Uma Proposta para a Incorporação do Modelo de Features à Linguagem UML

Ivan Mathias Filho,
Carlos J.P. de Lucena
Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de São Vicente 225
Rio de Janeiro, RJ, 22453-900, Brasil
Email: {ivan,lucena}@inf.puc-rio.br

PUC-RioInf.MCC21/01 Julho de 2001

Resumo

O modelo de features é uma das técnicas mais bem sucedidas para fomentar a reutilização de artefatos de software desde os estágios iniciais do desenvolvimento. Este trabalho tem por objetivo apresentar uma proposta para a incorporação do modelo de features à linguagem UML, que se tornou a linguagem padrão para a modelagem de software orientada a objetos. A especificação precisa e cuidadosa desta proposta, que utiliza técnicas formais e informais tais como notação gráfica, linguagem natural e linguagem formal, permitirá a construção de ferramentas que dêem suporte ao desenvolvimento de modelos de features para famílias de aplicações, e a instanciação de membros destas famílias a partir de tais modelos. Além disso, a utilização do padrão XMI na representação dos modelos de features facilitará a integração destes com muitas das ferramentas CASE existentes no mercado.

Palavras chave: Reutilização, Análise de Domínios, UML, XML, Orientação a Objetos.

Abstract

Feature modeling is one of the most successful techniques in use to promote the reuse of software artifacts as of the initial stages of development. The purpose of this paper is to present a proposal for the incorporation of feature modeling into the UML language, which has become a standard for the modeling of object-oriented software. The precise and careful specification of this proposal, which uses formal and informal techniques such as graphic notation, natural language and formal language, will permit the building of tools that provide support to the development of feature models for families of applications, and the instantiation of members of these families based on these models. Moreover, the use of the XMI standard in the representation of the feature models will facilitate their integration with the many CASE tools already existing on the market.

Keywords: Reuse, Domain Analysis, UML, XML, Object-Oriented.

1 Introdução

A adoção de técnicas que favoreçam a reutilização de artefatos de software desde os estágios iniciais do desenvolvimento é um requisito fundamental para construir softwares mais baratos, eficientes, e que atendam às necessidades dos clientes no momento adequado [1].

Uma das técnicas mais usadas para se atacar o problema acima, chamada de Análise de Domínio [2], é estudar os aspectos comuns e as variações existentes em uma família de aplicações [3], e organizá-las de maneira que sirvam de base para a construção de artefatos de software genéricos para um dado domínio de aplicação. Tais artefatos poderão ser futuramente adaptados para serem reutilizados na construção de um novo membro da família.

Durante o final da década de 1980 e início da década de 1990 a comunidade de Engenharia de Software assistiu ao aparecimento de vários métodos de Análise de Domínios. Apesar das inúmeras diferenças existentes, podemos dizer, de maneira genérica, que tais métodos são funcionalmente equivalentes, invariavelmente apresentando operações de agregação, classificação, generalização e parametrização [2].

Apesar das similaridades existentes, o método FODA (Feature Oriented Domain Analysis) [4], desenvolvido no Software Engineering Institute (SEI), destacou-se dos demais não somente pela solidez e coerência da sua abordagem, mas também pela vasta documentação disponível e pelo vários casos de estudo apresentados por inúmeras publicações. O método FODA foi posteriormente aperfeiçoado por um dos seus criadores, Kyo C. Kang, na Universidade de Pohang (Coréia), dando origem ao FORM, Feature Oriented Reuse Method [5].

Dentre as várias contribuições feitas pelos dois métodos citados acima, podemos destacar o Modelo de Features. Embora tal conceito não fosse uma novidade na época do desenvolvimento do FODA, o tratamento sistemático e o papel de catalisador do processo de reutilização dado às Features constituiu-se em algo realmente inovador.

Uma Feature pode ser informalmente definida como sendo “uma característica essencial das aplicações de um domínio” [5]. A principal virtude do uso das Features está no fato de elas capturarem e organizarem a terminologia usada por especialistas e usuários de um domínio de aplicação, constituindo-se assim no vocabulário de uma linguagem de domínio usada para facilitar a comunicação entre os especialistas em software e os usuários.

O objetivo deste trabalho é integrar o Modelo de Features à linguagem UML, permitindo que este conceito possa ser utilizado nos processos de desenvolvimento de software definidos pelos métodos de análise e design orientados a objetos existentes no mercado. Em especial, estamos interessados em aplicar esta técnica no desenvolvimento e na instanciação de frameworks orientados a objeto, dado que tanto as Features como os frameworks têm por objetivo sistematizar a construção de linhas de produtos de software [6].

A seção 2 fornece uma breve descrição dos principais aspectos estruturais e semânticos do Modelo de Features. Na seção 3 é descrita, em linhas gerais, a maneira pela qual a linguagem UML é especificada. Na seção 4 são apresentadas todas alterações feitas no metamodelo da UML para dar suporte ao Modelo de Features. Na seção 5 são apresentados as nossas conclusões e os nossos trabalhos futuros.

2 O Modelo de Features

Um modelo de features é uma representação hierárquica que visa capturar os relacionamentos estruturais entre as features de um domínio de aplicação. Em [5] são propostos três tipos de relacionamentos distintos entre as Features: **composição**, **generalização** e **implementado-por**. Além disso, em uma **composição**, as Features podem ser **opcionais**, **obrigatórias** e **alternativas**.

Neste ponto cabe estabelecer a diferença entre um Modelo de Features e uma instância do mesmo modelo. Um Modelo de Features é uma representação das características disponibilizadas aos usuários pelos membros de uma família de aplicações. Uma instância de um modelo de features determina as características de um membro específico desta família. Logo, quando uma Feature é classificada como **opcional**, estamos querendo dizer que a característica que ela representa pode estar ou não presente em um dado membro da família. Da mesma forma, uma Feature **obrigatória** representa uma característica que tem que estar presente em todos os membros de uma família. Por último, um grupo de Features **alternativas** define um grupo de características onde uma, e somente uma, pode estar presente em um membro da família.

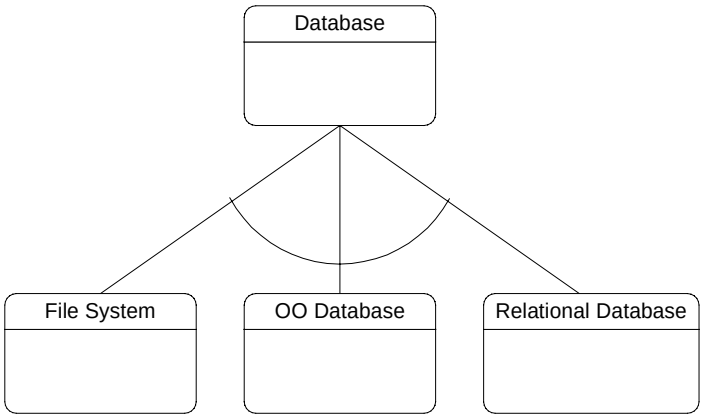

Figura 1 – O laço envolvendo as Features representa uma composiçao alternativa

O presente trabalho, diferentemente de [5], considera supérflua a inclusão do relacionamento de **generalização** entre as Features. Não que a semântica que tal relacionamento carrega seja desnecessária para um melhor entendimento do modelo. Porém, no nosso entender, o relacionamento de composição, da maneira como é propoosto pelos autores do FORM, já fornece tal informação. Para demonstrar isto vamos recorrer a alguns exemplos extraídos de [5].

A Figura 1 mostra uma Feature, Database, composta por três Features alternativas que descrevem três mecanismos diferentes de persistência em um EBBS (Eletronic Bulletin Board System). Pela semântica da

composição alternativa, descrita anteriormente, no processo de instanciação de um membro da família EBBS apenas um dos métodos de persistência poderá ser escolhido. Note também que, qualquer que seja a Feature escolhida, poderemos dizer, neste caso, que tal escolha "é uma espécie de Database"; caracterizando assim o relacionamento de generalização.

Em um outro exemplo extrído do mesmo domínio de EBBS, a Figura 2 mostra duas possíveis escolhas para a implementação de uma interface textual com os usuários finais: Formated Display e Unformated Display. Neste caso, entretanto, foi usado o relacionamento de generalização (representado pelas linhas pontilhadas ligando os nós) entre as duas e a Feature Textual Display. Está bastante claro, no entanto, que em ambos os casos o que se deseja representar é o conceito de generalização, embora dois relacionamentos distintos, com notações distintas, tenham sido usados. Logo, no nosso entender, a composição alternativa é suficiente para representar o relacionamento de generalização.

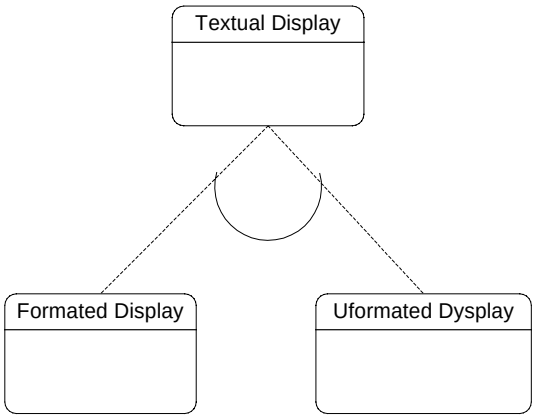

Figura 2 – As linhas pontilhadas representam a relação de generalização.

O relacionamento **implementado-por** não será considerado como algo particular ao Modelo de Features, dado que, como será visto na seção 3, o relacionamento de dependência presente na UML, acrescido do estereótipo <<realize>, já cumpre esta função.

Além dos relacionamentos estruturais citados anteriormente, são definidas em [5] duas regras de composição entre as Features. Para definirmos precisamente a semântica de tais regras temos que olhar novamente para um Modelo de Features e para as instâncias deste modelo. Duas Features **A** e **B** estão relacionadas pela regra **requires** (**A requires B**) quando a presença da característica **A** em um membro da família implicar na presença da característica **B**. Pela segunda regra, se duas features **A** e **B** estão relacionadas pela regra **mutexWith** (**A mutexWith B**), a presença de uma das duas características em um membro da família implica na ausência da outra.

Finalmente, o FORM propõe uma classificação para as Features de acordo com o tipo de informação que elas representam. Desta maneira, as Features são divididas em quatro camadas com diferentes níveis de abstração:

- Capability Layer
- Operating Environment Layer
- Domain Technology Layer
- Implementation Technique Layer

A camada de Capability representa as Features que caracterizam os serviços, as funções, as operações, e a performance que são comuns a um domínio de aplicação. Algumas destas Features representam aspectos funcionais enquanto outras representam aspectos não-funcionais. Na camada de Operating Environment são localizadas as Features que descrevem o ambiente operacional onde uma aplicação será usada. Características tais como plataformas de hardware, sistemas operacionais, e protocolos de comunicação são representadas nesta camada. Nas camadas seguintes, Domain Technology e Implementation Techniques, são representadas as Features que tratam de detalhes de mais baixo nível de abstração. De acordo com [5], as Features da camada de Domain Technology são mais específicas a um determinado domínio de aplicação, enquanto que as da camada de Implementation Techniques são mais genéricas podendo, desta forma, ser utilizadas em outros domínios.

A distribuição das Features em camadas com diferentes níveis de abstração não traz nenhuma dificuldade adcional à integração do Modelo de Features com a UML. O mecanismo de pacotes presentes na UML poderá ser usado para organizar o Modelo de Features em camadas. Cada uma das Features será colocada em um pacote que irá representar uma das quatro camadas propostas (Figura 3). Desta forma, o relacionamento entre Features de diferentes camadas deverá seguir as regras de visibilidade e caminhos (paths) determinados pelo metamodelo da UML.

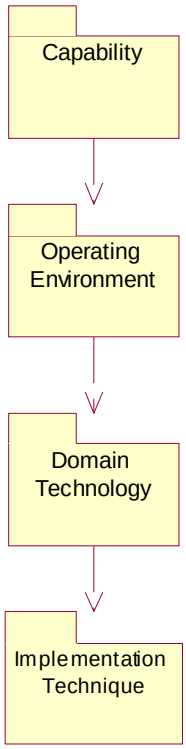

Figura 3 – As camadas do Modelo de Features.

3 A Linguagem UML

A UML (Unified Modeling Language) é uma linguagem para especificar, construir, visualizar, e documentar artefatos de software, que incorpora a maioria dos conceitos utilizados pela comunidade de orientação a objetos na modelagem de sistemas de software [7].

Embora a UML não seja comprometida com nenhuma linguagem de programação em particular, é inegável que a sua concepção sofreu a influência de algumas linguagens de programação orientadas a objetos, especialmente C++, Java e Smalltalk. O mesmo pode ser dito em relação ao processo de desenvolvimento de software. Apesar de não estar formalmente comprometida com nenhum processo em particular, a UML foi concebida para dar suporte a processos dirigidos por casos de uso, centrados na arquitetura, iterativos e incrementais [8].

3.1 A Especificação da UML

A notação e a semântica da UML são descritas, de maneira semiformal, através de uma combinação de notação gráfica, linguagem natural e linguagem formal [7]. O objetivo principal da abordagem adotada é descrever a UML de maneira precisa sem entretanto ter que utilizar linguagens formais tais como Objective-Z ou VDM++; linguagens estas que foram projetadas para serem usadas por pessoas com um sólido *background* matemático, inacessível à maioria dos desenvolvedores encontrados nos dias atuais.

A especificação da linguagem UML é dividida em três partes:

- Sintaxe Abstrata
- Regras de Boa Formação
- Semática

A sintaxe abstrata é descrita através de um subconjunto da própria UML; basicamente diagramas de classes complementados por descrições em linguagem natural. As regras de boa formação são descritas usando uma linguagem formal chamada OCL (Object Constraint Language) [8], além de linguagem natural. A semântica é quase que totalmente descrita em linguagem natural, entretanto notações adicionais são introduzidas dependendo do modelo a ser descrito.

3.2 O Metamodelo da UML

O metamodelo da UML é definido como sendo uma das camadas de uma arquitetura de quatro camadas:

- Meta-metamodelo
- Metamodelo
- Modelo
- Objetos

Nesta arquitetura o meta-metamodelo descreve a infraestrutura necessária para a definição dos metamodelos. No caso da UML, o papel de meta-metamodelo é exercido pelo OMG Meta-Object Facility (MoF) [10]. Logo, a UML pode ser vista como uma instância do MoF.

A função principal da camada de metamodelo é definir uma linguagem para a especificação de modelos. Ou seja, definir uma linguagem com a qual possamos modelar os aspectos de um determinado domínio de aplicação, como por exemplo reservas de passagens aéreas, automação bancária e vendas de produtos on-line. Portanto, o metamodelo da UML descreve a estrutura da própria linguagem UML.

Um modelo é uma instância de um metamodelo, cujo objetivo é definir uma linguagem que descreva certos aspectos de um domínio aplicação. As instâncias do modelo são os objetos; abstrações de software das entidades do “mundo real”.

Com o objetivo de facilitar a compreensão da linguagem, o metamodelo da UML foi organizado em pacotes (elemento definido pela própria UML). No nível mais alto da arquitetura existem três pacotes (Figura 4). O pacote Foundation é responsável pela definição dos artefatos que serão usados para descrever as estruturas estáticas de um modelo, tais como classes, associações, tipos de dados, estereótipos e etc. O pacote Bahavioral Elements define os elementos responsáveis pela descrição dos aspectos dinâmicos, tais como casos de uso, colaborações, máquinas de estado e etc. Por último, o pacote Model Management define como os elementos da modelagem são organizados em modelos, pacotes e subsistemas.

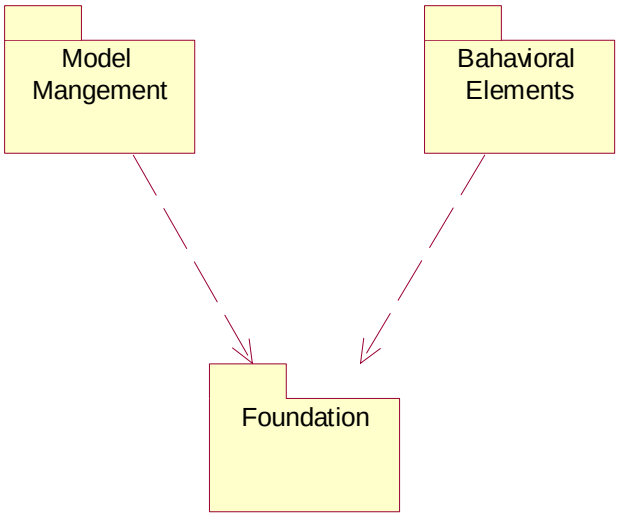

Figura 4 – Os pacotes da camada de mais de nível mais alto.

Dos três pacotes citados anteriormente apenas o Foundation será abordado mais detalhadamente, dado que o processo de integração das Features à UML utiliza apenas os elementos de modelagem nele definidos. Além disso, o modelo de Features é eminentemente estático, não necessitando assim de nenhum elemento definido no pacote Bahavioral Elements.

O pacote Foundation é subdivido em três outros pacotes: Core, Extension Mechanisms, e Data Types (Figura 5).

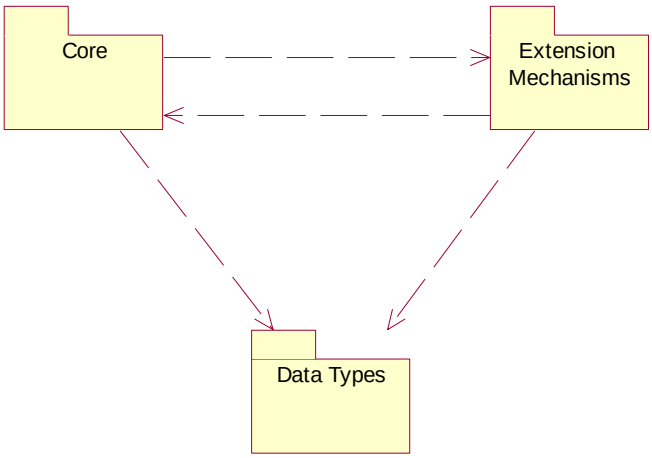

Figura 5 – A explosão do pacote Foundation.

O pacote Core contém os conceitos básicos para a construção de um metamodelo elementar, além de fornecer a infraestrutura necessária para a adição de elementos tais como metaclasses, meta-associações, e

meta-atributos. O pacote Extension Mechanisms fornece os meios para que novos elementos possam ser incorporados à UML sem que a estrutura básica desta tenha que ser modificada. Finalmente, o pacote Data Types define os tipos de dados básicos usados nas especificações dos outros pacotes.

As seções seguintes, que tratam da incorporação do modelo de features à UML, irão interagir mais de perto com os elementos definidos no pacote Foundation; logo, deixaremos para as próximas seções o estudo mais detalhado dos subpacotes do Foundation.

4 O Modelo de Features e a UML

Nesta seção serão abordados os mecanismos usados para a incorporação do Modelo de Features à UML. Antes de começar a detalhar o processo em si é necessário estabelecer certos preliminares. Primeiro cabe esclarecer o porquê da não utilização dos estereótipos, o mecanismo clássico de extensão da UML. Muitos trabalhos que têm por objetivo acrescentar novas construções à UML abusam dos estereótipos para dar semântica própria aos novos elementos de modelagem. Este mecanismo, apesar de muito útil, não permite que mudemos a características fundamentais dos elementos sobre quais os estereótipos são aplicados. Ou seja, se aplicarmos um estereótipo sobre uma classe o elemento resultante continua sendo uma classe, com todos os aspectos estruturais e semânticos atribuídos às classes. Isto quer dizer que o elemento resultante poderá ter atributos estruturais, operações e métodos, além de poder gerar múltiplas instâncias. As únicas coisas que são permitidas aos estereótipos são a adição de novos **tagged values** e novas restrições aos elementos de um modelo. Por exemplo, poderíamos criar um estereótipo aplicável às classes que definisse uma restrição não permitindo a instanciação das classes sobre as quais o estereótipo fosse aplicado. Neste caso, o elemento resultante seria uma classe abstrata.

Outro aspecto a destacar é que o roteiro do processo de integração é o mesmo proposto em [7] para a especificação da linguagem UML. Ou seja, primeiro será apresentada a sintaxe abstrata juntamente com a descrição, em linguagem natural, de alguns aspectos semânticos. Depois serão estabelecidas as regras de boa formação. Além disso, são estabelecidas regras de instanciação de um Modelo de Features.

4.1 Sintaxe Abstrata

A figura 6, extraída de [7], mostra uma visão parcial das classes e dos relacionamentos existentes no pacote Core. Este pacote, como já antecipa o próprio nome, define meta-classes e meta-relacionamentos fundamentais para o entendimento da meta-modelo de UML.

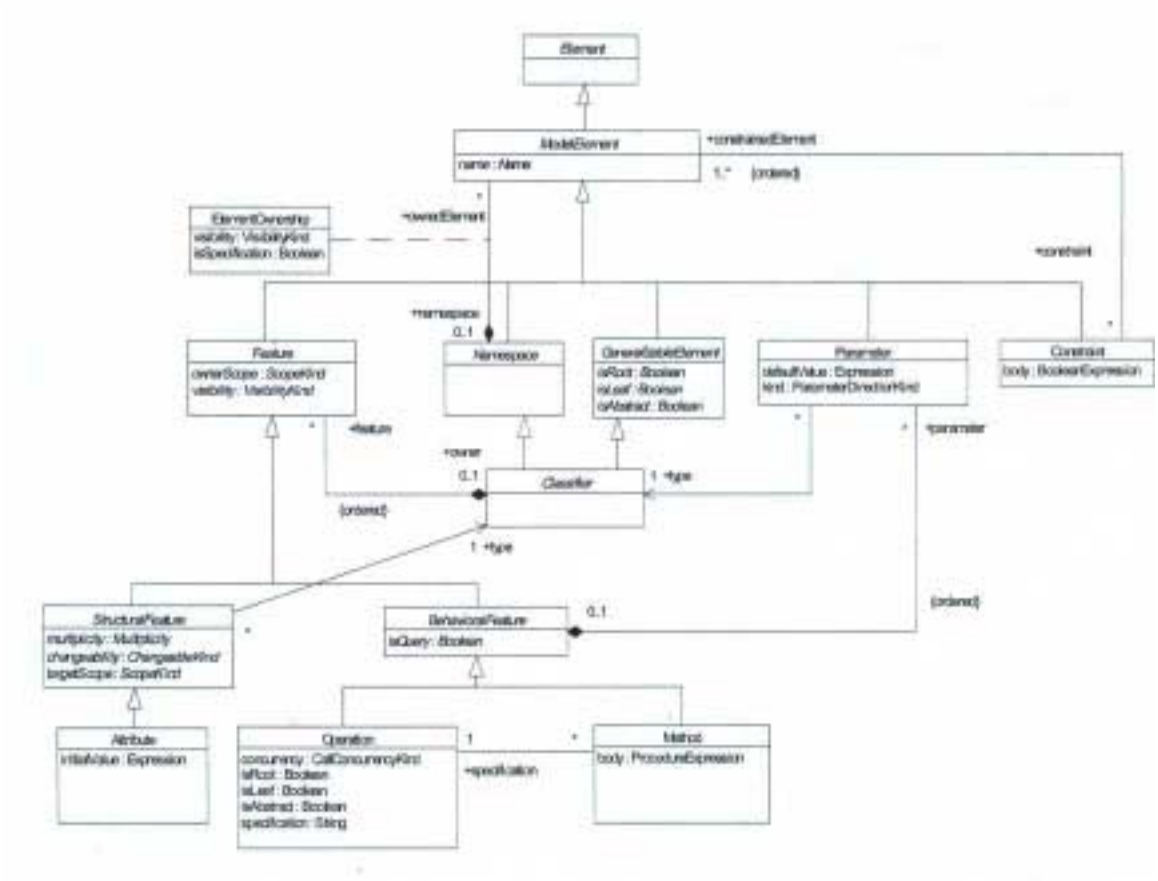

Figura 6 – A explosão do pacote Foundation.

A primeira observação a ser feita é que qualquer elemento de modelagem deve ser uma subclasse direta ou indireta de **ModelElement**, herdando desta o atributo **Name**, que dá nome a qualquer elemento de modelagem presente na UML. A segunda observação é que doravante as Features, como têm sido chamadas até agora as características de um domínio de aplicação, serão chamadas de Domain Features, para que não haja nenhuma confusão com as operações e com os atributos de uma classe, ambos definidos como subclasses da meta-classe **Feature** (como pode ser visto na Figura 6).

A primeira decisão de *design* no que tange ao processo de integração é fazer com que uma Domain Feature seja definida como uma subclasse direta de **ModelElement**. Procedendo assim evitaremos a "contaminação" das Domain Features por atributos e relacionamentos herdados de outras meta-classes. Por exemplo, se tivéssemos tomado a decisão de herdar diretamente de **Classifier**, as Domain Features herdariam, além da semântica do próprio **Classifier**, o relacionamento com a meta-classe Feature. Ou seja, estaríamos dizendo que uma Domain Feature possui atributos, operações e métodos. Da mesma forma, ao tomarmos a decisão de que a composição alternativa (descrita na seção 2) carrega consigo a semântica do relacionamento de generalização, eliminamos a necessidade de ter que definir a meta-classe Domain Feature como uma subclasse de GeneralizableElement.

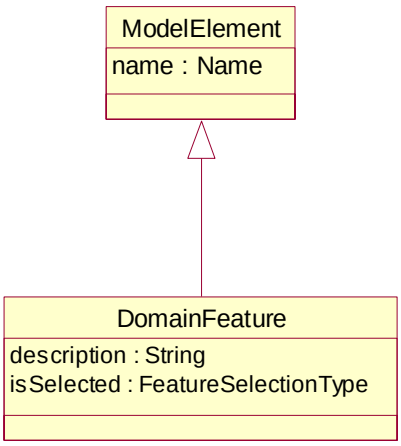

Figura 7 – A definição da classe DomainFeature.

A Figura 7 mostra a classe DomainFeature definida como uma subclasse direta de **ModelElement**. O atributo **description** permite que se acrescente informações textuais adicionais para um melhor entendimento do palpel de uma Domain Feature dentro de um domínio de aplicação. O outro atributo presente na classe DomainFeature, **isSelected**, só tem significado para as instâncias das Domain Features; isto é, o valor do atributo é **undefined** antes da instanciação. Se uma Domain Feature de um modelo for instanciada; isto é, se tal característica fizer parte de um membro específico de uma família de aplicações, então o valor do atributo **isSelected** será a constante booleana **true**. Caso contrário, o valor do atributo será **false**. Para definir o conjunto de valores válidos para o atributo **isSelected** introduzimos o tipo enumerado **FeatureSelectionType** no pacote DataType (Figura 8).

Uma vez definida a classe DomainFeature, é hora de definir o relacionamento de composição entre as Domain Features. A Figura 9 mostra que os relacionamentos em UML são modelados por uma classe chamada **Relationship**, que herda diretamente de **ModelElement**. Além disso, podemos ver as várias espécies de relacionamentos presentes na UML olhando para as subclasses de **Relationship**, tais como **Generalization**, **Flow**, e **Association**. Existe ainda a subclasse **Dependency**, que embora não apareça na Figura 9 é a subclasse de **Relationship** que define o relacionamento de dependência.

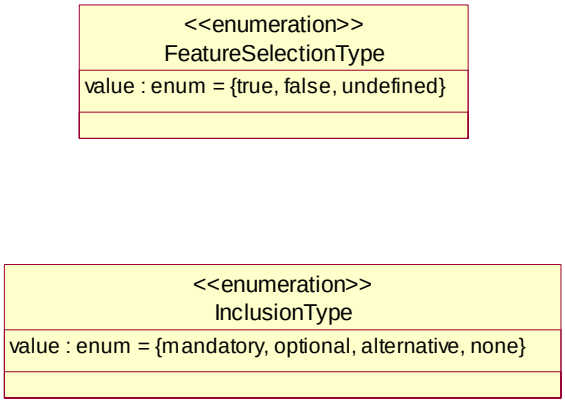

Figura 8 – A definição dos tipos enumerados.

No caso da composição entre Domain Features tomamos a decisão de criar uma nova classe para modelar este relacionamento (Figura 10). Veja que se definíssemos a classe **DomainFeatureComposition** como uma subclasse de **Association**, como poderia parecer natural à primeira vista, herdaríamos também o relacionamento desta com a classe **Classifier**. Dado que uma **DomainFeatureComposition** define uma associação entre Domain Features, o relacionamento herdado da classe **Association** seria bastante inconveniente.

Uma **DomainFeatureComposition** é um relacionamento hierárquico (1 para N) entre uma Domain Feature que faz o papel de todo (whole), e muitas outras Domain Features que fazem o papel das partes (part). Veja que as Domain Features que participam da composição não estão ligadas diretamente à classe **DomainFeatureComposition**, e sim à classe **FeatureCompositionEnd**. Caso assim não o fizéssemos, uma Domain Feature sempre seria alternativa, opcional ou obrigatória por si só, independentemente das composições das quais participasse. Ao introduzirmos a classe **FeatureCompositionEnd** transferimos tal atributo para o nó de uma composição (atributo **inclusion**), que está associado a uma única Domain Feature. Logo, uma Domain Feature pode participar de diversas composições e ter um atributo de inclusão diferente em cada uma delas, dando assim mais liberdade aos projetistas durante a construção de um modelo. Para definir o conjunto de valores válidos para o atributo **inclusion** introduzimos o tipo enumerado **InclusionType** no pacote DataType (Figura 8).

O outro atributo da classe **FeatureCompositionEnd**, **isWhole**, é um atributo booleano que indica se uma Domain Feature faz o papel de todo (whole) ou de uma parte (part) em uma composição. Este atributo foi introduzido no modelo apenas para facilitar a construção de algumas regras de boa formação, como será visto a diante.

A última alteração feita na sintaxe abstrata da UML foi a introdução de uma classe, chamada **DomainFeatureDependency**, para a especificação das regras de composição **requires** e **mutexWith**

(Figura 11). Tal classe, definida como subclasse de **Dependency**, associa duas Domain Features nos papéis de **source** e **target**. A semântica da dependência é definida pela associação da classe **DomainFeatureDependency** com um estereótipo (classe **Stereotype**), cujo nome será **Requires** ou **MutexWith**. A introdução dos papéis é devida à construção de regras de boa formação para as instâncias dos modelos.

O último comentário que deve ser feito em relação à sintaxe abstrata se deve ao relacionamento entre as Domain Features e os outros elementos de modelagem tais como os casos de uso, as classes e as colaborações. Para tal não foi necessária a inclusão de nenhuma construção específica, já que um tipo de dependência, chamada **Abstraction**, rotulada pelo estereótipo **trace**, já foi previamente introduzida no meta-modelo da UML com o objetivo de relacionar elementos de diferentes modelos.

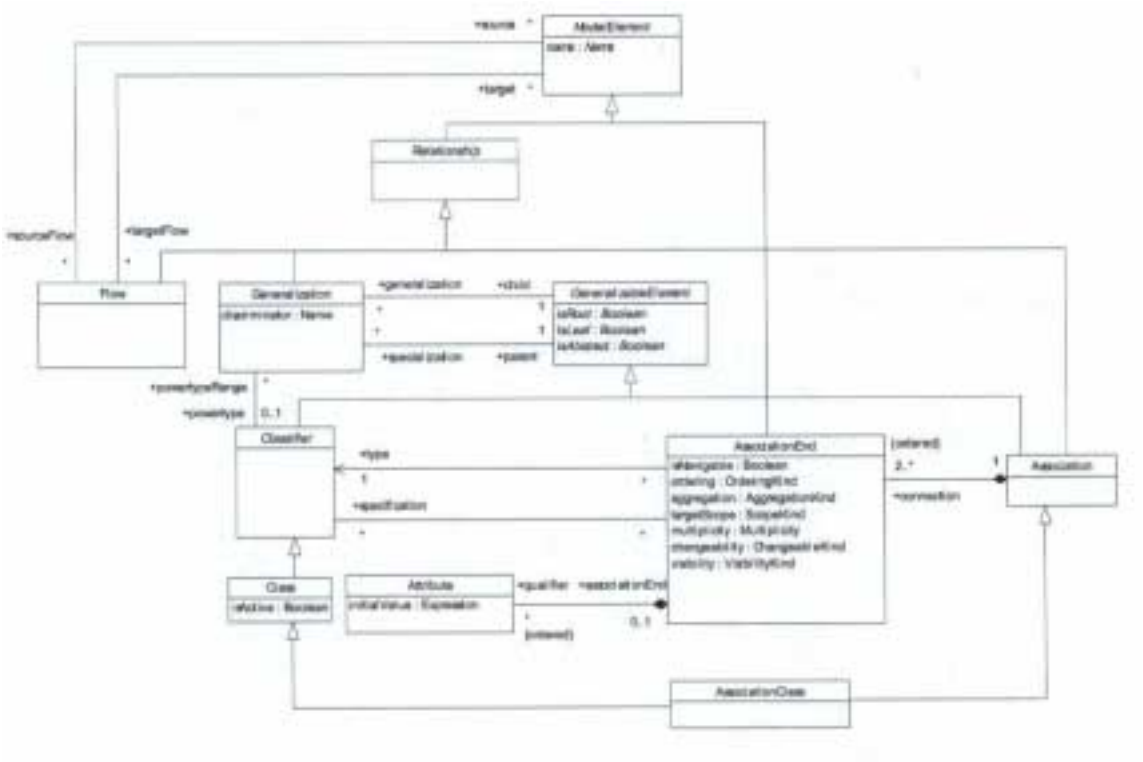

Figura 9 – Os relacionamentos da UML.

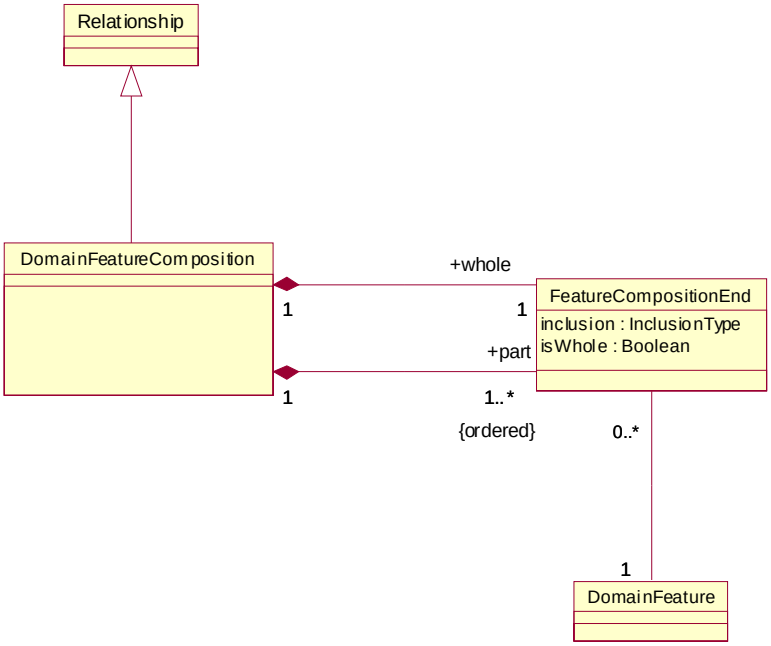

Figura 10 – A composição de DomainFeatures.

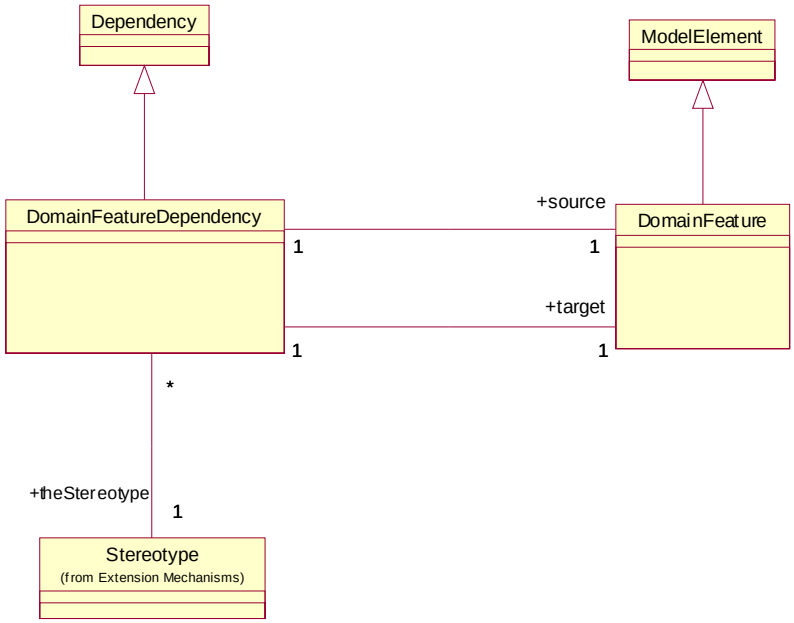

Figura 11 – As regras **requires** e **mutexWith**.

4.2 Regras de Boa Formção

Os elementos de modelagem introduzidos na sintaxe abstrata da UML não são suficientes para entender completamente a semântica das Domain Features. É necessário complementar os diagramas de classes vistos na seção anterior com algumas regras de boa formação. Para tal, usaremos um *mix* de linguagem natural e expressões escritas na linguagem OCL [9]. Uma expressão escrita em OCL é bastante simples e fácil de ser entendida pelos programadores de um modo geral, já que a sua sintaxe é bastante semelhante a das linguagens de programação como Java ou C++. Em geral, usa-se a linguagem OCL para escrever invariantes, pré e pós-condições. Normalmente as construções OCL são expressões de navegação com as quais estabelecemos certas restrições na construção dos modelos.

Apesar de não fazer parte da sintaxe OCL, usaremos como regra definir o contexto da expressão antes de escrevê-la. O contexto da expressão será um elemento de modelagem, no nosso caso uma classe, a partir do qual dar-se-á o início da navegação. Usaremos sempre a palavra chave **self** para fazermos referência ao elemento que define o contexto da expressão.

Antes de começarmos a listar as regras de boa formação vale a pena falar sobre dois operadores fundamentais da OCL. O operador ponto (**.**) é usado de maneira muito semelhante aos seus correspondentes nas linguagens C++ e Java; ou seja, ele serve para selecionar um elemento de modelagem, um atributo de um elemento de modelagem, ou uma operação em uma expressão de navegação. O operador *collection* (**->**) é usado todas a vezes que um *link* em uma expressão de navegação resulta em uma coleção de elementos.

DomainFeature

[1] Uma DomainFeature só poderá fazer o papel de todo (whole) em uma única composição.

```
self.featureCompositionEnd->select(isWhole)->size<=1
```

[2] Uma DomainFeature, no papel de todo, composta apenas por DomainFeatures (partes) opcionais será também uma DomainFeature opcional.

```
(self.featureCompositionEnd.isWhole and
      self.featureCompositionEnd.domainFeatureComposition.part->forAll(p:
      FeatureCompositionEnd | p.inclusion = #optional))
implies
(self.featureCompositionEnd.inclusion = #optional)
```

[3] Uma DomainFeature não pode ser um *template*, logo não possui parâmetros de template.

```
self.templateParameter->size=0
```

[4] Uma DomainFeature não é usada para modelar os aspectos comportamentais de um domínio de aplicações, logo não possui nenhuma máquina de estado associada a ela.

```
self.behavior->size=0
```

DomainFeatureComposition

[1] Se, em uma composição, um FeatureCompositionEnd for alternativo, então todo os outros também o serão.

```
self.part->select(inclusion = #alternative)->size=0 or
self.part->select(inclusion = #alternative)->size = self.part->size
```

DomainFeatureDependency

[1] Os dois tipos de DomainFeatureDependency são **requires** e **mutexWith**.

```
self.stereotype.name='requires' or self.stereotype.name='mutexWith'
```

4.3 Regras de Instanciação

O objetivo desta seção é estabelecer restrições que devem ser atendidas pelas instâncias de um Modelo de Features. Vale a pena lembrar que uma instância de tal modelo determina as características de uma família de aplicações que serão disponibilizadas, aos usuários, por um membro da família. Como foi feito com as reagras de boa formação, as regras de instanciação também serão expressas em OCL.

DomainFeature

[1] Uma DomainFeature poderá ter no máximo uma instância.

```
self.allInstances->size<=1
```

[2] Se uma DomainFeature for obrigatória em pelo menos uma composição então ela terá que ser instanciada.

```
self.featureCompositionEnd->select(inclusion = #mandatory)->size>0 implies
self.isSelected = true
```

DomainFeatureComposition

[1] Em uma composição alternativa apenas uma, e somente uma, DomainFeature (part) pode ser instanciada.

```
self.part->select(inclusion = #alternative)->size>0 implies
self.part.domainFeature->select(isSelected = true)->size=1
```

[2] Quando em uma composição uma, DomainFeature no papel de todo (whole) for instanciada, pelo menos uma das DomainFeatures no papel de parte (part) tem que ser instanciada.

```
self.whole.domainFeature.isSelected=true implies
self.part.domainFeature->select(isSelected=true)->size>=1
```

DomainFeatureDependency

[1] Se duas DomainFeatures estão relacionadas pela regra **requires**, então a instanciação da DomainFeature no papel de source implica na instanciação da DomainFeature no papel de target.

```
self.stereotype.name='requires' and self.source.isSelected=true
implies
self.target.isSelected=true
```

[2] Se duas DomainFeatures estão relacionadas pela regra **mutexWith**, então a instanciação de uma delas implica na não-instanciação da outra.

```
(self.stereotype.name='mutexWith' and self.source.isSelected=true
implies
self.target.isSelected=false) or
(self.stereotype.name='mutexWith' and self.target.isSelected=true
implies
self.source.isSelected=false)
```

5 Conclusão e Trabalhos Futuros

O presente trabalho descreve cuidadosamente uma proposta de incorporação do modelo de features [5] à linguagem UML. Embora a alteração do meta-modelo da UML só possa ser efetivada através da aceitação da mesma por parte da OMG, é nossa intenção construir uma ferramenta que permita a integração do Modelo de Features com os modelos gerados por ferramentas CASE tais como o Rational Rose [11], o ArgoUML [12] e o Together [13]. Tal integração será possível graças à utilização do padrão XMI (uma especificação da linguagem UML no padrão XML [14]) na representação dos modelos gerados por tais ferramentas. Além disso, a ferramenta a ser construída permitirá a verificação das regras de instanciação durante o processo de especificação de um membro de uma família de aplicações.

6 Bibliografia

[1] Prieto-Díaz, R. *Historical Overview*. In Advances in Software Reuse: Selected Papers from the Second International Workshop on Software Reusability, p.1-16, March 1993, Lucca, Italy. Edited by Ruben Prieto-Diaz and William B. Frakes, IEEE Computer Society Press, 1993.

[2] Arango, G. *Domain Analysis Methods*. In Advances in Software Reuse: Selected Papers from the Second International Workshop on Software Reusability, p.17-49, March 1993, Lucca, Italy. Edited by Ruben Prieto-Diaz and William B. Frakes, IEEE Computer Society Press, 1993.

[3] Parnas, D.L. *On the Design and Development on Program Families*. IEEE Transaction on Software Engineering, Vol. SE-2, No. 1, March 1976.

[4] Kang, K.C.; Cohen, S.G.; Hess, J.A.; Novak, W.E. and Peterson, A.S. *Feature-Oriented Domain Analysis (FODA) Feasibility Study (CMU/SEI-90-TR-21)*. Pittsburgh, PA, Software Engineering Institute, Carnegie Mellon University, Nov 1993.

[5] Kang, K.C.; Kim, S.; Lee, J.; Kim, K.; Shin, E.; Huh, M. *FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architecture*. Department of Computer Science and Engineering, Pohang University of Science and Technology, Korea, 1998.

[6] Passeti, A.; Pree, W. *Two Novel Concepts for Systematic Product Line Development*. in "Software Product Lines - Experience and Research Directions", P. Donohoe (ed), Kluwer Academic Publishers, 2000 (First Software Product Line Conference, organized by CMU/SEI, Denver, Colorado, 28-31 August 2000).

[7] OMG Unified Modeling Language Specification, Version 1.3, June 1999. Found at http://www.omg.org/technology/uml/index.htm.

[8] Jacobson, I.; Booch, G.; Rumbaugh, J. *The Unified Software Development Process*. Addison-Wesley, Reading, Massachusetts, February 1999.

[9] Warmer, J.; Kleppe, A. *The Object Constraint Language – Precise Modeling with UML*. Addison-Wesley, Reading, Massachusetts, October 1998.

[10] OMG Meta-Object Facility, Version 1.3, April 2000. Found at http://www.omg.org/technology/documents/formal/mof.htm.

[11] Rational Rose description at http://www.rational.com/products/rose/index.jsp.

[12] Argo/UML Description found at http://www.argouml.org.

[13] Together Description found at http://www.togethersoft.com.

[14] XML Specification found at http://www.w3.org/XML/.