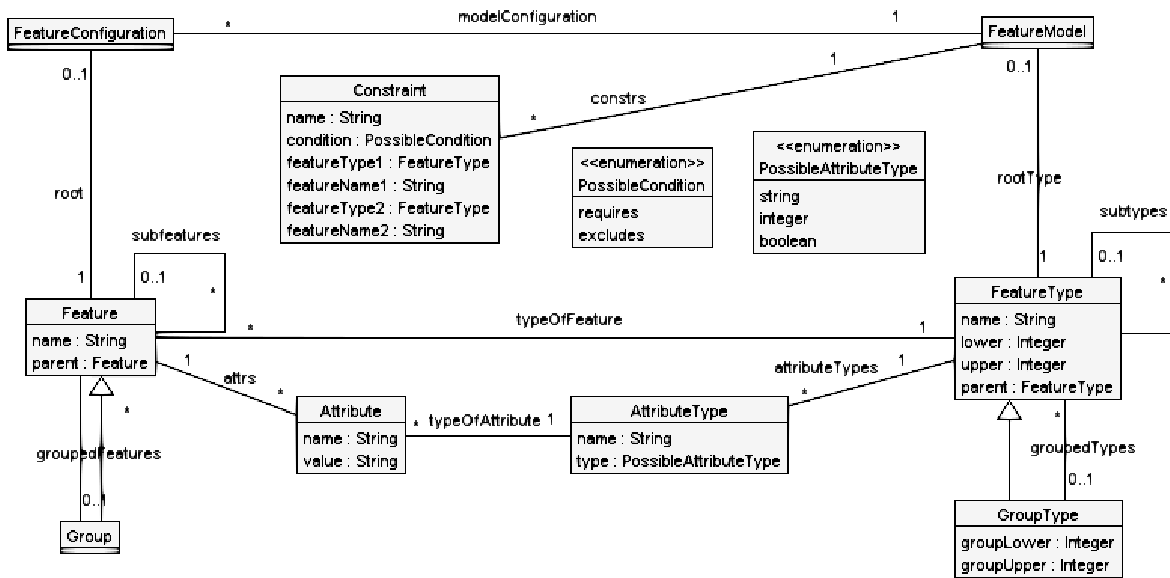


Figura 4 O metamodelo OOFM.

Dentre estas estruturas, *FeatureType* (Figura 4) é uma classe que descreve uma feature num FM. Ela contém os atributos definidos para representar informações de features, tais como: o *name* (identificação ou valor da feature); a cardinalidade (*upper* e *lower*); o *parent* (*FeatureType* imediatamente acima na árvore de features); os *subtypes* (coleção de objetos *FeatureType* imediatamente abaixo); e os *attributeTypes* (um *FeatureType* pode ter vários *AttributeTypes* associados).

Para cada FM modelado, features podem ser organizadas num *grupo de features* (*group)* (Kang et al., 1990). Um *grupo* é uma feature especializada cujas subfeatures são restringidas pelas informações definidas no *grupo* (Kang et al., 1990). Neste sentido, como features podem ser representadas por instâncias *FeatureType*, *GroupType* (Figura 4) é uma *FeatureType* especializada que representa cada *grupo* criado num FM.

Num FM, cada grupo precisa definir quais subfeatures serão agrupadas e de que forma (e.g. opcional ou mandatória e intervalo de cardinalidades). Assim, cada *GroupType* define quais instâncias *FeatureType* disponíveis na coleção *subtypes* são agrupadas como *groupedTypes*, bem como quais cardinalidades (*groupUpper* e *groupLower*) serão seguidas por cada grupo de instâncias *FeatureType* criado. Segundo Czarnecki, Helsen e Eisenecker

(2005), cada feature pode conter atributos que complementam informações num FM. No metamodelo OOFM, *FeatureType* também pode definir os possíveis *atributos de uma feature*, os quais são representados pelos atributos *name* e *type* de instâncias *AttributeType* associadas (Figura 4). O atributo *name* representa ou o tipo do atributo (configuração indireta) ou o próprio valor do atributo (configuração direta). Já o atributo *type* representa os possíveis tipos de dados que podem ser armazenados como um valor de um atributo (e.g. *string*, *integer*, *boolean*).

FeatureModel (Figura 4) é a estrutura responsável pela definição de FMs desejados. Trata-se de uma classe que organiza a estrutura do FM pela definição do elemento *rootType* (nó raiz do FM), o qual representa a primeira instância *FeatureType* disponível num FM. *FeatureModel* também restringe possíveis FCs derivadas através de objetos *Constraint* (Figura 4) instanciados. Cada objeto *Constraint* indica quais possíveis instâncias de *Feature* podem existir numa *FeatureConfiguration*, a depender dos objetos *FeatureType* associados via relacionamentos *requires* e *excludes* definidos (Kang et al., 1990).

Segundo Asikainen, Männistö e Soininen (2006, 2007), FCs também podem ser obtidas pela criação de novas e equivalentes instâncias de features de FMs, caracterizando assim uma "configuração indireta" de features. No metamodelo OOFM, features numa FC indireta são representadas por objetos da classe *Feature* (Figura 4), os quais representam valores instanciados de um *FeatureType*. Assim, para cada objeto *Feature* instanciado, são representados: os atributos *name* e *parent* (valor da instância e feature pai associada); coleções de subfeatures (*subfeatures*); coleções de atributos de features (*attrs*); e um relacionamento *typeOfFeature* indicando o *FeatureType* associado da respectiva *Feature* indireta.

Seguindo a perspectiva de associação entre FMs e FCs, *Group* (Figura 4) é uma especialização da classe *Feature* que mostra quais *subfeatures* são agrupadas como *groupedFeatures*. Objetos *Feature* agrupados devem respeitar a cardinalidade de *GroupType*, bem como devem estar associados a instâncias *FeatureType* capazes de serem agrupadas (*groupedTypes*).

Continuando com a perspectiva de associação entre FMs e FCs, para a classe *Attribute* (Figura 4), esta descreve quais informações podem ser representadas num atributo

de feature de uma FC. Todas as instâncias *Attribute* possuem um *name*, um *value* corrente e uma referência a um *AttributeType* associado (via relacionamento *typeOfAttribute*).

E para finalizar, a classe *FeatureConfiguration* (Figura 4) representa uma configuração indireta de um *FeatureModel* associado. Cada *FeatureConfiguration* tem basicamente: um *root*, descrevendo a *Feature* inicial de uma FC; e um *modelConfiguration*, indicando o seu *FeatureModel* associado. Cada estrutura referência criada num *FeatureModel* (e.g. *Constraint*, *FeatureType*, *GroupType* e *AttributeType)* deve ser seguida pelos respectivos elementos associados criados na *FeatureConfiguration* associada (e.g. *Feature*, *Group* e *Attribute*).

Com a definição das estruturas OOFM capazes de representar FMs e FCs diversos, o passo seguinte consiste em definir como as coleções de *FeatureTypes* e *Features modeladas* (e.g. *subtypes*, *subfeatures*, *groupedTypes* e *groupedFeatures*) podem ser manipuladas. Neste sentido, diversas operações de features têm sido propostas nos últimos anos. Tratam-se de operações que geram novos FMs a partir de coleções de features pré-existentes, sendo em geral agrupadas em quatro categorias operacionais: *Seleção*, *Inclusão*, *Mapeamento* e *Atualização*.

De fato, analisando operações do tipo "Reduction" (Reiser e Weber, 2006), "Select", "Remove" (Czarnecki, Helsen e Eisenecker, 2004) e "Dividing" (Sochos, Riebisch e Philippow, 2006), percebe-se como objetivo principal destas operações a *Seleção* de subárvores de um FM. Tais operações aplicam uma simplificação do FM original com o objetivo de obter apenas as features desejadas de um modelo.

Com relação às operações de features "Refinement" (Reiser e Weber, 2006), "Clone" (Czarnecki, Helsen e Eisenecker, 2004), "Adding" (Sochos, Riebisch e Philippow, 2006) e "Add New Alternative" (Alves et al., 2006), estas fornecem FMs especializados através da *Inclusão* de novas features ao modelo.

Algumas operações de *Mapeamento* de features foram propostas no sentido de refinar e reorganizar associações entre features, tais como: "Integrating", "Reordering" (Sochos, Riebisch e Philippow, 2006), "Move", "Regroup" (Reiser e Weber, 2006) e "Merge Optional and Alternative" (Alves et al., 2006).

Finalmente, transformações de *Atualização* de features, capazes de mudar o valor corrente de features, podem ser exemplificadas pela operação "Assign" (Czarnecki, Helsen e Eisenecker, 2004) e pelas *deviations* "*change ...*" disponíveis (Reiser e Weber, 2006).

Tendo como base estes grupos de manipulação de features, algumas operações de features são propostas para a OOFM. Assim, para o grupo *Seleção*, têm-se as seguintes operações de features: *exclude* (*FeatureType* e *Feature*), *removeAttributeType*, *removeAttribute*, *removeConstraint*, *getFeatureTypeByPath*, *getFeatureByPath* e *getAttribute*. Tratam-se de operações que permitem a identificação e a exclusão de instâncias OOFM desejadas.

Para o grupo *Inclusão*, tem-se as operações: *add* (*FeatureType* e *Feature*), *copy* (*FeatureType* e *Feature*), *clone* (*FeatureType* e *Feature*), *addAttributeType*, *addAttribute* e *addConstraint*. Tratam-se de operações capazes de adicionar novas instâncias OOFM com seus respectivos valores em FMs e FCs disponíveis.

O grupo *Mapeamento* foca na mudança/reorganização de estruturas OOFM representativas de features. Ele é composto pelas seguintes operações: *move* (*FeatureType* e *Feature*), *merge* (*FeatureType* e *Feature*), *setToGroupType*, *setToGroup*, *setToFeature* e *setToFeatureType*.

E para finalizar, operações do grupo *Atualização* são aplicadas em instâncias *Feature* e *Attribute* pela execução de operações *update* e *setAttribute* disponíveis.

FeatureConfiguration
add(featureRef : Feature, newName : String, newType : FeatureType) : Feature exclude(featureRef : Feature) merge(feature : Feature, featureRef : Feature) clone(featureRef : Feature) : Feature copy(fromFeature : Feature, toFeature : Feature) : Feature move(fromFeature : Feature, toFeature : Feature) addAttribute(featureRef : Feature, newName : String, newType : AttributeType, newValue : String) : Attribute removeAttribute(featureRef : Feature, attribute : Attribute) getAttribute(featureRef : Feature, attributeName : String, attributeType : AttributeType) : String setAttribute(featureRef : Feature, attributeName : String, attributeType : AttributeType, value : String) setToGroup(featureRef : Feature, newType : GroupType, newGroupedFeatures : Set(Feature)) : Group setToFeature(groupRef : Group) : Feature

Figura 5 Operações de features da classe *FeatureConfiguration*.

A Figura 5 ilustra algumas destas operações de features OOFM, juntamente com seus respectivos parâmetros. Tratam-se de operações especificadas para a classe

FeatureConfiguration, focadas na manipulação de objetos *Feature*, *Group* e *Attribute* desejados.

2.2.1 Restrições de Modelagem

Para garantir a geração de modelos OOFM bem formados, algumas restrições OOFM foram definidas no intuito de evitar a geração de árvores de features inválidas.

Desta forma, considerando as restrições definidas para cada estrutura OOFM, tem-se:

- O valor de *name* em *FeatureType*, *AttributeType* e *Constraint* devem ser únicos nas respectivas coleções *subtypes*, *attributeTypes* e *constrs*;
- A combinação de valores *name* e *typeOfFeature* em instâncias *Feature* deve ser única na respectiva coleção *subfeatures*;
- A combinação de valores *name* e *typeOfAttribute* em instâncias *Attribute* deve ser única na respectiva coleção *attrs*;
- Cardinalidades *FeatureType* devem ser seguidas por *Features* criadas a partir do respectivo tipo;
- *GroupTypes* devem ter duas ou mais instâncias distintas de *subtypes* como *groupedTypes*;
- *Features* de um *Group* só podem ser agrupadas (via *groupedFeatures*) se seus *FeatureTypes* estiverem agrupados no *GroupType* do respectivo *Group* (via *groupedTypes*);
- O número de *Features* associadas em *groupedFeatures* de um *Group* é limitada pela cardinalidade definida pelo *GroupType* associado;
- Todos os valores *parent* em instâncias *Feature* e *FeatureType* devem ser definidos;
- *rootType*, *root* e features raiz iniciais de árvores de features clonadas têm o valor de *parent* referenciando a si próprio;

- O valor *parent* das instâncias de features em subfeatures deve ser uma referência à própria feature contêiner das respectivas subfeatures;
- O *typeOfFeature* de todos os valores *parent* de instâncias *Feature* devem ser iguais ao valor *parent* de seus respectivos valores *typeOfFeature*;
- Referências circulares entre features e subfeatures numa árvore de features devem ser evitadas;
- Todas as instâncias *FeatureType*, *AttributeType* e *Constraint* usadas por uma instância *FeatureConfiguration* devem estar disponíveis no *FeatureModel* associado;
- Regras definidas em *Constraints* devem ser seguidas por instâncias *FeatureConfiguration* associadas;
- *Constraints* definidas referenciando instâncias *FeatureType* e *Feature* similares são inválidas;
- Toda instância OOFM referenciada por demais instâncias OOFM deve estar disponíveis durante seu uso;
- A estrutura *FeatureModel* de instâncias *FeatureClass* devem apresentar uma estrutura *FeatureModel* compatível com a da sua respectiva *FeatureClass* pai (*super*);
- Para cada *FeatureClass*, todas as suas *FeatureConfigurations* devem ter o mesmo (e corrente) *FeatureModel* como referência.

Da mesma forma, considerando as operações de features OOFM previamente descritas, tem-se as seguintes restrições de modelagem:

- Todos os parâmetros iniciais de uma operação de features devem ser válidos;
- Operações *exclude*, *removeAttributeType*, *removeAttribute* e *removeConstraint* devem destruir o respectivo elemento OOFM indicado. Este também deve ser removido da respectiva coleção de elementos OOFM associado (*subtypes*, *subfeatures*, *attributeTypes*, *attrs* ou *constrs*);
- Operações *add* devem criar novas instâncias de elementos OOFM de acordo com os valores fornecidos em parâmetros. Cada nova instância deve ser

incluída, na respectiva coleção de elementos OOFM associada (*subtypes*, *subfeatures*, *attributeTypes*, *attrs* ou *constrs*);

- Operações *merge* devem incluir uma feature referência (parâmetro inicial) como uma subfeature da feature destino indicada (parâmetro subsequente);
- Operações *clone* devem criar e retornar uma árvore de features similar à árvore de features fornecida por uma feature referência (parâmetro);
- Operações *copy* devem criar uma árvore de features similar à árvore de features fornecida por uma feature referência (parâmetro inicial). Esta nova árvore será incluída na coleção de features (*subtypes* ou *subfeatures*) da feature alvo (parâmetro subsequente);
- Operações *move* devem incluir na coleção de features (*subtypes* ou *subfeatures*) da feature alvo a feature referência indicada, a qual será devidamente removida da sua feature original (*parent*);
- Operação *update* deve mudar o valor *name* da instância *Feature* indicada;
- Operações *setToGroupType* e *setToGroup* devem mudar instâncias *FeatureType* e *Feature* indicadas para grupos de instâncias equivalentes;
- Operações *setToFeatureType* e *setToFeature* devem mudar instâncias *GroupType* e *Group* indicadas para instâncias de features equivalentes;
- Operações *getFeatureTypeByPath* e *getFeatureByPath* devem retornar as respectivas instâncias *FeatureType* e *Feature* indicadas pelo *path* na árvore de features;
- Operações *getAttribute* e *setAttribute* devem retornar e atualizar instâncias *Attribute* de acordo com parâmetros especificados.

2.3 ESPECIFICAÇÃO FORMAL

Algumas estratégias de formalização de features foram propostas na literatura nos últimos anos (Alves et al., 2006; Schobbens, Heymans e Trigaux, 2006) com o objetivo de

compensar a ambiguidade, falta de precisão e expressividade representada pelas abordagens de modelagem de features da época. No geral, tratam-se de formalizações focadas na definição de notações algébricas e gramáticas correspondentes para a representação de FMs. Como resultado, elas evitam interpretações errôneas e confusas fornecidas por definições informais de FMs, tais como features duplicadas e organizações inválidas de árvores de features.

Uma especificação formal também foi definida para a técnica OOFM (Sarinho e Apolinário, 2012). Ela descreve expressões Object Constraint Language (OCL) (OMG, 2000) para as estruturas e restrições OOFM definidas. OCL é capaz de descrever restrições formais em linguagens OO e outros artefatos de modelagem de software, evitando assim modelos e construções indesejadas de sistemas via restrições declaradas (OMG, 2000).

2.3.1 Expressões OCL para Estruturas OOFM

Como restrições textuais foram descritas para indicar modelos OOFM bem formados, esta subseção descreve importantes expressões OCL equivalentes capazes de definir restrições formais para as estruturas OOFM propostas. Trata-se de uma representação direta de tais restrições textuais em objetos, coleções e operações OCL, sendo estes organizados em invariantes, pré-condições e pós-condições OCL necessárias (OMG, 2000).

Neste sentido, invariantes OCL são especificadas para possíveis instâncias *FeatureType* e *AttributeType* (Figura 6), as quais definem:

- Valores distintos de *name* para cada instância disponível em *subtypes*, evitando assim valores duplicados de *FeatureType name* nesta coleção (linhas 2 e 3);
- Um *parent* especificado para todas as instâncias *FeatureType*, garantindo assim *FeatureTypes* relacionados com *FeatureTypes* pré-existentes (linha 4);
- A ausência de referências circulares nas instâncias *FeatureType* criadas, via verificação de resultado da função *hasCircularReference* (linha 5);

- A similaridade entre uma instância *FeatureType* e o valor *parent* de seus respectivos *subtypes*, garantindo assim uma estrutura de árvore e um mesmo *rootType* final para todas as instâncias *FeatureType* numa árvore de features (linha 6);
- Valores *FeatureType.lower* especificados sempre menores ou iguais que os valores especificados em *FeatureType.upper* (linha 7);
- Valores *name* distintos para cada instância em *attributeTypes*, evitando assim valores duplicados de *AttributeType name* nesta coleção (linhas 8 e 9); e
- A existência de referências *AttributeType* válidas (não excluídas) para cada instância *attributeTypes* disponível (linhas 10-12).

```
1.   context FeatureType
2.   inv: FeatureType.allInstances() -> forAll(ft | ft.subtypes ->
3.              forAll(st | ft.subtypes -> select (name = st.name) -> size() < 2))
4.   inv: FeatureType.allInstances() -> forAll(ft | not ft.parent.name.isUndefined())
5.   inv: FeatureType.allInstances() -> forAll(ft | not ft.hasCircularReference(ft))
6.   inv: FeatureType.allInstances() -> forAll(ft | ft.subtypes -> forAll(st | st.parent = ft))
7.   inv: FeatureType.allInstances() -> forAll(ft | ft.lower <= ft.upper)
8.   inv: FeatureType.allInstances() -> forAll(ft | ft.attributeTypes ->
9.              forAll(at | ft.attributeTypes -> select (name = at.name) -> size() < 2))
10.  inv: FeatureType.allInstances() -> forAll(ft | if (ft.attributeTypes -> size() > 0)
11.             then ft.attributeTypes -> forAll(ftat | AttributeType.allInstances() -> exists(at | ftat = at))
12.             else true endif)
```

Figura 6 Expressões OCL para *FeatureType e AttributeType*.

Já para instâncias *GroupType*, tem-se as seguintes invariantes (Figura 7):

- Para representar um grupo de features, pelo menos duas ou mais referências *FeatureType* disponíveis devem ser agrupadas em coleções *groupedTypes* (linha 2);
- Para todas as instâncias *FeatureType* alocadas em *groupedTypes*, apenas *subtypes* correntes podem ser agrupados (linha 3);
- Apenas instâncias *subtypes* distintas podem ser usadas em coleções *groupedTypes* (linha 4). Ou seja, não pode haver *FeatureTypes* duplicados em *groupedTypes*; e
- Valores *GroupType.groupLower* especificados devem ser sempre menores ou iguais aos valores especificados em *GroupType.groupUpper* (linha 5).

```
1.   context GroupType
2.   inv: GroupType.allInstances() -> forAll(gt | (gt.groupedTypes -> size() >= 2) and
3.        (gt.groupedTypes -> forAll(gts | gt.subtypes -> exists(st | st.name = gts.name))) and
4.        (gt.groupedTypes -> forAll(gts | gt.groupedTypes-> select (name = gts.name) -> size() < 2)))
5.   inv: GroupType.allInstances() -> forAll(gt | gt.groupLower <= gt.groupUpper)
```

Figura 7 Expressões OCL para *GroupType*.

Considerando a representação de invariantes OCL para elementos representativos de FCs, tem-se para instâncias *Feature* (Figura 8):

- Valores *name* e *typeOfFeature* distintos para cada instância em *subfeatures*, evitando assim instâncias *Feature* duplicadas nesta coleção (linhas 2 e 3);
- Um *parent* válido para instâncias *Feature*, garantindo assim *Features* relacionadas com *Features* pré-existentes (linha 4);
- Ausência de referências circulares nas instâncias *Feature* criadas, via verificação de resultado da função *hasCircularReference* (linha 5);
- Similaridade de referências *FeatureType* entre *parent.typeOfFeature* e *typeOfFeature.parent*, garantindo assim a equivalência de *FeatureTypes* e *Features* nos seus respectivos FM e FC (linha 6);
- A similaridade entre uma instância *Feature* e o valor *parent* de suas respectivas *subfeatures*, garantindo assim uma estrutura de árvore e um mesmo *root* final para todas as instâncias *Feature* numa árvore de features (linha 7); e
- O número de instâncias *Feature* com *FeatureType* similares num ramo corrente de features devem seguir os valores de cardinalidade *lower* e *upper* indicados pelo respectivo *FeatureType* (linhas 8-10).

Considerando as invariantes OCL para instâncias *Attribute* (Figura 8), estas definem que:

- Instâncias *Attribute* alocadas em coleções *Feature.attrs* possuem valores *name* e *typeOfAttribute* distintos, evitando assim instâncias *Attribute* duplicadas nesta coleção (linhas 12 e 13);

- Referências *attrs* possuem instâncias *Attribute* válidas (não excluídas) para uso (linhas 14-16); e
- Instâncias *Attribute* alocadas em *attrs* trabalham com instâncias *AttributeType* válidas (não excluídas) em *typeOfAttribute* (linhas 17-19).

```
1.  context Feature
2.  inv: Feature.allInstances() -> forAll(f | f.subfeatures -> forAll(sf | f.subfeatures ->
3.          select (name = sf.name and typeOfFeature.name = sf.typeOfFeature.name) -> size() < 2))
4.  inv: Feature.allInstances() -> forAll(f | not f.parent.name.isUndefined())
5.  inv: Feature.allInstances() -> forAll(f | not f.hasCircularReference(f))
6.  inv: Feature.allInstances() -> forAll(f | f.parent.typeOfFeature = f.typeOfFeature.parent)
7.  inv: Feature.allInstances() -> forAll(f | f.subfeatures -> forAll(sf | sf.parent = f))
8.  inv: Feature.allInstances() -> reject(f | f.parent = f) -> forAll(f |
9.          f.parent.subfeatures -> select(typeOfFeature = f.typeOfFeature) -> size() >= f.typeOfFeature.lower and
10.         f.parent.subfeatures -> select(typeOfFeature = f.typeOfFeature) -> size() <= f.typeOfFeature.upper)
11.
12. inv: Feature.allInstances() -> forAll(f | f.attrs -> forAll(at | f.attrs ->
13.         select (name = at.name and typeOfAttribute.name = at.typeOfAttribute.name) -> size() < 2))
14. inv: Feature.allInstances() -> forAll(f | if (f.attrs -> size() > 0)
15.         then f.attrs -> forAll(fa | Attribute.allInstances() -> exists(a | fa = a))
16.         else true endif)
17. inv:Feature.allInstances() -> forAll(f | if (f.attrs -> size() > 0)
18.         then f.attrs -> forAll(fa | f.typeOfFeature.attributeTypes -> exists(at | fa.typeOfAttribute = at))
19.         else true endif)
```

Figura 8 Expressões OCL para *Feature* e *Attribute*.

Finalmente, para instâncias *Group*, as invariantes OCL declaradas (Figura 9) determinam que:

- Cada *Feature* disponíveis numa *subfeatures* só pode ser agrupada uma vez no respectivo *groupedFeatures* do *Group* (linhas 2-4);
- Para uma *Feature* ser agrupada em *groupedFeatures*, e necessário que seu *FeatureType* associado também tenha sido agrupado em *groupedTypes* (linhas 5-7); e
- O tamanho de um *groupedFeatures* deve seguir a cardinalidade indicada no respectivo *GroupType* (linhas 8-10).

```
1.   context Group
2.   inv: Group.allInstances()-> forAll(g | g.groupedFeatures ->
3.             forAll(gf | g.subfeatures -> select(name = gf.name and
4.                                     typeOfFeature.name = gf.typeOfFeature.name) -> size() = 1))
5.   inv: Group.allInstances()-> forAll(g | g.groupedFeatures -> forAll(gf |
6.             g.typeOfFeature.oclAsType(GroupType).groupedTypes ->
7.                       exists(gt | gt.name = gf.typeOfFeature.name)))
8.   inv: Group.allInstances()->
9.             forAll(g | g.groupedFeatures -> size() >= g.typeOfFeature.oclAsType(GroupType).groupLower and
10.                      g.groupedFeatures -> size() <= g.typeOfFeature.oclAsType(GroupType).groupUpper)
```

Figura 9 Expressões OCL para *Group*.

2.3.2 Expressões OCL para Operações OOFM

Seguindo as restrições textuais propostas para manipulação de features OOFM, esta subseção apresenta as principais expressões OCL especificadas para operações de features declaradas nas classes *FeatureModel* (Figura 10). Tratam-se de pré e pós-condições OCL que restringem valores fornecidos como parâmetro e indicam resultados esperados para cada operação de features efetuada.

Desta forma, para a classe *FeatureModel*, tem-se que pré e pós condições são definidas para as operações *add*, *exclude*, *merge*, *clone*, *copy*, *move*, *addAttributeType*, *removeAttributeType*, *setToGroupType*, *setToFeatureType*, *addConstraint* e *removeConstraint* (sendo algumas ilustradas na Figura 10). No caso da operação *getFeatureTypeByPath*, esta é representada como uma função OCL, tornando-se assim desnecessário definir pré e pós-condições para a mesma.

Considerando a operação *add* (linhas 1 e 2), tem-se pós-condições OCL que definem a criação de uma instância *FeatureType* como resultado (linhas 4-5). Esta nova instância é configurada com os valores fornecidos por parâmetros e é incluída na coleção *subtypes* do parâmetro *featureTypeRef* (linha 6). O parâmetro *featureTypeRef* indica o destino final da nova instância *FeatureType* resultante.

Para a operação *exclude* (linha 8), pós-condições OCL definem que a referência *FeatureType* indicada no parâmetro *featureTypeRef* deve ser destruída (linha) e removida da

coleção *subtypes* de *featureTypeRef.parent* (linhas 10 e 11). A garantia de que *featureTypeRef* foi destruída também é especificada (linha 12).

No mesmo contexto, para a operação *merge* (linha 14), pós-condições OCL definem que esta operação deve incluir uma referência *FeatureType* (indicada pelo parâmetro *featureType*) na coleção *featureTypeRef.subtypes* (linha 16). O valor de *featureType.parent* também deve ser configurado com o valor do parâmetro *featureTypeRef* (linha 17), consolidando assim o novo parentesco de *featureType*.

Para a operação *clone* (linha 19), de acordo com as suas pós-condições, é necessário criar uma nova instância *FeatureType* resultado capaz de apresentar uma estrutura de árvore de features similar a de *featureTypeRef*. Esta verificação de similaridade é confirmada ou não pela função *areSiblings,* a qual compara recursivamente as estruturas da árvores de features indicadas (linha 21).

```
1.  context FeatureModel::add(featureTypeRef: FeatureType, newName: String,
2.                      newLower: Integer, newUpper: Integer): FeatureType
3.  pre: not featureTypeRef.isUndefined()
4.  post: result.name = newName and result.lower = newLower and result.upper = newUpper and
5.          result.parent = featureTypeRef and result.oclIsNew()
6.  post: featureTypeRef.subtypes = featureTypeRef.subtypes@pre -> including(result)
7.
8.  context FeatureModel::exclude(featureTypeRef: FeatureType)
9.  pre: featureTypeRef <> self.rootType and not featureTypeRef.isUndefined()
10. post: featureTypeRef@pre.parent@pre.subtypes = featureTypeRef@pre.parent@pre.subtypes@pre ->
11.         reject (st | st.name = featureTypeRef@pre.name)
12. post: featureTypeRef.name.isUndefined()
13.
14. context FeatureModel::merge(featureType: FeatureType, featureTypeRef: FeatureType)
15. pre: featureType <> self.rootType and not featureTypeRef.isUndefined()
16. post: featureTypeRef.subtypes = featureTypeRef.subtypes@pre -> including(featureType)
17. post: featureType.parent = featureTypeRef
18.
19. context FeatureModel::clone(featureTypeRef: FeatureType): FeatureType
20. pre: not featureTypeRef.isUndefined()
21. post: areSiblings(featureTypeRef, result, result, false) and result.oclIsNew()
22.
23. context FeatureModel::copy(fromFeatureType: FeatureType, toFeatureType: FeatureType): FeatureType
24. pre: not fromFeatureType.isUndefined() and not toFeatureType.isUndefined()
25. post: areSiblings(fromFeatureType, result, result, false) and result.oclIsNew()
26. post: toFeatureType.subtypes = toFeatureType.subtypes@pre -> including(result)
27. post: result.parent = toFeatureType
28.
29. context FeatureModel::move(fromFeatureType: FeatureType,toFeatureType: FeatureType)
30. pre: fromFeatureType <> self.rootType and not fromFeatureType.isUndefined() and not
    toFeatureType.isUndefined()
31. post: fromFeatureType.parent = toFeatureType
32. post: toFeatureType.subtypes = toFeatureType.subtypes@pre -> including(fromFeatureType)
33. post: fromFeatureType@pre.parent@pre.subtypes = fromFeatureType@pre.parent@pre.subtypes@pre ->
34.         reject (st | st.name = fromFeatureType@pre.name)
```

Figura 10 Pré e pós condições de algumas operações de *FeatureModel*.

Com uma estratégia similar, a operação *copy* (linha 23) apresenta pós-condições que devem criar uma nova instância *FeatureType* resultado, bem como verificar se ela apresenta uma estrutura de árvore de features similar a de *fromFeatureType* (linha 25). Esta nova instância *FeatureType* resultado deve ser incluída na coleção *toFeatureType.subtypes* (linha 26), e seu *parent* deve ser igual a referência *toFeatureType* (linha 27). Assim, a inclusão da cópia de *fromFeatureType* na coleção *toFeatureType.subtypes* é efetuada, e a operação *copy* de features é caracterizada.

Finalmente, para as pós-condições declaradas na operação *move* (linha 29), estas indicam que se deve aplicar uma mudança de referência nos atributos *parent* e nas coleção *subtypes* das instâncias *FeatureType* alvo e destino (*fromFeatureType* e *toFeatureType*, respectivamente). A idéia é: configurar *fromFeatureType.parent* para ser igual a *toFeatureParent* (linha 31); remover *fromFeatureType* da coleção *fromFeatureType.parent.subtypes* (linhas 33 e 34); e incluir *fromFeatureType* na coleção *subtypes* do parâmetro *toFeatureType* (linha 32). Assim, um *move* de *fromFeatureType* para *toFeatureType.subtypes* é efetuado.

Para algumas destas operações de *FeatureModel* acima descritas, pré-condições são especificadas para restringir o uso de *rootType* como *featureTypeRef* (linhas 9, 15 e 30), evitando assim manipulações indevidas de features na estrutura *FeatureModel* proposta. Da mesma forma, para todas as operações de features em *FeatureModel*, tem-se que o uso de instâncias *FeatureType* indefinidas deve ser evitado (pré-condições nas linhas 3, 9, 15, 20, 24 e 30), garantindo assim a correta avaliação das pós-condições especificadas.

2.3.3 Um Perfil OOFM

Buscando complementar o significado semântico da técnica OOFM, esta subseção mostra a incorporação dos elementos estruturais e comportamentais da OOFM numa perspectiva OO disponível e consolidada. A idéia é reutilizar conceitos sintáticos e semânticos

de orientação a objetos pré-existentes, estendendo-os para dar suporte aos conceitos de features modelados e formalizados na OOFM.

Neste sentido, um Perfil OOFM foi definido (Figura 11), capaz de combinar estruturas e operações representativas de features com metaclasses UML disponíveis na superestrutura UML (Sarinho e Apolinário, 2012). Ele é baseado no mecanismo de Perfil da UML (OMG, 2009), uma abordagem formal lightweight de extensão da UML (Booch, Rumbaugh e Jacobson, 1998).

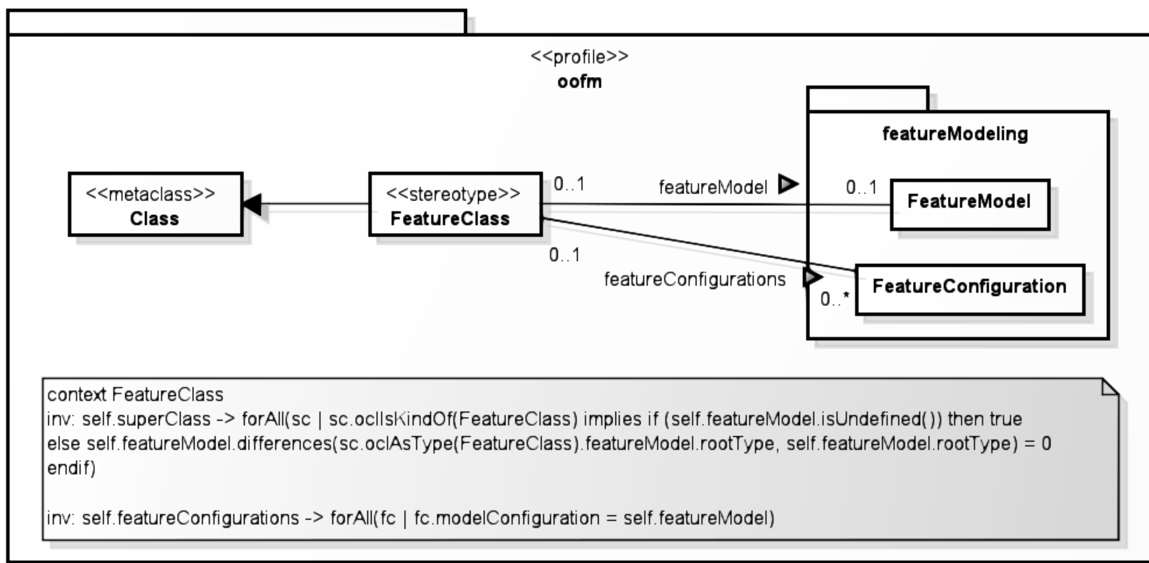


Figura 11 Perfil OOFM.

Através do Perfil OOFM, regras OO (e.g. associação, herança, visualização) se tornam disponíveis para cada modelo OOFM projetado. Ou seja, uma mesclagem necessária para consolidar o esquema de herança entre instâncias *FeatureClass*, onde:

- O estereótipo *<<feature-class>>* garante para cada *FeatureClass* declarada a semântica da metaclasse *Class*; e
- Uma invariante OCL garante para todas instâncias *FeatureClass* que nenhuma diferença entre FMs de *super* e *sub FeatureClasses* será encontrada.

Como resultado, o Perfil OOFM fornece uma sintaxe e semântica OO integrada aos recursos OOFM propostos para a modelagem de features (Sarinho e Apolinário, 2012). Ele também consolida uma abordagem formal EMOF+OCL (Garcia, 2008) para a OOFM, um

importante recurso para a produção de modelos OOFM bem formados dentro das estruturas e restrições OOFM definidas.

2.4 FERRAMENTAS DE SUPORTE

Uma importante missão da modelagem de features é facilitar a análise da variabilidade de sistemas (Kang et al., 1990). Trata-se de um objetivo alcançado com sucesso através das construções gráficas de árvores de features com seus respectivos relacionamentos entre features (Cechticky et al., 2004).

Representações gráficas de modelos OOFM podem ser efetuadas a partir de diagramas UML e respectivas ferramentas de apoio. Uma destas ferramentas é o OOFM Environment (Sarinho e Apolinário, 2012), uma coleção de especificações baseadas na OOFM modeladas por instruções de programação de features capazes de serem interpretadas via UML-based Specification Environment (USE) (Gogolla, Büttner e Richters, 2007).

Entretanto, OOFM Environment produz modelos OOFM, segundo o padrão gráfico da UML, através de rotinas modeladoras de features programadas por projetistas de features. Ou seja, é praticamente uma atividade de programação de features ao invés de uma atividade "gráfica" de modelagem de features.

A modelagem direta na UML de modelos OOFM também não é uma tarefa simples de ser efetuada, uma vez que existem várias restrições OCL impostas para cada modelo OOFM gerado que precisam ser respeitadas. Ou seja, existem grandes chances de se gerar modelos OOFM inválidos sem a utilização de um ferramental de apoio durante a modelagem.

Mais ainda, uma quantidade extra de entidades, atributos e relacionamentos de features é necessária na modelagem OOFM quando comparada com técnicas tradicionais de modelagem de features existentes. Um exemplo desta dificuldade é ilustrado na Figura 12, a qual mostra um modelo OOFM representativo do tradicional domínio de automóveis baseado em features (Kang et al., 1990; Deursen e Klint, 2002). Neste modelo é possível observar algo

em torno de 4 a 5 vezes mais elementos representativos de features quando comparado aos tradicionais modelos propostos por Kang et al. (1990) e Deursen e Klint (2002).

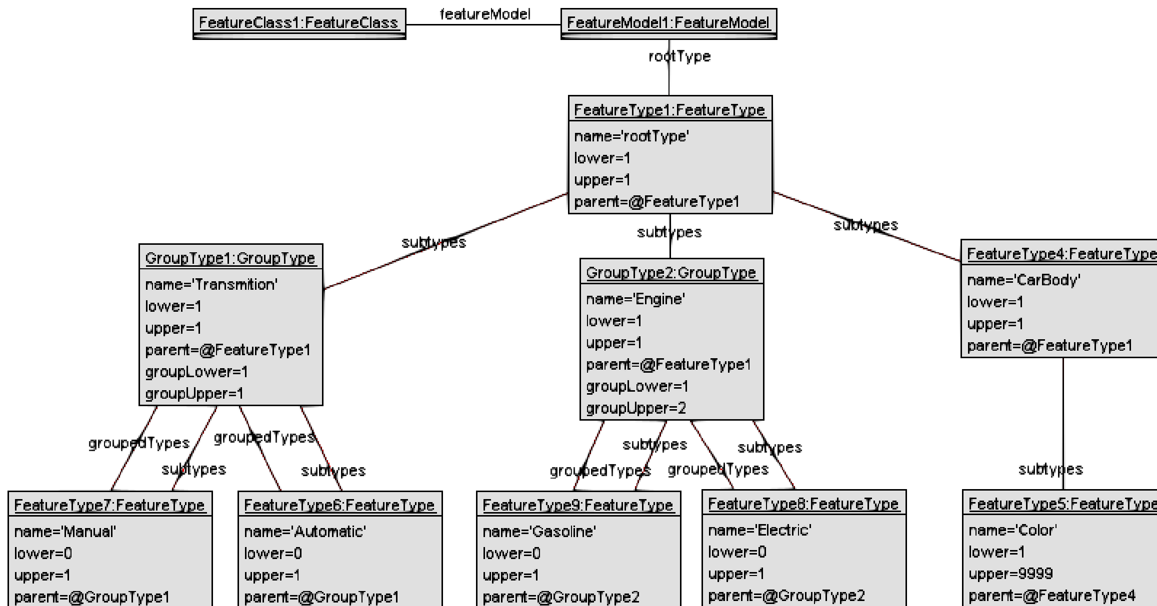


Figura 12 Modelo OOFM para o domínio de automóveis.

Pensando nestas "dificuldades UML" na geração de modelos OOFM, definiu-se uma Domain Specification Language (DSL) (Van Deursen, Klint e Visser, 2000) para representar árvores OOFM. Trata-se de um Extensible Markup Language (XML) denominado OOFM-XML que permite a representação de: elementos OOFM (e.g. *FeatureType*, *Feature*, *AttributeType*, etc.); cardinalidades de features (e.g. *lower*, *upper*, *groupLower*, *groupUpper*); identificação de features (e.g. *Feature name*, *FeatureConfiguration id*); e associações relevantes entre features (e.g. *featureType* de uma *Feature*, *isGrouped*, *isGroupedType*).

O objetivo deste OOFM-XML é evitar preocupações com restrições OCL e com rotinas de programação de features durante a produção de FMs e FCs baseados na OOFM. Ou seja, ele fornece os elementos e relacionamentos mínimos necessários para a produção de modelos OOFM bem-formados. A Figura 13 mostra um exemplo deste OOFM-XML, representando um FM e um FC simplificados do domínio de automóveis ilustrado na Figura 12.

Um *FeatureClass Editor* também foi desenvolvido para a técnica OOFM. Trata-se de uma ferramenta que fornece um suporte gráfico para modelos OOFM, bem como a geração e interpretação de arquivos OOFM-XML equivalentes. Seu funcionamento consiste em

executar operações OOFM via menus de interface, permitindo a montagem completa de árvores OOFM dentro das restrições impostas pela técnica. A verificação de compatibilidade entre o FM e seus FCs modelados também é efetuada pela ferramenta via comando *verify* no menu de interface, garantindo assim a produção de FCs válidos nas *FeatureClasses* modeladas.

```
<FeatureClass>
   <FeatureModel>
      <FeatureType name='RootType' lower='1' upper='1'>
         <GroupType name='Transmition' lower='1' upper='1' groupLower='1' groupUpper='1'>
            <AttributeType name='Code' type='string'/>
            <FeatureType name='Automatic' lower='0' upper='1' isGroupedType='true'></FeatureType>
            <FeatureType name='Manual' lower='0' upper='1' isGroupedType='true'></FeatureType>
         </GroupType>
         <GroupType name='Engine' lower='1' upper='2' groupLower='1' groupUpper='2'>
            <FeatureType name='Gasoline' lower='0' upper='1' isGroupedType='true'></FeatureType>
            <FeatureType name='Electric' lower='0' upper='1' isGroupedType='true'></FeatureType>
         </GroupType>
      </FeatureType>

      <Constraint name='C1' condition='excludes'
         featureType1='/RootType/Engine/Electric' featureName1=''
         featureType2='/RootType/Transmition/Automatic' featureName2='volks'/>
   </FeatureModel>

   <FeatureConfiguration id='FC1'>
      <Feature featureType='RootType' name='root'>
         <Group featureType='Transmition' name='transmition'>
            <Attribute name='code' attributeType='Code' value='IKF00001'/>
            <Feature featureType='Automatic' name='fiat' isGrouped='true'></Feature>
         </Group>
         <Group featureType='Engine' name='engine'>
            <Feature featureType='Electric' name='15kVA' isGrouped='true'></Feature>
            <Feature featureType='Gasoline' name='1000cc' isGrouped='true'></Feature>
         </Group>
      </Feature>
   </FeatureConfiguration>
</FeatureClass>
```

Figura 13 Exemplo OOFM-XML para o domínio de automóveis.

A Figura 14 ilustra o *FeatureClass Editor* trabalhando com o exemplo OOFM-XML para o domínio de automóveis (Figura 13). O lado esquerdo da imagem apresenta os elementos modelados para a árvore de features OOFM. Já o lado direito apresenta o diagram OOFM visualizado pelo framework *Prefuse* (Prefuse, 2011), o qual oferece uma interação diferenciada para as árvores de features desenhadas. A Figura 14 também ilustra instâncias *FeatureType* com diferentes cores no diagrama OOFM, destacando assim os elementos agrupados nos seus respectivos *GroupTypes*.

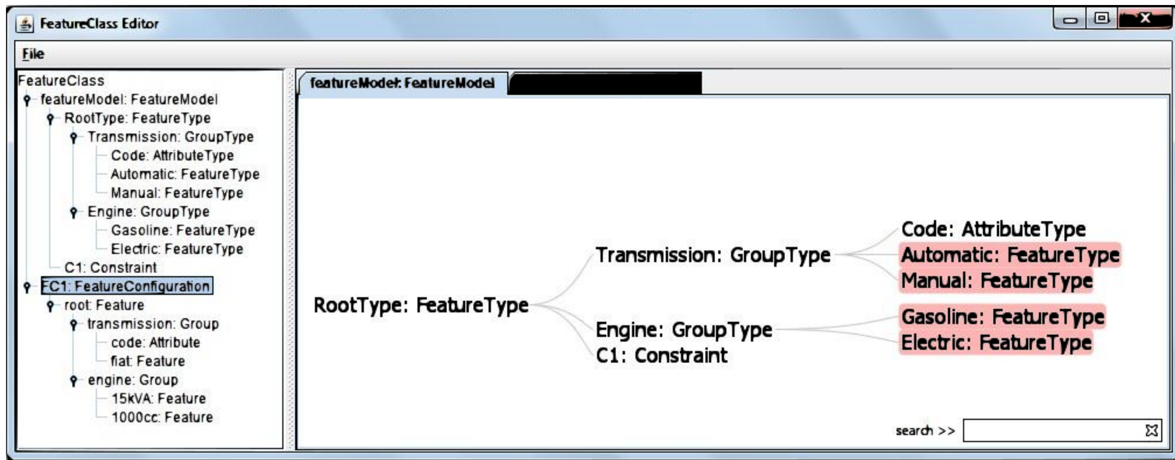


Figura 14 *FeatureClass Editor* trabalhando com o exemplo OOFM-XML para o domínio de automóveis.

2.5 CONCLUSÃO DO CAPÍTULO

Este capítulo apresentou os princípios base da técnica OOFM. Para tal, conceitos-chave da OOFM foram descritos, juntamente com sua abordagem de formalização e ferramentas de suporte.

Com relação aos conceitos chave da OOFM, combinou-se árvores de features Forfamel com relacionamentos Composition Model e operações de features disponíveis. O processo de formalização da OOFM definiu restrições OCL e modelou um Perfil UML, dando um significado sintático e semântico aos recursos OOFM especificados. Facilidades gráficas e de processamento para a modelagem OOFM também foram obtidas via *FeatureClass Editor* e OOFM-XML, respectivamente.

Como resultado, importantes recursos para definição e uso da OOFM se tornaram disponíveis, fornecendo soluções textuais, gráficas e formais para modelar features OOFM desejadas. No geral, tratam-se de recursos base importantes para a produção padronizada de SPLs e de sistemas concretos com base na OOFM, tema este que começa a ser abordado nos próximos capítulos deste trabalho.

Capítulo

3

Este capítulo aborda o uso da OOFM na produção de SPLs avançadas. Para tal, um OOFM Framework baseado no padrão Modelo-Visão-Controle (MVC) foi desenvolvido. Um OOFM Process baseado em atividades de SPL Engineering (SPLE) também foi especificado. Tratam-se de recursos abstratos que serão especializados e configurados durante a produção de SPLs de domínios específicos e de sistemas concretos desejados.

3 LINHAS DE PRODUTO DE SOFTWARE BASEADAS NA OOFM

Segundo Clements e Northrop (2001), Software Product Line (SPL) é um conjunto de sistemas intensivos de software compartilhando um conjunto de features comuns e gerenciadas que satisfaz necessidades específicas de um mercado ou missão particular, e que são desenvolvidos de um conjunto comum de recursos base de uma maneira prescrita.

No intuito de prover SPLs, Czarnecki (2004) descreveu FMs como o ponto inicial para desenvolver arquiteturas de famílias de sistemas numa perspectiva de *solution-space*. Para Rosenmüller e Siegmund (2010), SPLs podem ser descritas usando FMs e instâncias de SPLs (programas concretos) podem ser definidas pela seleção de features requeridas. Mais ainda, Beuche e Dalgarno (2006) descreveram importantes atividades de Engenharia de SPLs (SPLE) baseadas em FMs. Neste sentido, pela possível representação de SPLs via FMs e buscando cobrir o aspecto de variabilidade "produção de sistemas" identificado, este capítulo descreve a produção de SPLs com base na OOFM.

Assim, considerando a produção de sistemas concretos via SPLs baseadas em features, é possível verificar que estes podem ser diretamente obtidos com os recursos OOFM já definidos. Basta utilizar FMs e FCs representados em *FeatureClasses* nas demais classes de um sistema concreto via abordagem de implementação da variabilidade identificada (Gurp,

Svahnberg e Bosch, 2000). Desta forma, FMs utilizados indicarão a variabilidade da SPL produzida, enquanto que FCs instanciados representarão os produtos concretos derivados.

Buscando garantir um tratamento diferenciado aos sistemas produzidos a partir das *FeatureClasses* modeladas, desenvolveu-se um OOFM Framework para a OOFM (Sarinho, Apolinário e Almeida, 2012). Trata-se de um framework (Fayad, Schmidt e Johnson, 1999) baseado no padrão Model-View-Controller (MVC) (Krasner e Pope, 1988) que permite: a integração de features espalhadas em FMs e FCs distintos; a atualização e inspeção dinâmica de features definidas em *FeatureClasses* distintas; e a adaptação de features modeladas em artefatos de implementação de sistemas existentes. Ou seja, trata-se de um framework que oferece os recursos necessários para a produção de MPLs e DSPLs de uma maneira padronizada com base na técnica OOFM proposta.

Um OOFM Process também foi desenvolvido para o OOFM Framework. Ele apresenta atores, atividades e transições baseadas em atividades de SPL Engineering (SPLE) (Beuche e Dalgarno, 2006) que trabalham com recursos definidos na OOFM e no OOFM Framework. Tais atividades de SPLE permitem o sequenciamento da produção de recursos OOFM concretos para um domínio de software particular.

Juntos, OOFM Framework e OOFM Process representam recursos abstratos importantes da técnica OOFM que auxiliam na geração de SPLs padronizadas e avançadas para domínios de software específicos (Figura 2).

3.1 OOFM FRAMEWORK

Segundo Fayad, Schmidt e Johnson (1999), frameworks são capazes de ditar o fluxo geral de controle em um programa qualquer, fornecendo um comportamento default e extensível sendo na maioria dos casos protegidos por modificações indesejadas de usuários. Seguindo esta afirmação, um OOFM Framework foi desenvolvido no intuito de “gerenciar” a programação de features de modelos OOFM projetados (Sarinho, Apolinário e Almeida, 2012).

Trata-se de um mapeamento dos recursos propostos no Perfil OOFM (Sarinho e Apolinário, 2012) para códigos concretos de programação OO (Warmer e Kleppe, 2003). A idéia é realizar algo como: para cada estrutura representativa de feature definida no Perfil OOFM, tem-se uma classe equivalente numa linguagem de programação OO escolhida. Desta forma, classes representativas de features (e.g. *FeatureType*, *Feature*), relacionamentos de features (*root*, *rootType*, entre outros), operações de features (*merge*, *clone*, etc) e demais recursos OOFM se tornam implementáveis em linguagens OO desejadas.

Mais ainda, de acordo com os recursos OO disponíveis para cada linguagem OO destino, tem-se que:

- Diferentes tipos de estratégias de validação de features podem ser aplicadas nas árvores de features definidas (e.g. exceções, verificação de resultados de funções);
- Importantes atributos de qualidade para um sistema concreto podem ser considerados durante o mapeamento (e.g. desempenho, complexidade, robustez); e
- Estratégias similares de uso dos recursos OOFM modelados podem ser aplicadas, mesmo sendo implementados em linguagens OO distintas (portabilidade).

A Figura 15 mostra uma descrição parcial de um código Java (Java, 2012) representativo do OOFM Framework, apresentando operações e validações de instâncias *FeatureType* para cada *FeatureModel* representado.

Padrões estruturais e comportamentais (e.g.: *observer* e *adapter* (Gama et al., 1996), MVC (Krasner e Pope, 1988)) também foram aplicados no OOFM Framework (Figura 16). Através do padrão MVC, separou-se preocupações estáticas e dinâmicas da OOFM envolvidas na produção de sistemas concretos. Como resultado, garantiu-se ao *Controller* um papel de *middleware* entre features documentadas (*Model*) e recursos de implementação desejados (*View*).

Focando na representação do *Model* de um sistema (Figura 16), foram aplicadas estruturas *FeatureState* e *FeatureContext*, juntamente com estruturas representativas de *features*.. *FeatureState* e *FeatureContext* são derivados do padrão *state* (Gamma et al., 1994),

o qual é usado pela programação OO para representar o estado de um objeto em tempo de execução. Já as estruturas representativas de *features*, estas são formadas por instâncias das classes *Feature*, *Group*, *Attribute*, *FeatureType*, *GroupType* e *AttributeType*, as quais foram declaradas no pacote *featureModeling* mapeado a partir do metamodelo do Perfil OOFM.

```
1.  // instantiate a new FeatureType that must be included as subtype of the indicated  featureTypeRef
2.  public void add(FeatureType featureTypeRef, String newName, Integer newLower,
3.                                                          Integer newUpper) throws Exception{
4.        this.merge(new FeatureType(newName, newLower, newUpper), featureTypeRef);
5.  }
6.
7.  //  includes the indicated featureType as subtype of the featureTypeRef
8.  public void merge(FeatureType featureType, FeatureType featureTypeRef) throws Exception{
9.        featureType.setParent(featureTypeRef);
10.       featureTypeRef.addSubtype(featureType);
11. }
12.
13. //  excludes the subtyped featureType from the featureType.parent
14. public void exclude(FeatureType featureType) throws Exception{
15.       featureType.getParent().removeSubtype(featureType);
16.       featureType.setParent(null);
17. }
18.  …
19. //  verify all proposed restrictions for feature objects in a FM, such as null parents, rootType values, ...
20. public void validate() throws FeatureTypeException, ConstraintException {
21.       verifyConstraintNames();
22.       verifyConstraintFeatureTypes();
23.       verifySimilarFeatureTypesInAConstraint();
24.       verifySubtypeNames(this.rootType);
25.       verifyNullParents(this.rootType);
26.       verifyAttributeNames(this.rootType);
27.       …
28.       // verify rootType data
29.       if ((!this.rootType.getName().equals("rootType")) || (this.rootType.getParent() == null) ||
30.            (this.rootType.getLower() != 1) || (this.rootType.getUpper() != 1))
31.                throw new FeatureTypeException("Invalid rootType data.", this.rootType); ...
32. }
```

Figura 15 Descrição parcial de operações e validações do OOFM Framework em Java.

No geral, tratam-se de estruturas base para representar árvores de features em modelos OOFM projetados. Elas também facilitam a integração de features em árvores distintas, com um *FeatureState* representando múltiplas instâncias de *Feature,* e com um *FeatureContext* representando múltiplas instâncias *FeatureState* quando necessário.

FeatureBehavior, *FeatureObserver*, *FeatureTypeObserver* e *FeatureStateObserver* são estruturas comportamentais definidas para representar o *Controller* de um sistema (Figura 16). Estruturas *FeatureBehavior* são baseadas no padrão *command* (Gamma et al., 1994), o qual define objetos que encapsulam as respectivas ações e seus parâmetros. Instâncias *FeatureObserver*, *FeatureTypeObserver* e *FeatureStateObserver* são baseadas no padrão *observer* (Gama et al., 1994), onde objetos são registrados para observar um evento o qual

pode ser repassado para outro objeto. Estas estruturas são importantes para monitorar atualizações dinâmicas aplicadas em estruturas *Model* criadas. Elas também permitem a execução de operações de features estaticamente declaradas ou dinamicamente definidas.

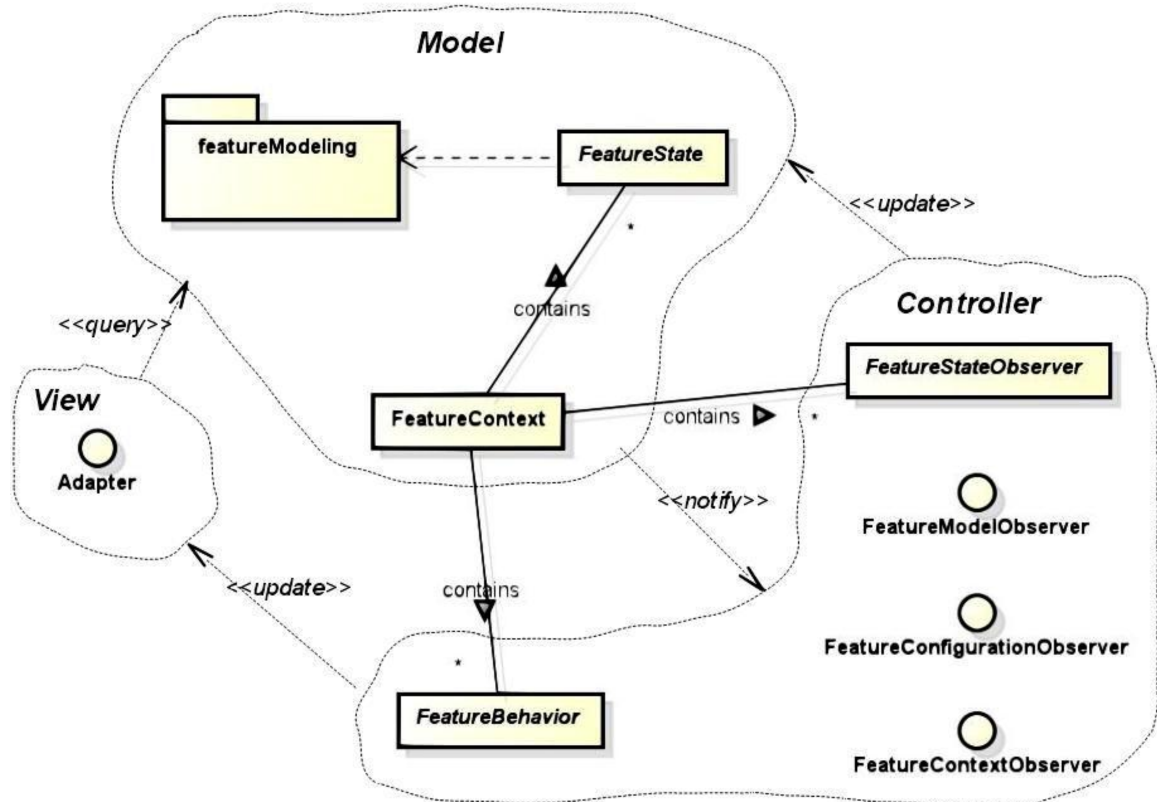


Figura 16 Arquitetura MVC do OOFM Framework.

Para a representação da *View*, instâncias *Adapter* são fornecidas para aplicar valores do *Model* em cada tecnologia de implementação utilizada (Figura 16). Cada instância *Adapter* é baseada no padrão *adapter* (Gamma et al., 1994), o qual permite que classes com interfaces incompatíveis trabalhem juntas, encapsulando sua própria interface dentro de uma interface compatível com uma classe já existente. Instâncias *Adapter* são a principal interface entre a variabilidade OOFM identificada (para um domínio de software específico) e o sistema concreto desejado (baseado numa solução de implementação existente que será adaptada).

Como resultado, o OOFM Framework se torna uma importante extensão da OOFM para a criação de sistemas concretos baseados em FMs e FCs estáticos e dinâmicos. Ele também fornece características importantes para a implementação de SPLs avançadas, tais como: a combinação de features espalhadas em FMs distintos através de instâncias

FeatureState; a atualização dinâmica de *FeatureClasses* distintas via instâncias *FeatureBehavior*; e a inspeção dinâmica de atualizações de features via *feature observers* definidos.

3.2 ATIVIDADES DE SPLE PARA RECURSOS OOFM

Com base nas atividades de SPLE propostas por Beuche e Dalgarno (2006) (à esquerda na Figura 17), definiu-se **3 passos principais** para a produção de recursos OOFM concretos voltados para uma missão ou mercado particular de software (Sarinho, Apolinário e Almeida, 2012) (à direita na Figura 17):

1) A representação de FMs via recursos OOFM;
2) A geração de FCs via operações OOFM; e
3) A adaptação de artefatos de implementação de sistemas para recursos OOFM.

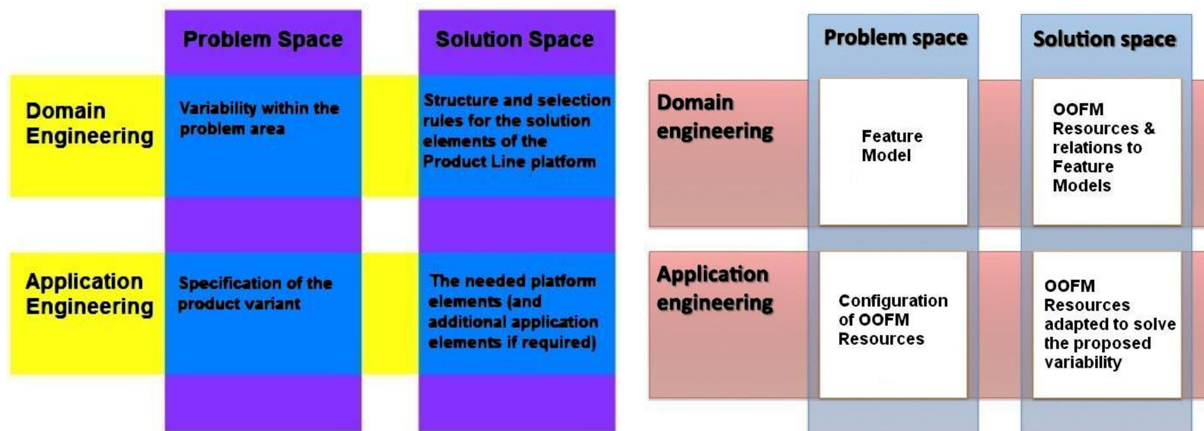


Figura 17 Atividades de SPLE (à esquerda) e suas adaptações para recursos OOFM (à direita).

Desta forma, considerando as atividades dos *Problem* e *Solution Spaces* para a seção *Domain Engineering* (o **primeiro passo**) (Figura 17), diferentes tipos de

FeatureClasses podem ser criadas no intuito de representar FMs desejados. Elas podem ser usadas para modelar árvores de features tradicionais, bem como FMs e FCs estendidos por recursos OO.

Também, considerando o uso do OOFM Framework, várias *features*, *states*, *behaviors* e *observers* podem ser declaradas no intuito de representar características "avançadas" de features. *FeatureStates* representam múltiplas *features* de uma maneira integrada. Atualizações aplicadas em *features* e *states* instanciados são inspecionadas via *observers* atachados. *FeatureBehaviors* (ativadas por *observers* ou não) quando executados permitem a atualização de múltiplas *features* e *states* desejadas.

Desta forma, enquanto que elementos *states* definem novas relações estáticas num FM, *observers* e *behaviors* definem novas relações dinâmicas entre elementos de um FM. Ou seja, tratam-se de novos "relacionamentos" para FMs que, juntos com o conceito de *FeatureClass* previamente definido, permitem a modelagem de FMs avançados de acordo com os interesses do projetista de features.

De acordo com o *Problem Space* de *Application Engineering* (o **segundo passo**) (Figura 17), é necessário produzir FCs desejados através dos recursos OOFM definidos. Modelos UML baseados no Perfil OOFM e interpretações de XMLs produzidos pelo *FeatureClass Editor* podem ser usados para instanciar *FeatureClasses* com os FCs desejados. Também, para cada *FeatureClass* declarada, várias instâncias de *features*, *states*, *behaviors* e *observers* podem ser produzidas via programação de features em métodos de features declarados. Juntas, estas abordagens se complementam para gerar FCs desejados via recursos OOFM.

Finalmente, considerando o *Solution Space* de *Application Engineering* (o **terceiro passo**) (Figura 17), vários *adapters* podem ser criados e configurados no intuito de implementar a variabilidade proposta de um sistema desejado. Cada *adapter* realiza a interface *Adapter* do OOFM Framework, fornecendo comportamentos específicos para os métodos *initialization* e *adaptation* da interface. O método *initialization* é basicamente disparado na inicialização de um sistema, aplicando a carga inicial de features correntes instanciadas. Já o método *adaptation* é executado sempre que determinadas atualizações em features monitoradas precisam ser aplicadas num sistema. Ou seja, tratam-se de métodos que

aplicam o estado corrente de FCs modelados em sistemas (ou recursos de implementação) adaptados durante as suas execuções.

AdapterObservers também são usados para controlar *adapters* criados. Eles verificam se atualizações aplicadas em valores de *features* (usadas neste caso por instâncias *FeatureState*) devem ser visualizadas ou não no sistema adaptado. Caso sim, o método *adaptation* é executado de forma a realizar as atualizações necessárias no sistema adaptado.

3.3 OOFM PROCESS

Com as atividades de SPLE mapeadas na técnica OOFM, percebe-se que diferentes tipos de *atores*, *atividades* e *transições* podem ser aplicados em cada passo descrito. Tratam-se de elementos que podem ser combinados em diagramas de atividades para modelar: o processo geral do negócio; a lógica capturada em um único cenário de uso; ou a lógica detalhada de uma regra de negócio (Booch, Rumbaugh e Jacobson, 1998).

Assim, tem-se no OOFM Process (Figura 18) um diagrama de atividades que ilustra o processo de modelagem OOFM baseado em atividades de SPLE adaptadas. Dentre os atores identificados no OOFM Process, destacam-se: o *Domain Specialist*, o *OOFM Specialist*, o *Application Creator* e o *Technology Specialist*.

O *Domain Specialist* é responsável pela identificação de features representativas de um domínio de software. Estas features podem ser ou específicas para uma família de softwares ou genéricas para qualquer família de softwares possível de ser representada neste domínio de software.

O *OOFM Specialist* é responsável pela extensão do OOFM Framework, criando novas estruturas OOFM para domínios de software específicos. Ele segue as features identificadas pelo *Domain Specialist*, as quais guiam a especialização dos recursos abstratos disponíveis no OOFM Framework.

O *Application Creator* gera aplicações concretas a partir dos recursos do OOFM Framework e das features desejadas num domínio de software específico. Trata-se de um

produtor de FCs que gera features a partir de uma família de software projetada (FM associado). Features geradas também podem ser usadas para contextualizar estruturas OOFM modeladas com a ajuda do *OOFM Specialist*. FMs especializados também podem ser criados pelo *Application Creator* a partir de FMs genéricos, antes de se construir os FCs desejados para cada aplicação de domínio projetada.

Finalmente, o *Technology Specialist* trabalha na integração dos recursos do OOFM Framework com os recursos da tecnologia de implementação escolhida. Trata-se de um ator que fornece as estruturas *Adapter* necessárias para se criar sistemas concretos, bem como ajuda o *Application Creator* a usar tais estruturas *Adapter* quando necessário.

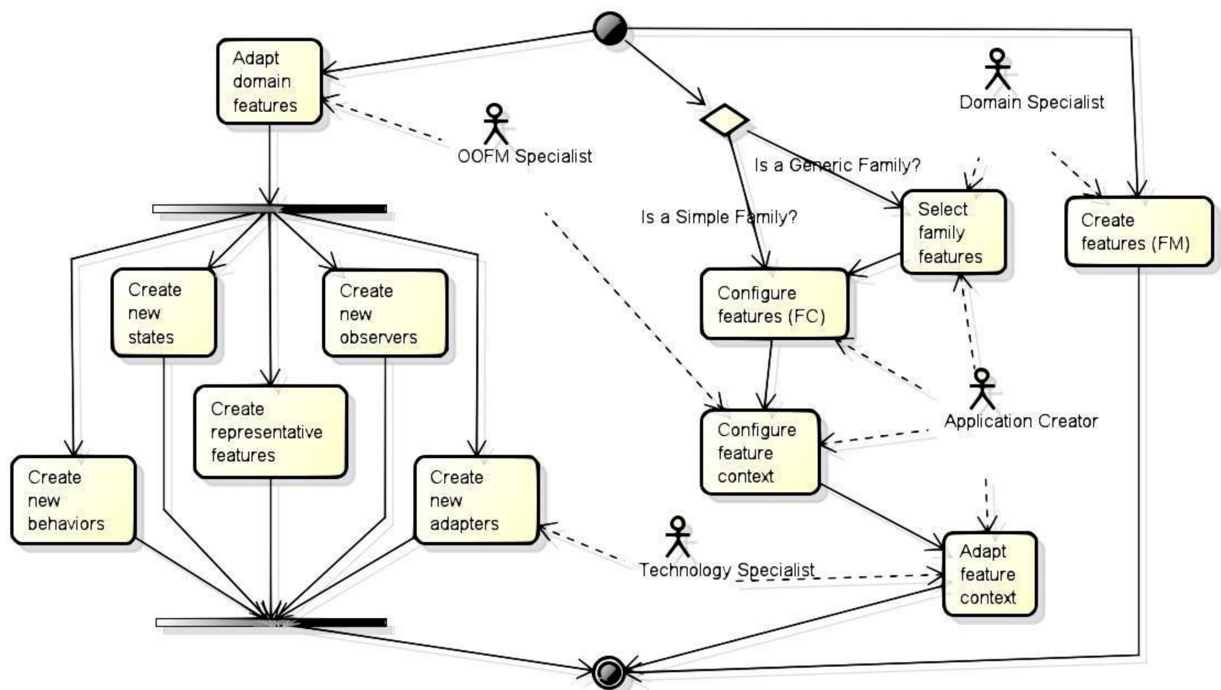


Figura 18 Diagrama de Atividades do OOFM Process.

Ou seja, *Domain Specialist* e *OOFM Specialist* executam atividades relacionadas ao **primeiro passo**, criando recursos OOFM representativos de FMs de uma SPL de domínio específico. *Application Creator* executa na maior parte do seu tempo atividades do **segundo passo**, gerando FCs com base nos recursos de FM especializados. Entretanto, caso algum recurso de FM tenha se mantido abstrato mesmo após a sua especialização para um domínio específico, *Application Creator* pode atuar como produtor de recursos de FM especializados, assumindo o papel de um *Domain Specialist* ou de um *OOFM Specialist*. Já o *Technology*

Specialist fica responsável pelas atividades do **terceiro passo**, adaptando os recursos OOFM definidos para os respectivos artefatos de implementação do sistema.

3.4 CONCLUSÃO DO CAPÍTULO

Este capítulo apresentou uma abordagem de criação padronizada de SPLs avançadas com base na OOFM. Para tal, foram descritos o OOFM Framework e as atividades de SPLE organizadas no OOFM Process.

OOFM Framework é basicamente formado por um conjunto de estruturas que se caracterizam como uma Product Line Architecture (PLA) capaz de representar FMs e FCs avançados via recursos OOFM. Já o OOFM Process define os atores e atividades necessárias para converter as estruturas abstratas do OOFM Framework numa SPL concreta e especializada. Atividades do OOFM Process também indicam como produzir sistemas concretos com base na SPL definida.

Como resultado, uma nova abordagem de desenvolvimento de SPLs foi obtida com a OOFM, sendo esta formada por FMs e FCs avançados aplicados numa PLA simples e padronizada. Ou seja, uma abordagem inversa às técnicas tradicionais de produção de SPL, as quais são baseadas em modelos FODA simples manipulados por PLAs distintas e complexas (Sarinho, Apolinário e Almeida, 2012).

O próximo capítulo irá apresentar um exemplo de SPL baseada na OOFM para o domínio dos jogos digitais denominada Feature-based Environment for Digital Games (FEnDiGa), bem como seu uso na produção de uma versão simplificada do jogo Pacman, corroborando assim a capacidade da OOFM em fornecer sistemas de software concretos.

Capítulo

4

Este capítulo apresenta o Feature-based Environment for Digital Games (FEnDiGa), uma especialização do OOFM Framework e do OOFM Process para o domínio dos Jogos Digitais. FEnDiGa foi aplicada na produção de uma versão simplificada do jogo digital Pacman, demonstrando assim a capacidade da OOFM em gerar sistemas concretos desejados.

4 IMPLEMENTANDO JOGOS DIGITAIS COM A OOFM

OOFM Framework melhora a técnica OOFM ao fornecer um tratamento estático e dinâmico de FMs e FCs modelados. OOFM Process apresenta atores, atividades e transições de atividades de SPLE com base em recursos definidos na OOFM. Tratam-se de recursos importantes para a OOFM que auxiliam na produção de sistemas concretos desejados. Entretanto, tratam-se também de estruturas abstratas projetadas para trabalhar apenas com recursos genéricos representativos de features. Neste sentido, versões especializadas do OOFM Framework e do OOFM Process precisam ser fornecidas, no intuito de tratar a variabilidade de domínios de software específicos bem como gerar sistemas concretos desejados.

Com relação ao domínio dos jogos digitais, trata-se de um domínio de grande variabilidade e complexidade, tanto no aspecto conceitual como também no aspecto de implementação de jogos digitais específicos. De fato, trabalhar com jogos digitais significa trabalhar com diferentes tipos de: simulações (e.g. esportes, aventuras, lutas, batalhas), hardwares (e.g. jogos móveis, jogos para web, acelerômetros), interações humanas (e.g. imersão, multiplayer) e estórias complexas (e.g. jogos derivados do cinema, séries de RPGs) (Sarinho e Apolinário, 2008). Ou seja, trata-se de um excelente campo de prova para a abordagem OOFM proposta. Este capítulo apresenta a geração de uma SPL OOFM para o

domínio dos jogos digitais denominada *Feature Environment for Digital Games* (FEnDiGa) (Sarinho, Apolinário e Almeida, 2011-2012). Trata-se de uma especialização do OOFM Framework e do OOFM Process, tendo como base os modelos de features NESI (Sarinho e Apolinário, 2008) e GDS (Sarinho e Apolinário, 2009) representativos de jogos digitais.

FEnDiGa permite a integração das visões conceituais e de implementação de jogos representadas nas features dos modelos NESI e GDS, respectivamente. FEnDiGa também permite a adaptação direta de FMs representativos de jogos digitais para múltiplas game engines, algo importante para garantir a portabilidade do G-Factor (Binsubaih e Maddock, 2008) em jogos digitais modelados.

4.1 MODELOS NESI E GDS

O modelo *Narrative-Entertainment-Simulation-Interaction* (NESI) é basicamente uma coletânea de features baseada em conceitos de jogos digitais identificados na literatura (Figura 19) (Sarinho e Apolinário, 2008). Seu objetivo principal é definir um jogo digital como uma combinação de quatro visões conceituais organizadas em features: *Narrativa*, *Entretenimento*, *Simulação* e *Interação*.

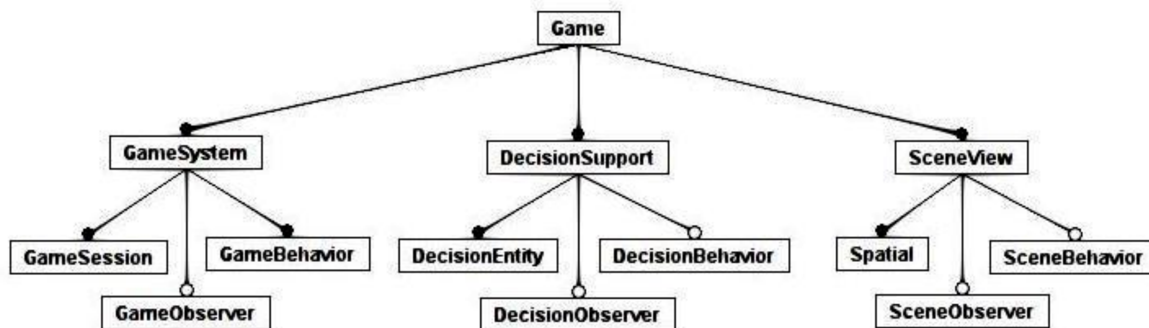


Figura 19 Diagrama simplificado do modelo NESI.

Segundo Sarinho e Apolinário (2008), uma *Narrative* é um *Flow*, um script dinâmico tentando alcançar *Goals* seguindo algumas *Rules* definidas para o jogo. *Entertainment* é representado pela *Immersion* do jogador durante o jogo, seguido por uma

proposta de *Theme* para integrar o jogador e o jogo na realidade proposta. *Simulation* é uma combinação de *Elements* (recursos para jogar) e suas *Relationships*, que acontecem num *Environment* definido (espaços para jogar). A *Interaction* humana é representada pelas features *Control* (entrada do jogo) e *Presentation* (saída do jogo), consolidadas pela feature *GamePlay* que indica a execução e o estado do jogo como um todo.

Com estas features principais, outras subfeatures relacionadas a conceitos de jogos foram definidas, a exemplo das subfeatures de *Element* demonstradas na Figura 20. Como resultado, foram fornecidas mais de 350 features conceituais de jogos digitais capazes de serem configuradas e manipuladas por projetistas de jogos.

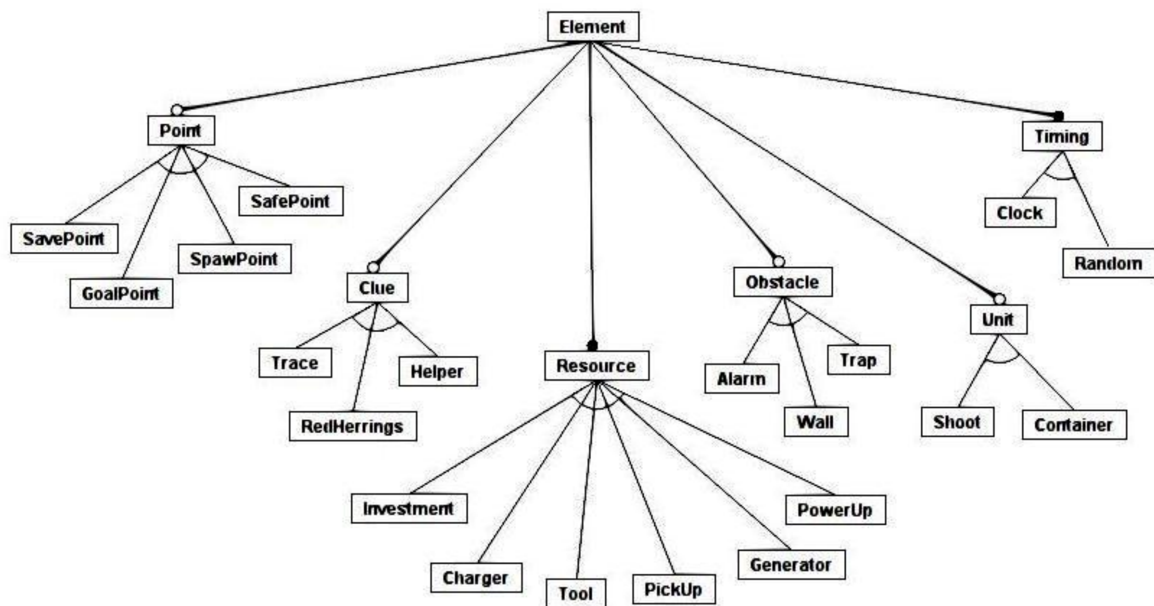


Figura 20 Modelo de features da subfeature *Element*.

Por outro lado, percebendo que o modelo NESI se encontrava muito distante da implementação real de um jogo digital, desenvolveu-se um segundo trabalho de modelagem de features focando apenas em recursos e trabalhos voltados para a implementação dos jogos digitais (Sarinho e Apolinário, 2009). Como resultado, obteve-se o modelo de features *GameSystem-DecisionSupport-SceneView* (GDS).

Segundo Sarinho e Apolinário (2009), o modelo de features GDS (Figura 21) busca definir um jogo como uma combinação de três features principais: *GameSystem*,

DecisionSupport e *SceneView*, onde cada uma delas descreve configurações genéricas e aspectos comportamentais identificados em recursos de implementação de jogo digitais.

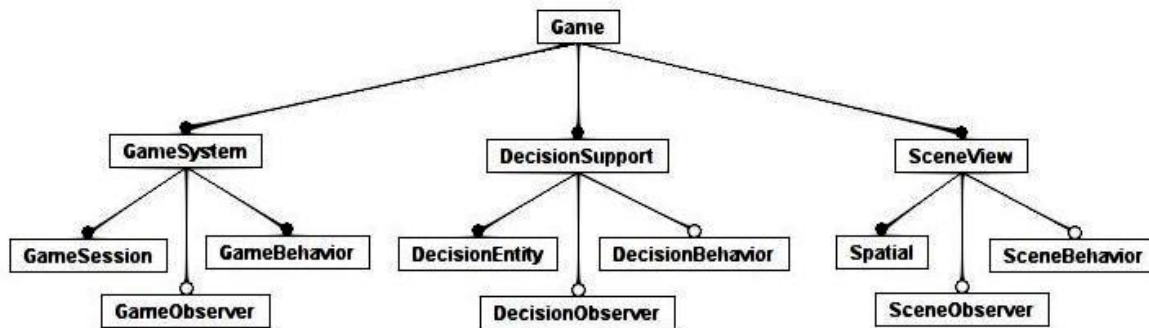


Figura 21 Diagrama simplificado do modelo GDS.

A feature *GameSystem* é o ponto principal de junção do jogo. Ela é responsável pelo controle da execução do jogo, descrevendo *GameBehaviors*, *GameContext* e *GameObservers* disponíveis para o jogo. Pela execução de *GameBehaviors*, ativados por *GameObservers* ou não, o *Player* corrente da *GameSession* pode disparar ações que afetam todos os dados definidos no jogo, tais como *DecisionEntities* e *SceneNodes* por exemplo.

A feature *DecisionSupport* é um esforço para integração de estratégias de tomada de decisão usadas por diferentes jogos digitais. Ela apresenta *DecisionEntities* (*Scenario*, *Agent*), o *ContextState* de cada *DecisionEntity* (*FiniteState* ou *NeuralNet*), e *DecisionBehaviors* predefinidas, capazes de ler e alterar valores de *DecisionEntities* e *ContextStates* durante as respectivas execuções.

A feature *SceneView* (Figura 22) é uma coleção de *SceneNodes* distribuídos por sessões *Spatial*, tendo vários *SceneBehaviors* e *SceneObservers* capazes de executar ações de cena num jogo. Um *SceneNode* representa uma hierarquia de informações sobre a cena, com uma *Location* e um *BoudingVolume* específico para detecção de colisão em cada nó de cena. Cada *SceneNode* pode representar também hierarquias de informações de *AudioNode*, *GraphicNode* e *PhysicsNode* simultaneamente.

Como resultado, a partir das features principais: *GameSystem*, *DecisionSupport* e *SceneView*, e das subfeatures: *AudioNode*, *GraphicNode* e *PhysicsNode*, um total de 6 modelos de features relacionadas a implementação de jogos são definidos, produzindo mais de 250 features gerais a serem manipuladas pelo desenvolvedor de jogos.

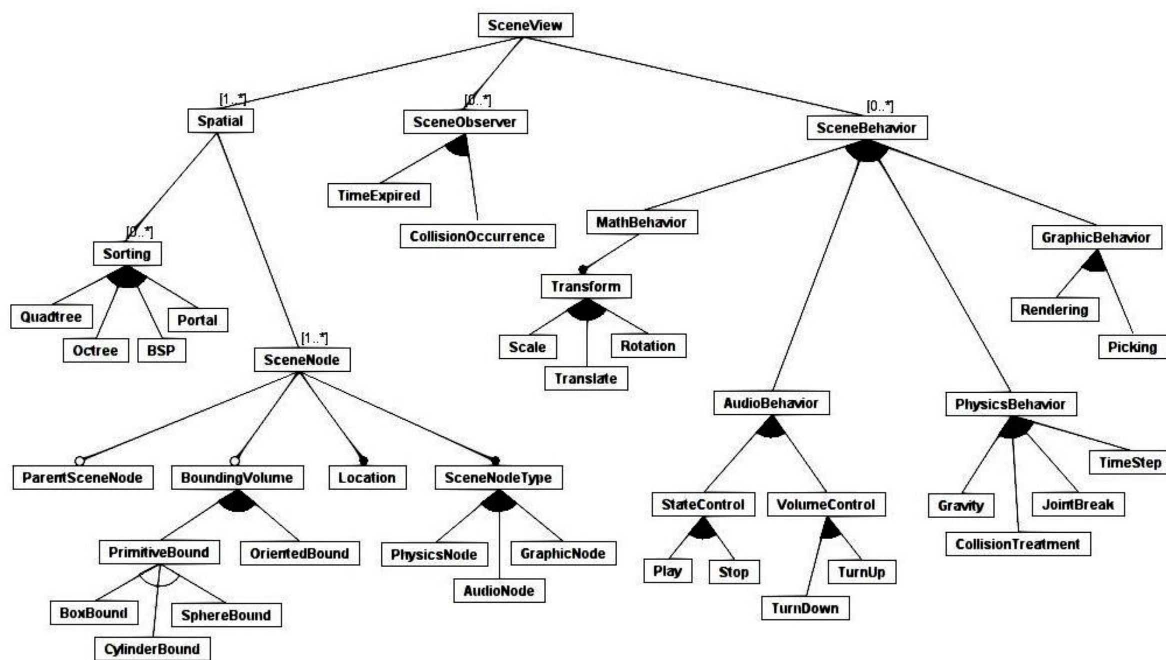


Figura 22 Modelo de features da subfeature *SceneView*.

4.2 FEATURE-BASED ENVIRONMENT FOR DIGITAL GAMES

Como descrito previamente, versões especializadas do OOFM Framework e do OOFM Process precisam ser fornecidas para atender uma missão ou mercado particular de software (Sarinho, Apolinário e Almeida, 2012). Esta subseção descreve a produção do *Feature-based Environment for Digital Games* (FEnDiGa), uma da especialização do OOFM Framework e OOFM Process para o domínio dos jogos digitais (Sarinho, Apolinário e Almeida, 2011-2012).

Assim, considerando o OOFM Process (Figura 18), é possível verificar a diagramação de duas principais linhas de atividades a serem seguidas, sendo a primeira voltada para a *especialização* e a segunda voltada para a *seleção* e *configuração* de recursos OOFM definidos. As atividades de especialização focam na produção de *features*, *states*, *behaviors*, *observers* e *adapters* para um domínio de software específico. Já as atividades de

seleção e configuração busca gerar um sistema concreto com base nos recursos OOFM especializados.

Com a aplicação dos modelos NESI e GDS para a representação de features de jogos digitais, as atividades de especialização do FEnDiGa Process (Figura 23) irão estender os recursos do OOFM Framework para *representar*, *integrar*, *manipular*, *monitorar* e *adaptar* features NESI e GDS definidas. Já as atividades de seleção e configuração do FEnDiGa Process (Figura 23) irão trabalhar com os recursos NESI e GDS representados no FEnDiGa para produzir um jogo digital desejado.

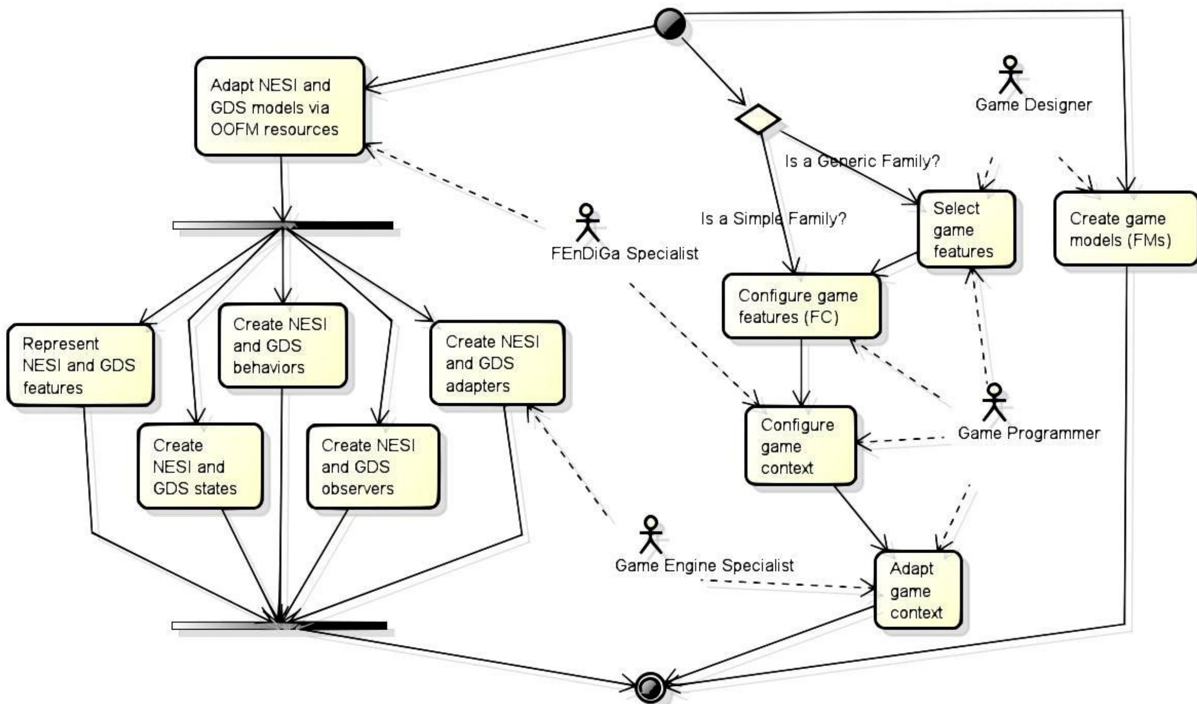


Figura 23 FEnDiGa Process.

FEnDiGa Framework (Figura 24) é a especialização do OOFM Framework para *representar*, *integrar*, *manipular*, *monitorar* e *adaptar* features NESI e GDS propostas. Trata-se de uma distribuição e integração das features NESI e GDS modeladas com base nos recursos abstratos disponibilizados pelo OOFM Framework.

Considerando a atividade de representação de features NESI e GDS no processo de geração do FEnDiGa Framework, instâncias *FeatureType* são criadas para representar o *Model* de um jogo digital, tais como *Narrative*, *Element*, *DecisionSupport*, *Avatar* e *GameSystem*, entre outras. Trata-se da representação direta de features NESI e GDS

modeladas usando os recursos OOFM propostos. Cada *FeatureType* instanciado é agrupado em instâncias *FeatureModel* da respectiva *FeatureClasses* NESI ou GDS (*Narrative*, *Entertainment*, *Simulation*, *Interaction*, *GameSystem*, *DecisionSupport* ou *SceneView*) declarada no pacote *features* da Figura 24.

Diferentes tipos de *FeatureStates* capazes de combinar features distintas de jogos digitais também são definidos no FEnDiGa Framework, tais como *ActState*, *CollisionSystemState*, *ElementState*, *EnvironmentState*, *GameSessionState*, *ImmersionState*, *NodeState*, *PlayerState* e *SpatialState* (pacote *states* na Figura 24). Tratam-se de elementos identificados como pontos chave comuns para a integração de features NESI e GDS modeladas. Eles também foram identificados como pontos de referência na representação de informações desejadas por game engines a serem adaptadas posteriormente.

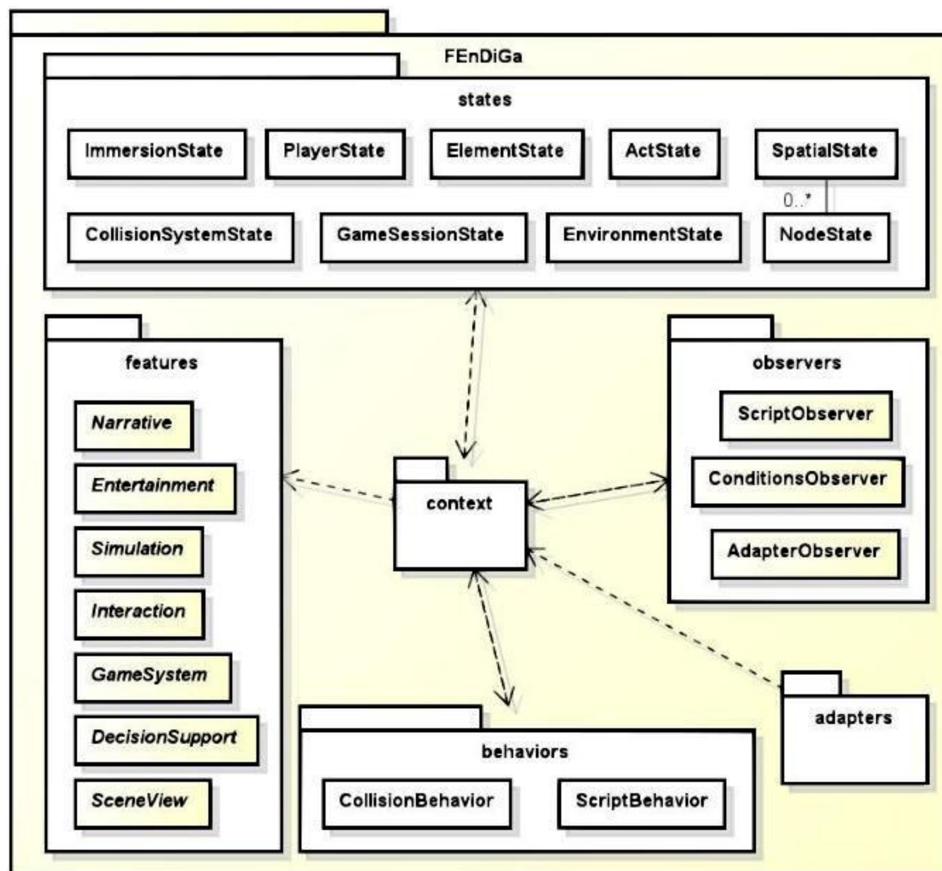


Figura 24 Visão modular do FEnDiGa Framework.

Estruturas especializadas de *FeatureBehavior* e *FeatureStateObserver* também são declaradas para tratar a dinâmica dos jogos digitais no FEnDiGa Framework, tais como

ScriptBehavior, *CollisionBehavior*, *ConditionObserver*, *AdapterObserver* e *ScriptObserver* (pacotes *behaviors* e *observers* na Figura 24). Tratam-se de estruturas capazes de representar comportamentos bem como tratar eventos em jogos digitais diversos, os quais são representados nas features *Action*, *Relationship*, *GameBehavior*, *DecisionBehavior, SceneBehavior*, *GameObserver*, *DecisionObserver*, *SceneObserver*, entre outras dos modelos NESI e GDS.

E para finalizar, alguns *adapters* também são implementados para integrar estruturas do FEnDiGa Framework em motores de jogos disponíveis. Para tal, estruturas *Adapter* especializadas foram criadas para os motores GTGE (Golden Studios, 2011) (*PlayFieldAdapter*, *SpriteAdapter*), JGame (JGame, 2011) (*JGObjectAdapter*, *JGObjectCollectionAdapter*) e JMonkeyEngine3 (JME3, 2011) (*StaticPhysicsNode2DAdapter*, *AnimatedPhysicsNode2DAdapter*). Tratam-se de classes adaptadoras capazes de trabalhar com *states* definidos na representação de jogos digitais 2D.

4.3 DESENVOLVENDO O JOGO SIMPLIFIEDPACMAN

Considerando as atividades de seleção e configuração definidas no FEnDiGa Process, esta subseção irá demonstrar como os recursos do FEnDiGa Framework serão utilizados para produzir um jogo digital desejado, neste caso o SimplifiedPacman (Sarinho e Apolinário, 2009).

Trata-se de uma versão simplificada do clássico Pacman, apresentando: o personagem pacman; apenas um fantasma como inimigo; diversas pílulas e paredes; tratamento de colisão entre elementos do jogo; quantidade de vidas do jogador; e um tratamento para a derrota ou vitória no jogo ("*game over*" ou "*game victory*") (Sarinho e Apolinário, 2009).

Com relação ao processo de produção do SimplifiedPacman, duas atividades se destacam de acordo com o FEnDiGa Process definido: 1) a seleção e respectiva configuração de *features*, *states*, *behaviors* e *observers* do jogo; e 2) a adaptação de *states* definidos no jogo

para game engines trabalhadas. Outra atividade importante é a criação de *FeatureTypes* representativas do FM do jogo. Entretanto, como não há necessidade de features extras para o SimplifiedPacman, pode-se considerar que esta atividade já foi realizada com a devida construção do FEnDiGa Framework dando suporte aos FMs NESI e GDS modelados.

Para realizar a seleção e respectiva configuração de *features*, *states*, *behaviors* e *observers* do SimplifiedPacman, utilizou-se algumas instruções de programação de features para criar novas instâncias necessárias a representação do jogo. A Figura 25 ilustra um pouco deste processo de configuração do jogo digital descrevendo um trecho do método *createSimplifiedPacmanGame* (Sarinho, Apolinário e Almeida, 2011-2012). Trata-se do método responsável pela configuração de estruturas do FEnDiGa Framework representativas do negócio do jogo SimplifiedPacman (Sarinho, Apolinário e Almeida, 2011-2012).

```
1.   // creating features to represent the Pacman character
2.   Feature pacmanElement = Simulation.addElement(simulationConfiguration,
3.                                   new FeatureParam("pacman", "Entity", "???"));
4.   Feature pacmanAvatar = Entertainment.addAvatar(entertainmentConfiguration,
5.                                   new FeatureParam("pacman", "Player", "???"),
6.                                   new FeatureParam[]{new FeatureParam("life", "Other", "1"),
7.                                           new FeatureParam("speedX", "Dextery", "0"),
8.                                           new FeatureParam("speedY", "Dextery", "0")});
9.   ...
10.
11.  // ElementState constructor returns a valid Pacman state based on Pacman features
12.  ElementState pacmanState = new ElementState(this.featureContext, ...,
13.                                          pacmanElement, ... , pacmanAvatar, ...);
14.  ...
15.
16.  // executes gameVictoryBehavior when number of pills == 0
17.  Feature pillsEndObserver = SceneView.addSceneObserver(sceneViewConfiguration,
18.                FeatureParam("pillsEnd", "SpatialEvaluation",
19.                       "featureState.getGroupNodeIDCount('pillGroup') == 0"),
20.                String[]{"gameVictoryBehavior"});
21.
22.  // creates pillsEnd observer using the feature pillsEndObserver
23.  ScriptObserver pillsEnd = new ScriptObserver(this.featureContext, ..., pillsEndObserver);
24.  ...
```

Figura 25 Descrição parcial do método *createSimplifiedPacmanGame*.

Numa visão detalhada deste trecho de código: o *state Pacman* é produzido por *features Pacman* instanciadas (linhas 1-13); e o *SpatialState* é monitorado por um *ScriptObserver* especializado chamado *pillsEnd* (linha 23), o qual executa o *gameVictoryBehavior* quando zero *pills* se tornam disponíveis no jogo (linhas 16-20).

Com relação à adaptação de *states* definidos no SimplifiedPacman para game engines trabalhadas, pretende-se utilizar as estruturas *Adapter* definidas no FEnDiGa Framework para integrar *states* modelados com game engines destino.

Um exemplo deste processo de integração entre *adapters* e *states* pode ser verificado na Figura 26. Ela ilustra a seleção de objetos *Adapter* (linhas 1, 15 e 21) e a sua inclusão em instâncias *AdapterObserver* (linhas 14 e 19) capazes de monitorar *states* específicos (via método *FeatureState.attach*) do jogo SimplifiedPacman (Sarinho, Apolinário e Almeida, 2011-2012).

```
1.  private SimplifiedPacmanPlayFieldAdapter gamePlayAdapter;
2.  …
3.
4.  public void initResources() {
5.          SpatialState gamePlayState = (SpatialState) this.simplifiedPacman.getFeatureContext().
6.                                                          getFeatureState("gamePlayState");
7.  …
8.
9.  // gets the gamePlayState and creates all NodeState adapters necessary to perform the game
10.  this.gamePlayAdapter.initialization(gamePlayState);
11. …
12.
13.  // configuring gamePlayAdapter as an observer of any gamePlayState changes
14.  gamePlayState.attach(new AdapterObserver(this.simplifiedPacman.getFeatureContext(),
15.                                                          this.gamePlayAdapter));
16.  ...
17.
18.  // configuring pacman adapter to verify pacmanState updates
19.  ElementState pacmanState = (ElementState) pacmanState.attach(
20.          new AdapterObserver( this.simplifiedPacman.getFeatureContext(),
21.                                          this.gamePlayAdapter.getAdapter("pacman")));
22.  ...
```

Figura 26 Descrição parcial do método *initResources*.

A Figura 27 ilustra o resultado final obtido para o jogo SimplifiedPacman após a configuração e adaptação das estruturas fornecidas pelo FEnDiGa Framework. Ela mostra: o negócio do jogo na classe SimplifiedPacman independente de implementação (o G-Factor de um jogo (Binsubaih e Maddock, 2008)); cenas obtidas para cada motor de jogo adaptado (GTGE (Golden Studios, 2011), JGame (JGame, 2011) e JME3 (JME3, 2011)); e versões especializadas de estruturas do FEnDiGa Framework desenvolvidas para ganhos de performance.

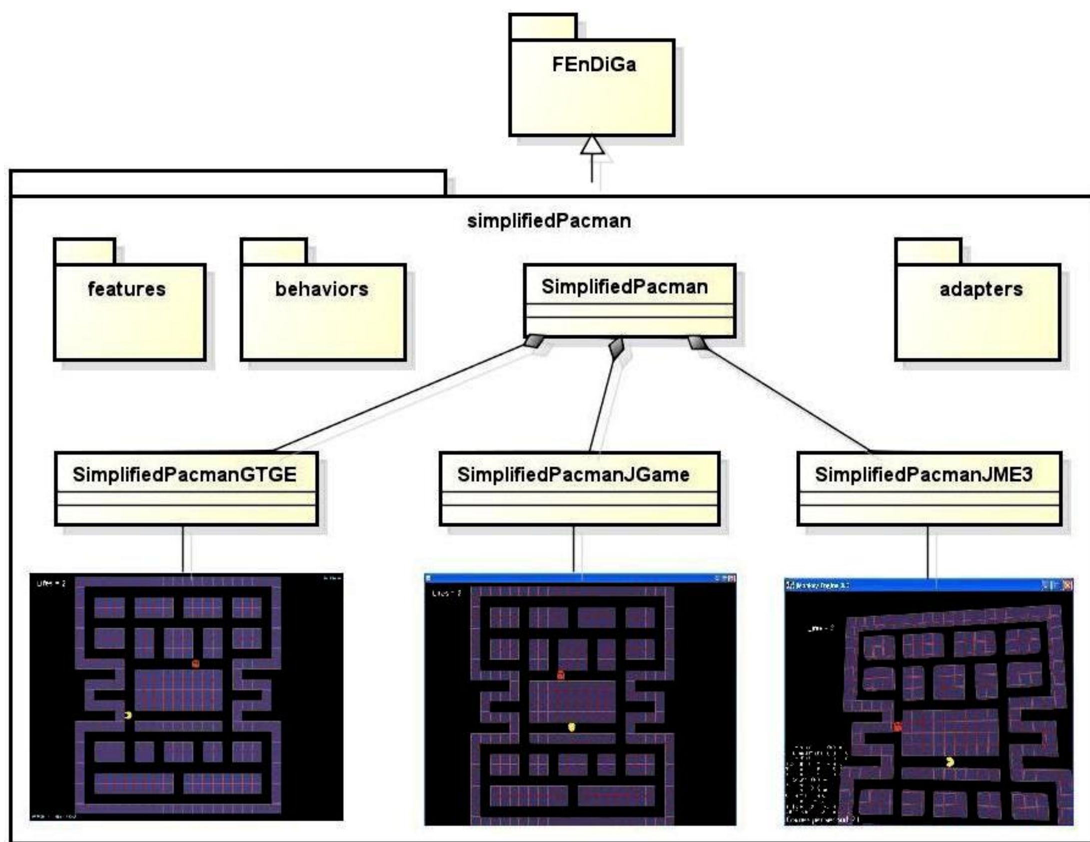


Figura 27 Arquitetura e imagens do jogo *SimplifiedPacman* via FEnDiGa.

4.4 CONCLUSÃO DO CAPÍTULO

Este capítulo apresentou a especialização do OOFM Framework e do OOFM Process para o domínio dos jogos digitais com base nos modelos NESI e GDS desenvolvidos. O modelo NESI descreve features conceituais representativas de jogos digitais encontradas na literatura. Já o modelo GDS foca em features de implementação do jogo, destacando recursos disponibilizados em game engine diversas.

Inicialmente, trabalhou-se na criação do FEnDiGa Process, definindo quais atividades especializadas deveriam ser executadas para integrar e configurar os modelos NESI e GDS nos recursos OOFM definidos. Em seguida o FEnDiGa Framework foi construído, gerando *features*, *states*, *behaviors* e *observers* associados aos modelos NESI e GDS

utilizados. Também foram adaptadas três game engines Java para trabalharem com *states* definidos no FEnDiGa Framework. Ao final, o FEnDiGa Framework foi devidamente configurado e adaptado, gerando como resultado o jogo SimplifiedPacman funcionando em 3 motores de jogos distintos.

Assim, como resultado final deste capítulo, obteve-se uma SPL para jogos digitais capaz de gerar jogos concretos via configuração e adaptação de recursos OOFM modeladas para jogos digitais diversos. Trata-se de uma validação importante da capacidade da OOFM em gerar SPLs e sistemas concretos de domínios específicos, atendendo como resultado o aspecto de variabilidade "produção de sistemas". O próximo capítulo irá dar continuidade ao processo de verificação e validação da OOFM, através da análise de questões de corretude, representatividade, complexidade e usabilidade da OOFM. Uma análise comparativa da OOFM com demais técnicas de modelagem de features também será efetuado com relação aos aspectos de variabilidade identificados.

Capítulo

5

Este capítulo descreve a abordagem de verificação aplicada na técnica OOFM proposta. Basicamente, são analisadas: a capacidade de fornecer documentos OOFM corretos; a representatividade de árvores de features com recursos OOFM; a complexidade de modelos OOFM gerados; e a usabilidade de recursos OOFM na modelagem de features. Uma tabela comparativa baseada nos aspectos de variabilidade identificados também é apresentada neste capítulo.

5 AVALIANDO A TÉCNICA OOFM

Este capítulo apresenta importantes atividades de avaliação executadas na técnica OOFM proposta, as quais verificam a sua corretude, representatividade, complexidade, usabilidade e cobertura de aspectos de variabilidade identificados.

Para avaliar a corretude da OOFM, definiu-se um ambiente formal de avaliação da OOFM. Este OOFM Environment (Sarinho e Apolinário, 2012) é capaz de verificar a produção de modelos OOFM bem-formados segundo o Perfil OOFM + expressões OCL especificadas. Sua intenção é verificar se é possível gerar documentos OOFM corretos de acordo com as regras sintáticas e semânticas previamente definidas para a OOFM.

Com relação à representatividade da OOFM, efetuou-se um comparativo de elementos OOFM com recursos de técnicas de modelagem de features disponíveis. Um mapeamento formal de OOFM-XML para *Simple XML Feature Model* (SXFM) (SPLOT, 2012) também foi produzido. A intenção aqui é mostrar a capacidade da OOFM em gerar recursos de features compatíveis com demais técnicas de modelagem de features comparadas.

Métricas estruturais de modelagem de features (Bagheri e Gasevic, 2011) foram coletadas para avaliar a complexidade da OOFM. Comparativos de atividades de manutenção e de reuso de features OOFM com demais técnicas de modelagem de features também foram

efetuadas. Tratam-se de atividades avaliativas que buscam mostrar a similaridade da complexidade OOFM com relação às técnicas de modelagem de features avaliadas.

Um experimento também foi realizado com a OOFM no intuito de determinar a sua usabilidade na modelagem de árvores de features. Para tal, estudantes de Engenharia de Computação da Universidade Estadual de Feira de Santana (UEFS) foram convidados a trabalhar com a OOFM na representação de modelos de features novos e pré-existentes. Ao final da atividade responderam um questionário avaliativo sobre a técnica, verificando a sua Utilidade, Facilidade de Uso, Facilidade de Aprendizagem e Satisfação (Lund, 2001).

E para finalizar, uma tabela comparativa de aspectos de variabilidade entre a OOFM e técnicas de modelagem de features identificadas também foi elaborada. Sua intenção aqui é mostrar a cobertura que a OOFM oferece aos aspectos de variabilidade identificados.

5.1 GERANDO DOCUMENTOS CORRETOS

Considerando a produção de modelos OOFM válidos, é necessário verificar se:

- Árvores de features corretas podem ser modeladas ou não;
- Hierarquias OO corretas podem ser representadas pela técnica OOFM; e
- Modelos OOFM corretos são produzidos após a execução de operações de features.

Neste sentido, e analisando o uso do Perfil OOFM, levantou-se as seguintes questões a serem avaliadas:

- **Q1**: Árvores de features OOFM corretas podem ser geradas de acordo com as estruturas e regras definidas na OOFM?
- **Q2**: Hierarquias de features OOFM corretas podem ser geradas de acordo com as estruturas e regras definidas na OOFM?
- **Q3**: Operações OOFM definidas fornecem modelos OOFM corretos de acordo com seus resultados de saída esperados?

No intuito de avaliar estas questões, preparou-se um OOFM Environment (Sarinho e Apolinário, 2012). Trata-se de uma coleção de especificações que animam o Perfil OOFM proposto via interpretação do USE System (Gogolla, Büttner e Richters, 2007). OOFM Environment gera configurações de objetos corretas e esperadas de acordo com as estruturas e restrições OOFM propostas. Cada configuração de objetos representa uma instância de modelo OOFM, e uma configuração de objetos válida representa uma solução válida de modelo OOFM para uma estratégia de produção OOFM escolhida.

Com relação ao USE System, trata-se de um método formal lightweight para a geração automática de configurações de objetos grandes não triviais, capaz de ilustrar diagramas de classe complexos bem como trabalhar com cenários complexos projetados (Gogolla, Büttner e Richters, 2007).

```
1.  class FeatureType                       // represents a FM feature
2.  attributes
3.     name: String                         // main identification of a FeatureType
4.     lower: Integer                       // lower quantity of subfeatures
5.     upper: Integer                       // higher quantity of subfeatures
6.     parent: FeatureType                  // current FeatureType is a subftype of parent
7.  ...
8.  end
9.
10. class GroupType < FeatureType           // FeatureType specialization
11. attributes
12.    groupLower: Integer                  // lower quantity of groupedFeatures
13.    groupUpper: Integer                  // higher quantity of groupedFeatures
14. end
15.
16. class AttributeType                     // represents a feature attribute in a FM
17. attributes
18.    name: String                         // main identification of an AttributeType
19.    type: PossibleAttributeType          // suggested type for the attribute value
20. end
21.
22. class FeatureClass end                  // represents the <<feature-class>> stereotype
23.
24. // 0..1 FeatureType instances may have 0..* FeatureType instances  as subtypes
25. association Subtypes between FeatureType [0..1] FeatureType [0..*] role subtypes end
26.
27. // possible types of an feature attribute
28. enum PossibleAttributeType {string, integer, boolean}
29.
30. // one FeatureType instance may have 0..* AttributeType instances
31. association AttributeTypes between FeatureType [1] AttributeType [0..*] role attributeTypes end
32.
33. // many FeatureClass instances may have many FeatureClass instances as superClass
34. association SuperClass between FeatureClass[0..*]  FeatureClass[0..*] role superClass end
35. …
```

Figura 28 Representação textual de elementos do Perfil OOFM.

Com o objetivo de aplicar o USE System na técnica OOFM, primeiro é necessário representar modelos OOFM de uma maneira textual (Figura 28). Neste sentido, o USE

System propõe uma sintaxe textual simplificada representativa de modelos UML. Ela usa características definidas em diagramas UML (tais como classes, associações, entre outras), capazes de serem animadas e validadas pela respectiva suite. Expressões OCL também podem ser representadas nesta sintaxe textual, permitindo a especificação adicional de restrições de integridade nos modelos textuais definidos.

Uma descrição parcial dos elementos estruturais da OOFM na sintaxe USE System pode ser analisada na Figura 28. Nela são explanadas estruturas *FeatureType*, *GroupType* e *AttributeType* juntamente com suas associações *subtypes*, *PossibleAttributeType* e *attributeTypes*. Vale salientar que uma representação de classe UML ao invés de uma extensão da metaclasse *Class* (algo possível em Perfis UML) foi aplicada para o estereótipo <<*feature-class*>>. Isto é necessário uma vez que o USE System não suporta a criação direta de estereótipos, e uma solução para fornecer a semântica de *Class* para cada *FeatureClass* declarada precisa ser aplicada.

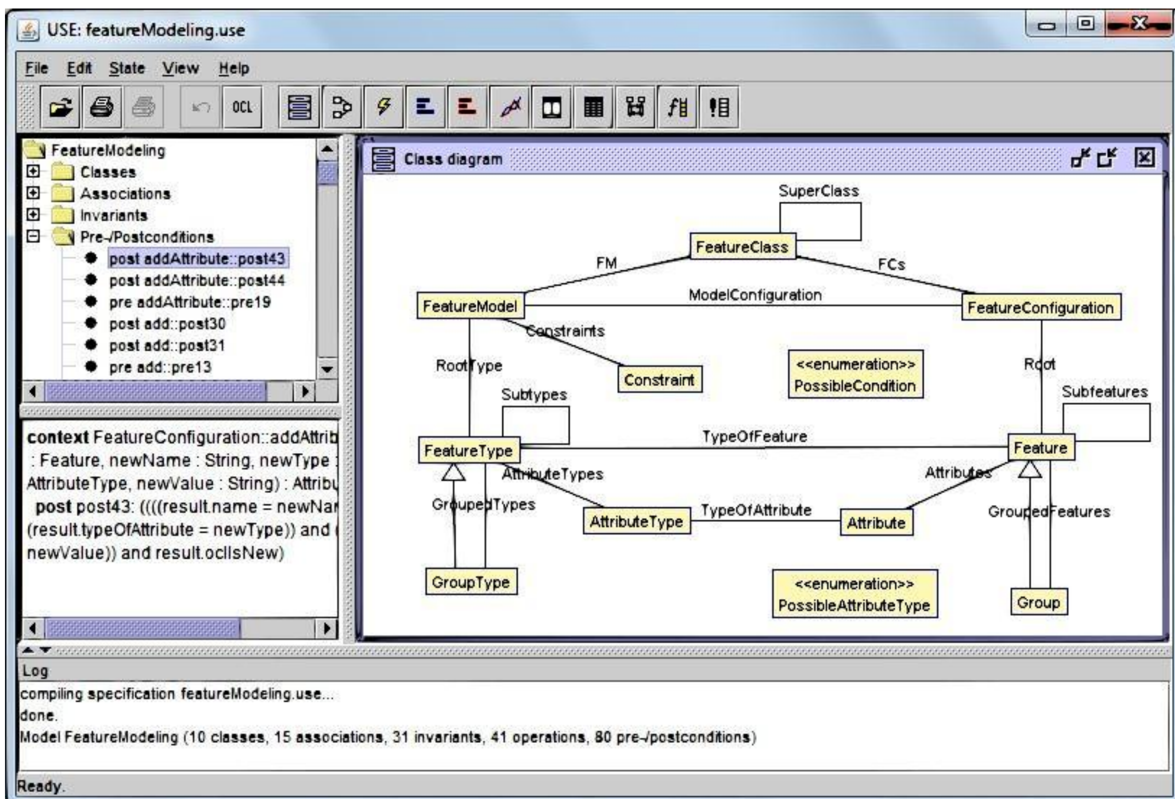


Figura 29 Compilação e criação de elementos estruturais, comportamentais e restritivos da OOFM via ferramento USE System.

Cada elemento OOFM codificado, segundo a sintaxe USE System, precisa ser devidamente compilado pela ferramenta USE System. Trata-se de uma atividade que valida à sintaxe do código produzido, preparando o modelo e as regras OCL propostas para gerarem configurações de objetos OOFM desejadas. A Figura 29 ilustra o resultado final deste processo de compilação, onde os elementos OOFM codificados são sintaticamente verificados na ferramenta USE System.

Com a definição de uma sintaxe textual da OOFM, o passo seguinte consiste na definição de procedimentos ASSL capazes de mudar o estado de modelos OOFM representados. Isto é necessário porque, como uma linguagem de especificação, expressões OCL representam um sistema sem efeitos colaterais (OMG, 2000). De fato, quando uma expressão OCL é avaliada, ela simplesmente retorna um valor, sem nenhuma alteração no modelo. Isto significa que o estado de um sistema nunca muda pelo uso da OCL, embora uma expressão OCL possa ser usada para especificar e validar uma mudança de estado aplicada.

De acordo com Gogolla, Bohling e Richters (2003), a tarefa de um procedimento ASSL é construir uma configuração de objetos ou parte dele através da sua interpretação via USE System. Cada configuração de objetos representa um novo estado do sistema para um modelo, o qual deve ser validado de acordo com as estruturas e restrições OOFM definidas para o modelo. De fato, quando uma configuração de objetos completa é construída via comandos ASSL, todas as invariantes são checadas, buscando definir se a configuração de objetos gerada pode ser aceita ou não. Se uma invariante falha, a configuração de objetos é descartada e a avaliação ASSL é encerrada. Uma outra opção é executar um *backtracking* se uma invariante falhar, onde uma possível alternativa de configuração de objetos no procedimento ASSL é escolhida para ser avaliada.

Alguns exemplos destas situações ASSL descritas são ilustradas no código a seguir (Figura 30), as quais mostram procedimentos ASSL definidos com base nas regras de restrição OOFM especificadas. Assim, para o procedimento *generateFeatureModels* (linhas 2-17), instâncias *Sequence* de *FeatureType* e *FeatureModel* são trabalhadas pelas variáveis *rootTypes* e *featureModels*. Sua intenção aqui é produzir e iterar instâncias *FeatureType* e *FeatureModel* disponíveis, produzindo como resultado novas árvores de features. Neste procedimento, comandos ASSL *CreateN* são usados para instanciar *n* objetos de um

respectivo tipo fornecido como parâmetro. Comandos ASSL *Insert* também são usados para inserir instâncias em associações definidas e disponíveis no modelo.

```
1.  // generate FeatureModel instances with their RootTypes
2.  procedure generateFeatureModels(n: Integer)
3.  var rootTypes: Sequence(FeatureType), featureModels: Sequence(FeatureModel),
4.                                                          featureClass: FeatureClass;
5.  begin
6.      rootTypes := CreateN(FeatureType, [n]);
7.      featureModels := CreateN(FeatureModel, [n]);
8.      for i:Integer in [Sequence{1..n}] begin
9.              [rootTypes->at(i)].name := ['rootType'];
10.             [rootTypes->at(i)].parent := [rootTypes->at(i)];
11.             [rootTypes->at(i)].lower := [1];
12.             [rootTypes->at(i)].upper := [1];
13.             Insert (RootType, [featureModels->at(i)], [rootTypes->at(i)]);
14.             featureClass := Try([FeatureClass.allInstances->asSequence]);
15.             Insert (FM, [featureClass], [featureModels->at(i)]);
16.     end;
17. end;
18.
19. // creates n FeatureTypes for a featureTree
20. procedure generateFeatureTypes(n: Integer)
21. var featureTypes: Sequence(FeatureType), parent: FeatureType, name: String;
22. begin
23.     featureTypes := CreateN(FeatureType, [n]);
24.     for i:Integer in [Sequence{1..n}] begin
25.             name := Any([Sequence{'Ada','Bel','Cam','Day','Eva','Flo','Gen','Hao','Ina','Jen'}]);
26.             parent := Try([FeatureType.allInstances->asSequence]);
27.             [featureTypes->at(i)].name:= [name];
28.             [featureTypes->at(i)].parent := [parent];
29.             [featureTypes->at(i)].lower := Any([Sequence{0..1}]);
30.             [featureTypes->at(i)].upper := Any([Sequence{1..3}]);
31.             Insert (Subtypes, [parent], [featureTypes->at(i)]);
32.     end;
33. end;
34.
35. // creates a new FeatureType and put it in the featureTree
36. procedure addFeatureType(featureTypeRef: FeatureType, newName: String,
37.                                               newLower: Integer, newUpper: Integer)
38. var newFeatureType: FeatureType;
39. begin
40.     newFeatureType := Create(FeatureType);
41.      [newFeatureType].name:= [newName];
42.      [newFeatureType].parent := [featureTypeRef];
43.      [newFeatureType].lower := [newLower];
44.      [newFeatureType].upper := [newUpper];
45.     Insert (Subtypes, [featureTypeRef], [newFeatureType]);
46. end;
```

Figura 30 Exemplos de procedimentos ASSL.

Considerando o procedimento *generateFeatureTypes* (linhas 20-33), uma *Sequence* de ***n*** instâncias *FeatureType* é tratada e inserida em árvores de features. Para tal, *FeatureTypes* criados e armazenados na variável *featureTypes* são configurados e inseridos na respectiva associação *Subtypes* (papel *subtypes* no Perfil OOFM). Para os comandos ASSL *Any* e *Try* executados neste procedimento, ambos fornecem um valor indefinido com base numa *Sequence* recebida como parâmetro. A diferença é o comportamento de *backtracking* do

comando *Try*, escolhendo outro valor da *Sequence* se a configuração de objetos produzida falha após a execução do procedimento.

Para o procedimento *addFeatureType* (linhas 36-46), uma produção simples e direta de configuração de objetos é efetuada. O objetivo deste procedimento é configurar os atributos da respectiva variável *newFeatureType* com os parâmetros *featureTypeRef*, *newName*, *newLower* e *newUpper* fornecidos. A inserção da variável *newFeatureType* como uma associação *Subtypes* do parâmetro *featureTypeRef* também é efetuada, via comando *Insert (Subtypes, [featureTypeRef], [newFeatureType])* descrito.

Além destes 3 procedimentos ASSL descritos na Figura 30, outros 35 importantes procedimentos ASSL também foram implementados (resumidamente descritos na Tabela 1). Tratam-se de procedimentos capazes de produzir configurações de objetos válidas segundo as regras de restrição OOFM propostas.

Tabela 1 Descrição resumida dos procedimentos ASSL implementados.

Procedimento ASSL	Descrição
generateFeatureClasses (n: Integer)	Constrói *n* instâncias de *FeatureClass*
generateFeatureClassHierarchy (n: Integer)	Constrói hierarquias de *n* instâncias de *FeatureClass*
generateFeatureModels (n: Integer)	Constrói *n* instâncias *FeatureModel* com seus respectivos *RootTypes*
generateFeatureConfigurations (n: Integer)	Para instâncias *FeatureModel* aleatórias, gerar *n* instâncias *FeatureConfiguration* com seus respectivos *Feature root* associados
generateFeatureTypes (n: Integer)	Constrói *n* instâncias *FeatureType* para árvores aleatórias de *FeatureType*
generateFeatures (n: Integer)	Constrói *n* instâncias *Feature* para árvores aleatórias de *Feature*
generateAttributeTypes (n: Integer)	Para instâncias *FeatureType* aleatórias, gerar *n* instâncias *AttributeType*
generateAttributes (n: Integer)	Para instâncias *Feature* aleatórias, gerar instâncias *Attribute*
generateConstraints (n: Integer)	Constrói possíveis instâncias *Constraint* para instâncias *FeatureModel* aleatórias
generateGroupTypes (n: Integer, groupSize: Integer)	Transforma instâncias *FeatureType* aleatórias em instâncias *GroupType*
generateGroups (n: Integer, groupSize: Integer)	Transforma instâncias *Feature* aleatórias em instâncias *Group*
addFeatureType (featureTypeRef: FeatureType, newName: String, newLower: Integer, newUpper: Integer)	Constrói uma nova instância *FeatureType* que será adicionada em *featureTypeRef*
addFeature (featureRef: Feature, newName: String, newType: FeatureType)	Constrói uma nova instância *Feature* que será adicionada em *featureRef*
addAttributeType (featureTypeRef: FeatureType, newName: String, newType: PossibleAttributeType)	Constrói uma nova instância *AttributeType* que será adicionada em *featureTypeRef*
addAttribute (featureRef: Feature, newName: String, newType: AttributeType, newValue: String)	Constrói uma nova instância *Attribute* que será adicionada em *featureRef*
addConstraint (featureModelRef: FeatureModel, newName: String, newCondition: PossibleCondition,	Constrói uma nova instância *Constraint* que será adicionada em *featureModelRef*

newFeatureType1: FeatureType, newFeatureName1: String, newFeatureType2: FeatureType, newFeatureName2: String)	
setToGroupType (featureTypeRef: FeatureType, newGroupLower: Integer, newGroupUpper: Integer, newGroupedTypes: Sequence(FeatureType))	Transforma instâncias *FeatureType* específicas em instâncias *GroupType*
setToGroup (featureRef: Feature, groupTypeRef: GroupType, newGroupedFeatures: Sequence(Feature))	Transforma instâncias *Feature* específicas em instâncias *Group*
setToFeatureType (groupTypeRef: GroupType)	Transforma instâncias *GroupType* específicas em instâncias *FeatureType*
setToFeature (groupRef: Group)	Transforma instâncias *Group* específicas em instâncias *Feature*
mergeFeatureType (featureType: FeatureType, featureTypeRef: FeatureType)	Aplica nas *subtypes* de *featureTypeRef* a *featureType* indicada
mergeFeature (feature: Feature, featureRef: Feature)	Aplica nas *subfeatures* de *featureRef* a feature indicada
mergeTree (featureType: FeatureType, featureTypeRef: FeatureType, featureRef: Feature)	Aplica nas *subtypes* de *featureTypeRef* a *featureType* indicada, juntamente com todas as suas subfeatures tipificadas
excludeFeatureType (featureTypeRef: FeatureType, featureModel: FeatureModel)	Exclui a *featureTypeRef* do *featureModel* indicado
excludeFeature (featureRef: Feature, featureConfiguration: FeatureConfiguration)	Exclui a *featureRef* da *featureConfiguration* indicada
removeAttribute (featureRef: Feature, attr: Attribute)	Exclui a instância *Attribute attr* da *featureRef* indicada
removeAttributeType (featureTypeRef: FeatureType, attributeType: AttributeType)	Exclui *attributeType* da *featureTypeRef* indicada
removeConstraint (featureModel: FeatureModel, constraintRef: Constraint)	Exclui *constraintRef* da *featureModel* indicada
copyFeatureType (fromFeatureType: FeatureType, toFeatureType: FeatureType)	Copia a feature *fromFeatureType* e adiciona o resultado em *toFeatureType.subtypes*
copyFeature (fromFeature: Feature, toFeature: Feature, finalFeatureType: FeatureType)	Copia a feature *fromFeature* e adiciona o resultado em *toFeature.subfeatures*
copyFeatureTypeTree (fromFeatureType: FeatureType, toFeatureType: FeatureType, featureModel: FeatureModel, depth: Integer)	Copia a árvore de features *fromFeatureType* e adiciona o resultado em *toFeatureType.subtypes*
copyTree (fromFeatureType: FeatureType, toFeatureType: FeatureType, toFeature: Feature, featureModel: FeatureModel, depth: Integer)	Copia a árvore de features *fromFeatureType* e adiciona o resultado em *toFeatureType.subtypes*, juntamente com suas *subfeatures* que são adicionadas em *toFeature.subfeatures*
moveFeatureType (fromFeatureType: FeatureType, toFeatureType: FeatureType)	Move a feature *fromFeatureType* para *toFeatureType.subtypes*
moveTree (fromFeatureType: FeatureType, toFeatureType: FeatureType, toFeature: Feature)	Move a feature *fromFeatureType* para *toFeatureType.subtypes*, juntamente com suas *subfeatures* que são movidas para *toFeature.subfeatures*
cloneFeatureType (featureTypeRef: FeatureType, featureModel: FeatureModel, depth: Integer)	Clona a feature *fromFeatureType* indicada juntamente com seus *subtypes* até a *depth* indicada
cloneTree (featureTypeRef: FeatureType, featureModel: FeatureModel, depth: Integer)	Clona a feature *fromFeatureType* indicada juntamente com seus *subtypes* e *subfeatures* até a *depth* indicada
setAttribute (attr: Attribute, value: String)	Atualiza o valor *name* de *attr*
update (featureRef: Feature, value: String)	Atualiza o valor *name* de *featureRef*

A partir destes procedimentos ASSL descritos, Scripts ASSL podem ser usados para produzir configurações válidas de objetos OOFM em diferentes estratégias de produção. Em outras palavras, scripts de avaliação podem ser preparados para responder as questões **Q1**, **Q2** e **Q3** propostas para a OOFM.

Assim, considerando a questão **Q1**, a idéia é verificar se a abordagem OOFM é capaz de fornecer pelo menos uma instância *FeatureModel* correta, juntamente com a sua respectiva instância *FeatureConfiguration* correta (Sarinho e Apolinário, 2012). Para tal,

expressões OCL definidas foram exploradas numa estratégia de criação de múltiplos e imprevisíveis snapshots de modelos OOFM. Neste sentido, um script ASSL foi especificado (Figura 31) para gerar elementos OOFM aleatórios e corretos via procedimentos ASSL declarados (Tabela 1) (Sarinho e Apolinário, 2012). Como resultado, múltiplas árvores de features distintas e corretas foram geradas nas avaliações deste script ASSL. Ou seja, é possível gerar árvores de features corretas segundo as regras formais definidas para a OOFM.

```
1.  // initiates all loaded elements
2.  use> reset
3.  // loads the oofm profile
4.  use> open featureModeling.use
5.  // ASSL procedure call
6.  use> gen start buildFeatureModel.assl generateFeatureClasses(1)
7.  // accept the valid snapshot provided by the procedure evaluation
8.  use> gen result accept
9.  use> gen start buildFeatureModel.assl generateFeatureModels(1)
10. use> gen result accept
11. use> gen start buildFeatureModel.assl generateFeatureConfigurations(1)
12. use> gen result accept
13. use> gen start buildFeatureModel.assl generateFeatureTypes(6)
14. use> gen result accept
15. use> gen start buildFeatureModel.assl generateFeatures(4)
16. use> gen result accept
17. use> gen start buildFeatureModel.assl generateAttributeTypes(3)
18. use> gen result accept
19. use> gen start buildFeatureModel.assl generateAttributes(3)
20. use> gen result accept
21. use> gen start buildFeatureModel.assl generateConstraints(1)
22. use> gen result accept
23. use> gen start buildFeatureModel.assl generateGroupTypes(1,2)
24. use> gen result accept
25. use> gen start buildFeatureModel.assl generateGroups(1,2)
26. use> gen result accept
```

Figura 31 Script ASSL para a geração de árvores de features OOFM.

Para a questão **Q2**, uma estratégia de produção de *FeatureClasses* hierárquicas foi aplicada (Sarinho e Apolinário, 2012). Ela verifica a possibilidade de fornecer pelo menos um conjunto de instâncias *FeatureModel* para uma hierárquica de *FeatureClasses*. A idéia consiste em estressar a invariante OCL responsável por verificar se existe nenhuma diferença entre *FeatureModels* herdados de *FeatureClasses* numa hierárquica. Como resultado, múltiplas instâncias *FeatureClass*, *FeatureModel* e *FeatureType* foram criadas e avaliadas no script ASSL (Figura 32), gerando hierarquias de *FeatureClasses* distintas, corretas e aleatórias. Ou seja, é possível gerar hierarquias de features corretas segundo as regras formais definidas para a OOFM.

```
1.    use> reset
2.    use> open featureModeling.use
3.    use> gen start buildFeatureModel.assl generateFeatureClassHierarchy(3)
4.    use> gen result accept
5.    use> gen start buildFeatureModel.assl generateFeatureModels(3)
6.    use> gen result accept
7.    use> gen start buildFeatureModel.assl generateFeatureTypes(6)
8.    use> gen result accept
```

Figura 32 Script ASSL para a geração de hierarquias de features OOFM.

Finalmente, considerando a questão **Q3**, é necessário validar as pré-condições e pós-condições definidas para as operações de features da OOFM. Para tal, scripts ASSL que executam operações OOFM específicas durante a produção de snapshots OOFM desejados foram implementados. Assim, 12 scripts ASSL foram desenvolvidos como casos de teste para as operações de features OOFM propostas (resumidamente descritos na Tabela 2). Tratam-se de scripts de teste capazes de verificar as restrições OCL de manipulação de features propostas no Perfil OOFM.

Tabela 2 Descrição resumida dos scripts de teste das operações de features da OOFM.

Scripts de Teste	Descrição Resumida
creatingTreeTC.cmd	Constrói uma árvore de features inicial, através da criação aleatórica de instâncias *FeatureClass*, *FeatureModel*, *FeatureConfiguration*, *FeatureType*, *Feature*, *Contraint*, *AttributeType* e *Attribute*.
excludingFeaturesTC.cmd	A partir da árvore de features criada em *creatingTreeTC*.cmd, este script exclui instâncias *FeatureType*, *Feature*, *Contraint*, *AttributeType* e *Attribute* disponíveis.
copingFeatureTypesTC.cmd	A partir da árvore de features criada em *creatingTreeTC*.cmd, este script copia toda a árvore de instâncias *FeatureType* a partir de *f1Type* para ela mesma (*subtypes* de *f1Type*).
copyTreeTC.cmd	Similar ao script *copingFeatureTypesTC*.cmd, mas agora incluindo a cópia das instâncias *Feature* associadas a *f1Type*.
copyTree2TC.cmd	A partir da árvore de features criada em *creatingTreeTC*.cmd, este script copia a árvore de features partir de *f2Type* para as features *rootType* e *root* da mesma árvore de features.
movingFeatureTypesTC.cmd	A partir da árvore de features criada em *creatingTreeTC*.cmd, este script move a *FeatureType f3Type* para a *FeatureType f2Type*.
moveTreeTC.cmd	A partir da árvore de features criada em *creatingTreeTC*.cmd, este script move a árvore de features partir de *f2Type* para as features *rootType* e *root* da mesma árvore de features.
cloningFeatureTypesTC.cmd	A partir da árvore de features criada em *creatingTreeTC*.cmd, este script transforma *f1Type* em *GroupType* (instâncias *FeatureType f2Type* e *f3Type* como *groupedTypes*) e clona toda a árvore de instâncias *GroupType* de *f1Type*.
cloneTreeTC.cmd	A partir da árvore de features criada em *creatingTreeTC*.cmd, este script transforma *f1Type* em *GroupType* (instâncias *FeatureType f2Type* e *f3Type* como *groupedTypes*), e *f1* em *Group* (instâncias *Feature f21* e *f22* como *groupedFeatures*). No final, toda a árvore de features e de grupos a partir do *GroupType f1Type* e do *Group* f1 são clonadas.

cloneTree2TC.cmd	A partir da árvore de features criada em *creatingTreeTC*.cmd, este script clona toda a árvore de features de *f2Type*.
mergeTreeTC.cmd	Aplica a árvore de features criada em *cloneTreeTC*.cmd na *FeatureType f1Type* (*subtypes* de *rootType*) e na *Feature f1* (*subfeatures* de *root*).
groupingFeaturesTC.cmd	A partir da árvore de features criada em *creatingTreeTC*.cmd, este script agrupa e desagrupa as features *f1Type* e *f1* criadas.

Com o objetivo de ilustrar o processo de avaliação de um destes scripts ASSL, o código a seguir (Figura 33) apresenta uma descrição parcial da execução do script *creatingTreeTC.cmd*. Basicamente, três passos são apresentados neste código: 1) a criação das instâncias *FeatureClass*, *FeatureModel* e *FeatureConfiguration* com as respectivas features *rootType* e *root* de uma maneira aleatória; 2) a inclusão de uma nova *FeatureType* chamada "*f1Type*" como *subtypes* de *rootType*; e 3) a inclusão de uma nova instância *Feature* chamada "*f1*" como *subfeatures* de *root*.

Neste código (Figura 33), também é possível verificar que os comandos *!openter* e *!opexit* são usados para aplicar as pré-condições e pós-condições de restrição nos passos do script. A idéia é verificar se os parâmetros fornecidos e as configurações de objetos produzidas no procedimento ASSL seguem as restrições OCL propostas para a respectiva operação de features OOFM.

```
1.  // preparing an initial feature tree
2.  use> reset
3.  use> open featureModeling.use
4.  use> gen start buildFeatureModel.assl generateFeatureClasses(1)
5.  use> gen result accept
6.  use> gen start buildFeatureModel.assl generateFeatureModels(1)
7.  use> gen result accept
            8.  use> gen start buildFeatureModel.assl generateFeatureConfigurations(1)
9.  use> gen result accept
10.
11. // adding FeatureType
12. use> !openter FeatureModel.allInstances->asSequence->first add(FeatureType.allInstances->
13.                 select (name='rootType')->asSequence->first, 'f1Type', 1, 1)
14. use> gen start featureModelingOperations.assl addFeatureType(FeatureType.allInstances->
15.                 select(name='rootType')->asSequence->first, 'f1Type', 1, 1)
16. use> gen result accept
17. use> !opexit FeatureType.allInstances->select(name='f1Type')->asSequence->first
18.
19. // adding Feature
20. use> !openter FeatureConfiguration.allInstances->asSequence->first
21.         add(Feature.allInstances-> select(name='root')->asSequence->first, 'f1',
22.                 FeatureType.allInstances->select(name='f1Type')->asSequence->first)
23. use> gen start featureModelingOperations.assl addFeature(Feature.allInstances->
24.                 select(name='root')->asSequence->first, 'f1',
25.                 FeatureType.allInstances->select(name='f1Type')->asSequence->first)
26. use> gen result accept
27. use> !opexit Feature.allInstances->select(name='f1')->asSequence->first
28. ...
```

Figura 33 Descrição parcial do Script ASSL *creatingTreeTC.cmd*.

Como resultado final da avaliação dos 12 scripts ASSL especificados, árvores de features corretas e esperadas foram representadas após a avaliação de cada um. Ou seja, resultados de saída corretos e esperados foram confirmados para cada manipulação de feature definida na OOFM.

5.2 MODELANDO RECURSOS COMPATÍVEIS

Considerando a representatividade de árvores de features, é importante verificar se um modelo OOFM pode apresentar informações de features compatíveis com técnicas de modelagem de features existentes. Na tentativa de responder esta questão, estruturas e operações OOFM propostas devem ser associadas a recursos de modelagem de features equivalentes disponíveis. Para tal, três questões específicas devem ser avaliadas:

- **Q4**: OOFM consegue representar as mesmas estruturas e relacionamentos de features disponíveis nas técnicas tradicionais de modelagem de features?
- **Q5**: OOFM permite atividades similares de atualização de FMs e FCs?
- **Q6**: OOFM permite abordagens similares de reuso de FMs e FCs?

Neste sentido, para a questão **Q4**, elementos e relacionamentos representativos de features modelados em FMs e FCs tradicionais tais como features, grupos, agrupamento de features e atributos foram fornecidos por modelos OOFM equivalentes. Numa breve análise do que foi abordado até o momento neste trabalho, é possível verificar que estruturas identificadas em FODA, CBFM e Forfamel (principais abordagens de documentação de árvores de features) podem ser modeladas por estruturas OOFM correspondentes (feature num FM → *FeatureType*, feature num FC → *FeatureType* direta ou *Feature* indireta, entre outras).

Figura 34 Modelo OOFM equivalente ao domínio de e-shop via CBFM.

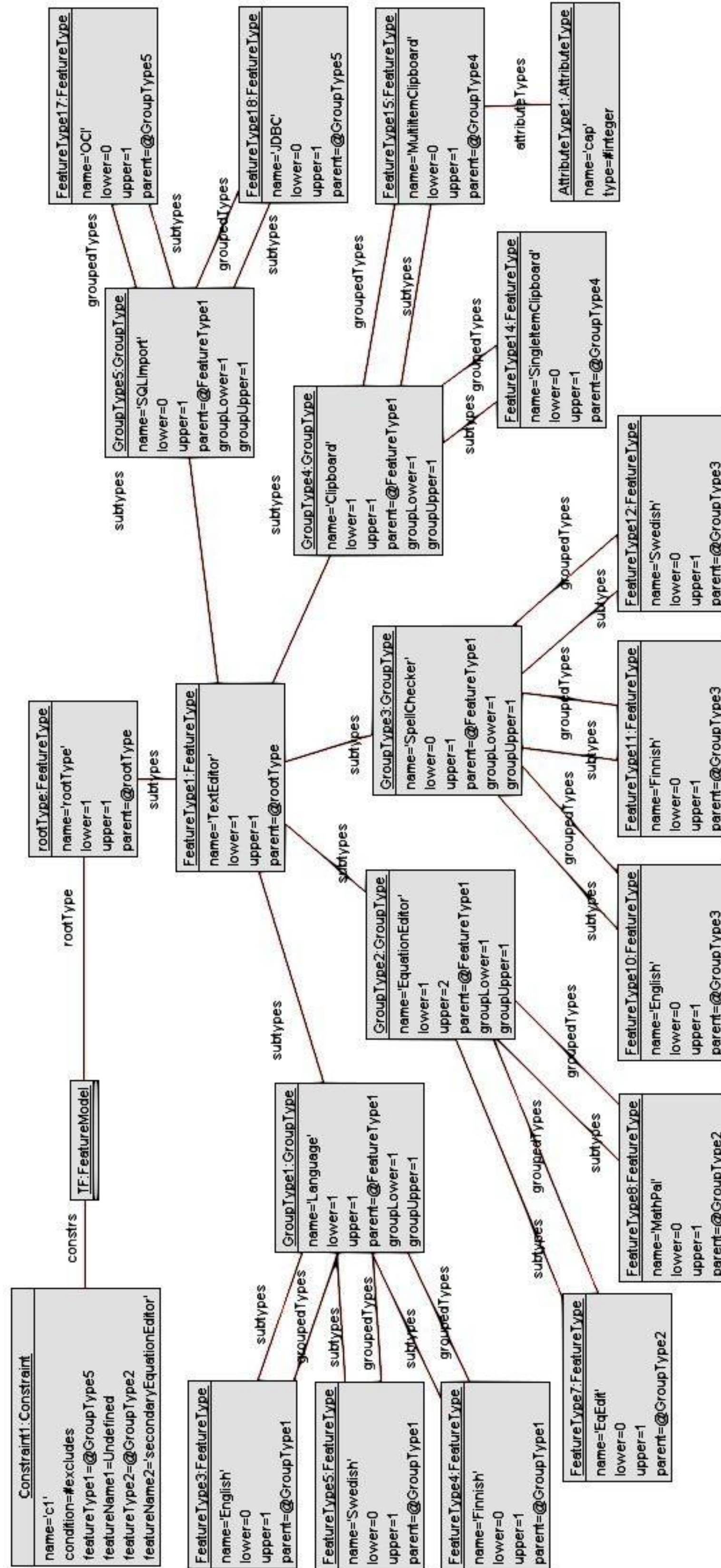


Figura 35 Modelo OOFM equivalente ao domínio de editores de texto via Forfamel.

Buscando demonstrar esta capacidade de representação de elementos representativos de FMs e FCs tradicionais, um modelo OOFM equivalente foi produzido para cada abordagem de documentação de features identificada. Neste sentido, as Figuras 12, 34 e 35 ilustram exemplos de modelos OOFM representativos do domínio de automóveis FODA (Deursen e Klint, 2002), do domínio de e-shop CBFM (Czarnecki e Kim, 2005) e do domínio de editores de texto Forfamel (Asikainen, Männistö e Soininen, 2006), respectivamente. Como resultado, elementos e relacionamentos representativos de features foram similarmente modelados dentro dos recursos disponibilizados pela técnica OOFM proposta.

Um transformador XSL de OOFM-XML para *Simple XML Feature Model* (SXFM) também foi produzido como resposta para a questão **Q4**. Trata-se de uma abordagem formal para a representação de recursos equivalentes de features via estruturas OOFM. SXFM é mantido pelo Software Product Lines Online Tools (SPLOT, 2012), um ambiente web para modelar, validar e armazenar árvores de features. Como resultado, demonstrou-se que elementos e relacionamentos de features equivalentes podem ser formalmente obtidos com a técnica OOFM proposta.

```
// provide root, grouped, optional and mandatory features based on FeatureType values
<xsl:template match="FeatureType">
  <xsl:for-each select="ancestor::node()[name()!='FeatureClass' and name()!='FeatureModel']">&#09;</xsl:for-each>
  <xsl:choose>
    <xsl:when test="parent::node()[name()='GroupType']">: </xsl:when>
    <xsl:when test="@name='RootType'">:r </xsl:when>
    <xsl:when test="@lower=0">:o </xsl:when>
    <xsl:otherwise>:m </xsl:otherwise>
  </xsl:choose>
  <xsl:choose>
    <xsl:when test="@name='RootType'">root (root_id)</xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="@name"/>
      (id_<xsl:value-of select="@name"/>_<xsl:value-of select="generate-id(.)"/>)
    </xsl:otherwise>
  </xsl:choose>
  <xsl:apply-templates/>
</xsl:template>

// apply GroupType values according to each identation level
<xsl:template match="GroupType">
  <xsl:for-each select="ancestor::node()[name()!='FeatureClass' and name()!='FeatureModel']">&#09;</xsl:for-each>
  :g [<xsl:value-of select="@groupLower"/>, <xsl:value-of select="@groupUpper"/>]
  <xsl:apply-templates/>
</xsl:template>
```

Figura 36 Descrição parcial do transformador XSL de OOFM para SXFM.

A Figura 36 ilustra alguns trechos do transformador XSL proposto, descrevendo a produção de árvores de features SXFM com base nas instâncias *FeatureType* e *GroupType*. Já

a Figura 37 ilustra o resultado da transformação do OOFM-XML para o domínio de automóveis (Figura 13) via transformador XSL proposto (Figura 33).

```
<?xml version="1.0" encoding="UTF-8"?>
<feature_model name="Feature Model">
	<feature_tree>
		:r root (root_id)
			:g [1, 1]
				: Automatic (id_Automatic_N65558)
				: Manual (id_Manual_N65564)
			:g [1, 2]
				: Gasoline (id_Gasoline_N65578)
				: Electric (id_Electric_N65584)
	</feature_tree>
	<constraints>
		C1:~id_Electric_N65584 or ~id_Automatic_N65558
	</constraints>
</feature_model>
```

Figura 37 Modelo SPLOT equivalente ao modelo OOFM de automóveis gerado via transformador XSL proposto.

Para a questão **Q5**, operações de features de técnicas de modelagem de features foram comparadas com operações OOFM definidas (Sarinho e Apolinário, 2012). A idéia foi verificar se operações OOFM conseguem fornecer FMs e FCs similares de acordo com as operações de features disponíveis nas técnicas Staged Configuration (Czarnecki, Helsen e Eisenecker, 2004), Multi-Level Feature Tree (Reiser e Weber, 2006), Feature-Architecture Mapping (Sochos, Riebisch e Philippow, 2006) e Feature Refactoring (Alves et al., 2006). A A Tabela 3 ilustra estas possíveis associações, mostrando como resultado a capacidade da OOFM de executar manutenções em árvores de features equivalentes de acordo com as técnicas de modelagem de features comparadas.

Tabela 3 Operações OOFM equivalentes a operações de modelagem de features comparadas.

Técnica de Modelagem de Features	Operação de Feature Disponível	Manipulação de Feature OOFM Equivalente
Staged Configuration	refining a feature cardinality	*add*, *exclude* ou *copy*
	refining a group cardinality	*setToGroupType* e *setToFeatureType*
	removing a grouped feature from a group	*setToGroupType* e *setToFeatureType*
	selecting a grouped feature from a group	*setToGroupType* e *setToFeatureType*
	assigning a value to an attribute	*setAttribute*
	cloning a solitary subfeature	*clone*

Multi-Level Feature Tree	change name (FC)	*update*
	change name (FM)	Combinação de *exclude* e *add FeatureType*
	change attribute	*setAttribute*
	change cardinality	Combinação de *exclude* e *add FeatureType*
	Removal	*exclude para* todas as *subfeatures* disponíveis
	Move	*move*
	Refinement	*add*, *copy* ou *move*
	Reduction	*exclude* ou *move*
	Regrouping	*setToGroupType* e *setToFeatureType*
Feature-Architecture Mapping	Adding	*add*
	Integrating	*merge*
	Dividing	Combinação de *clone* e *exclude*
	Reordering	*move*
Feature Refactoring	Collapse refactoring	*setToGroupType* e *setToFeatureType*
	Add refactoring	*add*
	Convert Mandatory to Optional	Combinação de *exclude* e *add FeatureType*
	Convert Alternative to Optional	*setToFeatureType*
	Pull and Push refactoring	*move*
	Remove formula	*removeConstraint*
	Merge refactoring	*merge*
	Replace refactoring	Combinação de *exclude* e *add FeatureType*

Considerando a questão **Q6**, abordagens de reuso de features devem ser comparadas com as possibilidades de reuso de features propostas pela OOFM (Sarinho e Apolinário, 2012). Neste sentido, a técnica Multi-Level Feature Trees (Reiser e Weber, 2006), descreveu como reusar modelos de features através de associações propostas entre instâncias *FeatureModel* e *Feature* declaradas, as quais definem quais tipos de features referências podem ser reusadas por features referenciadoras. Usando a técnica OOFM, o mesmo resultado pode ser obtido por um *merge* de partes clonadas de um FM "referência" para outro FM disponível, criando assim features referenciadoras de features referência clonadas.

Também, para a técnica Context Variability Model (Hartmann e Trew, 2008), esta utiliza um FM para configurar outros FMs num outro nível de abstração (neste caso num nível de contexto). Como as tradicionais restrições de FMs (*requires* e *excludes*) são usadas para definir um FM como nível de contexto na Context Variability Model, instâncias *Constraint* da técnica OOFM também podem ser aplicadas em FMs e FCs associados para criar um nível de contexto com resultados similares.

Mais ainda, para o Composition Model (Rosenmüller e Siegmund, 2010), tem-se que FMs modelados podem ser reusados por relacionamentos UML de associação e de herança entre FMs distintos. OOFM também permite tal reuso de FMs via relacionamentos UML disponíveis, através de estruturas *FeatureClass* criadas que pode ser associadas e herdadas por diferentes tipos de instâncias *FeatureClass* desejadas. Como resultado, tem-se que as estratégias de encapsulamento e de reuso do estado-da-arte da modelagem de features apresentadas podem ser totalmente cobertas pela técnica OOFM proposta.

5.3 AVALIANDO ATRIBUTOS DE COMPLEXIDADE

Considerando a complexidade da técnica OOFM, é importante verificar se os resultados de representação, manipulação e reuso de features com a OOFM são mais difíceis de serem os obtidos do que com as técnicas tradicionais de modelagem de features. Com base neste argumento, pode-se levantar as seguintes questões:

Q7: Um modelo OOFM é mais difícil de ser estruturalmente representado?

Q8: Um modelo OOFM é mais difícil de ser manipulado?

Q9: Um modelo OOFM é mais difícil de ser reusado?

Com o objetivo de comparar a complexidade estrutural de modelos OOFM com relação às abordagens tradicionais de modelagem de features (questão **Q7**), se faz necessário quantificar quantos elementos e relacionamentos de features são necessários para representá-los. Entretanto, alguns poucos estudos preliminares de definição de métricas estruturais de FMs, sem as devidas avaliações teóricas e empíricas necessárias, foram conduzidos nos últimos anos (Bagheri e Gasevic, 2011).

Tentando cobrir o máximo de características estruturais de FMs e FCs possíveis, tem-se que algumas métricas de modelagem de features foram propostas por Bagheri e Gasevic (2011), tais como: Number of Features (NF), Number of Top Features (NTop), Number of Leaf Features (NLeaf), Depth of Tree (DT), Cyclomatic Complexity (CC), Cross-

Tree Constraints (CTC), Ratio of Variability (RoV), Coefficient of Connectivity (CoC), Flexibility of Configuration (FoC) e Number of Valid Configurations (NVC).

No geral, tratam-se de métricas de features que avaliam atributos de qualidade de sistemas para FMs projetados (e.g. analisabilidade, modificabilidade, compreensibilidade) (Al-Kilidar, Cox e Kitchenham, 2005), mostrando seu impacto casual em aspectos de escala de modelagem de features (Bagheri e Gasevic, 2011). Bagheri e Gasevic (2011) também identificaram as métricas NLeaf, CC, NVC e FoC como um subconjunto de métricas suficientes para avaliar a complexidade da manutenabilidade de FMs projetados.

Com base nestas medidas estruturais, tem-se que a complexidade estrutural de modelos OOFM pode ser tanto avaliada como comparada com relação a demais abordagens de modelagem de features existentes. A Tabela 4 mostra a avaliação destas métricas com base nos modelos OOFM (Figuras 34 e 35) e em seus respectivos FMs equivalentes.

Tabela 4 Métricas de modelagem de features para modelos OOFM equivalentes.

	FMs Modelados			
	Forfamel	**OOFM-Forfamel**	**CBFM**	**OOFM-CBFM**
NF	15	18	[23,∞]	[23,∞]
NTop	5	5	2	2
NLeaf	9	12	[12,∞]	[12,∞]
CC	1	1	0	0
CTC	2/15	2/18	0	0
RoV	15/6	17/6	[19/8,∞]	[19/8,∞]
CoC	15/15	29/18	22/23	28/23
FoC	11/15	14/18	11/23	11/23
NVC	[288,∞]	[288,∞]	[896,∞]	[896,∞]
DT	3	3	6	6

Como resultado, considerando a produção mínima de possíveis features, tem-se que números similares de métricas de features foram encontrados para os elementos representativos de FMs (Tabela 4). Também é possível verificar resultados similares de manutenabilidade, segundo as métricas NLeaf, CC, NVC e FoC, entre os modelos de features CBFM e Forfamel e seus modelos OOFM equivalentes propostos.

Todavia, algumas exceções podem ser identificadas para métricas de elementos de grupo, onde um relacionamento extra de *groupedType* é modelado para cada feature

agrupada. Também, para os resultados do modelo OOFM-Forfamel, tem-se que instâncias extras de *FeatureType* foram modeladas com o intuito de evitar referências circulares entre as instâncias *SpellCheck* e *Language* (Asikainen, Männistö e Soininen, 2006). Como resultado, é possível verificar o uso de alguns elementos estruturais extras em modelos OOFM projetados, quando comparados com as abordagens tradicionais de modelagem de features existentes.

Considerando a questão **Q8**, é necessário analisar o uso das operações de features OOFM. Neste sentido, é possível verificar que operações de features OOFM podem ser usadas de uma maneira simples e direta na maioria das operações de features identificadas (Tabela 3).

Contudo, também é possível verificar que seqüências (iteradas ou não) de operações de features e métodos de features OOFM acabam se tornando necessárias para algumas operações de features. Como exemplo, a operação *dividing* proposta na Feature-Architecture Mapping (FArM) só pode ser obtida com a combinação das operações *clone* e *exclude* da OOFM. Já a operação *removal* proposta na Multi-Level Feature Trees precisa efetuar várias chamadas *exclude* para cada subfeature identificada na feature que vai ser removida. Como resultado, manipulações OOFM são tão simples quanto às manipulações de FMs tradicionais existentes (aplicação direta), exceto quando composições de operações de features OOFM são necessárias para representar uma atualização de FM similar.

Outro aspecto que precisa ser avaliado com relação à questão **Q8** se refere às limitações estruturais aplicadas na manutenção de FMs quaisquer. De fato, para cada modelo OOFM, se um FM representado possui pelo menos um FC associado, toda e qualquer atualização de FM não pode invalidar seus respectivos FCs associados. Operações tradicionais de manipulação de features não possuem este problema, uma vez que são baseadas nas técnicas FODA e CBFM as quais usam o mesmo modelo para representar FMs e FCs. Como resultado, apenas a técnica OOFM possui restrições na manutenção de FMs quando houver FCs associados pré-existentes (Forfamel não apresentou operações de manipulação de features).

Finalmente, para a questão **Q9**, analisando a técnica Context Variability Model, é possível verificar que seu nível de contexto pode ser diretamente representado por instâncias *Constraint* da OOFM. Também, para a técnica Multi-Level Feature Trees, FMs referência e referenciados podem ser produzidos de uma maneira similar através de sequências de

operações *merge* e *clone* da OOFM. Mais ainda, para o Composition Model, FMs associados e herdados podem ser diretamente representados por FMs disponíveis em cada instância *FeatureClass* herdada ou associada.

Como resultado, o encapsulamento e o reuso OOFM pode ser aplicado de uma maneira simples e similar de acordo com o encapsulamento e o reuso estado-da-arte da modelagem de features existente. Contudo, como herança de FMs e dependências de FCs são tratadas pela técnica OOFM, uma rigorosa validação estrutural é esperada para cada instância OOFM modelada. Desta forma, pode-se ter um maior custo na produção de hierarquias de FMs e FCs OOFM com relação às atuais abordagens de encapsulamento e reuso de features aplicadas.

5.4 AVALIANDO A USABILIDADE NA MODELAGEM

Considerando o uso da OOFM, é importante verificar a sua usabilidade na produção de árvores de features quaisquer. Para tal, o *FeatureClass Editor* foi avaliado em um experimento de usabilidade com 24 estudantes de Engenharia de Computação da Universidade Estadual de Feira de Santana (UEFS) durante um curso de Análise de Sistemas no período 2012.1.

Inicialmente, a ferramenta *FeatureClass Editor* foi introduzida numa atividade de replicação do exemplo OOFM para o domínio de automóveis (Figura 14). A intenção aqui foi familiarizar os alunos com os comandos e a forma de uso do *FeatureClass Editor* no desenvolvimento de uma árvore de features simples e pré-definida.

Em seguida, eles receberam instruções para modelar novos FMs e FCs para jogos digitais clássicos, tais como *Pacman* e *Pitfall*, usando a respectiva ferramenta. Uma atividade prévia para modelar estes jogos digitais usando UML foi realizada anteriormente neste curso.

Ao final, os alunos foram convidados a responder um questionário de avaliação sobre a usabilidade da ferramenta na produção de árvores de features OOFM. Este questionário foi baseado nas 30 questões USE propostas por Lund (2001) as quais focam na

avaliação de importantes domínios de usabilidade, tais como: Utilidade, Facilidade de Uso, Facilidade de Aprendizagem e Satisfação.

A Tabela 5 ilustra a média das respostas obtidas neste experimento, numa escala de *discordo*, *neutro* e *concordo* para cada questão de usabilidade respondida. Dentre os resultados obtidos, a maioria dos estudantes *concorda* com a usabilidade da OOFM para os domínios de usabilidade avaliados.

Tabela 5 Média dos resultados do questionário de usabilidade da OOFM.

	Discordo	Neutro	Concordo
Utilidade	9%	11%	80%
Facilidade de Uso	17%	20%	64%
Facilidade de Aprendizagem	17%	7%	76%
Satisfação	24%	20%	56%

Os melhores resultados de usabilidade foram obtidos para as questões de: produtividade, efetividade, facilidade de uso, "atinge as necessidades", aprendizado rápido e satisfação. Trata-se de uma consequência direta da simplicidade e dos poucos comandos necessários para se modelar FMs e FCs OOFM com o *FeatureClass Editor*. Já os piores resultados foram obtidos nas questões de: material de suporte, inconsistência e de "emoção" (e.g. maravilhosa, prazerosa, "sinto que eu precisava dela") sobre a ferramenta. Uma consequência direta da falta de documentação e bug encontrados na ferramenta durante o experimento.

5.5 COMPARANDO ASPECTOS DE VARIABILIDADE

Como descrito previamente, diferentes tipos de técnicas de modelagem de features têm sido propostas nos últimos anos. Elas apresentaram importantes aspectos de variabilidade bem como extensões diversas de técnicas de modelagem de features pré-existentes. Esta seção apresenta algumas destas técnicas especializadas, descrevendo suas idéias gerais, benefícios e

limitações. No final, características importantes destas técnicas foram sumarizadas, mostrando uma comparação entre seus aspectos de variabilidade e a técnica OOFM proposta.

A primeira técnica descrita é a abordagem Staged Configuration proposta por Czarnecki, Helsen e Eisenecker (2004). Ela objetiva reduzir a complexidade de FMs modelados pela aplicação de diferentes transformações de modelagem de features. Assim, uma versão parcialmente configurada de um modelo de features global é fornecido por esta técnica, quando trabalhada no contexto de uma sub-linha de produto particular. Contudo, pelo uso deste mecanismo de scale-down, apenas especializações do modelo global podem ser obtidas, e.g. pela "aproximação de cardinalidades ou pelo mascaramento de features desnecessárias, enquanto que outras formas de alterações locais são frequentemente necessárias" (Reiser e Weber, 2006).

Reiser e Weber (2006) descreveram como representar FMs usando modelos referência-referenciados. A idéia foi trocar a representação de modelos globais e extremamente largos por modelos independente e formalmente interrelacionados. Como resultado, obteve-se uma nova forma de gerenciar famílias de produto altamente complexas via Multi-Level Feature Trees (Reiser e Weber, 2006). Todavia, considerando as relações hierárquicas propostas entre features, é possível verificar que FMs herdados podem excluir features reusadas do FM original da hierarquia. Assim uma relação fraca entre features hierárquicas é obtida por esta abordagem de modelagem de features.

Hartmann e Trew (2008) propuseram um FM para contextualizar outros FMs usando a mesma estratégia de relacionamentos entre features (*requires* e *excludes*) em outro nível de abstração (o nível de contexto). O Context Variability Model (CVM) resultante restringe FMs, tornando possível a produção de MPLs que suportam várias dimensões no espaço de contexto (Hartmann e Trew, 2008). Contudo, atualizações em FMs restringidos podem invalidar o CVM configurado, sendo necessário um conjunto específico de operações de features para evitar a produção de árvores de features inconsistentes.

A abordagem Composition Model (Rosenmüller e Siegmund, 2009) foi proposta para projetar e configurar MPLs via recursos OO. Ela foi baseada em relacionamento de classes UML, tais como agregação e herança, as quais são aplicadas na associação de SPLs distintos baseados em FMs. Contudo, esta técnica de modelagem de features não descreve em detalhes como árvores de features distintas podem ser associadas por relacionamentos UML.

Como consequência, importantes dúvidas podem ser levantadas sobre a compatibilidade hierárquica entre *super* e *sub* FMs e FCs, bem como sobre o gerenciamento de FCs durante a manutenção e o reuso de FMs.

Forfamel (Asikainen, Männistö e Soininen, 2006) também aplicou recursos OO na modelagem de features. Ele representou FMs e FCs com classes *FeatureType* e *Feature* juntamente com uma semântica restritiva. Forfamel também descreveu regras de construções para FMs, e definiu quais possíveis FCs podem ser instanciados. Contudo, apenas *FeatureTypes* abstratos foram descritos no intuito de tratar o encapsulamento de features. Mais ainda, nenhuma operação relacionada à atualização de features foi apresentada, algo necessário para gerenciar e evoluir FMs e FCs modelados.

Clafer (Bak, Czarnecki e Wasowski, 2010) definiu uma linguagem de modelagem de classes com suporte de primeira-classe a modelagem de features. Sua semântica foi definida em termos de uma abordagem generativa de construções Alloy (o transformador Clafer-to-Alloy) (Bak, Czarnecki e Wasowski, 2010). Todavia, Clafer requer mais melhorias como linguagem para se tornar mais usável (Bak, Czarnecki e Wasowski, 2010), ela não segue as estruturas de modelagem de features tradicionais (ausência de grupos de features, por exemplo), e uma semântica padrão para especificações Clafer bem-formadas também não foi fornecida.

Kumbang (Asikainen, Männistö e Soininen, 2007) representou features com recursos UML. Ele unifica modelagem de features com arquitetura de software no intuito de modelar a variabilidade de software em SPLs. Para tal, Kumbang provê estruturas representativas de features capazes de serem usadas em configurações arquiteturais fornecidas por declarações *component-connector*. Kumbang foi rigorosamente descrita usando linguagem natural e recursos de um Perfil UML. Ele mostra a importância da distinção entre instâncias de feature types e features que aparecem em FMs e FCs (Asikainen, Männistö e Soininen, 2007). Contudo, Kumbang precisa de uma linguagem de restrições para capturar as interações complexas entre suas diferentes entidades de features (Asikainen, Männistö e Soininen, 2007). Também não foram especificadas operações de features para atividades de manutenção em árvores de features Kumbang.

Considerando o uso de instruções de programação para modelar FMs, Acher, Collet e Lahire (2011) apresentaram as principais características de FAMILIAR, um DSL

para gerenciar FMs. Ele fornece várias operações que dão suporte ao "entendimento" de FMs, bem como algumas operações para manipular, renomear e remover features projetadas. Um ambiente de execução capaz de interpretar instruções FAMILIAR também foi fornecido, permitindo a produção de scripts modularizados e parametrizados no intuito de gerar MPLs. Dentre as limitações identificadas nesta abordagem, o nome de uma feature deve ser único dentro de um FM, nenhum suporte a restrição entre features foi descrito, e nenhum intervalo de cardinalidade entre features foi apresentado.

Finalmente, considerando o uso de Java na produção de FMs, Sunkle et al. (2008) propôs uma extensão de Java que implementa features com entidades de primeira-classe. Através desta extensão, uma vez que features podem ser definidas num programa, diferentes tipos de FMs podem ser gerados por features programadas (Sunkle et al., 2008). Como resultado, esta extensão Java permite a criação de diferentes SPLs para cada conjunto de features programadas. Dentre as limitações de modelagem de features identificadas: nenhum suporte a cardinalidade de features foi encontrado, todas as features devem ter nomes distintos num FM, apenas duas operações de features foram descritas (sem nenhum conteúdo formal definido), a representação de restrições entre features é implícita, entre outras.

A Tabela 6 sumariza estes recursos e limitações identificados na modelagem de features, sendo estes organizados por aspectos de: documentação, manutenção, encapsulamento, reuso, semântica formal, FMs e FCs compatíveis, ferramentas de suporte e programação de features. Esta tabela também descreve a técnica OOFM com base nos respectivos aspectos de variabilidade sumarizados.

Como resultado, OOFM apresenta:

- Uma documentação diferenciada de features via integração da UML com representação de FMs e FCs diretos e indiretos;
- Uma API e recursos de programação de features formalmente definidos para manipulação de FMs e FCs diversos;
- Regras de encapsulamento e reuso para hierarquias de FMs e FCs formalmente definidas;
- Garantia de compatibilidade entre FMs e FCs diretos e indiretos modelados; e
- Ferramental de suporte capaz de gerar SPLs e sistemas concretos.

Tabela 6 Aspectos de variabilidade identificados na OOFM e demais técnicas de modelagem de features.

Feature Modeling Techniques	**Feature Modeling Aspects**							
	Documentation	***Maintenance***	***Encapsulation***	***Reuse***	**Formal Semantics**	**Compatible FMs and FCs**	**Tool Support**	**Feature Programming**
Staged Configuration	FODA	Scale-down operations (select, clone)	None	Scaled-Down feature trees	Propositional Logic	Yes	Partially by external tools	No
Multi-Level Feature Trees	CBFM	Deviation operations	None	Referred and Referring features	Propositional Logic	No	Self-made IDE	No
Context Variability Model	CBFM	None	None	Contextual model	Propositional Logic	Yes	Self-made IDE	No
Composition Model	UML + FODA	None	None	OO relationships	Propositional Logic (partially described)	Not described.	Configuration generator	No
Forfamel	OO + CBFM (indirect configuration)	None	Abstract property	OO resources	Weight Constraint Rule Language (WCRL)	Yes	Self-made IDE	No
Clafer	OO + limited CBFM	OO resources	OO resources	OO resources	One Alloy specification for each Clafer model	Yes	Alloy Translator	No
Kumbang	UML + CBFM (indirect configuration)	None	UML resources	UML resources	Weight Constraint Rule Language (WCRL)	Yes	Kumbang Modeller and Configurator	No
FAMILIAR	FODA without constraints	Reasoning and maintenance operations	None	Feature modeling scripts	Propositional Logic (partially described)	Yes	Java + XText	Yes
Features as First-class Entities	FODA	OO resources + 2 feature operations	OO resources	OO resources	None	Yes	JastAdd	Yes
OOFM	UML + Similar Forfamel structure (indirect configuration)	UML resources + feature operations	UML resources	UML resources and feature operations	UML Profile + OCL Expressions	Yes	OOFM Environment and OOFM Framework	Yes

Em outras palavras, OOFM se apresenta como uma solução diferenciada de modelagem de features, capaz de cobrir todos os aspectos de variabilidade identificados nas técnicas de modelagem de features analisadas.

5.6 CONCLUSÃO DO CAPÍTULO

Este capítulo apresentou importantes avaliações de corretude, representatividade, complexidade e usabilidade para a técnica OOFM.

De fato, a capacidade de representar FMs e FCs corretos foi verificada de acordo com as questões Q1, Q2 e Q3. A representatividade de elementos, relacionamentos, operações e abordagens de reuso de features também foi confirmada de acordo com as questões Q4, Q5 e Q6. A complexidade similar de documentar, manipular e reusar features também foi inferida para a maioria dos casos de acordo com as questões Q7, Q8 e Q9. A usabilidade da OOFM em modelar FMs e FCs também foi confirmada num experimento de modelagem de features. Finalmente, o suporte completo e integrado dos aspectos de variabilidade identificados também foi confirmado para a técnica OOFM.

Como resultado, a capacidade de produzir documentos corretos, equivalentes e simples de uma maneira usável e completa (de acordo com os aspectos de variabilidade identificados) foi confirmada na OOFM. O próximo capítulo irá apresentar as conclusões finais, contribuições, limitações e trabalhos futuros para a OOFM.

Capítulo
6

Este capítulo apresenta as conclusões deste trabalho, juntamente com as suas contribuições e limitações identificadas. Também são apresentados alguns trabalhos futuros previstos com a OOFM.

6 CONCLUSÕES

Esta tese apresentou uma nova técnica de modelagem de features denominada Modelagem de Features Orientada a Objetos (OOFM). Para tal, conceitos-chave da OOFM foram descritos, juntamente com sua abordagem de formalização e ferramentas de suporte.

A criação de SPLs baseadas na OOFM também foi explorada, mostrando a estrutura do OOFM Framework e atividades de SPLE organizadas no OOFM Process. Um jogo digital também foi desenvolvido com base nos OOFM Framework e OOFM Process, corroborando a capacidade da OOFM em gerar sistemas concretos desejados.

Importantes avaliações também foram efetuadas na técnica OOFM, demonstrando a sua capacidade de produzir documentos corretos, equivalentes e simples de uma maneira usável e completa de acordo com os aspectos de variabilidade identificados.

Dentre as contribuições da OOFM, trata-se de uma solução integrada de aspectos de variabilidade que permite a análise e o projeto da variabilidade de sistemas em termos de features. Para tal, modelos e sistemas OOFM podem ser gerados pelos seguintes recursos apresentados: Perfil OOFM (modelagem UML), OOFM Environment (scripts formais), *FeatureClass Editor* (ferramenta de suporte) e OOFM Framework (programação OO).

OOFM é capaz de produzir múltiplos FMs e FCs (para MPLs), bem como gerar estruturas OO avançadas para trabalhar com features em tempo de execução (para DSPLs). Ou seja, OOFM é capaz de produzir SPLs avançadas baseadas em features, tendo como base a

aplicação de programação de features na produção de árvores de features dinâmicas e parametrizáveis.

Mais ainda, com o uso da OOFM, FMs e FCs avançados podem ser usados na geração de sistemas concretos via Product Line Architecture (PLA) padronizada (o OOFM Framework). Assim, uma nova abordagem de desenvolvimento de SPLs foi obtida, sendo esta contrária às abordagens tradicionais de SPL baseadas em modelos FODA simples manipulados por PLAs distintas e complexas (Sarinho, Apolinário e Almeida, 2012).

Dentre as limitações identificadas na OOFM, existe uma grande influência das estruturas de programação de features na produção de modelos OOFM avançados. Trata-se de algo que pode ser observado com o OOFM Environment baseado em scripts ASSL, com o OOFM Framework baseado em programação OO, e com o Perfil OOFM baseado em expressões OCL. Ou seja, tem-se uma inversão na forma de representação e uso de features pela OOFM, contrariando o padrão gráfico tradicionalmente adotado pelas abordagens de modelagem de features existentes.

Como possível solução para este problema, pretende-se criar mais ferramentas visuais de modelagem para a OOFM, fortalecendo assim a filosofia gráfica de modelagem de features na OOFM. Ferramentas e repositórios web de modelos e recursos OOFM também serão desenvolvidos no futuro, algo similar à iniciativa SPLOT (2012) já consolidada na literatura.

Também existe um forte acoplamento entre FMs e FCs na OOFM, o que impede a atualização aleatória e desregrada de FMs com múltiplos FCs associados. Assim, a OOFM impõe restrições diversas de uso na modelagem de features, caminhando no sentido contrário às abordagens tradicionais e "livres" de modelagem de features existentes. Entretanto, trata-se também de algo positivo, no sentido de que a OOFM indica claramente o que pode e o que não pode ser feito em seus FMs e FCs modelados.

Novas estratégias de implementação de features também serão exploradas com a OOFM no futuro, a exemplo de hierarquias, padrões e frameworks de features. A idéia é aplicar estratégias já consagradas de reuso de software na análise e no projeto da variabilidade de sistemas diversos.

Um bom exemplo de trabalho futuro nesta linha seria a produção de um framework de features para jogos digitais, dando continuidade ao que foi apresentado até o

momento com o FEnDiGa e os modelos NESI e GDS. Também seria interessante a aplicação da OOFM em outros domínios de software, a exemplo dos sistemas embarcados e dos sistemas de informação, corroborando assim a aplicação da OOFM em importantes domínios de software existentes.

REFERÊNCIAS

ACHER, M.; COLLET, P.; LAHIRE, P. **A Domain-Specific Language for Managing Feature Models.** ACM SAC'11, March 21-25, TaiChung, Taiwan, 2011.

ALVES, V.; GHEYI, R.; MASSONI, T.; KULESZA, U.; BORBA, P.; LUCENA, C. **Refactoring Product Lines.** Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE '06), pp. 201–210, ACM Press, 2006.

AL-KILIDAR, H.; COX, K.; KITCHENHAM, B. **The Use and Usefulness of the ISO/IEC 9126 Quality Standard**. International Symposium on Empirical Software Engineering, 2005.

ASIKAINEN, T.; MÄNNISTÖ, T.; SOININEN, T. **A Unified Conceptual Foundation for Feature Modelling.** 10th International Software Product Line Conference (SPLC'06), pp.31-40, 2006.

ASIKAINEN, T.; MÄNNISTÖ, T.; SOININEN, T. **Kumbang: A Domain Ontology for Modelling Variability in Software Product Families.** Advanced Engineering Informatics, 21, pp. 23-40, 2007, Elsevier.

BABAR, M.; CHEN, L.; SHULL, F. **Managing Variability in Software Product Lines.** IEEE Software, May/June 2010, pp. 89-91,94. ISSN 0740-7459.

BAGHERI E.; GASEVIC, D. **Assessing the Maintainability of Software Product Line Feature Models using Structural Metrics.** Software Quality Journal 19(3):579-612, Springer, 2011.

BAK, K.; CZARNECKI, K.; WASOWSKI, A. **Feature and Meta-Models in Clafer: Mixed, Specialized, and Coupled.** SLE'10 Proceedings of the Third international conference on Software language engineering, Springer-Verlag Berlin, 2010. ISBN: 978-3-642-19439-9.

BENAVIDES, D.; RUIZ-CORTÉS, A.; TRINIDAD, P.; SEGURA, S. **A Survey on the Automated Analyses of Feature Models.** Jornadas de Ingenierıa del Software y Bases de Datos (JISBD), 2006.

BEUCHE, D.; DALGARNO, M. **Software product line engineering with feature models.** In: Software Acumen, 2006. Available at http://www.methodsandtools.com/PDF/mt200604.pdf.

BINSUBAIH, A.; MADDOCK, S. **Game Portability Using a Service-Oriented Approach. International Journal of Computer Games Technology**. Volume 2008, Article ID 378485, 7 pages. Hindawi Publishing Corporation, doi:10.1155/2008/378485.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. **The Unified Modeling Language User Guide.** Addison-Wesley, 1998, ISBN-10: 0201571684.

CECHTICKY, V.; PASETTI, A.; ROHLIK, O.; SCHAUFELBERGER, W. **XML-based feature modeling.** Software Reuse: Methods, Techniques and Tools: 8th International Conference, ICSR 2004, Madrid, Spain, July 5-9, 2009, Proceedings, Vol. 3107 of Lecture Notes in Computer Science, Springer-Verlag, Heidelberg, Germany, pp. 101–114.

CHEN, L.; BABAR, A.; ALI, N. **Variability Management in Software Product Lines: A Systematic Review.** In: SPLC'09, San Francisco, CA, USA, pp. 81–90, 2009.

CLEMENTS, P.; NORTHROP, L. **Software Product Lines: Practices and Patterns - 3rd edition.** Addison-Wesley Professional. 2001. ISBN-13: 978-0201703320.

CZARNECKI, K. **Overview of Generative Software Development**. Proceedings of Unconventional Programming Paradigms (UPP), pp 313-328, Springer-Verlag, 2004.

CZARNECKI, K.; HELSEN, S.; EISENECKER, U. **Staged Configuration Using Feature Models.** Software Product Lines, Lecture Notes in Computer Science, Springer, 2004.

CZARNECKI, K.; HELSEN, S.; EISENECKER, U. **Formalizing Cardinality-Based Feature Models and Their Specialization.** Software Process Improvement and Practice, 10(1):7–29, jan/mar, 2005.

DEURSEN, V.; KLINT, P. **Domain-Specific Language Design Requires Feature Descriptions.** Journal of Computing and Information Technology, 10(1): 1–17, 2002. Available at http://homepages.cwi.nl/~arie/papers/fdl/fdl.pdf .

FAYAD, E.; SCHMIDT, C.; JOHNSON R. **Building Application Frameworks Object-Oriented Foundations of Framework Design.** John Wiley Sons. 1999.

GAMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design Patterns: Elements of Reusable Object-Oriented Software.** Addison-Wesley Professional. 1994. ISBN 0201633612.

GARCIA, M. **Formalization of QVT-Relations: OCL-based Static Semantics and Alloy-based Validation.** Proceedings of the Second Workshop on MDSD Today, pp 21–30, October 2008.

GOGOLLA, M.; BOHLING, J.; RICHTERS, M. **Validation of UML and OCL Models by Automatic Snapshot Generation.** Proc. 6th Int. Conf. Unified Modeling Language (UML'2003), Springer, Berlin, 2003. LNCS 2863.

GOGOLLA, M.; BÜTTNER, F.; RICHTERS, M. **USE: A UML-Based Specification Environment for Validating UML and OCL.** Science of Computer Programming, 69:27-34, 2007.

GOLDEN STUDIOS. **Golden T Game Engine (GTGE).** Available from: http://www.goldenstudios.or.id/products/GTGE/ [Accessed 17 August 2011].

GURP, J.; SVAHNBERG, M.; BOSCH, J. **Managing Variability in Software Product Lines**. Landelijk Architectuur Congres, Amsterdam, 2000.

GURP, J.; SVAHNBERG, M.; BOSCH, J. **On the Notion of Variability in Software Product Lines**. Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA), pp. 45-55, 2001.
HARTMANN, H.; TREW, T. **Using Feature Diagrams with Context Variability to Model Multiple Product Lines for Software Supply Chains.** 12th International Software Product Line Conference, pp. 12-21, 2008, ISBN: 978-0-7695-3303-2.

JAVA. **Oracle Technology Network > Java.** Available from: http://www.oracle.com/technetwork/java/index.html [Accessed 30 November 2012].

JGAME. **A Java Game Engine for 2D Games.** Available from: http://www.13thmonkey.org/~boris/jgame/ [Accessed 17 August 2011].

JME3. **jMonkeyEngine.** Available from: http://jmonkeyengine.org/ [Accessed 17 August 2011].

KANG, K.; COHEN, S.; HESS, J.; NOVAK, W.; PETERSON, A. **Feature-Oriented Domain Analysis (FODA) Feasibility Study, Technical Report.** CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh , 1990.

KRASNER, E.; T. Pope. **A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80.** J. Object Oriented Program., v. 1, p. 26-49, 1988.

LUND, A. **Measuring Usability with the USE Questionnaire.** The Usability SIG Newsletter, 2001.

OMG. **Response to the UML 2.0 OCL RfP (ad/2000-09-03).** OMG Document ad/2003-01-07. 2000. Available from: http://www.omg.org/spec/OCL/2.2/, [Accessed 25 August 2011].

OMG. **UML Superstructure - version 2.2.** OMG Document Number: formal/2009-02-02. 2009. Available from: http://www.omg.org/spec/UML/2.2/Superstructure, [Accessed 25 August 2011].

PREFUSE. **Prefuse Information Visualization Toolkit.** Available from: http://prefuse.org/., [Accessed 30 November 2012].

REISER, M.; WEBER, M. **Managing Highly Complex Product Families with Multi-Level Feature Trees.** 14th IEEE International Requirements Engineering Conference IEEE, 2006.

ROSENMÜLLER, M.; SIEGMUND, N. **Automating the Configuration of Multi Software Product Lines**. ICB Research Report, Proceedings of the 4th International Workshop on Variability Modeling of Software-intensive Systems (VaMoS'10), pp. 123-130, 2010.

SARINHO, V.; APOLINÁRIO, A. **A Feature Model Proposal for Computer Games Design.** Proceedings of the VII Brazilian Symposium on Computer Games and Digital Entertainment, Belo Horizonte, p. 54-63, 2008.

SARINHO, V.; APOLINÁRIO, A. **A Generative Programming Approach for Game Development**. Proceedings of the VIII Brazilian Symposium on Computer Games and Digital Entertainment, Rio de Janeiro, p. 9-18, 2009.

SARINHO, V., APOLINÁRIO, A., **Combining Feature Modeling and Object Oriented Concepts to Manage the Software Variability.** IEEE International Conference on Information Reuse and Integration (IRI), Las Vegas, USA, pp. 344-349, 2010.

SARINHO, V.; APOLINÁRIO, A.; ALMEIDA, E. **A Feature-based Environment for Digital Games.** 1st Workshop on Game Development and Model-Driven Software Development, in conjunction with 10th International Conference on Entertainment Computing (ICEC), Vancouver, 2011.

SARINHO, V.; APOLINÁRIO, A.; ALMEIDA, E. **OOFM - A Feature Modeling Approach to Implement MPLs and DSPLs.** IEEE IRI 2012, August 8-10, Las Vegas, Nevada, USA, pp 740-742, 2012.

SARINHO, V.; APOLINÁRIO, A. **Detailing the UML Profile of the OOFM Technique.** In: Proceedings of 3rd Brazilian Workshop on Model Driven Development (WB-DSDM 2012), v. 8, pp 25-32, 2012, ISSN:2178-6097.

SCHOBBENS, P.; HEYMANS, P.; TRIGAUX, J. **Feature Diagrams: A Survey and a Formal Semantics.** 14th IEEE International Requirements Engineering Conference, 2006.

SOCHOS, P.; RIEBISCH, M.; PHILIPPOW, I. **The Feature-Architecture Mapping (FArM) Method for Feature-Oriented Development of Software Product Lines.** Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS '06), pp. 308-318, ISBN: 0-7695-2546-6.

SPLOT. **SPL On-line Tools.** Available from: http://www.splot-research.org/ [Accessed 30 November 2012].

SUNKLE, S.; ROSENMÜLLER, M.; SIEGMUND, N.; RAHMAN, S. S.; SAAKE, G.; APEL, S. **Features as First-class Entities – Toward a Better Representation of Features.** McGPLE Workshop, pp. 27-34, 2008.

SVAHNBERG, M.; GURP, J.; BOSCH, J. **A Taxonomy of Variability Realization Techniques. Software-Practice & Experience.** Volume 35, Issue 8 (July 2005), 705-754. New York: John Wiley & Sons, 2005. ISSN:0038-0644.

TRINIDAD, P.; RUIZ-CORTÉS, A.; PEÑA, J.; BENAVIDES, D. **Mapping Feature Models onto Component Models to Build Dynamic Software Product Lines.** International Workshop on Dynamic Software Product Line, 2007.

VAN DEURSEN, A.; KLINT, P.; VISSER, J. **Domain-Specific Languages: An Annotated Bibliography.** ACM Sigplan Notices 35.6, pp 26-36, 2000.

WARMER, J.; KLEPPE, A. **Object Constraint Language, The: Getting Your Models Ready for MDA, Second Edition.** Addison Wesley, 2003. ISBN: 0-321-17936-6.

APÊNDICE A – Questionário: Avaliação da Abordagem Object Oriented Feature Modeling (OOFM) na Modelagem de Features

Este questionário busca avaliar sua opinião quanto ao uso da técnica OOFM na modelagem de features. Trata-se da coleta de opiniões diversas com relação ao uso da ferramenta gráfica da OOFM (The *FeatureClass Editor*) na representação de árvores de features OOFM num exemplo tradicional da modelagem de features. Modelos de features relacionados a jogos digitais previamente projetados na disciplina Análise de Sistemas (2012.1) também serão trabalhados pelo *FeatureClass Editor* e avaliados neste questionário.

Qual o seu nome?

Identificação necessária para a organização dos dados do questionário. [ ]

OOFM me ajudou a ser mais efetivo na modelagem de features?
Condorda/Discorda que a ferramenta FeatureClass Editor ajudou na modelagem correta de seu modelo de features desejado.

	1	2	3	4	5	6	7	
discordo muito	○	○	○	○	○	○	○	concordo muito

OOFM me ajudou a ser mais produtivo na modelagem de features?
Condorda/Discorda que a ferramenta FeatureClass Editor lhe ajudou a ser mais produtivo na geração/produção de seu modelo de features desejado.

	1	2	3	4	5	6	7	
discordo muito	○	○	○	○	○	○	○	concordo muito

OOFM pode ser considerada uma ferramenta útil na modelagem de features?
Condorda/Discorda que vale a pena usar a ferramenta FeatureClass Editor na modelagem de features.

	1	2	3	4	5	6	7	
discordo muito	○	○	○	○	○	○	○	concordo muito

OOFM me dá mais controle sobre as possíveis atividades de modelagem de features?
Condorda/Discorda que a ferramenta FeatureClass Editor orienta as possíveis atividades necessárias para a modelagem de features.

	1	2	3	4	5	6	7	
discordo muito	○	○	○	○	○	○	○	concordo muito

OOFM permite que a modelagem de features seja realizada de uma maneira mais fácil?
Condorda/Discorda que a ferramenta FeatureClass Editor facilita a modelagem de features.

1 2 3 4 5 6 7

discordo muito	○	○	○	○	○	○	○	concordo muito

OOFM salva o meu tempo durante uma modelagem de features?
Condorda/Discorda que a ferramenta FeatureClass Editor reduz o tempo utilizado na modelagem de features.

	1	2	3	4	5	6	7	
discordo muito	○	○	○	○	○	○	○	concordo muito

OOFM atende as minhas necessidades de modelagem de features?
Condorda/Discorda que a ferramenta FeatureClass Editor oferece todos os recursos desejados para se efetuar uma modelagem de features.

	1	2	3	4	5	6	7	
discordo muito	○	○	○	○	○	○	○	concordo muito

OOFM faz tudo o que eu espero para a modelagem de features?
Condorda/Discorda que a ferramenta FeatureClass Editor oferece todos os recursos que vc imaginou para a modelagem de features.

	1	2	3	4	5	6	7	
discordo muito	○	○	○	○	○	○	○	concordo muito

OOFM é fácil?
Condorda/Discorda que a ferramenta FeatureClass Editor é fácil de entender.

	1	2	3	4	5	6	7	
discordo muito	○	○	○	○	○	○	○	concordo muito

OOFM é simples?
Condorda/Discorda que a ferramenta FeatureClass Editor é simples de entender.

	1	2	3	4	5	6	7	
discordo muito	○	○	○	○	○	○	○	concordo muito

OOFM é amigável?
Condorda/Discorda que a ferramenta FeatureClass Editor é simples de usar.

	1	2	3	4	5	6	7	
discordo muito	○	○	○	○	○	○	○	concordo muito

OOFM requer o menor número de passos possíveis para realizar uma modelagem de features?
Condorda/Discorda que a ferramenta FeatureClass Editor fornece uma solução ótima para a modelagem de features.

	1	2	3	4	5	6	7	

discordo muito	○	○	○	○	○	○	○	concordo muito

OOFM é flexível?
Condorda/Discorda que a ferramenta FeatureClass Editor fornece mais de uma abordagem para se fazer uma modelagem de features.

	1	2	3	4	5	6	7	
discordo muito	○	○	○	○	○	○	○	concordo muito

Usar OOFM é a maneira mais fácil?
Condorda/Discorda que a ferramenta FeatureClass Editor fornece a maneira mais fácil de se fazer uma modelagem de features.

	1	2	3	4	5	6	7	
discordo muito	○	○	○	○	○	○	○	concordo muito

Consigo usar a OOFM sem o uso de material de suporte?
Condorda/Discorda que a ferramenta FeatureClass Editor pode ser utilizada sem a ajuda de manuais ou materiais textuais de suporte.

	1	2	3	4	5	6	7	
discordo muito	○	○	○	○	○	○	○	concordo muito

Não percebi nenhuma inconsistência durante o uso da OOFM?
Condorda/Discorda que a ferramenta FeatureClass Editor não apresentou nenhum problema/inconsistência durante o seu uso.

	1	2	3	4	5	6	7	
discordo muito	○	○	○	○	○	○	○	concordo muito

Usuários ocasionais e regulares irão gostar de usar a OOFM?
Condorda/Discorda que a ferramenta FeatureClass Editor é agradável tanto para usuário ocasionais como para usuários regulares.

	1	2	3	4	5	6	7	
discordo muito	○	○	○	○	○	○	○	concordo muito

OOFM permite a recuperação rápida e fácil de possíveis erros de modelagem?
Condorda/Discorda que a ferramenta FeatureClass Editor permite a recuperação rápida e fácil de possíveis erros na modelagem de features.

	1	2	3	4	5	6	7	
discordo muito	○	○	○	○	○	○	○	concordo muito

Posso usar a OOFM com sucesso em qualquer momento na modelagem de features?
Condorda/Discorda que a ferramenta FeatureClass Editor pode ser usada com sucesso em qualquer momento na modelagem de features.

	1	2	3	4	5	6	7	
discordo muito	○	○	○	○	○	○	○	concordo muito

Eu aprendi a usar a OOFM rapidamente?
Condorda/Discorda que conseguiu aprender a usar a ferramenta FeatureClass Editor rapidamente.

	1	2	3	4	5	6	7	
discordo muito	○	○	○	○	○	○	○	concordo muito

Eu me relembrei rapidamente como se usa a OOFM?
Condorda/Discorda que conseguiu relembrar rapidamente o uso da ferramenta FeatureClass Editor.

	1	2	3	4	5	6	7	
discordo muito	○	○	○	○	○	○	○	concordo muito

É fácil de aprender a usar a OOFM?
Condorda/Discorda que considera fácil o aprendizado do uso da ferramenta FeatureClass Editor.

	1	2	3	4	5	6	7	
discordo muito	○	○	○	○	○	○	○	concordo muito

Eu me tornei rapidamente um profissional no uso da OOFM?
Condorda/Discorda que se tornou rapidamente um profissional no uso da ferramenta FeatureClass Editor.

	1	2	3	4	5	6	7	
discordo muito	○	○	○	○	○	○	○	concordo muito

Eu estou satisfeito com a OOFM?
Condorda/Discorda que está satisfeito com a ferramenta FeatureClass Editor.

	1	2	3	4	5	6	7	
discordo muito	○	○	○	○	○	○	○	concordo muito

Eu vou recomendar a OOFM para amigos?
Condorda/Discorda que irá recomendar a ferramenta FeatureClass Editor para amigos.

	1	2	3	4	5	6	7	
discordo muito	○	○	○	○	○	○	○	concordo muito

É divertido usar a OOFM?
Condorda/Discorda que é divertido usar a ferramenta FeatureClass Editor.

	1	2	3	4	5	6	7	
discordo muito	○	○	○	○	○	○	○	concordo muito

A OOFM trabalha do jeito que vc queria que ela trabalha-se?
Condorda/Discorda que a ferramenta FeatureClass Editor trabalha do jeito que vc esperava.

	1	2	3	4	5	6	7	
discordo muito	○	○	○	○	○	○	○	concordo muito

OOFM é maravilhosa?
Condorda/Discorda que a ferramenta FeatureClass Editor é maravilhosa.

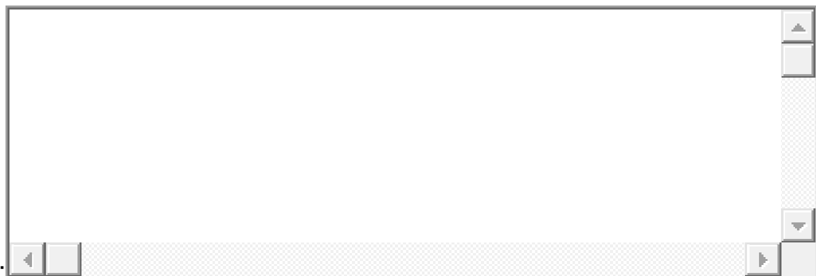
	1	2	3	4	5	6	7	
discordo muito	○	○	○	○	○	○	○	concordo muito

Vc sente que precisava de algo como a OOFM na sua vida?
Condorda/Discorda que precisava de algo como a ferramenta FeatureClass Editor na sua vida.

	1	2	3	4	5	6	7	
discordo muito	○	○	○	○	○	○	○	concordo muito

A OOFM é prazerosa de se usar?
Condorda/Discorda que a ferramenta FeatureClass Editor é prazerosa de se usar.

	1	2	3	4	5	6	7	
discordo muito	○	○	○	○	○	○	○	concordo muito

Pontos positivos da OOFM: 

Pontos negativos da OOFM: