A DOCUMENTAÇÃO E A INSTANCIAÇÃO DE FRAMEWORKS ORIENTADOS A OBJETOS.

Autor: Ivan Mathias Filho

Orientador: Prof. Doutor Carlos José Pereira de Lucena

IVAN MATHIAS FILHO

A DOCUMENTAÇÃO E A INSTANCIAÇÃO DE FRAMEWORKS ORIENTADOS A OBJETOS.

Tese apresentada ao Departamento de Informática da Pontifícia Universidade Católica do Rio de Janeiro como parte dos requisitos para a obtenção do título de Doutor em Ciências em Informática.

Orientador: Prof. Doutor Carlos José Pereira de Lucena

Departamento de Informática

Pontifícia Universidade Católica do Rio de Janeiro

Rio de Janeiro, 17 de Abril de 2002

Para minha esposa Luana, o grande amor da minha vida.

Agradecimentos

Aos meus filhos, Gabriel e Lívia, e a minha esposa, Luana, pelos vários finais de semana que deixamos de estar juntos.

Aos meus pais, Ivan e Terezinha, pelo amor e carinho com que me criaram.

Ao meu pai, que desde pequeno me ensinou a gostar e a admirar a nossa Ciência.

Ao meu orientador, Carlos José Pereira de Lucena, pela confiança e pelo apoio que me deu durante toda esta jornada.

Aos meus ex-colegas da FICAP e da Brahma, pelos valiosos ensinamentos que me foram passados no início da minha carreira.

Aos meus colegas Fernando Amorelli, Pedro Cardoso, Paulo César Moura Cabral e Ricardo Mutschaewski, com quem aprendi a ser um profissional de informática.

Aos meus colegas de coordenação do curso de Bacharelado em Informática da PUC-RIO, Sandra, Edmundo e Luiz Fernando, pela ajuda que sempre me prestaram.

Aos meus colegas do TECCOMM, pelas valiosas críticas que foram feitas durante a elaboração desta tese.

Ao meu grande amigo e colega Toacy Cavalcante de Oliveira, cuja estreita colaboração foi fundamental para a concretização deste trabalho.

Homenagem

Durante os quase oito anos em que fui aluno do Departamento de Informática da PUC-RIO, primeiro como aluno de mestrado, e depois como aluno de doutorado, tive o prazer de conviver com os professores Sérgio Eduardo Rodrigues de Carvalho, meu primeiro orientador, e José Lucas Mourão Rangel Netto. Infelizmente não posso compartilhar com eles este momento de alegria pois ambos já não mais se encontram entre nós. Entretanto, quero deixar aqui registrada a minha imensa satisfação e felicidade por ter sido aluno destes dois grandes professores.

Resumo

A economia competitiva dos dias atuais gera uma enorme demanda por produtos de software cada vez mais baratos, eficientes, confiáveis e que possam ser desenvolvidos sob uma forte restrição de custo e de tempo. Tais premissas sugerem que o desenvolvimento de software ocorra em um ambiente onde soluções previamente testadas e aprovadas possam ser modificadas, combinadas e adaptadas para serem empregadas na construção de novos produtos. Neste contexto, o uso de frameworks orientados a objetos, ou simplesmente frameworks, apresenta-se como uma das mais promissoras técnicas para a reutilização de código e, principalmente, de design. Contudo, um dos grandes entraves ao uso em larga escala da tecnologia de frameworks é o longo tempo necessário para que um *desenvolvedor* de aplicações se torne proficiente no uso de um determinado framework. Isto se deve, em grande parte, à inerente complexidade do design de um framework, que é projetado para atender aos requisitos de toda uma família de aplicações.

Buscando solucionar esta questão este trabalho irá apresentar uma descrição detalhada de um processo de instanciação de alto nível baseado no Modelo de *Features*. A função deste modelo é apresentar aos *desenvolvedores* de aplicação uma visão simplificada das características funcionais e tecnológicas disponibilizadas por um determinado framework. Neste contexto, o processo de instanciação aqui proposto irá alterar os modelos de design originais de um framework, a partir das escolhas feitas no Modelo de *Features*, e criar modelos de design para uma aplicação específica.

Abstract

The extreme competitiveness of contemporary economy generates a huge demand for cheaper, efficient and reliable software products, which often are developed under great pressures of time and budget. These premises suggests that software development must take place in an environment where proved solutions can be modified, combined and adapted to be used in the development of new products. Therefore, the use o object oriented frameworks, or frameworks for short, seems to be one of the most promising techniques for code and design reuse. Nevertheless, one of the obstacles that have to be removed before the widespread use of frameworks is the great amount of time for study necessary to be proficient in the use of a specific framework. This situation occurs due to the inherent complexity of the design of the frameworks, which are conceived to fulfill the requirements of an entire application family.

In this context, this work presents a detailed description of a high-level framework instantiation process based on the Features Model. This model is responsible for presenting to the application developers a simplified view of the main functional and technological characteristics of a framework. In a following step, the instantiation process modifies the original design of the framework to produce a new design for a specific member of an application family. All of the modification steps will be based on choices available in the Features Model.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A economia competitiva dos dias atuais gera uma enorme demanda por produtos de software cada vez mais baratos, eficientes, confiáveis e que possam ser desenvolvidos sob uma forte restrição de custo e de tempo. Tais premissas sugerem que o desenvolvimento de software ocorra em um ambiente onde soluções previamente testadas e aprovadas possam ser modificadas, combinadas e adaptadas para serem empregadas na construção de novos produtos [Díaz93, Jacobson97, Frakes94]. Neste contexto, o uso de frameworks orientados a objetos, ou simplesmente frameworks, apresenta-se como uma das mais promissoras técnicas para a reutilização de código e, principalmente, de design [Fayad99].

1.1 A Tecnologia de Frameworks

Embora não haja uma definição universalmente aceita, um framework pode ser visto como uma solução genérica para um conjunto de problemas relacionados, ou Domínio de Aplicação [Arango93]. Desta maneira, a reutilização de um framework se dá através da adaptação da solução geral às características particulares de um determinado problema.

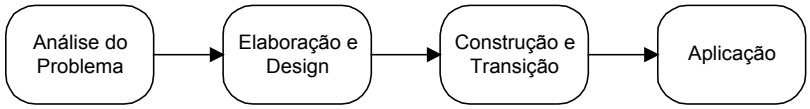
Em termos estruturais, um framework pode ser definido como uma aplicação orientada a objetos semicompleta, composta por uma parte fixa (*frozen-spots*), que incorpora os

aspectos invariáveis de um domínio de aplicação; e por Pontos de Adaptação, também chamados de *hot-spots* [Pree94, Pree97], onde estão contemplados os aspectos variáveis do domínio. Concretamente, o design de um framework é caracterizado por uma coleção de classes concretas e abstratas e de um padrão de colaboração entre estas classes [Johnson88, Wirfs-Brock89], que definem uma arquitetura de referência [Braun92] para toda uma família de aplicações.

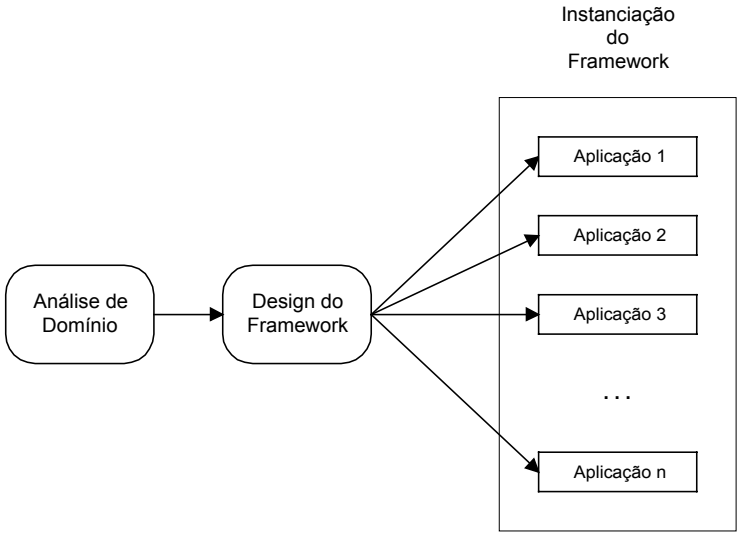
1.2 O Processo de Desenvolvimento de Frameworks

O processo de desenvolvimento de frameworks difere do processo tradicional de desenvolvimento orientado a objetos em dois aspectos fundamentais (Figura 1.1) [Markiewicz01]: em primeiro lugar, enquanto no desenvolvimento tradicional o objetivo é produzir uma única aplicação para atacar um problema bem definido, o desenvolvimento de frameworks irá produzir um design genérico para toda uma família de aplicações [Parnas76], que atacam não apenas um problema em particular, mas toda uma classe de problemas relacionados. Em segundo lugar, o processo de desenvolvimento de frameworks possui uma etapa peculiar, chamada de Instanciação, onde membros específicos de uma família de aplicações são produzidos.

Embora até a presente data não haja uma metodologia específica para o design de frameworks, podemos classificar as abordagens utilizadas em duas grandes categorias: a abordagem *a priori* e a abordagem baseada em *refactoring* [Miller99]. A abordagem *a priori*, também chamada de abordagem *top-down*, parte de uma etapa explícita de Análise de Domínio, onde são elucidados os requisitos comuns e não-comuns a toda uma classe de aplicações, para chegar até o design do framework. Esta abordagem, ilustrada na Figura 1.1, pode utilizar qualquer metodologia de desenvolvimento orientada a objetos que seja baseada em modelos de domínios; como, por exemplo, o método proposto em [Jacobson97].



Processo de Desenvolvimento OO Tradicional



Processo de Desenvolvimento de Frameworks

Figura 1.1 – O Processo de Desenvolvimento de Frameworks

A abordagem baseada em *refactoring*, também chamada de abordagem *bottom-up*, utiliza versões pré-existentes de aplicações para um dado domínio com o objetivo de produzir, através de múltiplas iterações, um design genérico baseado nos aspectos comuns e não-comuns encontrados nas diversas aplicações tomadas como base. Esta abordagem parte do princípio, largamente difundido e aceito pela comunidade de orientação a objetos, de que o desenvolvimento de frameworks requer uma grande dose de experiência e experimentação [Johnson88]. Logo, um framework deve ser construído através de sucessivas etapas de análise e síntese. Nas etapas de análise, o estudo das aplicações pré-existentes, somado ao conhecimento adquirido nas iterações anteriores, é usado para descobrir as estruturas básicas do domínio em mãos. Na etapa de síntese é então produzida, através da utilização de métodos de

generalizações sucessivas [Schmid97], uma coleção de componentes abstratos que irá definir uma arquitetura específica para um domínio de aplicações [Braun92]. Desta forma poderíamos dizer que a abordagem baseada em *refactoring* não produz o design de um framework, e sim converge[1] para um dado design.

O processo de adaptação de um framework às características particulares de um dado problema, e a conseqüente produção de uma aplicação que irá atacar tal problema, é chamado de Instanciação. Tal processo consiste em implementar código específico em Pontos de Adaptação pré-definidos, chamados de *hot-spots*. Dado que o design de um framework é, em última instância, um design orientado a objetos, o processo de instanciação realizar-se-á através de duas técnicas básicas: herança e composição.

Na instanciação por herança (Figura 1.2) a adaptação do framework será realizada através da utilização de mecanismos tais como herança e *overriding* de métodos [Pree99]. Ou seja, o comportamento genérico definido nas classes abstratas de um framework será especializado inserindo-se subclasses, definindo-se os métodos abstratos ou redefinindo-se os métodos que tenham implementação *default*. Eventualmente pode ser necessário introduzir novos atributos nas subclasses ou completar a definição de atributos que necessitem de valores *default* (*inicialização* de um atributo). Em outras ocasiões, quando for recomendável aumentar a flexibilidade de um design, poderá ser necessário aplicar *Design Patterns* tais como o *Strategy* e o *Template Method* [Gamma95] para redefinir o comportamento de um método.

O processo de instanciação por composição (Figura 1.3) utiliza componentes pré-fabricados na adaptação do framework. Neste caso não é necessário conhecer o design do framework com grande profundidade, já que a adaptação será feita *plugando-se* os componentes, que muitas vezes são fornecidos com o próprio framework, nos *hots-*

[1] Termo cunhado pelo professor Arndt Von Staa, do Departamento de Informática da PUC-RIO, em um seminário do PRONEX realizado na PUC-RIO em setembro de 2001.

spots. Em algumas ocasiões, quando o componente não se encaixar perfeitamente no *hot-spot*, será necessário utilizar *Design Patterns* tais como o *Adapter* e o *Bridge* [Gamma95] para que a *interface* do componente seja adaptada ao *hot-spot*.

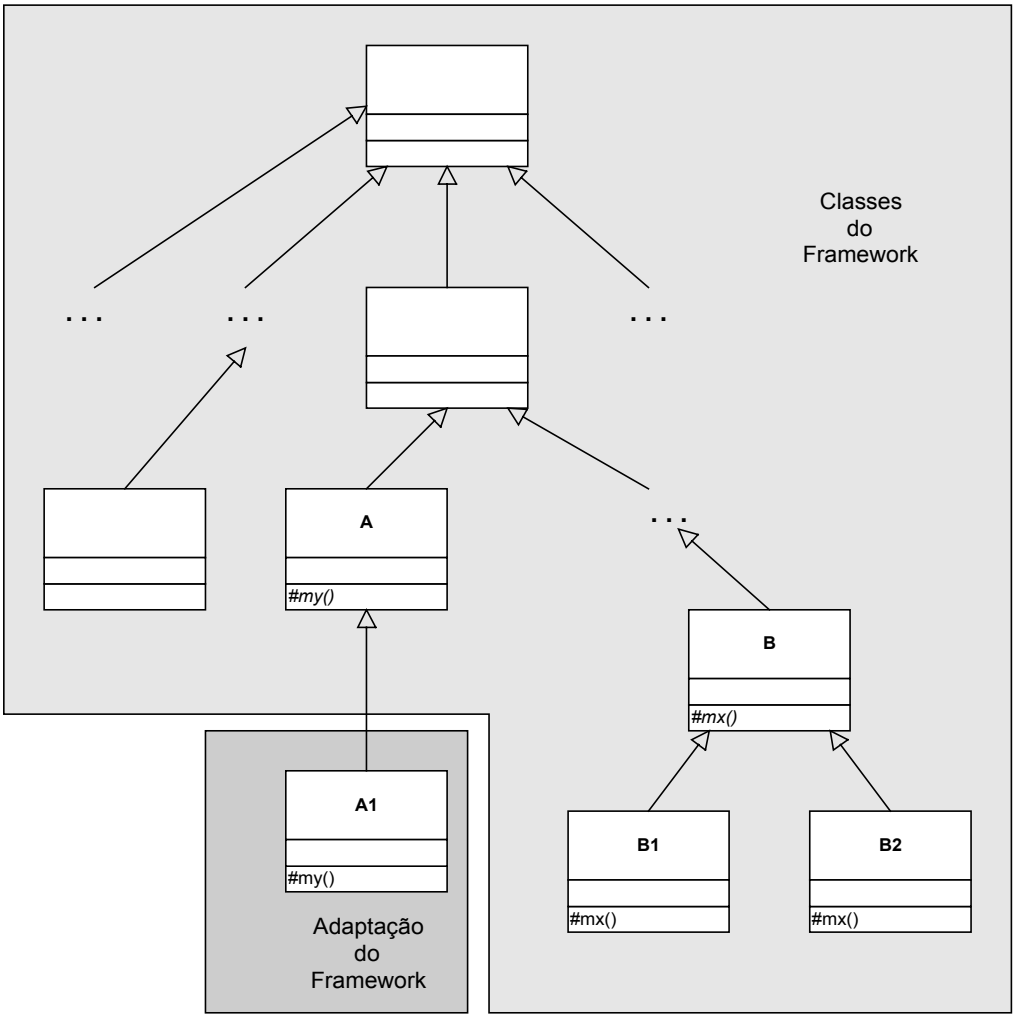


Figura 1.2 – Adaptação do Framework por Herança.

Quando um framework é adaptado através de herança diz-se que ele é um framework *white-box* [Johnson88, Pree99]. O termo *white-box* faz alusão ao fato de o design do framework ter que ser bem compreendido para que este método seja eficaz. No caso de a adaptação do framework ser realizada via composição será necessário apenas conhecer as *interfaces* externas dos componentes para que a instanciação possa ser realizada, não havendo necessidade de se conhecer os detalhes internos de

implementação dos componentes. Neste caso, o framework é chamado de *black-box* [Johnson88, Pree99].

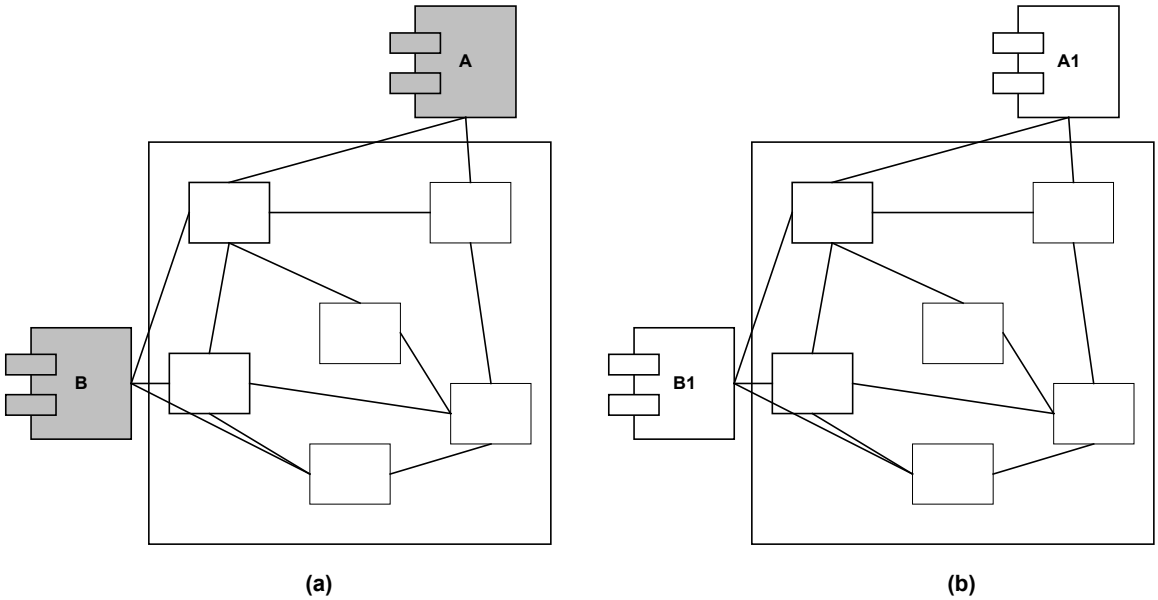


Figura 1.3 – Adaptação do Framework por Composição. Antes (a) e Após (b) a Especialização.

A classificação dos frameworks em *white-box* e *black-box* está longe de ser uma dicotomia. Na verdade um framework típico apresenta alguns *hot-spots* que são implementados por herança e outros que são implementados por composição. Logo, o que existe na verdade é um espectro de variação [Pressman92], onde os frameworks são distribuídos ao longo de um eixo, sendo posicionados mais perto de um extremo caso o método de adaptação utilizado seja predominantemente por herança; ou do outro extremo, caso a adaptação se dê predominantemente por composição.

De um modo geral, à medida que um domínio vai se tornando mais maduro e as mudanças por ele sofridas vão diminuindo, os frameworks para tais domínios tendem a ser predominantemente *black-box*, já que o leque de variações existentes tende a se estabilizar. Logo, a disponibilidade de componentes pré-fabricados para estes domínios tende a aumentar, aumentando-se assim a probabilidade de que um componente

adequado seja encontrado. Desta maneira procura-se evitar a adaptação por herança, que de um modo geral é mais complexa e onerosa.

Como último comentário, é importante destacar que o uso da tecnologia de frameworks cria um ambiente propício à reutilização de componentes pré-fabricados, já que o estabelecimento de uma arquitetura padrão para um domínio de aplicação fornece um contexto bem definido para o uso de tais componentes [Johnson97], minimizando assim o chamado *architectural mismatch* [Garlan95]. Tal fenômeno ocorre quando componentes pré-fabricados não podem ser reutilizados, em conjunto ou isoladamente, pelo fato de terem sido projetados baseados em visões conflitantes dos ambientes nos quais eles seriam inseridos.

1.3 O Problema da Instanciação de Frameworks

Para que um framework possa ser efetivamente reutilizado o seu design tem que ser flexível o bastante para acomodar as diversas variações existentes em um domínio de aplicação. Entretanto, tal característica torna o design do framework complexo, o que dificulta a sua compreensão e, conseqüentemente, o processo de instanciação [Booch94]. Por exemplo, frameworks complexos para o design de interfaces gráficas, tais como a MFC (Microsoft Foundation Class) [Kruglinski98] e a JFC/Swing (Java Foundation Class) [Horstmann99], levam de seis a doze meses, dependendo da experiência dos programadores, para serem bem compreendidos e utilizados de maneira produtiva [Fayad99]. Logo, para que a reutilização de um framework seja economicamente viável seria necessário que os *desenvolvedores* das aplicações instanciadas a partir de tal framework fossem previamente proficientes na sua utilização, ou que o custo de aprendizado da ferramenta fosse amortizado através da sua reutilização em vários projetos. É perfeitamente compreensível, entretanto, que em muitas situações ambas as alternativas sejam inviáveis; principalmente quando o

framework for usado para gerar uma única instância. Este seria o caso, por exemplo, de uma empresa de locação de veículos que resolvesse adquirir um framework para produzir uma aplicação que controlasse os procedimentos de aluguel dos seus automóveis. Em tal situação seria inviável economicamente treinar toda uma equipe para dominar o design do framework, já que este seria instanciado uma única vez. Além disso, seria pouco provável, devido à especificidade da situação, que fossem encontrados, em tempo hábil, profissionais disponíveis no mercado que conhecessem o framework.

Para contornar este problema é necessário que a reutilização de um framework seja calcada em uma documentação que permita a instanciação de aplicações sem que seja necessário conhecer os meandros do design do framework. Idealmente, o processo de instanciação deve ser suportado por uma ferramenta que permita que os *desenvolvedores* de aplicação, ou usuários do framework, selecionem as funcionalidades desejadas e produzam diretivas que os guiem, de maneira semi-automática, na geração de uma aplicação que incorpore tais funcionalidades [Ortigosa00a]. Para isso é fundamental que a documentação do framework forneça uma visão de alto nível da funcionalidade por ele disponibilizada.

1.4 A Documentação de um Framework

A documentação de um framework pode ser dividida em duas grandes categorias, cada uma das quais voltada para um grupo de *desenvolvedores* específico. A primeira categoria é dirigida para os projetistas do framework, e tem por objetivo documentar todo o processo de design [Butler97]; desde a etapa de Análise de Domínio, passando pelo design propriamente dito e abordando também o aspecto da evolução do framework. Entretanto, como a questão da escolha de uma metodologia que dê suporte às etapas de design e evolução de frameworks não será abordada nesta

dissertação, não serão feitas também maiores considerações sobre o conjunto de documentos relacionados com estas etapas.

A outra categoria de documentação é voltada para os *desenvolvedores* de aplicação. Tais documentos devem ser projetados para dar o suporte adequado à seleção de um framework que ataque o problema em questão, e a sua posterior reutilização através da instanciação de uma aplicação específica. De acordo com [Johnson92] esta categoria de documentação deve ser composta pelos seguintes itens:

a. **O Propósito de um Framework** – este item tem por objetivo permitir que os *desenvolvedores* de aplicação selecionem um framework que atenda de maneira eficaz às suas necessidades. Ele deve ser simples e objetivo, de tal maneira que possa ser analisado por todos os usuários (*desenvolvedores* de aplicação) potenciais de um framework. Desta maneira, se um framework não for adequado a um determinado problema pode-se descartá-lo antes que seja gasto muito tempo e esforço para compreendê-lo. De um modo geral este item deve fornecer informações sobre o que é, e o que não é, coberto por um framework, além dos seus aspectos fixos e dos Pontos de Adaptação existentes [Butler97].

b. **Como o Framework Deve Ser Usado** – o próximo passo na reutilização de um framework é aprender a produzir aplicações a partir dele. Neste item é importante organizar as informações objetivamente para atender os usuários inexperientes. Este tipo de usuário geralmente não está interessado em conhecer os detalhes do design do framework e sim produzir uma aplicação que resolva o seu problema o mais rápido possível.

c. **O Design de um Framework** – por último devem ser fornecidos documentos mais específicos que apresentem de maneira mais detalhada o design do framework. Este tipo de documentação é mais voltado para usuários experientes,

que desejam conhecer bem o design do framework a fim de explorá-lo na sua capacidade máxima. Isto é feito através da combinação de *hot-spots* de maneira criativa e da produção de aplicações mais flexíveis do que foi anteriormente previsto pelos projetistas do framework.

1.5 O Objetivo da Tese e o Resumo da Solução Proposta

O objetivo desta tese é atacar o problema da instanciação de frameworks, descrito na seção 1.3 desta dissertação, propondo um novo enfoque para a documentação voltada para os *desenvolvedores* de aplicação, e descrevendo todo um processo de instanciação de frameworks baseado na documentação proposta. Além disso, será descrita uma ferramenta que irá permitir a instanciação semi-automática de frameworks a partir da implementação das idéias que serão aqui apresentadas.

O enfoque adotado deverá atender aos três quesitos para a documentação de frameworks propostos em [Johnson92], e já abordados na seção 1.4 desta dissertação. Além disso, a abordagem proposta está em sintonia com as considerações feitas em [Ortigosa00a]; isto é, a instanciação de frameworks deve ser baseada preferencialmente em uma visão das características funcionais e tecnológicas disponibilizadas pelos frameworks, sem que seja necessário recorrer ao estudo minucioso do seu design.

Antes de partir para explanação da abordagem é necessário tecer alguns comentários adicionais sobre o que está sendo proposto:

a. O enfoque ora em questão não exclui as formas de documentação tradicionais, que serão analisadas no Capítulo 2 desta dissertação. O que será advogado nesta tese é que a forma de documentação a ser apresentada, juntamente com a ferramenta

de auxílio à instanciação, deve ser vista como a maneira mais simples e direta de produzir aplicações baseadas em frameworks.

b. Os usuários de frameworks (*desenvolvedores* de aplicação) mais beneficiados com esta abordagem serão os mais inexperientes. Isto não quer dizer que os *experts* no uso de um determinado framework não sejam beneficiados com a proposta. No mínimo eles poderão usá-la para experimentar novas idéias que lhes tenham ocorrido. Todavia, como já foi visto na seção 1.3, os maiores encargos no uso de frameworks ocorrem quando usuários eventuais, com pouca ou nenhuma experiência no uso de um determinado framework, desejam instanciá-lo poucas vezes; apenas o necessário para produzir uma única aplicação.

c. A abordagem deve ser integrada aos ambientes CASE orientados a objetos tais como o Rational Rose [Rose], o Argo/UML [Argo] e o Together [Together]. Desta forma será mais simples disponibilizar o método de instanciação aqui proposto, uma vez que toda a interface gráfica e todo o *background* de persistência seriam reutilizados. Além disso, os usuários de frameworks poderiam aproveitar todo o conhecimento adquirido no uso de tais ferramentas.

d. O processo de instanciação aqui proposto é mais adequado à instanciação de frameworks *white-box*. Todavia, ele poderá ser aplicado com igual sucesso na instanciação de frameworks *black-box* se os componentes usados na instanciação forem classes pré-fabricadas. Desta maneira a aplicação de tais componentes será feita através de mecanismos de composição, que permitirão conectar um determinado componente a um Ponto de Adaptação de um framework.

1.5.1 A Descrição da Abordagem

O primeiro passo na direção de um processo de instanciação baseado nas funcionalidades disponibilizadas por um framework é a produção de uma documentação de alto nível a partir da qual os usuários possam escolher as características que serão incorporadas às aplicações a serem instanciadas. Esta tarefa está intimamente ligada aos métodos utilizados para a construção do framework. Por esta razão, as etapas que serão executadas para a produção de tal documentação não serão objeto de estudo desta tese, apenas uma descrição sucinta do processo será apresentada nos parágrafos seguintes. O presente trabalho partirá do princípio de que a documentação necessária ao processo de instanciação será disponibilizada, de maneira integrada ao design do framework, pela equipe responsável pelo desenvolvimento do mesmo.

O processo de documentação que será proposto, resumido na Figura 1.4, tem como entrada um modelo de classes, representado na linguagem UML [Booch99], que descreva a visão estática do design do framework. É importante destacar que este modelo, por usar a linguagem UML padrão, não é capaz de representar de maneira objetiva os *hot-spots* existentes no framework [Fontoura99]. Estes estarão dispersos através das inúmeras interfaces e classes abstratas presentes no design.

A partir deste modelo os projetistas do framework irão produzir a documentação que irá auxiliar o processo de instanciação. Desta documentação fará parte o Modelo de Classes original acrescido de algumas informações complementares. Tais informações têm por objetivo identificar claramente no design os *hot-spots* existentes no framework. Entretanto, por não estarem definidas na linguagem UML-padrão, a inclusão de tais informações implicará na necessidade de se alterar o meta-modelo da UML.

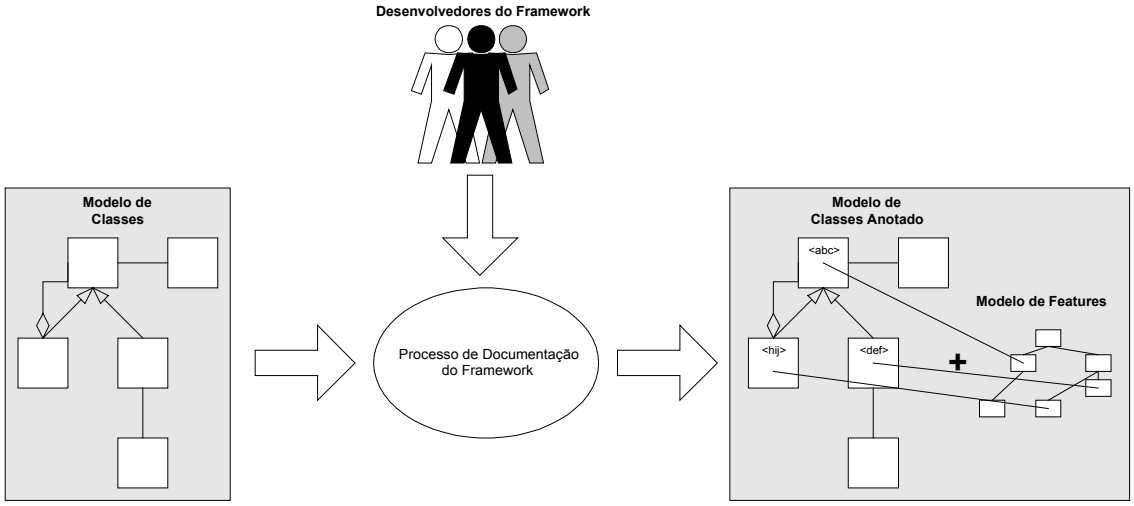


Figura 1.4 – Resumo do Processo de Documentação de Frameworks.

O outro componente da documentação é um modelo de alto nível que descreva as características funcionais e tecnológicas do framework. Este modelo será baseado em uma adaptação para a UML do Modelo de *Features*, proposto em [Kang93] e [Kang98], e que será visto de maneira mais detalhada no Capítulo 3 desta dissertação. É importante ainda destacar que os dois modelos estarão relacionados por *traces* (relacionamentos de dependência com o estereótipo <<trace>> [Booch99]) que partem das descrições das características do framework e chegam nos elementos do design que implementam os Pontos de Adaptação. Com isso será possível estabelecer um relacionamento entre as características funcionais e tecnológicas oferecidas por um framework e os elementos do design que implementam tais características.

No contexto da abordagem proposta o processo de instanciação de um framework começa com a seleção das características que serão incluídas na aplicação que será produzida. As características selecionadas (marcadas em cinza escuro no Modelo de *Features* da Figura 1.5), juntamente com o Modelo de Classes estendido, serão fornecidas como entrada para uma ferramenta (descrita no Capítulo 5 desta dissertação) que irá gerar um *script* contendo todos os passos que deverão ser

executados para instanciar uma aplicação com as características requeridas. A Figura 1.5 a seguir ilustra este processo.

Vale aqui destacar que a etapa anteriormente descrita não é totalmente automática, sendo preciso a intervenção dos *desenvolvedores* de aplicação para suprir algumas informações necessárias.

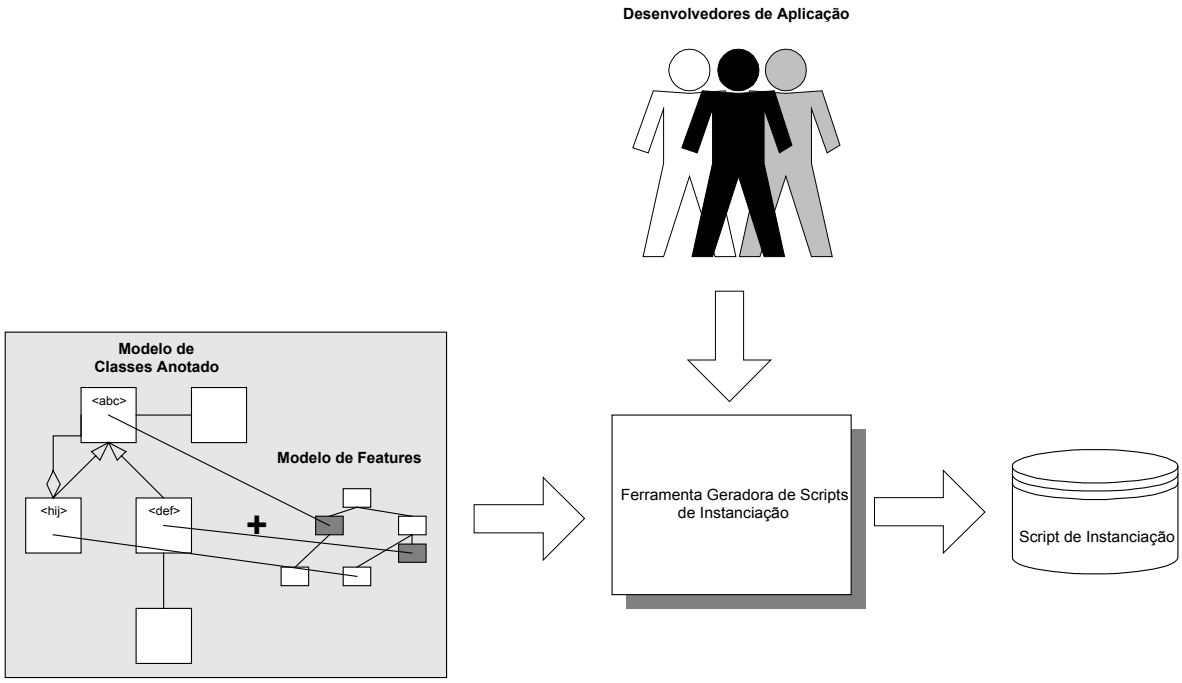


Figura 1.5 – A Geração do Script de Instanciação.

O último passo do processo de instanciação aqui proposto é executar o *script* gerado na etapa anterior. Isto será realizado através de uma ferramenta que irá produzir alterações no design original do framework (com a intervenção dos *desenvolvedores* de aplicação) de acordo com as instruções contidas no *script*. Na Figura 1.6 abaixo é possível observar o design modificado. As classes inseridas para atender aos requisitos da aplicação a ser produzida estão marcadas em cinza escuro.

Neste ponto é necessário fazer duas observações: em primeiro lugar temos que destacar que esta última etapa não é capaz de produzir um programa executável. Será preciso gerar o código a partir do design modificado e acrescentar o código novo,

específico para implementar as características escolhidas. Para executar esta tarefa, que está fora do escopo desta tese, será necessário consultar outros modelos do design [Larman98]., tais como contratos [Helm90], diagramas de seqüência, diagramas de transição de estados e etc.

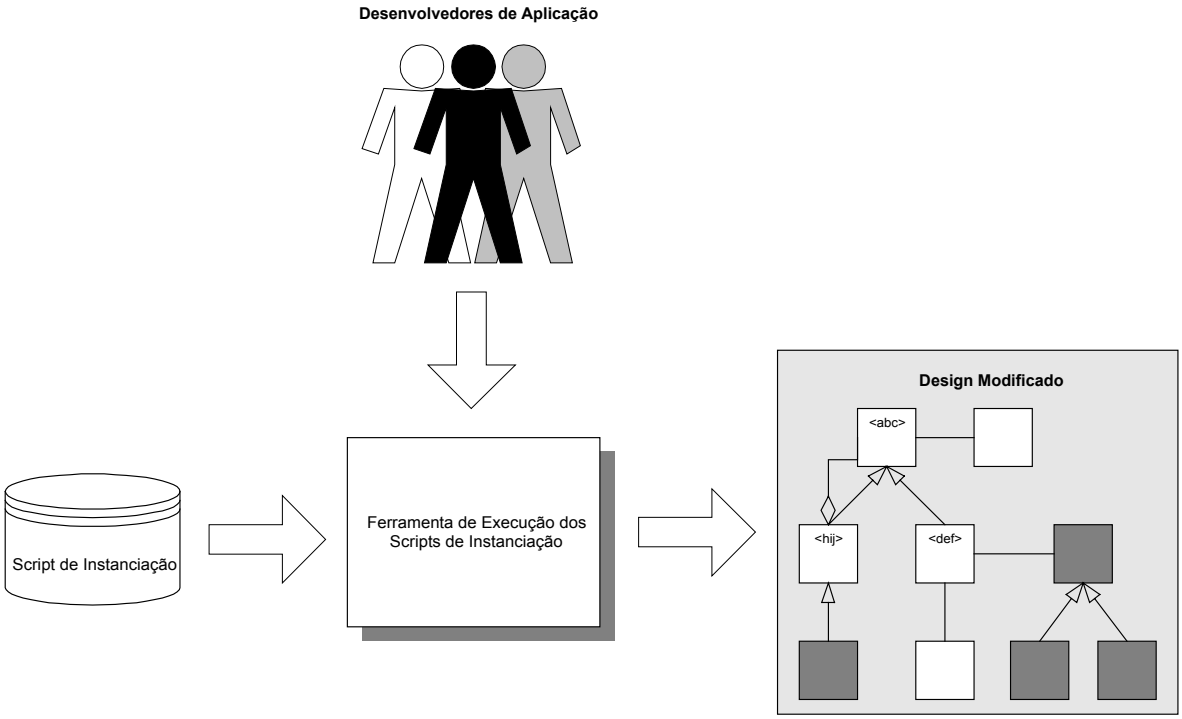


Figura 1.6 – A Produção do Design Modificado.

A segunda observação diz respeito à linguagem na qual será representado o *script* de instanciação e à ferramenta que irá executá-lo. Para suprir estes requisitos será utilizada a linguagem RDL (*Reuse Description Language*) proposta em [Oliveira01a]. No referido trabalho, além da linguagem em si, é descrita a implementação de uma máquina virtual capaz de executar, de maneira semi-automática, um *script* de instanciação representado em RDL.

1.6 Lista das Contribuições da Tese

De maneira sucinta, são as seguintes as contribuições feitas por esta tese:

a. A proposição de um método sistemático e objetivo para a instanciação de frameworks a partir das características funcionais e tecnológicas do mesmo.

b. A descrição precisa do Modelo de *Features* através da linguagem OCL.

c. A Integração do Modelo de *Features* com a linguagem UML e com outros modelos utilizados na descrição da arquitetura de um framework. Para tal será apresentado um conjunto de regras que irão governar todo o processo de integração dos modelos em questão. A definição precisa destas regras será significativamente facilitada pela tarefa descrita no item b.

d. A construção de uma ferramenta para dar suporte ao processo de instanciação de frameworks, desde a fase da seleção das características desejadas até a produção de um *script* de instanciação.

Além disso, no Apêndice D será apresentado um resumo das publicações que foram produzidas no decorrer do desenvolvimento desta tese.

1.7 Estrutura da Tese

O presente trabalho está estruturado da seguinte maneira:

Capítulo 2 Apresentação de alguns dos principais trabalhos que tratam da questão da documentação e da instanciação de frameworks.

Capítulo 3 Descrição detalhada do processo de integração do Modelo de *Features* com a linguagem UML.

Capítulo 4 Descrição do processo de integração do Modelo de *Features* com o Modelo de Classes que descreve a arquitetura de um framework. Também serão apresentados em detalhe todos os passos, e todas as

regras, referentes à geração de um script de instanciação a partir das escolhas feitas no Modelo de *Features*.

Capítulo 5 Descrição detalhada da ferramenta que deverá ser usada para dar suporte a todo o processo de instanciação de frameworks que será aqui proposto. Serão também fornecidos os detalhes da implementação de um protótipo da referida ferramenta.

Capítulo 6 Descrição de alguns estudos de casos que servirão como prova de conceito das idéias que serão apresentadas.

Capítulo 7 Apresentação das conclusões desta tese.

Capítulo 8 Apresentação de algumas propostas para trabalhos futuros.

Apêndice A Apresentação das regras do processo de integração descrito no Capítulo 3.

Apêndice B Definição das regras de construção do Modelo de *Features* e do processo de geração de um *script* de instanciação, ambos visto no Capítulo 4.

Apêndice C Apresentação de alguns modelos que descrevem os detalhes do design do protótipo da ferramenta descrita no Capítulo 5.

Apêndice D Descrição sucinta das publicações produzidas durante o desenvolvimento deste trabalho.

Capítulo 2

Trabalhos Relacionados

O objetivo deste capítulo é analisar algumas das principais propostas que tratam do problema da documentação de frameworks voltada para os *desenvolvedores* de aplicação. Embora não pretenda ser exaustiva, a relação dos trabalhos que serão abordados contém as mais importantes contribuições apresentadas ao longo dos últimos quinze anos, e que estão presentes nas referências bibliográficas da maioria das publicações que tratam da questão da produção de uma documentação que facilite o processo de instanciação de um framework.

Para permitir uma melhor compreensão por parte do leitor os trabalhos analisados a seguir foram classificados de acordo com a abordagem predominantemente utilizada, embora muitos deles sirvam-se de uma combinação de várias técnicas de documentação. Além disso, a apresentação segue a ordem cronológica de publicação dos trabalhos. Assim será possível fornecer uma visão panorâmica da evolução das propostas de documentação de frameworks ao longo do tempo.

Para finalizar, a última seção deste capítulo foi reservada para a apresentação de um quadro comparativo dos trabalhos analisados. Este quadro foi construído a partir dos requisitos de documentação apresentados em [Johnson92], e listados na seção 1.4 desta dissertação. Com isso será possível observar com mais clareza os aspectos positivos e negativos de cada um dos métodos apresentados, e estabelecer uma base

comparativa para avaliar a abordagem proposta por esta tese em relação às abordagens existentes atualmente.

2.1 Técnicas de Documentação Baseadas em *Cookbooks*

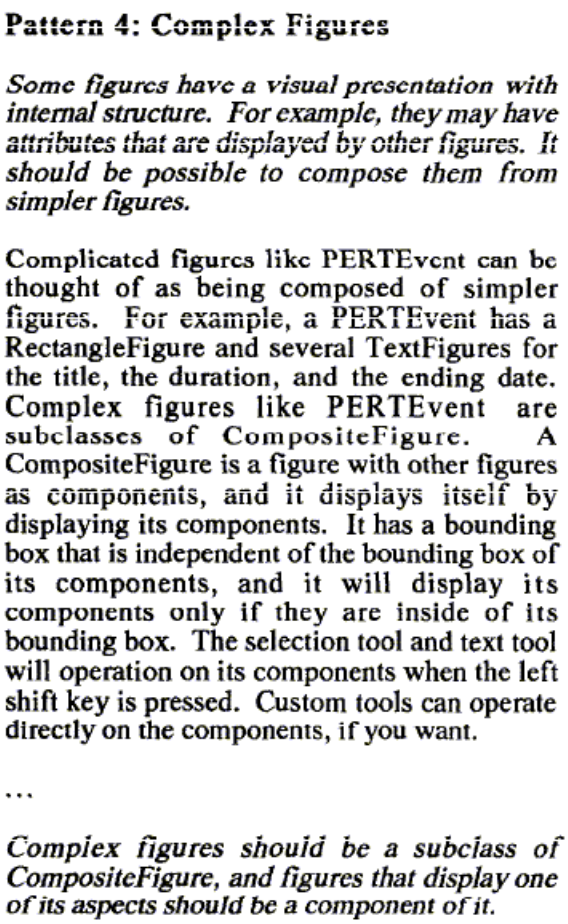
Um dos primeiros trabalhos a abordar explicitamente a questão de como documentar um framework de maneira que o processo de instanciação possa ser conduzido sem que os *desenvolvedores* de aplicação tenham que estar cientes dos detalhes do design foi a abordagem baseada em *cookbooks*, descrita em [Krasner88]. Um *cookbook* é um *tutorial* que ensina aos seus leitores como instanciar aplicações. Ele é usado para documentar um framework no que tange às descrições do seu propósito geral, dos seus componentes principais e de alguns aspectos interessantes do design. Além disso, a documentação é complementada com alguns exemplos que ilustram a maneira pela qual os componentes são usados. Os *cookbooks* são divididos em seções, chamadas de receitas, que descrevem problemas específicos e fornecem seqüências de passos que devem ser seguidas para que tais problemas sejam resolvidos.

O modelo de documentação apresentado em [Krasner88] é puramente narrativo e não-estruturado, o que torna difícil a construção de ferramentas que auxiliem o processo de instanciação. Além disso, o trabalho em questão não propõe nenhum mecanismo que descreva de maneira objetiva o design do framework e os relacionamentos existentes entre as receitas e os locais do design por elas afetadas.

Um outro método de documentação importante [Johnson92], que também pode ser classificado como sendo baseado em *cookbooks*, usa o conceito de padrões de instanciação[2] para descrever como um framework deve ser usado. Os padrões de instanciação são organizados na forma de um grafo direto [Gersting82], onde um nó

[2] Termo usado pelo autor desta dissertação para que não haja confusão entre os Patterns de [Johnson92] e os Design Patterns de [Gamma95].

representa um dado padrão de instanciação e um arco ligando dois nós representa uma referência direcionada de um padrão para outro. Cada padrão de instanciação descreve um problema que ocorre freqüentemente no domínio de aplicação de um framework, e a maneira pela qual tal problema é resolvido (Figura 2.1).



Pattern 4: Complex Figures

Some figures have a visual presentation with internal structure. For example, they may have attributes that are displayed by other figures. It should be possible to compose them from simpler figures.

Complicated figures like PERTEvent can be thought of as being composed of simpler figures. For example, a PERTEvent has a RectangleFigure and several TextFigures for the title, the duration, and the ending date. Complex figures like PERTEvent are subclasses of CompositeFigure. A CompositeFigure is a figure with other figures as components, and it displays itself by displaying its components. It has a bounding box that is independent of the bounding box of its components, and it will display its components only if they are inside of its bounding box. The selection tool and text tool will operation on its components when the left shift key is pressed. Custom tools can operate directly on the components, if you want.

...

Complex figures should be a subclass of CompositeFigure, and figures that display one of its aspects should be a component of it.

Figura 2.1 – Exemplo de um Padrão de Instanciação.

A descrição de um padrão de instanciação é estruturada da seguinte maneira:

a. *Descrição do Problema*.

b. *Discussão detalhada dos diferentes modos de solucionar o problema* – neste item são usados exemplos de soluções, são apresentados alguns detalhes do design e, eventualmente, são mostrados trechos do código do framework. Referências para outros padrões de instanciação também são comuns.

c. *Sumário da solução com referências para outros padrões de instanciação e para outras partes do framework*.

A documentação é organizada de maneira que o padrão inicial seja responsável pela apresentação do propósito do framework e pelo estabelecimento da seqüência na qual os demais padrões devem ser lidos, auxiliando assim os *desenvolvedores* de aplicação a navegar pela rede de padrões de instanciação que constituem a documentação de um framework. Embora o trabalho em questão não apresente nenhuma ferramenta para construir a documentação, é dito que o uso de ferramentas baseadas em hipertexto parece ser a melhor opção.

Diferentemente dos *cookbooks* propostos em [Krasner88], que descrevem a maneira típica pela qual um framework é usado, os padrões de instanciação apresentam-se como pares *problema-solução*. Desta forma, a solução de um problema maior será obtida através da decomposição deste em subproblemas, e da aplicação das soluções existentes para cada um dos subproblemas. Uma crítica que pode ser feita a esta abordagem é que, apesar de existirem referências para outros padrões relacionados, não existe uma descrição clara dos problemas que podem surgir quando da composição das soluções, devido aos inter-relacionamentos existentes entre elas.

Uma outra proposta, chamada *Active Cookbooks* [Sommerlad95, Pree95], ataca um dos pontos frágeis dos métodos baseados em *cookbooks* ao criar uma estrutura de hipertexto que permite a representação das interdependências existentes entre as receitas de um *cookbook*. O uso de hipertexto permite também o emprego de ferramentas gráficas que exibam os elementos do design associados a uma determinada receita (Figura 2.2), além de exemplos que possam ilustrar o processo de instanciação. Outra melhoria que merece ser destacada é a introdução de uma ferramenta que automatiza parcialmente o processo de instanciação com o auxílio dos

usuários do framework. Esta ferramenta gera parte do design de uma aplicação a partir da *execução* das receitas do *cookbook*.

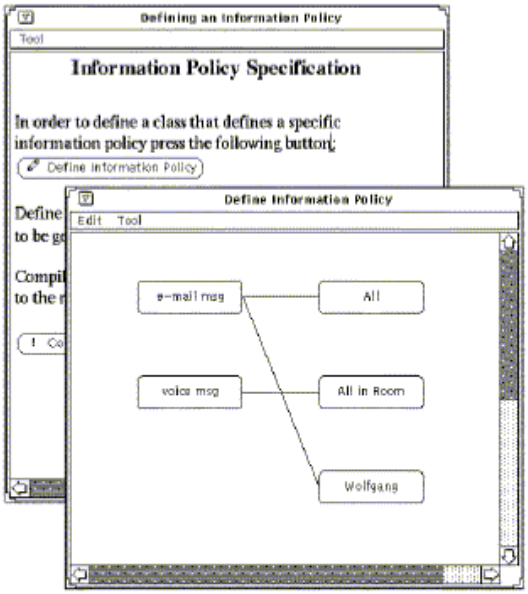


Figura 2.2 – Associação entre uma Receita e Parte do Design.

Apesar das melhorias introduzidas, os *Active Cookbooks* sofrem de uma importante deficiência inerente às abordagens baseadas em *cookbooks*: o método ainda conserva muito da forma de *tutorial*, não fornecendo uma estratégia adequada para a seleção das receitas que resolvem um determinado problema.

Em [Froehlich97] é apresentada uma abordagem muito semelhante aos padrões de instanciação propostos em [Johnson92]. Este trabalho, baseado no conceito de *hooks*, é direcionado para a descrição do propósito e do domínio de aplicação de um framework, além de fornecer diretivas de como ele deve ser usado. Cada *hook* fornece uma solução concisa e direcionada para um problema específico. Desta maneira, a solução para problemas complexos é obtida através da combinação de vários *hooks*, de modo semelhante aos padrões de instanciação de [Johnson92]. Um *hook* é descrito em uma linguagem própria e dividido em várias seções (Figura 2.3), embora nem todas sejam aplicáveis a todos os *hooks*.

- Name: a unique name, within the context of the framework, given to each hook.
- Requirement: the problem the hook is intended to help solve.
- Type: an ordered pair consisting of the method of adaption used and the amount of support provided for the problem within the framework.
- Area: the parts of the framework that are affected by the hook.
- Uses: the other hooks required to use this hook. The use of a single hook may not be enough to completely fulfill a requirement that has several aspects to it, so this section states the other hooks that are needed to help fulfill the requirement.
- Participants: the components that participate in the hook. These are both existing and new components.
- Changes: the main section of the hook which outlines the changes to the interfaces, associations, control flow and synchronization amongst the components given in the participants section. All changes, including those involving the use of other hooks, are intended to be made in the order they are given within this section.
- Constraints: limits imposed by using the hook, such as configuration constraints.
- Comments: any additional description needed.

Figura 2.3 – As Seções de um *Hook*.

Um exemplo de *hook* é mostrado na Figura 2.4. Nele são descritos, através da seção *Changes*, os passos necessários para incluir uma nova ferramenta na barra de ferramentas de uma aplicação gráfica baseada no framework *HotDraw* [Johnson92]. Embora a presente dissertação não vá entrar nos detalhes da linguagem usada neste exemplo (seção *Changes* da Figura 2.4), é importante destacar que não é citado nenhum mecanismo em [Froehlich97] que permita a execução de um *script* de instanciação baseado em tal linguagem. Uma abordagem muito mais completa e provida de suporte para a execução semi-automática de *scripts* de instanciação é apresentada em [Oliveira01a].

Name: New Tool Type
Requirement: A new type of tool is needed which
is not already provided within the framework.
Type: Enable a Feature, Parameter Pattern
Area: Tools
Participants: icon, cursor, manipulatingCursor,
eventTable, toolName, DefaultTools,
Reader, Figure, Command
Uses: Incorporate Tools, Choose Figure,
Choose Reader, Choose Command, DefaultEvents
Changes:
fill in icon, cursor, manipulatingCursor, toolName
DefaultEvents [eventTable]
repeat as needed
Figure = Choose Figure[ChosenFigure]
Reader = Choose Reader[ChosenReader]
Command = Choose Command
[ChosenCommand]
eventTable add (Figure, Reader, Command)
DefaultTools add (toolName, eventTable, icon,
cursor, manipulatingCursor)
Incorporate Tools[ChosenTools = (Tool, toolName)]
Comments:
When the tool is used, it finds the Figure it was
used on and a Reader for the type of input received,
and then executes the associated Command.
If there is no entry for a particular type of
Figure, an entry for that Figure's superclass will
be consulted.

Figura 2.4 – Exemplo de um *Hook*.

Uma deficiência do método baseado em *hooks* é a ausência de suporte para a seleção dos *hooks* necessários a produção de uma aplicação a partir de um framework, embora o trabalho em questão sugira que os *hooks* devam ser organizados por assunto ou por classes de problemas. Contudo, nenhum protótipo de ferramenta (um *browser* por exemplo) que facilite o estudo e a seleção dos *hooks* adequados foi apresentado. Outra lacuna que pode ser observada em [Froehlich97] é a inexistência de mecanismos que associem os *hooks* aos os elementos do design envolvidos. Esta deficiência dificulta a construção de ferramentas que possam conduzir o processo de instanciação com o mínimo de interferência dos usuários de um framework.

2.2 Técnicas de Documentação Baseadas na Descrição das Colaborações entre Objetos

Ao contrário das abordagens baseadas em *cookbooks*, que têm por objetivo permitir o uso de um framework sem que haja necessidade de entrar nos detalhes do design, alguns trabalhos partem do princípio de que só é possível utilizar adequadamente um framework se tais detalhes forem de amplo conhecimento dos seus usuários; especialmente os aspectos comportamentais do design, capturados sob a forma de colaborações entre objetos [Wirfs-Brock89]. Neste sentido, o trabalho de [Helm90] objetiva descrever os aspectos comportamentais de um framework através do conceito de contrato.

Um contrato é uma descrição formal das obrigações que os objetos têm que cumprir quando se relacionam uns com os outros (Figura 2.5). As obrigações estendem o conceito de tipo, baseado nas assinaturas das operações públicas, e incluem restrições no comportamento dos objetos, tais como a ordem na qual as operações devem ser invocadas. As restrições têm como objetivo representar as interdependências existentes entre os objetos no que se refere aos seus respectivos comportamentos. Os contratos definem também pré-condições [Meyer97], que devem ser satisfeitas pelos participantes para que um contrato possa ser instanciado, e *invariantes* [Meyer97], que devem ser mantidas pelos objetos participantes.

Duas importantes operações definidas sobre os contratos fornecem meios para expressarmos comportamentos complexos em termos de comportamento mais simples (composição de comportamentos). A primeira delas, chamada *Refinement*, permite a especialização das obrigações contratuais, enquanto a segunda, chamada *Inclusion*, permite que contratos possam ser definidos a partir de contratos mais simples (composição de contratos), chamados de sub-contratos.

```
contract SubjectView
    Subject supports [
        value : Value
        SetValue(val:Value) ↦ Δvalue {value = val}; Notify()
        GetValue() : Value ↦ return value
        Notify() ↦ ⟨|| v : v ∈ Views : v→Update() ⟩

        AttachView(v:View) ↦ {v ∈ Views}
        DetachView(v:View) ↦ {v ∉ Views}
        ]
    Views : Set(View) where each  View supports [
        Update() ↦ Draw()
        Draw() ↦ Subject→GetValue() {View reflects Subject.value}
        SetSubject(s:Subject) ↦ {Subject = s}
        ]
    invariant
        Subject.SetValue(val) ↦ ⟨∀v : v ∈ Views : v reflects Subject.value⟩
    instantiation
        ⟨|| v : v ∈ Views : ⟨Subject→AttachView(v) || v→SetSubject(Subject)⟩ ⟩
end contract
```

Figura 2.5 – Exemplo de um Contrato.

A capacidade de especificar de maneira precisa os aspectos dinâmicos de um design orientado a objetos, somada à habilidade de estender e compor comportamentos, faz da abordagem proposta por [Helm90] uma excelente técnica para documentar aspectos do design de um framework. Entretanto, é necessário complementá-la com outras técnicas de documentação, já que tanto a definição do propósito de um framework quanto a descrição de como ele deve ser instanciado não são cobertos por este método.

Outra questão a ser considerada é que os fluxos de controle entre os objetos de um framework não são representados de maneira tão clara como nos diagramas de colaboração existentes na linguagem UML [Larman98]. O fato de um contrato ser uma especificação puramente textual, representada em uma linguagem própria e diferente da linguagem de programação usada na implementação do framework, pode ser considerado como um fator que dificulta a assimilação desta abordagem pelos usuários de um framework.

Como último comentário vale aqui estabelecer a diferença entre os conceitos de contrato usados em [Helm90] e [Meyer92]. Enquanto o primeiro define as obrigações que os objetos têm que cumprir quando se relacionam uns com os outros, o segundo procura expressar o propósito de uma classe através da descrição das propriedades da sua interface (pré-condições, pós-condições e *invariantes*).

Uma outra abordagem muito promissora para descrever o comportamento de um sistema através da colaboração entre objetos é proposta em [Riehle98]. Neste trabalho os elementos fundamentais de modelagem deixam de ser exercidos pelas classes, vistas como abstrações de conceitos encontrados em um domínio de aplicação, e passam a ser exercidos pelos *roles* (papéis). Os *roles*, mais precisamente os *role types*, descrevem as visões que os objetos têm uns dos outros. Em um dado momento um objeto pode agir de acordo com vários *roles* distintos, permitindo assim que diferentes clientes possam ter visões distintas de um mesmo objeto. Além disso, um mesmo *role* pode ser exercido por objetos de diferentes classes.

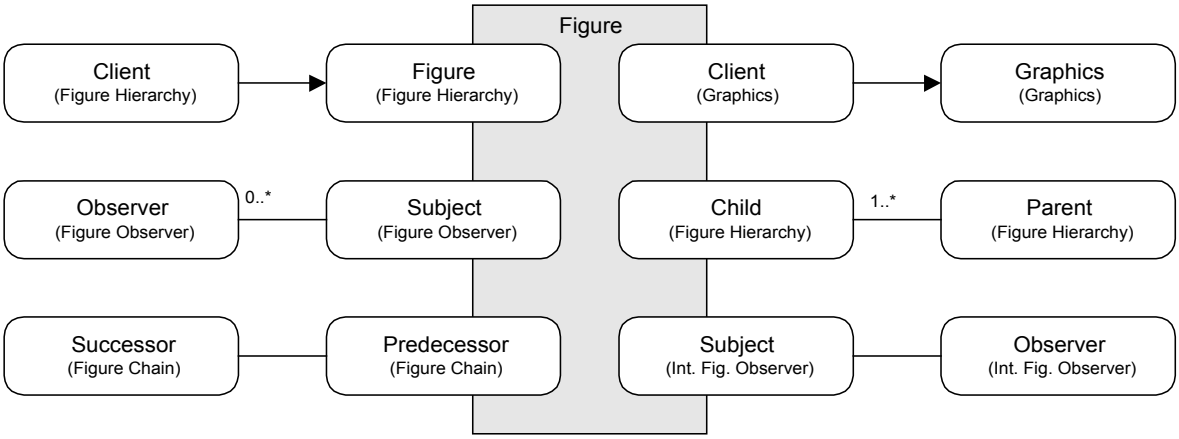


Figura 2.6 – A Classe *Figure* e Seus Diversos *Roles*.

A Figura 2.6 mostra a classe *Figure* (elemento com forma retangular) juntamente com os *roles* (elementos com as bordas arredondadas) que podem ser exercidos pelas suas instâncias. Na mesma figura podemos observar que a interação entre os *roles* é representada por um segmento de reta unindo dois *roles* distintos. A existência de uma

flecha designa uma interação unidirecional, enquanto que a ausência significa que a interação pode ser bidirecional.

Os *roles* são organizados em *role models* (Figura 2.7). Um *role model* descreve um grupo de colaborações com um propósito específico. Os *role models* definem as formas de interação entre os diversos *roles* e estabelecem algumas restrições sobre tais interações. Por exemplo, uma flecha com a extremidade vazada representa a restrição *role-implied*. Tal restrição, aplicada sobre uma interação entre dois *roles*, digamos (A,B), determina que um objeto no papel de A deve estar apto a cumprir também o papel de B. Por sua vez, um segmento de reta com as extremidades bloqueadas determina que um objeto no papel de A não pode exercer o papel de B, e vice-versa.

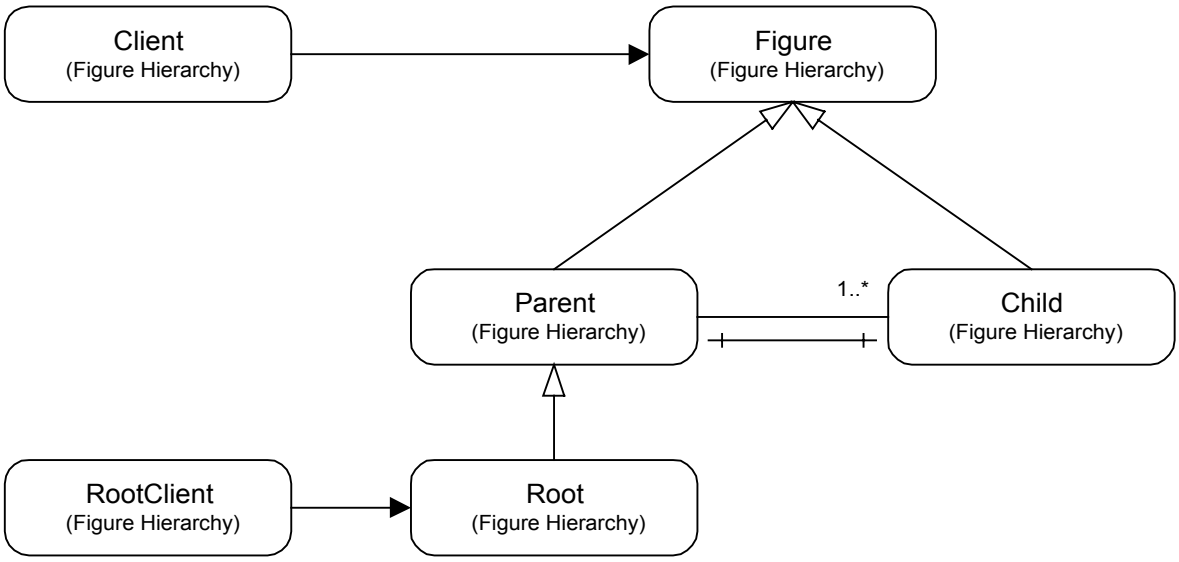


Figura 2.7 – O *Role Model Figure Hierarchy*.

Embora não sejam tão precisos na descrição das colaborações quanto os modelos baseados em contratos, os *role models* levam vantagem sobre estes no que tange à capacidade de representação gráfica e à simplicidade dos artefatos produzidos. Outro aspecto positivo desta abordagem é que os *roles* podem ser implementados através do mecanismo de interfaces disponibilizado pela linguagem Java, enquanto que os

contratos não são suportados por nenhuma linguagem de programação de uso corrente.

Em relação aos frameworks esta abordagem propõe a definição de um grupo de *roles*, chamado *integration role type set*, que deve ser implementado para que o framework seja instanciado. Neste sentido os *role types* cumprem papel semelhante aos *hot-spots*. Logo, na instanciação de um framework por herança (ver seção 1.2) os *roles* definidos no *integration role type set* deverão ser alocados às subclasses criadas durante o processo de instanciação. Já no caso de instanciação por composição (ver seção 1.2), devem ser criados grupos de *roles* nas classes clientes de tal maneira que haja correspondência entre estes e o grupo de *roles* definidos no *integration role type set*. Desta forma, os *roles* atuam como uma ponte entre as classes clientes e os serviços disponibilizados pelo framework.

Embora esta abordagem permita a criação de modelos bastante ilustrativos para a compreensão das interações que ocorrem em um framework, principalmente em relação aos diversos papéis que um objeto pode exercer, ela pouco acrescenta em relação ao processo de instanciação propriamente dito. Não foi concebida nenhuma ferramenta que automatize parte do processo de instanciação e nem foram propostos métodos que facilitem a criação de aplicações a partir de modelos de alto nível.

2.3 Técnicas de Documentação Baseadas em Múltiplos Modelos de Design

Os trabalhos que serão apresentados nesta seção propõem, de uma maneira geral, que a documentação de um framework seja composta por diferentes visões do design. Embora enfatizando detalhes específicos, os diversos modelos devem fornecer aos usuários de um framework uma visão integrada que permita o estudo minucioso dos

elementos do design que serão afetados pelo processo de instanciação. Por exemplo, em [Campbell92] o modelo de design de um framework é composto por um diagrama de hierarquia de classes, um diagrama de entidade-relacionamento [Batini92], uma descrição dos protocolos de interação dos componentes, uma descrição dos aspectos de sincronização entre os componentes e um diagrama de fluxos de controle. Apesar do trabalho em questão usar uma notação própria para representar todos estes modelos, não haveria nenhuma dificuldade para transpô-la para a UML. O diagrama de hierarquia de classes e o diagrama de entidade-relacionamento têm, em conjunto, o mesmo poder de expressão dos diagramas de classes da UML. A diferença está no fato de os relacionamentos de generalização/especialização estarem representados isoladamente no diagrama de hierarquia de classes, enquanto que as associações são capturadas pelo modelo de entidade-relacionamento. A descrição dos protocolos de interação é feita em um diagrama a parte, onde os métodos públicos dos componentes são apresentados.

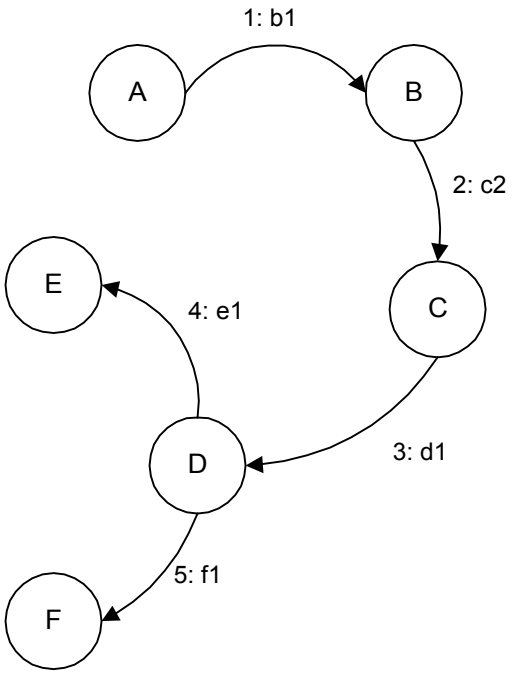


Figura 2.8 – Contrato Estabelecido por um Diagrama de Fluxos de Controle.

O diagrama de fluxos de controle, embora seja uma versão simplificada do diagrama de colaboração da UML [Larman98], vale a pena ser analisado com um pouco mais de cuidado. Este modelo nada mais é do que um grafo que representa de maneira abstrata o fluxo de mensagens [Wirfs-Brock90] entre os objetos de um framework. Os nós A, B, C, D, E e F, vistos na Figura 2.8, representam classes abstratas, enquanto que os arcos representam o envio de mensagens entre os objetos. O grafo como um todo representa uma seqüência válida de mensagens em um framework.

Para instanciar um dado framework será necessário criar subclasses concretas que cumpram o contrato estabelecido pelo diagrama de fluxos de controle. Por exemplo, sejam A', B', C', D', E' e F' subclasses concretas de A, B, C, D, E e F respectivamente. Desta maneira, o diagrama da Figura 2.9 seria uma realização do contrato estabelecido pelo diagrama da Figura 2.8.

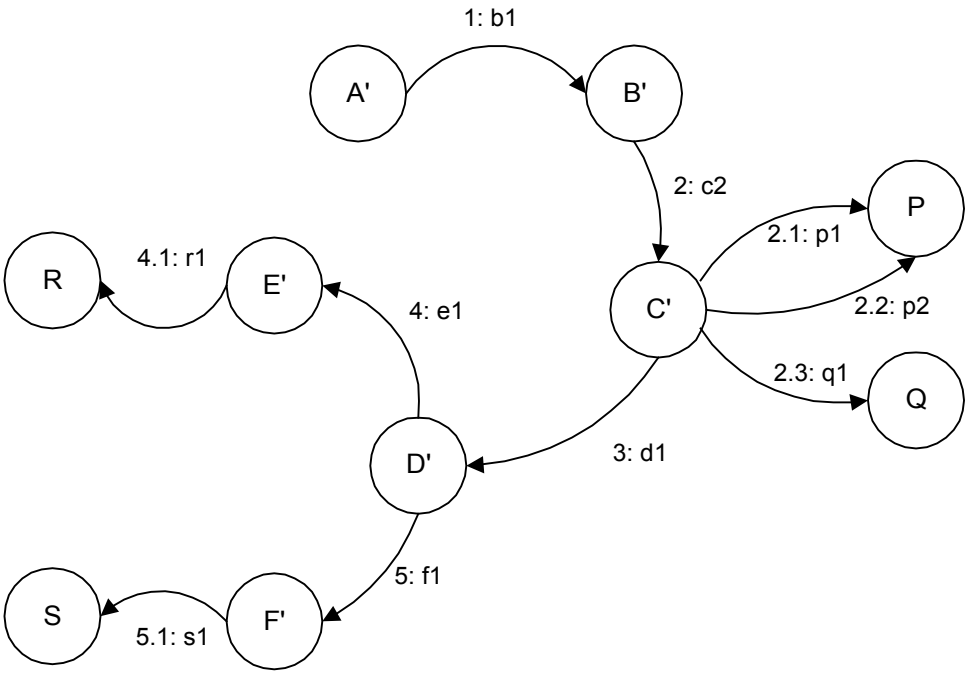


Figura 2.9 – Realização do Contrato Estabelecido pelo Diagrama de Fluxos de Controle.

Veja que o grafo acima acrescenta algumas classes (P, Q, R e S). Apesar disso, as classes concretas A', B', C', D', E' e F' cumprem o contrato estabelecido anteriormente,

embora acrescentem comportamento próprio. Esta técnica é uma maneira simplificada de representar graficamente os contratos propostos por [Helm90].

Embora permita descrever o design de um framework de maneira bastante detalhada este trabalho não fornece meios para documentar claramente o propósito de um framework, além de também não fornecer um roteiro objetivo para instanciá-lo.

Outro exemplo de abordagem que combina diferentes métodos de representar o design de um framework é [Lajoie95]. Este trabalho utiliza uma coleção de receitas para descrever como um framework deve ser adaptado. As receitas de [Lajoie95], chamadas de Motifs, são muito semelhantes aos padrões de instanciação propostos em [Johnson92]. Ou seja, um Motif é uma descrição textual, estruturada em seções, que descreve uma situação típica na qual o framework pode ser empregado, além de fornecer uma seqüência de passos para adaptá-lo.

Um Motif pode fazer referência para outros Motifs que estejam envolvidos em uma dada situação de instanciação. Isto forma uma rede de referências cruzadas que pode ser difícil de ser administrada, apesar de o trabalho em questão não fornecer nenhum suporte de ferramentas para tal. Assim como em [Johnson92], usa-se o primeiro Motif para descrever o domínio de aplicação de um framework, embora esta técnica não forneça uma visão objetiva dos aspectos variáveis e não-variáveis presentes em um framework.

A única diferença significativa deste trabalho em relação ao publicado em [Johnson92] é que os Motifs mantêm referências para os *Design Patterns* [Gamma95] e para os contratos [Helm90] envolvidos na adaptação de um framework. De resto podemos dizer que esta abordagem, apesar de corrigir algumas das falhas encontradas nas abordagens baseadas em *cookbooks* fornecendo, por exemplo, detalhes do design,

não propõe nenhum mecanismo que permita produzir aplicações a partir de uma descrição de alto nível das características presentes em um framework.

O último trabalho que será analisado nesta seção, [Gangopadhyay95], usa uma ferramenta CASE, chamada *ObjectChart*, que permite gerar um diagrama de seqüência (Figura 2.10) ilustrativo da colaboração entre os objetos de um grupo pré-selecionado de classes (Figura 2.11). Como a ferramenta *ObjectChart* é dotada de capacidade de animação, os autores do trabalho em questão propõem que ela seja usada para o estudo e o entendimento do design de um framework. Além disso, a própria ferramenta poderia ser usada no processo de instanciação, dado que através dela é possível alterar diretamente o design do framework.

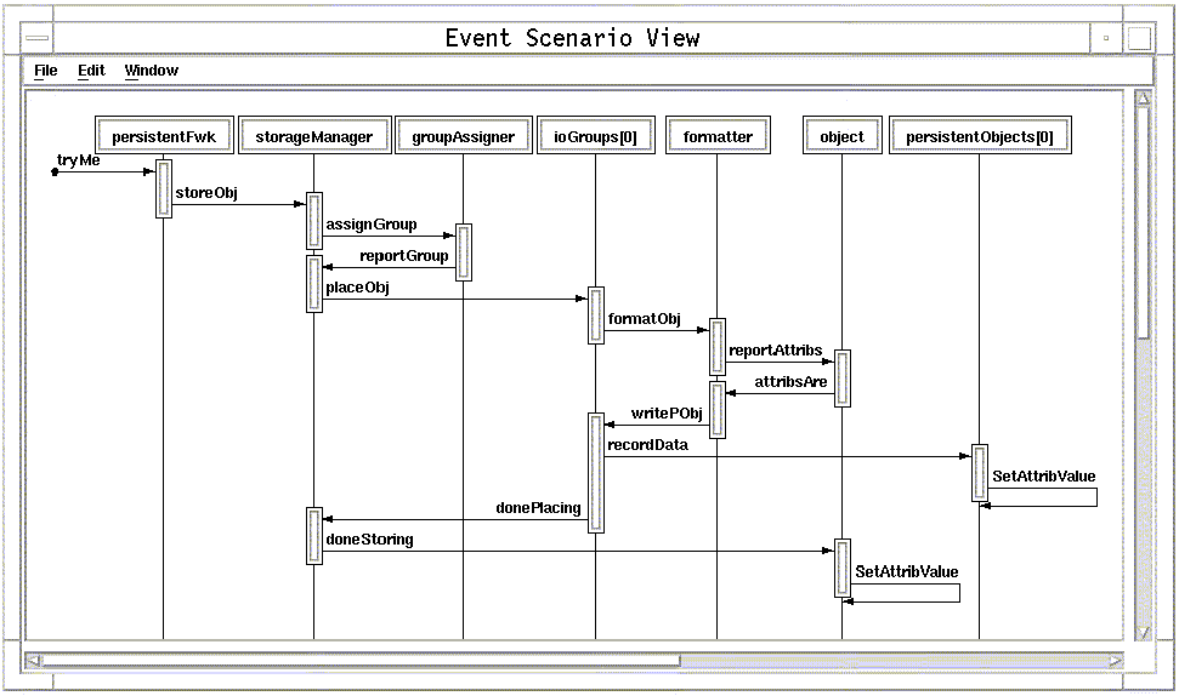


Figura 2.10 – Diagrama de Seqüência Gerado pela Ferramenta *ObjectChart*.

Esta técnica, entretanto, não ataca em nenhum momento a questão da escolha de um framework adequado para um domínio de aplicação, além de não fornecer um guia objetivo para adaptar um framework às necessidades dos seus usuários.

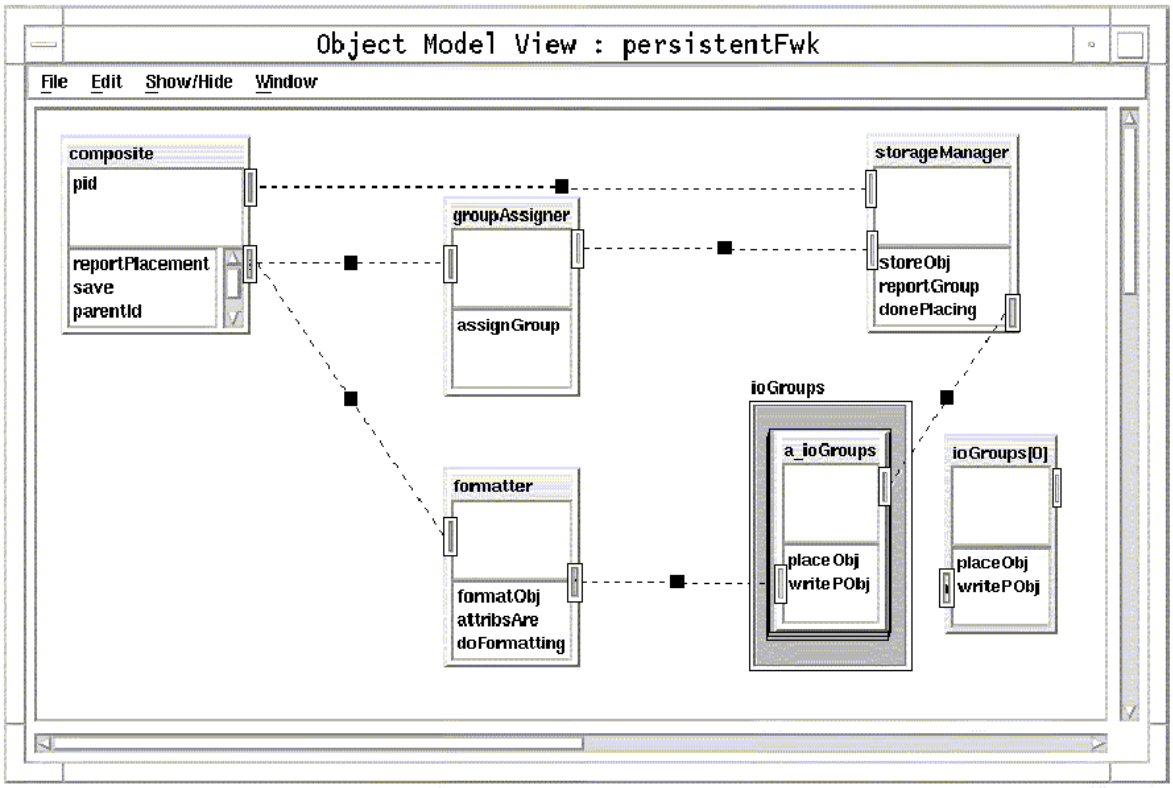


Figura 2.11 – Diagrama de Classes Produzido pela Ferramenta ObjectChart.

2.4 Técnicas de Documentação Baseadas em *Design Patterns*

Os trabalhos sobre *design patterns* [Gamma95] exerceram uma enorme influência na comunidade de orientação a objetos desde as primeiras publicações [Gamma93]. A partir desta data surgiram várias propostas que, de uma maneira ou de outra, procuram associar a instanciação de um framework à adaptação dos seus *hot-spots* através da aplicação de *design patterns*; embora o processo de instanciação vá muito além da simples utilização de alguns *Design Patterns* nos Pontos de Adaptação [Codenie97]. Logo, a documentação de um framework deveria permitir, segundo esta concepção, o estudo detalhado do design e a escolha dos *patterns* que melhor se prestassem à adaptação da arquitetura do framework aos requisitos de uma aplicação específica.

Um exemplo bastante significativo da abordagem supracitada é encontrado em [Lange95]. Neste trabalho é apresentada uma ferramenta que simula a execução de uma aplicação orientada a objetos e permite a visualização dos aspectos dinâmicos e estáticos de um design. A partir daí é possível então perceber a ocorrência de alguns *Design Patterns* e a colaboração existente entre os seus componentes principais. Por exemplo, a Figura 2.12 exibe um diagrama de objetos que mostra a interação entre os componentes de uma aplicação baseada no framework *InterViews* [Calder95]. Neste diagrama, muito semelhante ao diagrama de colaboração da UML, é possível observar a ocorrência do *pattern Observer* [Gamma95].

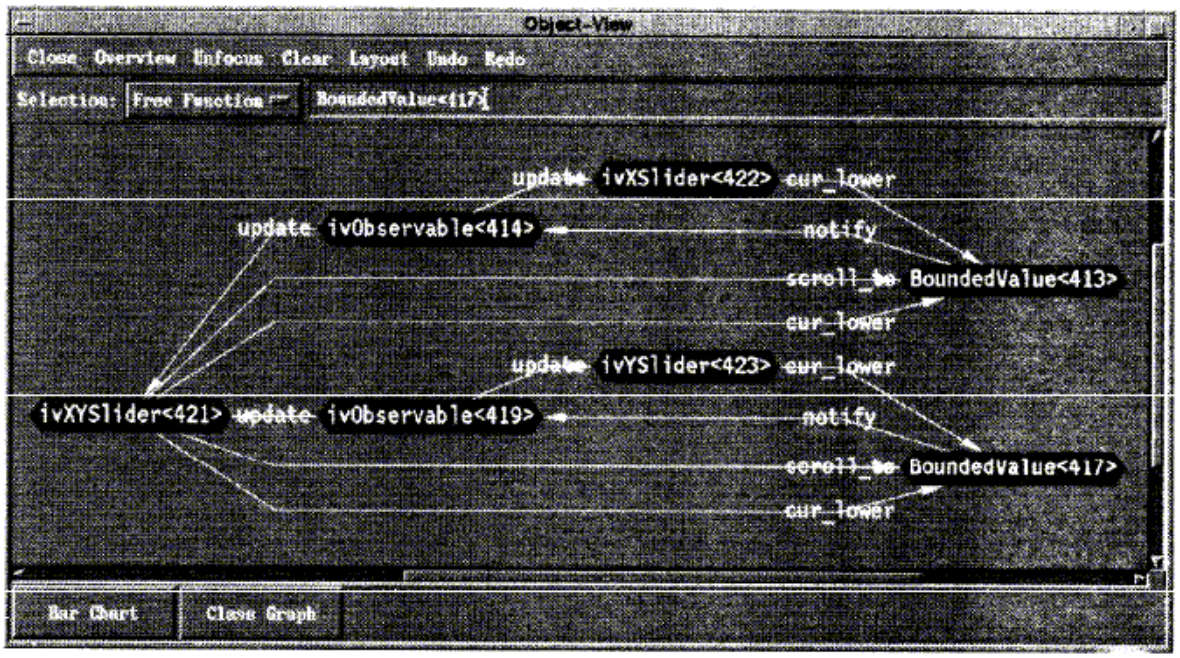


Figura 2.12 – Visualização do *Pattern Observer* Através de um Diagrama de Colaboração.

A Figura 2.13, por sua vez, exibe uma outra visão da colaboração entre os objetos da aplicação em questão. Nesta figura, assim como nos diagramas de seqüência da UML, o destaque é a seqüência da troca de mensagens entre os componentes.

Assim como outras abordagens voltadas exclusivamente para o estudo detalhado do design, a técnica de documentação proposta por este trabalho não torna explícito o

domínio de aplicação do framework, e nem ensina como usá-lo para produzir aplicações específicas.

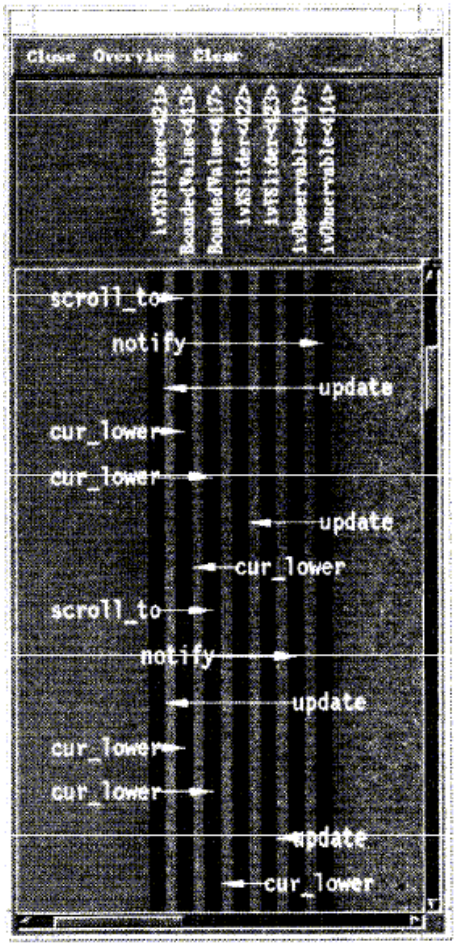


Figura 2.13 - Visualização do *Pattern Observer* Através de um Diagrama de Seqüência.

Uma outra proposta interessante nesta área, apresentada em [Fontoura99] e [Fontoura01], discute a extensão da linguagem UML com o objetivo de capturar alguns aspectos dos *Design Patterns*, que não são representados convenientemente pela versão atual da linguagem UML.

Tomemos o *pattern Strategy* [Gamma95] para exemplificar o uso da abordagem em questão. Este *pattern*, cuja representação através da linguagem UML pode ser vista na Figura 2.14, permite encapsular possíveis variações de uma dada operação (operação *algorithm* da Figura 2.14) através de uma interface estável (classe *Context* da Figura

2.14). A adaptação da operação é realizada instanciando-se uma subclasse da classe *Strategy* e implementando-se o novo algoritmo através da redefinição da operação *algorithm()*, herdada da classe *Strategy*. Entretanto, não é possível capturar todos os aspectos que envolvem o uso deste *pattern*, e com a riqueza de detalhes necessária, utilizando-se somente a linguagem UML-padrão.

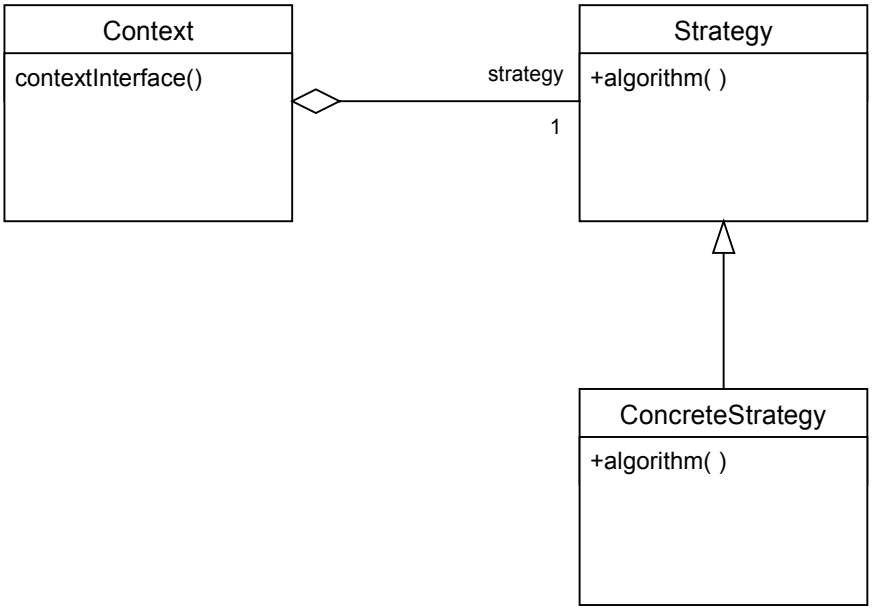


Figura 2.14 - Representação do *Pattern Strategy* Através da Linguagem UML-Padrão.

Para melhorar o poder de expressão da UML em relação aos *Design Patterns* a abordagem em questão propõe inserir algumas anotações nos seus componentes. Tais anotações têm por objetivo tornar claro qual aspecto do design deseja-se flexibilizar, além de indicar a maneira pela qual um dado *design pattern* deve ser instanciado.

Na Figura 2.15 é possível observar a introdução do *tagged value* [UML01] *variation* na representação do *pattern Strategy*. Esta anotação permite identificar a operação *algorithm* como sendo o *hot-spot* deste elemento do design. Um outro *tagged value*, *incomplete*, informa que diferentes algoritmos podem ser empregados através da criação de novas subclasses da classe *Stategy*, e da posterior redefinição da operação *algorithm*. A descrição do processo de instanciação é complementada por um diagrama

(Figura 2.16) que descreve os passos que devem ser executados para instanciar o *hot-spot*. Além disso, o processo de instanciação é auxiliado por uma ferramenta que aplica transformações [Cheatham89] no design original para gerar um novo design. Este novo design incorpora *Design Patterns* nos *hot-spots* do framework de acordo com as regras de instanciação estabelecidas pelas anotações inseridas nos Pontos de Adaptação.

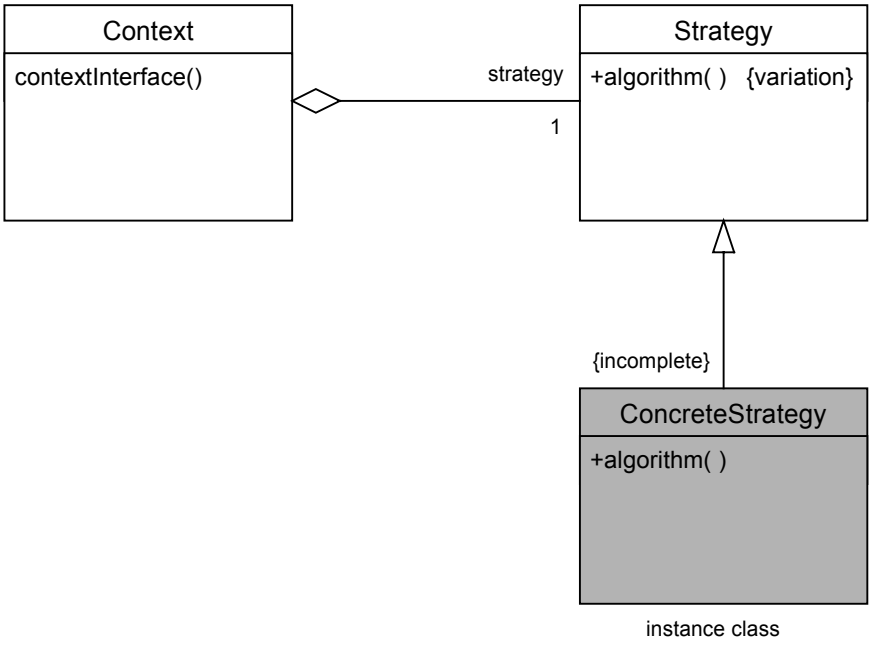


Figura 2.15 - Representação do *Pattern Strategy* Através de Uma Extensão da Linguagem UML .

A principal crítica a este trabalho é a importância demasiada que é dada aos *Design Patterns* no processo de instanciação de um framework. O uso de *patterns* tem mostrado ser uma excelente estratégia para tornar mais flexível um design orientado a objetos. Porém, isto não é suficiente para capturar todos os aspectos envolvidos na adaptação de um framework. Desta maneira, o foco do trabalho fica muito concentrado nos métodos de adaptação, não sendo dada a necessária atenção aos aspectos devem ser adaptados em um framework para atender as necessidades dos seus usuários. Outra importante questão que também não é abordada por este trabalho é existência de interdependências entre os Pontos de Adaptação. Esta é uma

questão que deveria ser bastante explorada dado que, nesta circunstância, a escolha dos *Design Patterns* mais adequados para adaptar um framework passaria a ser função de *tuplas* de *hot-spots* interdependentes, e não de cada *hot-spot* tomado de maneira isolada.

Apesar das críticas acima, é importante destacar que as alterações na UML propostas em [Fontoura99] e [Fontoura01] constituem-se em uma das principais contribuições na área de documentação de frameworks feitas nos últimos anos. Anotações semelhantes a estas serão usadas nesta dissertação para estabelecer uma ponte entre as características funcionais e tecnológicas oferecidas por um framework, e os elementos do design que têm que ser adaptados para que tais características possam ser implementadas.

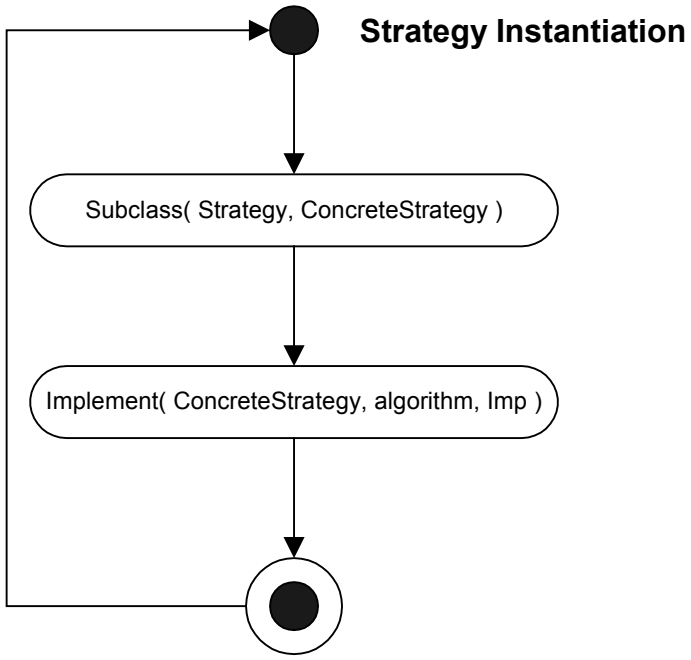


Figura 2.16 - Diagrama de Instanciação do *Pattern Strategy*.

2.5 Técnicas de Documentação Baseadas nas Funcionalidades Oferecidas por um Framework

O trabalho de [Ortigosa00a] apresenta um ambiente, chamado *HiFi* (*Helping in Framework Instantiation*), que tem por objetivo permitir a instanciação de aplicações a partir das funcionalidades disponibilizadas por um framework. Em linhas gerais, o processo de instanciação baseado no ambiente *HiFi* consta das seguintes etapas:

a. As funcionalidades do framework são apresentadas aos usuários, que então escolhem quais delas estarão presentes na aplicação que será produzida.

b. Baseado nas escolhas do item anterior, e nas regras de instanciação fornecidas com o framework, um plano de instanciação é gerado.

c. O plano de instanciação é executado.

A etapa de seleção das funcionalidades está baseada em uma documentação organizada na forma de hipertexto. À medida que os *hiperlinks* vão sendo navegados mais detalhes sobre as funcionalidades disponíveis vão sendo apresentados. Entretanto, a ferramenta não disponibiliza algumas informações importantes para configurar a aplicação que será produzida, tais como as características que são obrigatórias, as que são opcionais e as alternativas existentes para uma dada característica. Outro aspecto não fornecido pela documentação é a eventual existência de dependências entre as características disponibilizadas pelo framework.

As regras de instanciação são representadas através de duas notações distintas. Na primeira delas as regras são definidas através de pares (pré-condição, efeito), como pode ser visto na Figura 2.17.

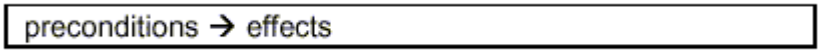


Figura 2.17 – Regra de Instanciação Definida Através de um Par (pré-condição, efeito).

Por exemplo, a regra ilustrada na Figura 2.18 define que é necessário executar duas operações para que seja possível incluir uma ferramenta (*Tool*) em uma aplicação gráfica baseada no framework *HotDraw*. A primeira operação é selecionar a ferramenta desejada, e a segunda é adicionar a ferramenta selecionada à aplicação que será produzida.

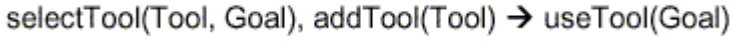


Figura 2.18 – Regra para a Inclusão de uma Ferramenta em uma Aplicação Baseada no Framework *HotDraw*.

A outra maneira de definir uma regra de instanciação é através de uma notação gráfica chamada *TOON* [Ortigosa00b]. Um exemplo do uso desta notação pode ser visto na Figura 2.19. Ela mostra que para inserir a funcionalidade *interactiveAnimation* é necessário antes aplicar a regra *makeAnimatedDrawing* (retângulo superior direito da Figura 2.19) e depois executar as seguintes tarefas: criar uma subclasse da classe *Tool* e implementar os métodos *activate* e *deactivate* nesta subclasse. Além disso, duas restrições são estabelecidas para estes dois métodos: o método *activate* deve invocar o método *startAnimation* e o método *deactivate* deve invocar o método *stopAnimation*. Ambos os métodos (*startAnimation* e *stopAnimation*) devem estar localizados em uma subclasse da classe *Drawing*, que já deve ter sido criada quando da execução desta regra.

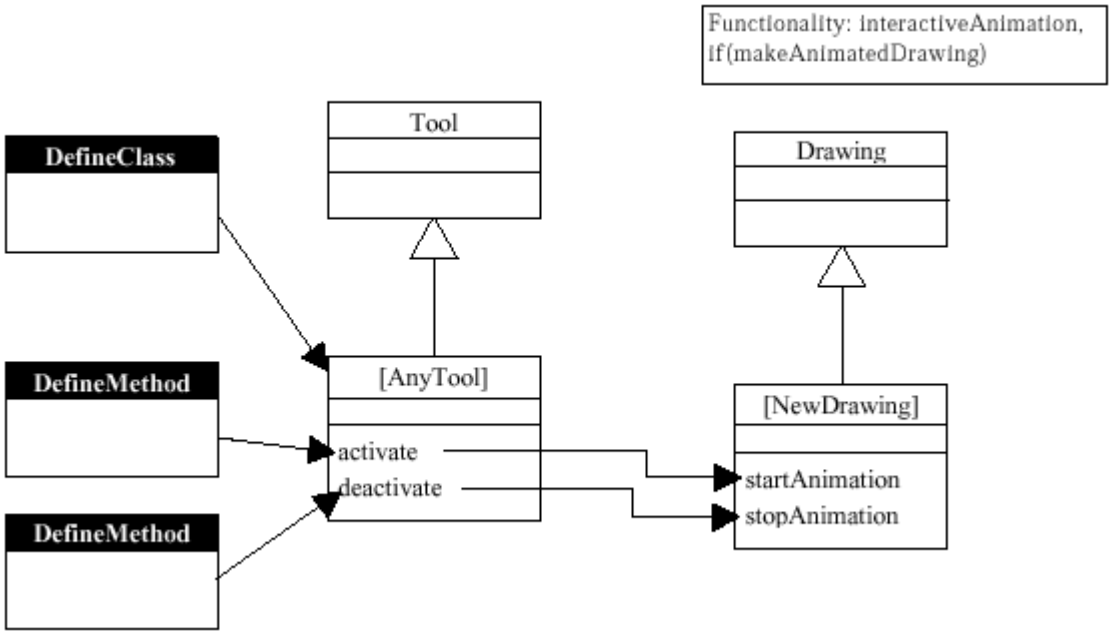


Figura 2.19 – Definição de uma Regra de Instanciação Através da Notação *TOON*.

Duas importantes questões não são devidamente esclarecidas na proposta publicada em [Ortigosa00a]. Primeiro, não é mostrado como as funcionalidades disponibilizadas pelo framework são relacionadas aos elementos do design correspondentes. Segundo, não é apresentado o método pelo qual um agente gera o plano de instanciação a partir das funcionalidades escolhidas; é dito apenas que ele *descobre* o que deve ser feito.

2.6 Considerações Finais

Nas seções anteriores foram analisados vários trabalhos relacionados ao problema da produção de uma documentação adequada à instanciação de aplicações a partir de um framework. O objetivo agora é tecer algumas considerações finais sobre estes trabalhos e apresentar um quadro comparativo (Tabela 2.1) das características principais de cada um deles. É importante relembrar, antes de irmos adiante, que estes trabalhos foram analisados em relação aos requisitos de documentação apresentados em [Johnson92] (ver seção 1.4 desta dissertação). Além disso, serão acrescentados mais dois itens no quadro comparativo: o primeiro trata da existência ou não de ferramentas que permitam automatizar parcialmente o processo de instanciação,

enquanto o segundo trata da existência ou não de mecanismos que suportem um processo de instanciação de alto-nível. Por processo de instanciação de alto-nível deve-se entender que a adaptação de um framework será baseada na seleção das características funcionais e tecnológicas por ele disponibilizadas; não sendo necessário para tal recorrer ao estudo detalhado do design.

Em relação ao primeiro requisito de documentação apresentado por [Johnson92], que trata da descrição do propósito de um framework, podemos afirmar que a maioria dos trabalhos se vale de texto livre para alcançar tal objetivo, embora muitos deles acrescentem ferramentas baseadas em hipertexto para facilitar a busca de informações.

Para descrever como um framework deve ser usado, segundo requisito apresentado em [Johnson92], o panorama não muda muito em relação às técnicas usadas para atender o primeiro requisito. De um modo geral, os métodos baseados em *cookbooks* utilizam texto livre, organizados sob a forma de receitas, para descrever os passos necessários à adaptação de um framework. As duas exceções ficam por conta dos *Active Cookbooks* de [Sommerlad95] e [Pree95], e dos *hooks* de [Froehlich97]. O primeiro deles, *Active Cookbooks*, permite associar as receitas com os elementos correspondentes do design através de *hiperlinks*. Deste modo foi possível construir uma ferramenta para automatizar parte do processo de instanciação. A segunda exceção, o método baseado em *hooks*, propõe uma linguagem mais precisa para descrever o *script* de instanciação, embora não tenha sido reportada a construção de nenhuma ferramenta que possa executar tal *script*.

Ainda em relação ao segundo requisito de documentação, podemos afirmar que praticamente todos os métodos baseados em descrições detalhadas do design não fornecem suporte adequado para descrever a maneira pela qual um framework deve

ser usado. Isto pode ser explicado em parte pelo fato de que tais abordagens partem do princípio de que o estudo minucioso do design é a base para a instanciação de aplicações a partir de um framework. A única exceção fica por conta do trabalho de [Fontoura99], que ataca esta questão através de diagramas de instanciação e anotações feitas nos diagramas de classe. O emprego destes recursos permite que a adaptação de um *hot-spot* possa ser executada, de maneira semi-automática, por uma ferramenta escrita em Prolog [Casanova87] e baseada em transformações [Cheatham89]. Por último, deve ser dito que apenas o ambiente *HiFi* [Ortigosa00a] propõe que a instanciação de aplicações deva ser baseada na descrição das funcionalidades disponibilizadas por um framework, embora este trabalho forneça apenas uma vaga descrição das ferramentas que permitiriam que tal propósito fosse alcançado.

Quanto ao terceiro requisito de documentação apresentado por [Johnson92], podemos dizer que quase todos os trabalhos analisados descrevem, de uma maneira ou de outra, aspectos do design de um framework. As exceções ficam apenas por conta de alguns dos trabalhos baseados em *cookbooks*, que não mencionam explicitamente como alguns dos detalhes de um design são apresentados aos usuários de um framework.

Neste ponto é oportuno fazer algumas observações adicionais para que os dados da Tabela 2.1 possam ser mais bem compreendidos:

a. Os trabalhos baseados na descrição das colaborações entre os objetos (seção 2.2), em múltiplos modelos do design (seção 2.3) e em *design patterns* (seção 2.4) foram reorganizados sob o título "*Técnicas de Documentação Baseadas em Descrições Detalhadas do Design*".

b. Quando um trabalho não mencionar explicitamente como ele trata um determinado requisito de documentação, a coluna correspondente a tal requisito será deixada em branco na linha correspondente ao trabalho em questão.

c. A coluna *Ferramenta p/ Instanciação* só foi preenchida com *Sim* se o trabalho correspondente possuir ferramenta que dê apoio explícito à etapa de adaptação do design do framework. Outros tipos de ferramentas não foram considerados neste quadro. Além disso, só foram levadas em conta as ferramentas que possuem pelo menos uma versão inicial em funcionamento, ou que tenham sido minuciosamente descritas. Esta foi a razão pela qual o trabalho de [Ortigosa00a] foi considerado como não tendo uma ferramenta que dê suporte à etapa de adaptação do design de um framework.

Trabalhos	Requisitos de Documentação de um Framework				
	O Propósito	Como Usar	Descrição do Design	Instanciação de Alto-Nível	Ferramenta p/ Instanciação
Técnicas de Documentação Baseadas em *Cookbooks*					
Cookbooks [Krasner88]	Texto livre	Receitas (texto livre)		Não	Não
Padrões de Instanciação [Johnson92]	Texto livre (estruturado)	Receitas (estruturado)		Não	Não
Active Cookbooks [Sommerlad95, Pree95]	Texto livre (hipertexto)	Receitas (hipertexto)	Notação gráfica	Não	Sim
Hooks [Froehlich97]	Texto livre (estruturado)	Linguagem de *script*		Não	Não
Técnicas de Documentação Baseadas em Descrições Detalhadas do Design					
Contratos [Helm90]			Linguagem formal	Não	Não
Roles [Riehle98]			Notação gráfica	Não	Não
Choices [Campbell92]			Notação gráfica	Não	Não
Motifs [Lajoie95]	Texto livre (estruturado)	Texto livre (estruturado)	Notação gráfica	Não	Não
ObjectChart [Gangopadhyay95]			Ferramenta CASE + Simulação	Não	Sim
Program Explorer [Lange95]			Ferramenta CASE + Simulação	Não	Não
UML-F [Fontoura99, Fontoura01]		Regras de instanciação	Notação gráfica	Não	Sim
Técnicas de Documentação Baseadas nas Funcionalidades Oferecidas por um Framework					
HiFi [Ortigosa00a]	Texto livre (hipertexto)	Regras de instanciação	Notação gráfica	Sim	Não

Tabela 2.1 – Tabela Comparativa das Técnicas de Documentação de Frameworks.

Antes de passarmos à análise dos dados contidos na tabela acima é importante reafirmar que a tese aqui defendida, e corroborada pelo trabalho de [Ortigosa00a], é que o foco do processo de instanciação deve estar voltado para a representação, e posterior seleção, das características de um framework que devem ser adaptadas para atender aos requisitos de uma aplicação específica. Neste sentido não há maiores

divergências entre a abordagem proposta nesta dissertação e os trabalhos baseados em *cookbooks*, dado que estes utilizam grupos de receitas relacionadas para descreverem as maneiras pelas quais um framework dever ser adaptado. Entretanto, como pode ser visto na Tabela 2.1, o caráter predominantemente textual da abordagem, somado à ausência de ferramentas que dêem suporte ao processo de instanciação, tornam os métodos baseados em *cookbooks* demasiadamente informais. Conseqüentemente, será inevitável que os usuários de um framework tenham que recorrer ao estudo cuidadoso do seu design para que possam traduzir as instruções contidas nas receitas em código que implemente as funções requeridas.

As abordagens baseadas em descrições detalhadas do design de um framework defendem, por sua vez, que o processo de instanciação deve estar focado nos métodos que devem ser utilizados para adaptar o design de um framework às necessidades dos seus usuários. Isto se reflete na ausência quase que total de documentos que atendam aos dois primeiros requisitos de documentação propostos em [Johnson92], como pode ser facilmente observado na Tabela 2.1. Na seção 1.3 desta dissertação foram apresentadas as razões pelas quais esta tese discorda de tais abordagens.

Pelos dados comparativos da Tabela 2.1, e pelo que foi descrito na seção 2.5 desta dissertação, o trabalho mais próximo da tese aqui defendida é o método *HiFi* [Ortigosa00a]. Entretanto, esta abordagem falha ao não fornecer uma maneira adequada de descrever o propósito de um framework e o seu domínio de aplicação. Isto compromete o processo de instanciação baseado nas características disponibilizadas por um framework na medida em que não torna explícito aos *desenvolvedores* de aplicação quais são as opções disponíveis para que as suas necessidades sejam atendidas. Outro aspecto negativo do método *HiFi* é que não são fornecidos maiores detalhes de como as características funcionais e tecnológicas de um

framework são refletidas no seu design. Isto explica em parte a razão pela qual a ferramenta de apoio ao processo de instanciação é vagamente descrita em [Ortigosa00].

Os próximos três capítulos desta dissertação irão mostrar como as duas falhas encontradas no método *HiFi* serão corrigidas dentro do contexto do método de instanciação de alto-nível proposto por esta tese.

Capítulo 3

O Modelo de *Features*

A partir deste capítulo começaremos a discutir a proposta contida nesta dissertação para atacar o problema da instanciação de aplicações a partir de um framework orientado a objetos. O primeiro passo em direção a este objetivo é fornecer aos *desenvolvedores* de aplicações um mapa das possibilidades disponibilizadas por um framework em termos da capacidade deste de produzir membros específicos de uma família de aplicações [Parnas76]. Para tal é necessário que a ferramenta a ser utilizada seja capaz de representar adequadamente os aspectos obrigatórios e opcionais de um dado domínio de aplicação. Como veremos a seguir, o Modelo de *Features* é uma escolha bastante adequada para cumprir esta tarefa de maneira precisa e objetiva.

3.1 O Modelo de *Features* na Análise de Domínios

Uma das técnicas mais usadas para estudar as variações existentes em uma família de aplicações e organizá-las de maneira que sirvam de base para a construção de artefatos de software genéricos para um dado domínio de aplicação, chama-se Análise de Domínios. 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 entre eles podemos dizer, de maneira genérica, que tais métodos são funcionalmente equivalentes,

invariavelmente apresentando operações de agregação, classificação, generalização e parametrização [Arango93].

Apesar das similaridades existentes, o método *FODA* (*Feature Oriented Domain Analysis*) [Kang93], 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 estudos de casos apresentados por inúmeras publicações [Cohen92, Griss98, Vici98 e Kwanwoo00]. 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* [Kang98].

Dentre as várias contribuições feitas pelos dois métodos supracitados, 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"* [Kang98]. 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 [Bosch e Hudak96] usada para facilitar a comunicação entre os especialistas em software e os usuários finais.

Nas seções seguintes serão apresentados os métodos utilizados para compatibilizar o Modelo de *Features* com a linguagem UML. Esta estratégia visa possibilitar a integração do Modelo de *Features* com os outros modelos usados para descrever a arquitetura de um framework. Desta forma será possível incorporar a abordagem proposta nesta dissertação aos ambientes CASE mais usados no desenvolvimento de

aplicações orientadas a objetos. As razões para tal integração foram apresentadas na seção 1.5 desta dissertação.

3.2 A Definição do 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 [Kang98] 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.

O presente trabalho, diferentemente de [Kang98], 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

proposto pelos autores do *FORM*, já fornece tal informação. Para demonstrar isto vamos recorrer a alguns exemplos extraídos de [Kang98].

A Figura 3.1 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*.

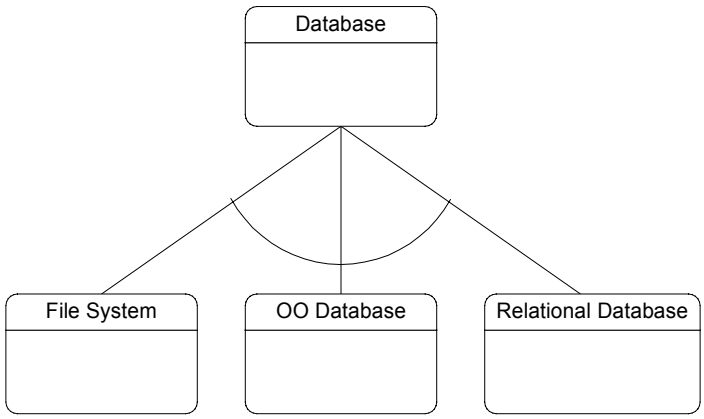


Figura 3.1 – Representação de uma Composição Alternativa na Notação Proposta em [Kang98].

Em um outro exemplo extraído do mesmo domínio de EBBS, a Figura 3.2 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) para designar que ambas as *Features* são especializações da *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.

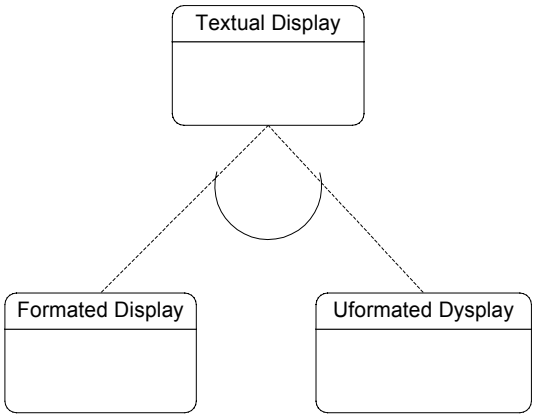


Figura 3.2 – Representação de uma Generalização na Notação Proposta em [Kang98].

O relacionamento *implementado-por* não será considerado como algo particular ao Modelo de *Features* dado que, como será visto na seção 3.3, o relacionamento de dependência presente na UML, acrescido do estereótipo <<realize>, já cumpre esta função.

Além dos relacionamentos estruturais citados anteriormente, são definidas em [Kang98] 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:

a. Capability Layer

b. Operating Environment Layer

c. Domain Technology Layer

d. 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 [Kang98], 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, serem utilizadas em outros domínios.

A distribuição das *Features* em camadas com diferentes níveis de abstração não traz nenhuma dificuldade adicional 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 3.3). Desta forma, o relacionamento entre *Features* de diferentes camadas deverá seguir as regras de visibilidade e caminhos (*paths*) determinados pelo meta-modelo da UML.

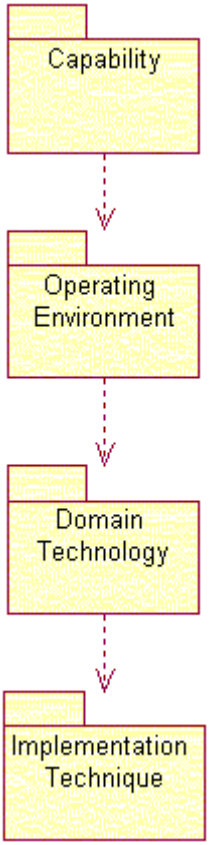


Figura 3.3 – Representação das Diversas Camadas de um Modelo de *Features*.

3.3 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 [UML01].

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 [Jacobson99].

3.3.1 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 [UML01]. O objetivo principal da abordagem adotada é descrever a UML de maneira precisa sem, entretanto, ter que utilizar linguagens formais tais como Object-Z [Duke94] ou VDM++ [Lano94]; linguagens estas que foram projetadas para serem usadas por pessoas com um sólido *background* matemático, inacessível à maioria dos *desenvolvedores* aplicações encontrados nos dias atuais.

A especificação da linguagem UML é dividida em três partes:

a. *Sintaxe Abstrata*

b. *Regras de Boa Formação*

c. *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*) [Warmer98], 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.

3.3.2 O Meta-modelo da UML

O meta-modelo da UML é definido como sendo uma das camadas de uma arquitetura de quatro camadas:

a. *Meta-metamodelo*

b. *Meta-modelo*

c. *Modelo*

d. *Objetos*

Nesta arquitetura o meta-metamodelo descreve a infraestrutura necessária para a definição dos meta-modelos. No caso da UML, o papel de meta-metamodelo é exercido pelo *OMG Meta-Object Facility (MoF)* [OMG00]. Logo, a UML pode ser vista como uma instância do *MoF*.

A função principal da camada de meta-modelo 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 meta-modelo da UML descreve a estrutura da própria linguagem UML.

Um modelo é uma instância de um meta-modelo, 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 meta-modelo da UML foi organizado em pacotes (elemento definido pela própria UML). No nível mais alto da arquitetura existem três pacotes (Figura 3.4). 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.

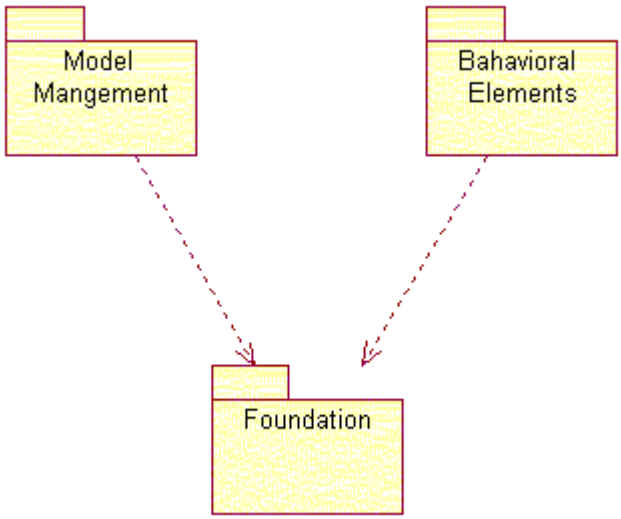


Figura 3.4 – Os Pacotes da Camada de Mais Alto Nível.

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 3.5).

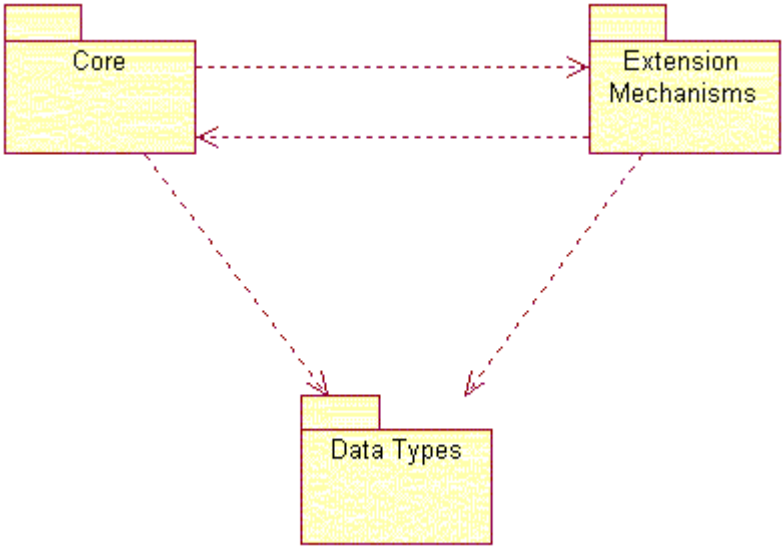


Figura 3.5 – A Subdivisão do Pacote *Foundation*.

O pacote *Core* contém os conceitos básicos para a construção de um meta-modelo elementar, além de fornecer a infraestrutura necessária para a adição de elementos tais como meta-classes, 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 (*lightweight extensibility*). 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 sub-pacotes do pacote *Foundation*.

3.3.3 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 (*lightweight extensibility*). 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 [UML01] 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, serão estabelecidas regras de instanciação de um Modelo de *Features*. Entretanto, para facilitar a leitura do texto com um todo, estes dois últimos itens foram postos no Apêndice A.

3.3.3.1 A Sintaxe Abstrata

A Figura 3.6, extraída de [UML01], 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.

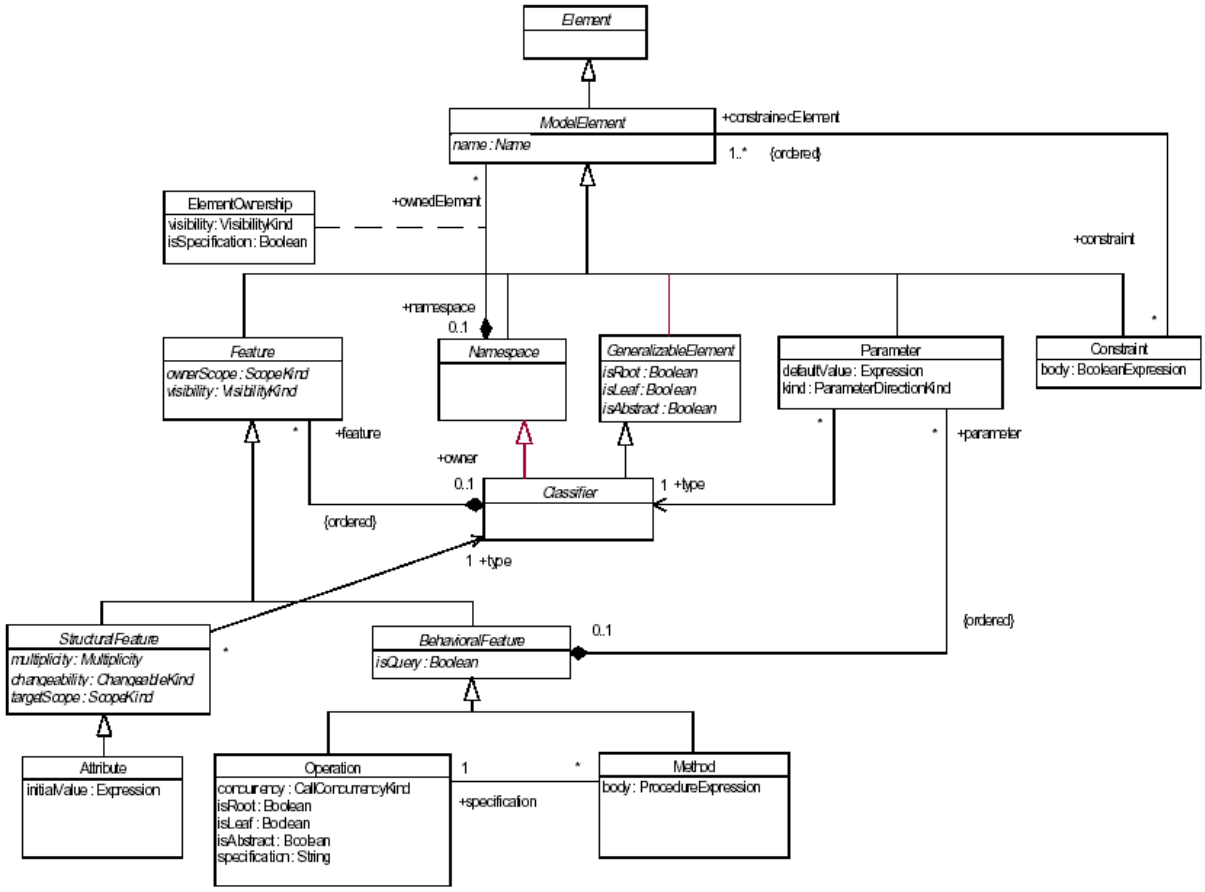


Figura 3.6 – As Meta-Classes e os Meta-relacionamentos do Pacote *Core*.

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 neste capítulo 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 3.6).

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 3.2) 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*.

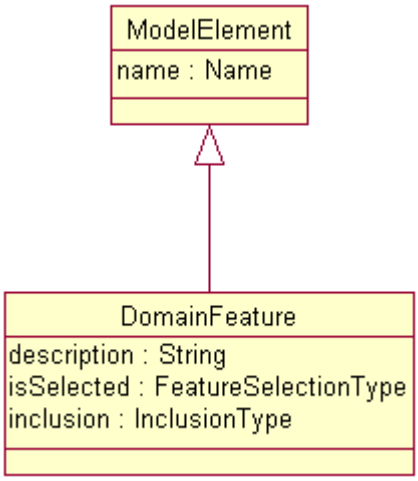


Figura 3.7 – A Definição da Classe *DomainFeature*.

A Figura 3.7 mostra a classe *DomainFeature* definida como uma subclasse direta de *ModelElement*. O atributo *description* permite acrescentar informações textuais adicionais para um melhor entendimento do papel de uma *Domain Feature* dentro de um domínio de aplicação. O atributo *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 3.8).

O último atributo da classe *DomainFeature*, *inclusion*, determina se uma dada característica (*Domain Feature*) é obrigatória ou não. Uma *Domain Feature* obrigatória deverá estar presente em todos os membros de uma família de aplicações, enquanto que uma *Domain Feature* opcional poderá estar presente apenas em alguns membros da família.

O tipo enumerado *InclusionType* foi introduzido no pacote *DataType* com o objetivo de definir os valores válidos para o atributo *inclusion* (Figura 3.8).

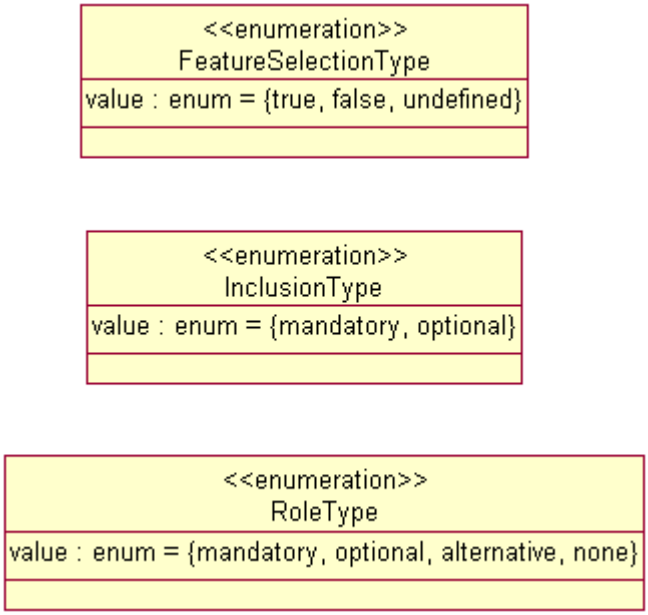


Figura 3.8 – A Definição dos Tipos Enumerados.

Uma vez definida a classe *DomainFeature*, é hora de definir o relacionamento de composição entre as *Domain Features*. A Figura 3.9 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 3.9 é a subclasse de *Relationship* que define o relacionamento de dependência.

No caso da composição entre *Domain Features* tomamos a decisão de criar uma nova classe para modelar este relacionamento (Figura 3.11). 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.

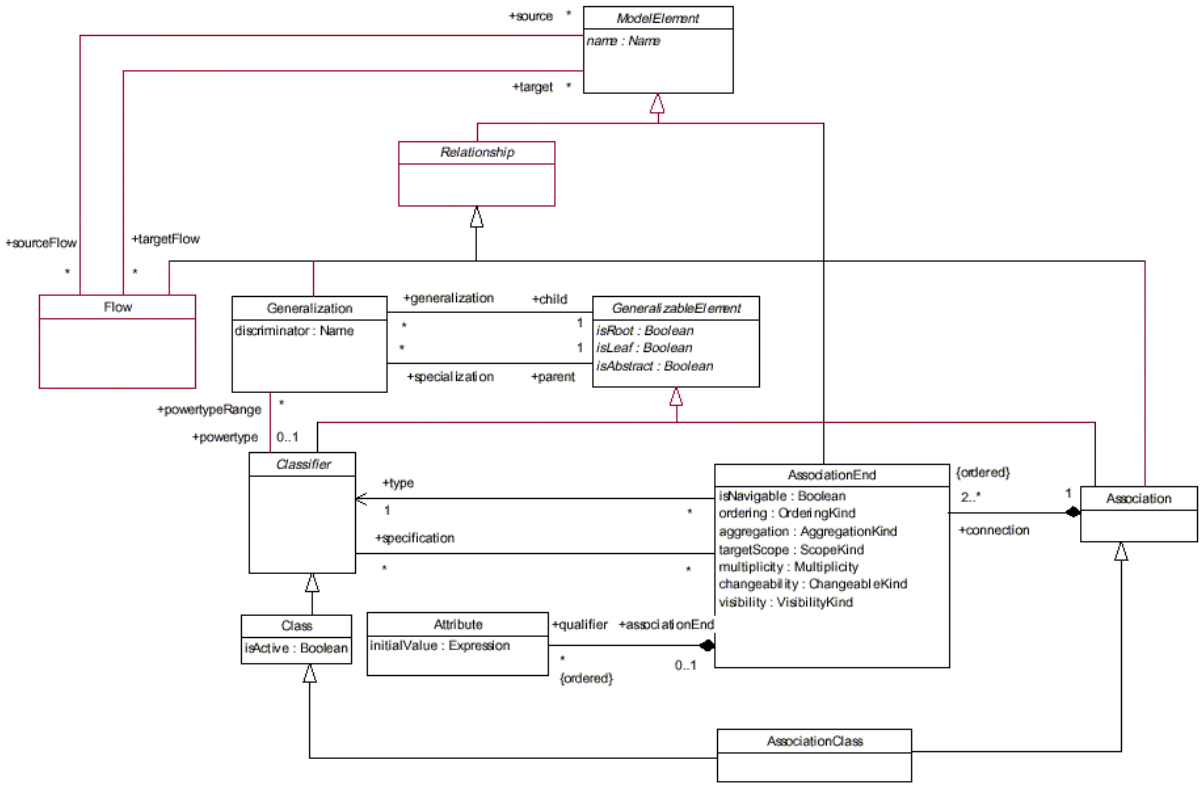


Figura 3.9 – Os Tipos de Relacionamentos da UML.

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*. Esta solução permite separar a questão da presença ou não de uma determinada *Domain Feature* em um membro de uma família de aplicações, do mecanismo de seleção de algumas características em uma composição de *Domain Features*. O exemplo a seguir irá justificar esta decisão de design.

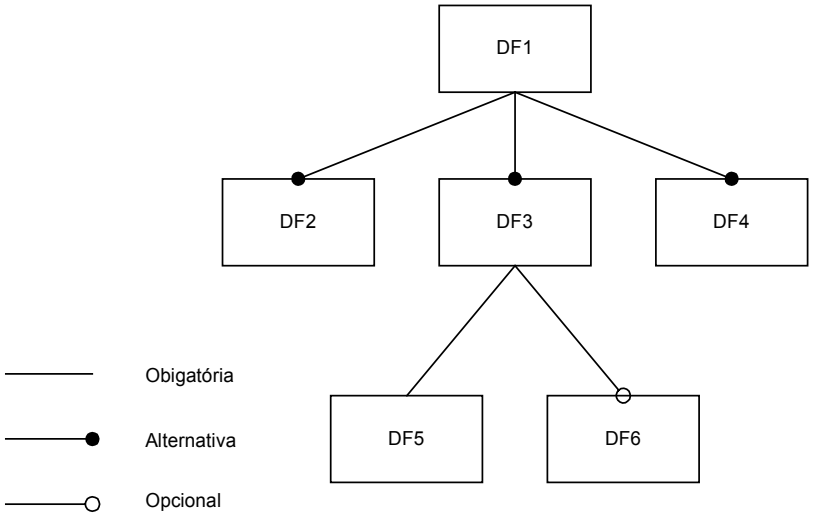


Figura 3.10 – Uma Composição Alternativa de *Domain Features*.

A Figura 3.10 acima mostra uma composição alternativa na qual a *Domain Feature* DF1 é composta pelas alternativas DF2, DF3 e DF4. Além disso, podemos observar que a *Domain Feature* DF3 é composta por DF5 e DF6. Como DF2, DF3 e DF4 são alternativas de DF1 podemos concluir que elas são opcionais, já que em uma composição alternativa no máximo uma característica pode ser escolhida (vide seção 3.2). O mesmo pode ser dito de DF5 e DF6, já que elas são componentes de uma característica opcional. Entretanto, o diagrama da Figura 3.10 estabelece que uma vez

selecionada a característica DF3 a *Domain Feature* DF5 deverá ser obrigatoriamente escolhida. Além disso, a característica DF6 torna-se também passível de escolha.

Para que a questão acima fosse resolvida de maneira satisfatória introduzimos o atributo *role* na classe *FeatureCompositionEnd*. Deste modo iremos permitir que uma *Domain Feature* opcional assuma o papel de uma característica obrigatória, opcional ou alternativa apenas no contexto de uma composição de *Domain Features*.

Para definir o conjunto de valores válidos para o atributo *role* introduzimos o tipo enumerado *RoleType* no pacote *DataType* (Figura 3.8).

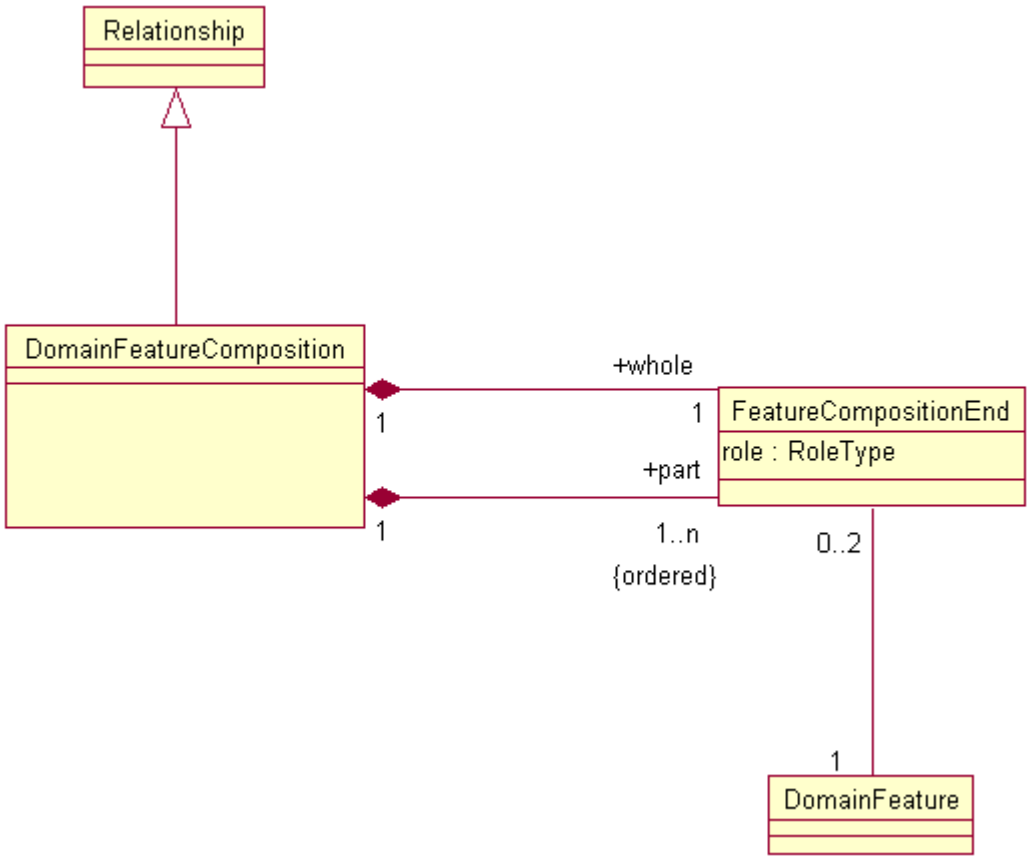


Figura 3.11 – A Composição de *Domain Features*.

Uma outra função que será exercida pelo atributo *role* da classe *FeatureCompositionEnd* (Figura 3.11) é indicar se uma *Domain Feature* faz o papel de todo (*whole*) ou de uma parte (*part*) em uma composição. Se o valor do atributo *role*

for igual à constante *none* a *Domain Feature* fará o papel de *whole*, caso contrário ela será uma parte em uma composição de *Domain Features*.

A introdução do atributo *role* irá facilitar bastante a construção de algumas regras de boa formação, como poderá ser visto no Apêndice A.

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 3.12). 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 (*source* e *target*) é devida à construção de regras de boa formação para as instâncias dos modelos.

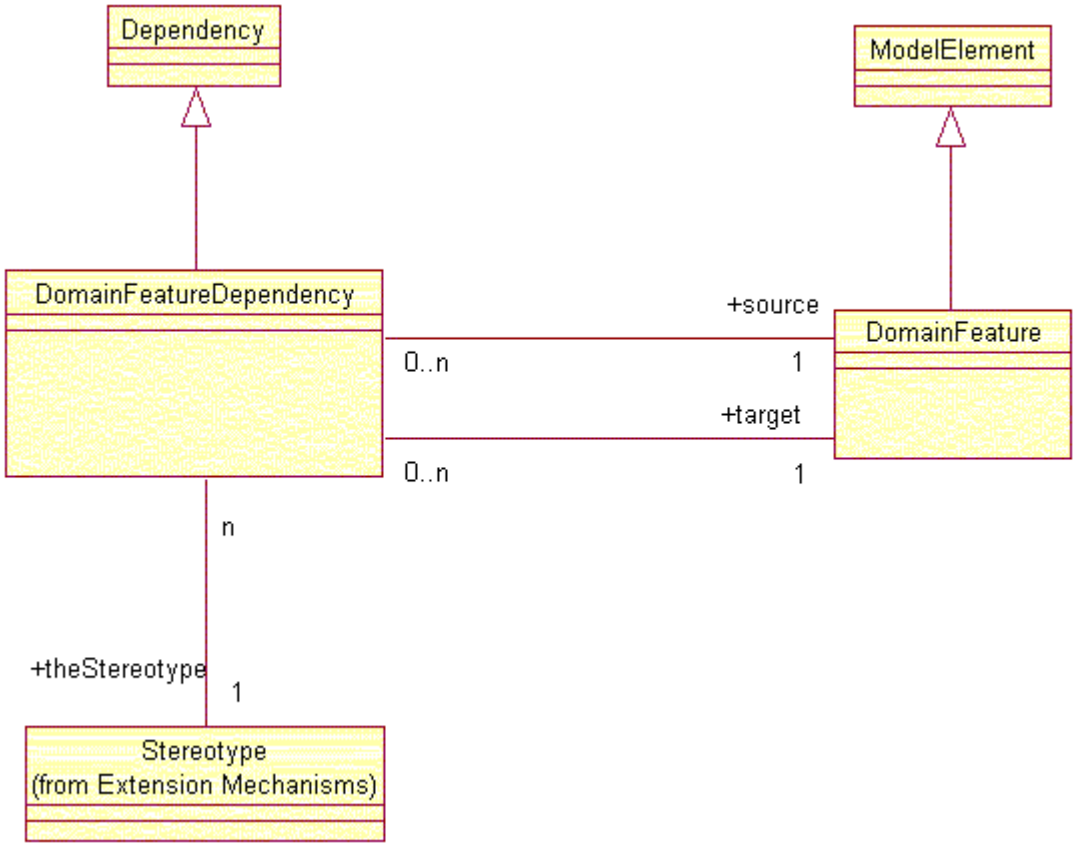


Figura 3.12 – Especificação das Regras *Requires* e *MutexWith*.

3.3.3.2 As Regras de Boa Formação

Os elementos de modelagem introduzidos na sintaxe abstrata da UML não são suficientes para entender de maneira detalhada a semântica das *Domain Features*. É necessário complementar os diagramas de classes vistos nas seções anteriores com algumas regras de boa formação. Entretanto, para não dificultar a leitura do texto com a introdução de expressões escritas em OCL, a definição das regras de boa formação do Modelo de *Features* será apresentada no Apêndice A desta dissertação.

3.4 Considerações Finais

Um Modelo de *Features* é um modelo semântico [Date99] capaz de representar de maneira estruturada as características obrigatórias e opcionais das aplicações de um dado domínio. Por este motivo ele se apresenta como uma boa opção para servir de

base para um processo de instanciação de frameworks que permita aos *desenvolvedores* produzir aplicações sem que seja preciso conhecer detalhadamente o design um framework.

Para tal é necessário compatibilizar o Modelo de *Features* com o meta-modelo que descreve a linguagem UML. Isto irá permitir a descrição precisa dos mecanismos que irão associar as características de um domínio de aplicação com as classes, os métodos e os atributos que compõem o design de um framework orientado a objetos. Em última instância a escolha de uma determinada *Feature* irá desencadear uma série de ações de reutilização [Oliveira01a] que serão aplicadas sobre estes elementos de design.

Capítulo 4

A Integração do Modelo de *Features* com o Design de um Framework

Neste capítulo iremos apresentar os mecanismos que serão utilizados para integrar um Modelo de *Features* que descreva as características opcionais e obrigatórias de um domínio de aplicações com o design de um framework orientado a objetos, a partir do qual serão instanciadas aplicações para este domínio. Tal integração permitirá adaptar o design do framework a partir da seleção de algumas das características apresentadas pelo Modelo de *Features*.

Para que tal objetivo possa ser alcançado será necessário inicialmente definir precisamente a maneira pela qual os Pontos de Adaptação existentes em um framework serão representados nos modelos de design do mesmo, já que a linguagem UML-padrão não dispõe de tal mecanismo. Posteriormente serão descritas as regras que irão nortear o processo de integração dos dois modelos. Por último, serão apresentados os procedimentos que serão realizados para alterar o design original do framework e inserir os novos componentes que farão parte da aplicação instanciada a partir das escolhas feitas no Modelo de *Features*.

Os exemplos que irão ilustrar os procedimentos apresentados neste capítulo foram baseados no design do framework DTFrame, apresentado em [Oliveira01a]. O framework DTFrame (*Drawing Tool Framework*) é um framework *white-box* que foi

desenvolvido para fornecer capacidade de desenho à ferramenta CASE 2GOOD [Carvalho98], desenvolvida no Laboratório de Métodos Formais (LMF) do Departamento de Informática da PUC-Rio.

4.1 A Representação dos Pontos de Adaptação

Uma questão primordial para a integração do Modelo de *Features* com o modelo de design de um framework é fazer a associação dos aspectos obrigatórios e opcionais de um domínio de aplicação, representados no Modelo de *Features*, com os elementos de design de um framework que implementam tais aspectos. Para tal é necessário responder a duas questões básicas:

a. Quais elementos de modelagem poderão representar os Pontos de Adaptação existentes em um framework?

b. Como diferenciar, em um design orientado a objetos, os aspectos variáveis dos aspectos invariáveis?

Apesar da grande variedade de elementos de modelagem existentes na UML os mecanismos clássicos de extensão de uma aplicação orientada a objetos [Lewis95, Budd91 e Meyer97] são as subclasses, a herança, o polimorfismo e a redefinição de métodos (*overriding*). Logo, o processo de adaptação de um framework predominantemente *white-box* às necessidades de uma aplicação específica envolve basicamente as seguintes operações:

I. Criação de novas subclasses a partir das classes existentes.

II. Inserção de novos atributos e métodos nas subclasses criadas.

III. Redefinição dos métodos das classes existentes nas subclasses criadas.

É importante mencionar que mesmo que levássemos em consideração elementos de design de maior granulação, como os *Design Patterns* [Gamma95], poderíamos decompor o processo de adaptação de tal elemento em sucessivas aplicações das três operações acima, respeitando-se uma determinada seqüência. Logo, segundo a abordagem proposta por esta dissertação, a adaptação de um framework será realizada, de modo semelhante à proposta contida em [Fayad99], executando-se um *script* de instanciação composto por operações derivadas das três operações básicas anteriores. Desta maneira, podemos afirmar que os Pontos de Adaptação de um framework estarão localizados nas classes, nos métodos e nos atributos que compõem o seu design.

Soluções para a segunda questão, a identificação dos Pontos de Adaptação nos modelos de design de um framework, podem ser encontradas nos trabalhos de [Fontoura01], [Fontoura99] e [Oliveira01a]. Estes trabalhos adotaram basicamente a mesma estratégia; ou seja, estender a linguagem UML através de estereótipos e *tagged values*, que uma vez associados aos elementos de modelagem apropriados (classes, atributos e métodos) permitem identificar claramente a existência de um Ponto de Adaptação. Mais ainda, os diferentes estereótipos e *tagged values* permitem identificar antecipadamente quais operações devem ser aplicadas para que a adaptação possa ser realizada.

A opção adotada no presente trabalho será adaptar a proposta de [Oliveira01a] às necessidades geradas pela introdução do Modelo de *Features* no processo de documentação e instanciação de frameworks. Ao invés de usarmos os mecanismos clássicos de extensão da UML (*lightweight extensibility*) iremos aplicar a mesma abordagem utilizada na incorporação do Modelo de *Features* à linguagem UML (Capítulo 3). Isto é, os elementos de modelagem que permitirão a identificação dos

Pontos de Adaptação de um framework serão introduzidos na UML através da alteração do meta-modelo da mesma (*heavyweight extensibility*). Esta opção é justificada não só pela necessidade já existente de alteração do meta-modelo da UML, em razão da incorporação do Modelo de *Features*, como também pela maior flexibilidade proporcionada por uma alteração mais profunda no meta-modelo.

A Figura 4.1 mostra a introdução de quatro novas classes no meta-modelo da UML. Elas permitem adicionar algumas propriedades aos elementos de modelagem vistos anteriormente (classes, métodos e atributos) para que estes possam representar adequadamente os Pontos de Adaptação de um framework.

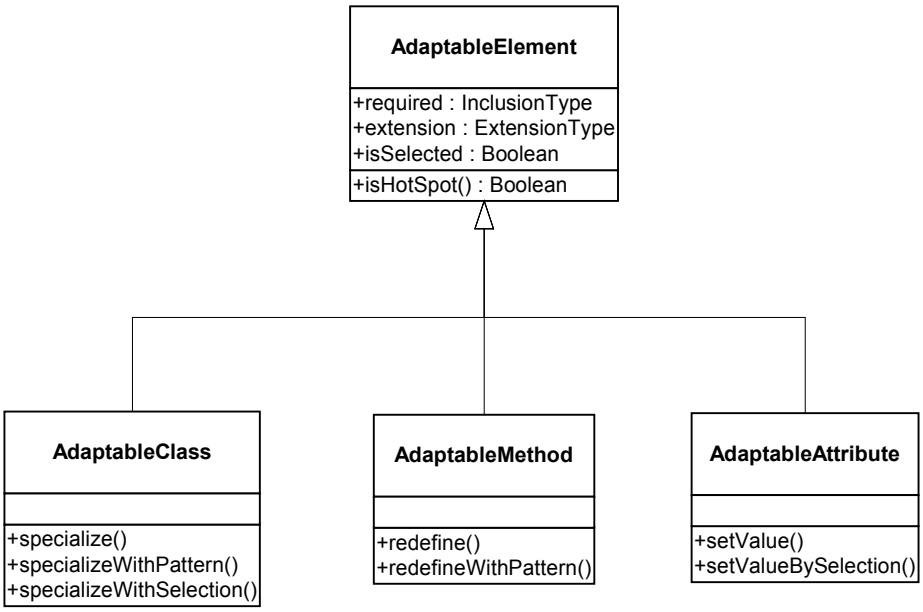


Figura 4.1 – Os Novos Elementos de Modelagem Introduzidos no Meta-modelo da UML.

A integração destas novas classes com as meta-classes que compõem a sintaxe abstrata da UML é feita através do mecanismo de herança múltipla (Figura 4.2). Desta forma é possível agregar novas funcionalidades às meta-classes já existentes sem, contudo, alterar a semântica original das mesmas [Oliveira01a].

Os novos elementos de modelagem, juntamente com os seus atributos, serão descritos a seguir. A descrição segue o mesmo modelo existente em [UML01].

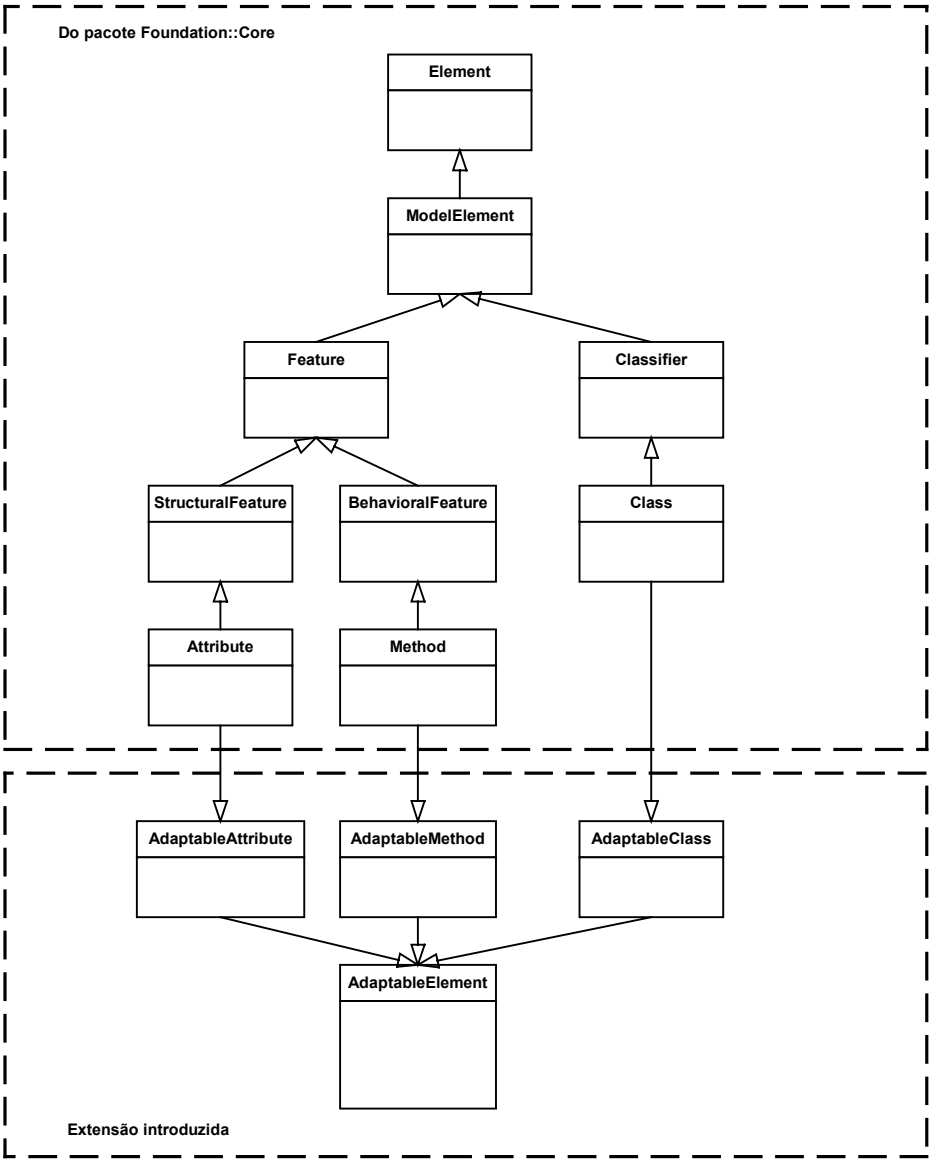


Figura 4.2 – Incorporação dos Novos Elementos de Modelagem à Sintaxe Abstrata da UML.

As Definições das Novas Classes Adicionadas à UML.

AdaptableElement

Raiz da hierarquia utilizada para representar um elemento adaptável.

Atributos

required Indica se a presença do elemento de modelagem é obrigatória ou não no design final. Os valores possíveis para este atributo (**mandatory** e **optional**) estão definidos no tipo de dado **InclusionType** (Figura 3.8). O valor **mandatory** indica que o elemento irá constar obrigatoriamente no design da aplicação que será instanciada. A semântica original do elemento de modelagem é preservada. O valor **optional** indica que a presença do elemento no design final não é obrigatória, caracterizando, assim, a existência de um Ponto de Adaptação.

extension os valores possíveis deste atributo (**ExtensionType**) indicam os tipos de Pontos de Extensão que os elementos de modelagem representam.

isSelected valor booleano que indica se um elemento de modelagem foi ou não selecionado para compor o design final.

AdaptableClass

Identifica que o elemento adaptável é uma classe.

AdaptableMethod

Identifica que o elemento adaptável é um método.

AdaptableAttribute

Identifica que o elemento adaptável é um atributo.

Os Elementos Adaptáveis, introduzidos no meta-modelo da UML, nos permitem representar dois mecanismos de flexibilização distintos. O primeiro deles, determinado pelo valor do atributo **required**, indica se um dado componente tem presença obrigatória ou não no design das aplicações instanciadas a partir de um framework. Embora este mecanismo de flexibilização não seja comumente abordado na literatura sobre frameworks, a sua existência está em perfeita sintonia com tipos de *Features* existentes (obrigatórias e opcionais). Além disso, a existência de componentes opcionais estende o campo de aplicação de um framework, pois determinadas características indesejáveis podem ser simplesmente descartadas [Oliveira01a].

O segundo mecanismo de flexibilização é determinado pelo valor do atributo **extension**. O domínio dos valores válidos para este atributo é definido pelo tipo enumerado **ExtensionType**. Este atributo indica a natureza da extensão que deve aplicada a um elemento adaptável. Os tipos de extensões são baseados nos mecanismos clássicos de extensão de uma aplicação orientada a objetos, vistos anteriormente.

O tipo enumerado **ExtensionType** deve ser incluído no pacote *Data Types*, onde estão localizados os tipos de dados utilizados pelos diversos pacotes que compõem o meta-modelo da UML (vide seção 3.3.3.1).

É a seguinte a definição do tipo enumerado **ExtensionType**:

ExtensionType

Tipo enumerado que determina o tipo de extensão existente em um elemento adaptável.

Valores

classExtension	valor válido quando o elemento de modelagem for uma classe adaptável. Define um ponto de extensão que deve ser adaptado através da criação de uma subclasse da classe em questão.
selectClassExtension	valor válido quando o elemento de modelagem for uma classe adaptável. Define um ponto de extensão que deve ser adaptado através da seleção de uma subclasse pré-existente da classe em questão.
methodExtension	valor válido quando o elemento de modelagem for um método adaptável. Define um ponto de extensão que deve ser adaptado através da redefinição do método em questão.
valueAssignment	valor válido quando o elemento de modelagem for um atributo adaptável. Define um ponto de extensão que deve ser adaptado através da atribuição de um valor inicial ao atributo em questão.
valueSelection	valor válido quando o elemento de modelagem for um atributo adaptável. Define um ponto de extensão que deve ser adaptado através da atribuição de um valor inicial ao atributo em questão. O valor deve ser selecionado a partir de uma lista pré-existente de valores válidos.
none	o elemento adaptável não possui ponto de extensão.

A caracterização dos dois mecanismos de flexibilização que podem ser aplicados a um componente adaptável de um framework nos permite definir um Ponto de Adaptação em função das propriedades de um Elemento Adaptável.

Definição: Um Ponto de Adaptação (*hot-spot*) é uma instância de uma das subclasses (**AdaptableClass**, **AdaptableMethod** e **AdaptableAttribute**) da meta-classe **AdaptableElement**, cujo valor do atributo **required** seja igual a constante **optional** ou cujo valor do atributo **extension** seja diferente da constante **none**.

A definição acima pode ser também expressa pela seguinte *invariante* escrita em OCL:

AdaptableElement

```
((isHotSpot()=true) implies (required=#optional or
extension<>#none)) and
((required=#optional or extension<>#none) implies
(isHotSpot()=true))
```

Em resumo, as modificações introduzidas no meta-modelo da UML nos permitem localizar precisamente os Pontos de Adaptação (*hot-spots*) existentes em um framework. Além disso, elas definem claramente os elementos de design que podem ser adaptados para produzir uma instância de um framework.

Uma última observação deve ser feita em relação à notação gráfica que será usada para representar os Pontos de Adaptação de um framework. Como o meta-modelo da UML não define nenhuma notação gráfica padrão para a exibição dos elementos de modelagem, apresentando apenas uma *stub* (**PresentationElement**) para tal (Figura 4.3), cada ferramenta *CASE* existente no mercado pode adotar a notação que melhor lhe convier. Logo, como nenhuma ferramenta *CASE* existente atualmente implementa o Modelo de *Features*, adotaremos a prática de listar textualmente todos os Pontos de Adaptação que existirem nos exemplos que virão a seguir.

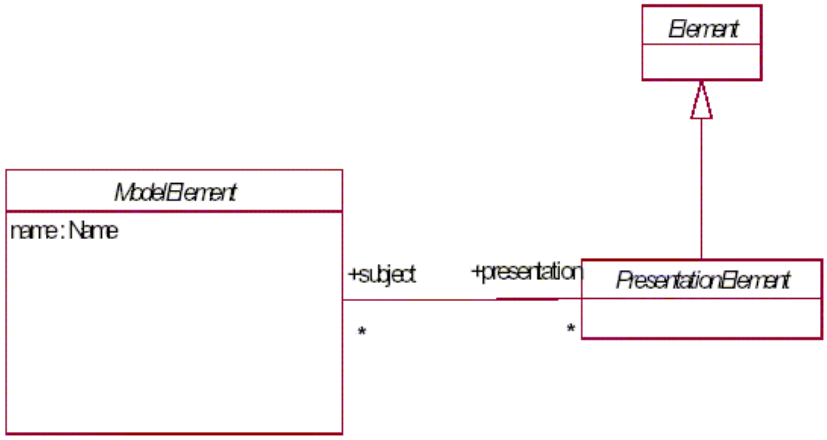


Figura 4.3 – Definição do *Stub* de Apresentação Gráfica dos Elementos de Modelagem da UML.

4.2 O Framework DTFrame

Para melhor ilustrar as questões que serão abordadas a seguir é oportuno neste momento apresentar os detalhes do framework DTFrame. A Figura 4.4 apresenta uma visão panorâmica do diagrama de classes principal do referido framework.

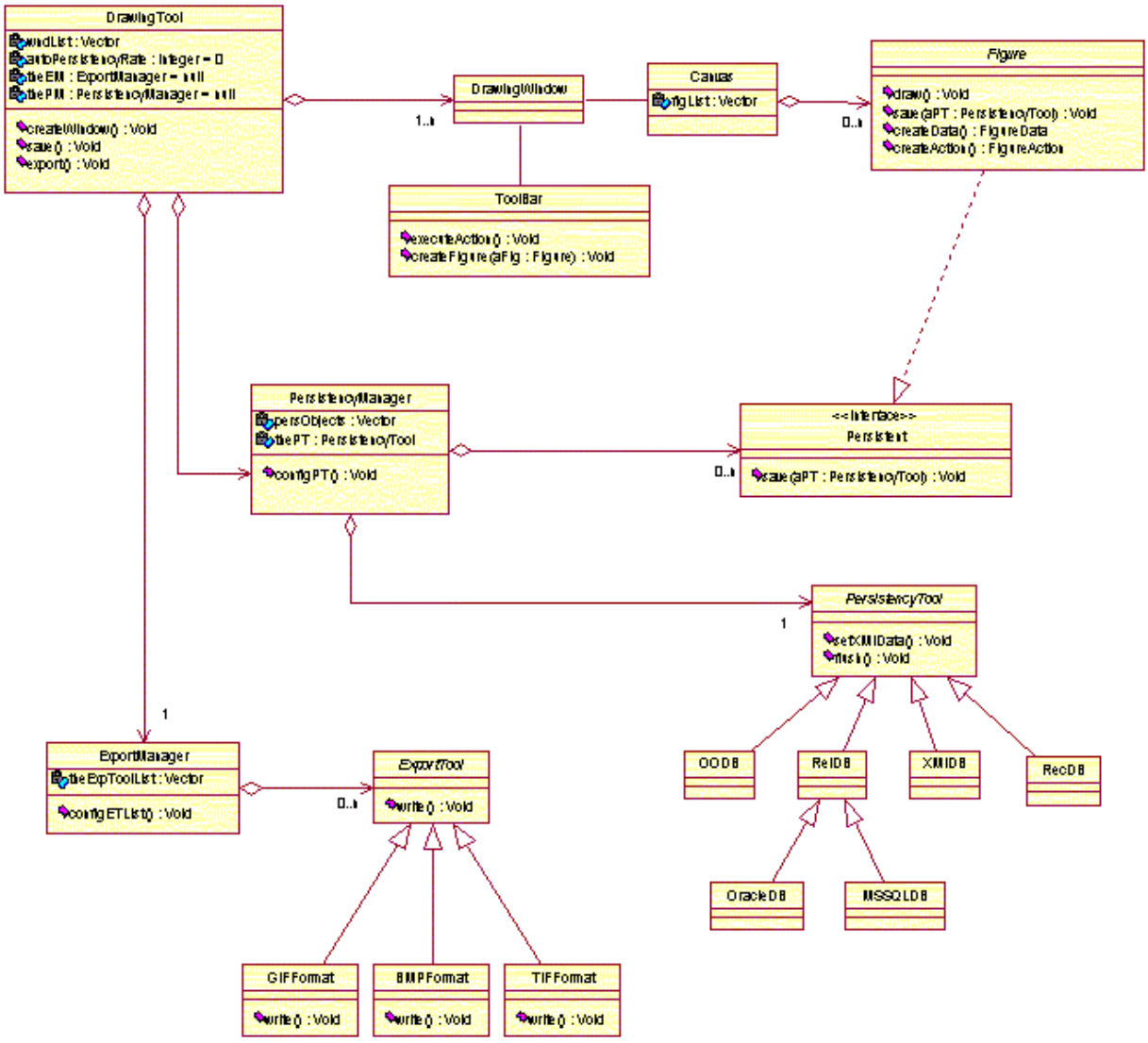


Figura 4.4 – Diagrama de Classes Principal do Framework DTFrame.

Este framework apresenta três grandes características que precisam ser devidamente configuradas para que se possa instanciar o subsistema de interface gráfica de uma ferramenta *CASE*. Tais características são os tipos de figuras existentes (elipse, círculo,

retângulo e etc.), o mecanismo de exportação de figuras e o mecanismo de persistência dos modelos criados pelos usuários.

Não será necessário descrever detalhadamente uma a uma as classes que compõem o framework DTFrame. As palavras escolhidas para nomear os atributos, os métodos e as classes são suficientes para que se compreenda a função exercida por cada um destes elementos de modelagem dentro do contexto do framework DTFrame. A estratégia adotada será apresentar os grupos de elementos que estão envolvidos no design de cada uma das três características citadas anteriormente.

A classe central do DTFrame chame-se **DrawingTool** (Figura 4.5). Ela é um *singleton* (Gamma95) que mantém relacionamentos com os principais componentes do framework. Esta classe não é em si um Ponto de Adaptação, entretanto alguns dos seus atributos o são. O atributo **theEM** é um elemento opcional que tem por objetivo manter o relacionamento entre a classe **DrawingTool** e o gerente de exportação de figuras (**ExportManager**). Este atributo é opcional, dado que, como será visto adiante, a capacidade de exportar figuras é uma característica opcional do DTFrame. A mesma observação vale para o atributo opcional **thePM**. Neste caso, todavia, o referido atributo mantém o relacionamento com o gerente de persistência **(PersistencyManager**).

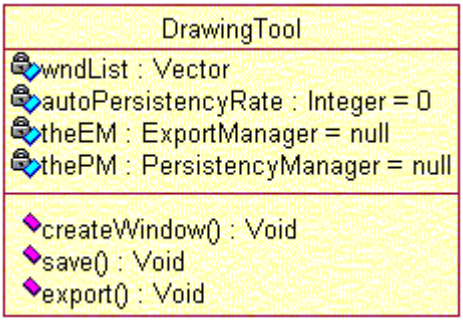


Figura 4.5 – A Classe DrawingTool.

Através do DTFrame é possível exportar os modelos criados com uma ferramenta *CASE* convertendo-se as figuras para os vários padrões existentes de armazenamento de imagens. Esta é uma característica opcional do framework que poderá ou não ser incorporada às aplicações instanciadas a partir do DTFrame. A Figura 4.6 mostra as classes envolvidas na implementação do mecanismo de exportação.

A classe **ExportManager** é um *singleton* que tem por objetivo gerenciar todo o mecanismo de exportação. Ela é um Ponto de Adaptação opcional uma vez que a sua inclusão no design de uma instância do framework irá depender da existência de capacidade de exportação de figuras na instância produzida. Já a classe abstrata **ExportTool**, além de ser um Ponto de Adaptação opcional, pelas mesmas razões da classe **ExportManager**, é também uma extensão do tipo *selectClassExtension*. Isto ocorre porque são fornecidas diversas subclasses de **ExportTool** para exportar figuras para diversos padrões pré-estabelecidos (**GIFFormat**, **TIFFormat** e **BMPFormat**). O método **write** é uma extensão do tipo *methodExtension* e precisa ser redefinido nas subclasses da classe **ExportTool**.

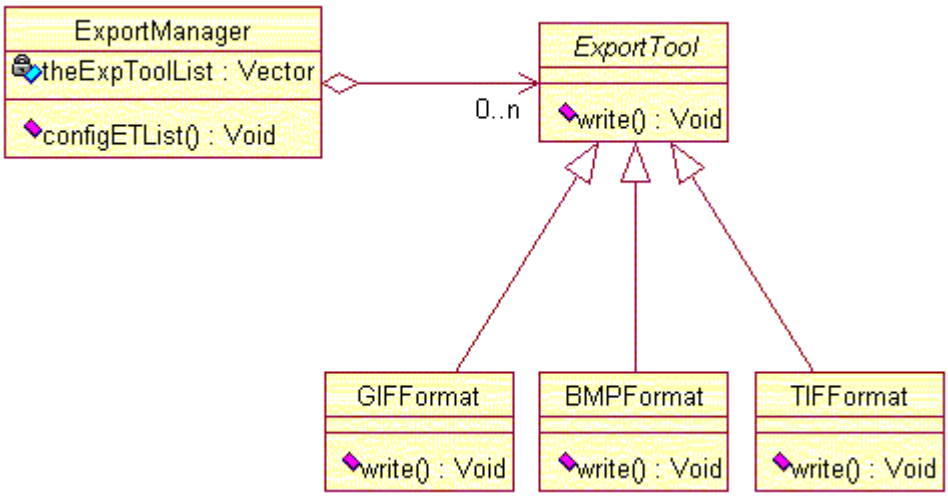


Figura 4.6 – As Classes Envolvidas no Mecanismo de Exportação.

O mecanismo de persistência é gerenciado pela classe *singleton* **PersistencyManager** (Figura 4.7). Ela é um Ponto de Adaptação opcional uma vez que a sua inclusão no design de uma instância do framework irá depender da existência de capacidade de persistência na instância produzida. A classe abstrata **PersistencyTool** além de ser um Ponto de Adaptação opcional, pelas mesmas razões da classe **PersistencyManager**, é também uma extensão do tipo *selectClassExtension* Isto ocorre porque são fornecidas diversas subclasses de **PersistencyTool** com objetivo de implementar os serviços de persistência através de vários sistemas pré-existentes de armazenamento de dados.

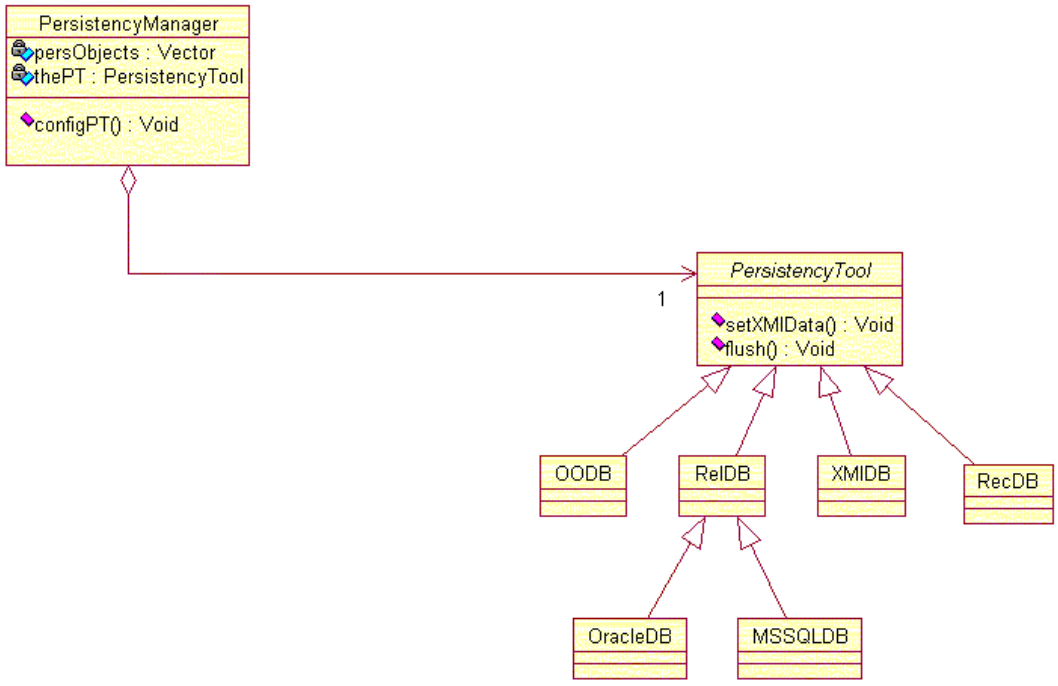


Figura 4.7 – As Classes Envolvidas no Mecanismo de Persistência.

A próxima classe a ser analisada é a classe abstrata **Figure** (Figura 4.8). A sua existência está associada à capacidade do DTFrame de fornecer diversos tipos de figuras para serem utilizadas nos modelos produzidos com a ferramenta *CASE*.

A classe **Figure** é uma classe obrigatória, não sendo, portanto, um Ponto de Adaptação opcional. Entretanto, esta classe é uma extensão do tipo *classExtension*. É

importante observar que, apesar de ser uma extensão, não são fornecidas subclasses pré-existentes junto com o framework. Neste caso cabe aos *desenvolvedores* de aplicação fornecerem, durante o processo de instanciação, as subclasses que irão implementar os tipos de figuras desejados.

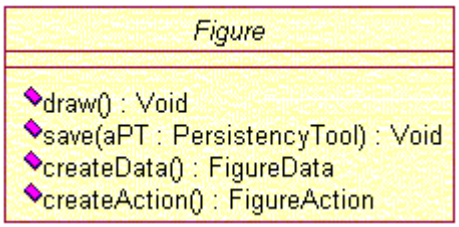


Figura 4.8 – A Classe Figure.

O último aspecto a ser observado é quanto ao uso de *Design Patterns* no framework em questão. A aplicação de *Design Patterns* deve ser executada principalmente durante o processo de construção do framework. Isto permitirá a obtenção de componentes flexíveis e fáceis de serem adaptados às necessidades específicas das aplicações que serão instanciadas. Os usuários dos frameworks (*desenvolvedores* de aplicação) podem, eventualmente, fazer uso de alguns *patterns* durante o processo de instanciação. Entretanto, as tarefas mais comuns que eles irão executar serão a inclusão de subclasses em uma hierarquia de generalização e a redefinição de métodos nas subclasses recém-inseridas.

É importante lembrar que o uso de *patterns* durante o processo de instanciação requer o estudo detalhado do design do framework para que se possa definir o tipo de flexibilização necessário e, conseqüentemente, o tipo de *Design Patterns* que será aplicado. Além disso, pode ser necessário adaptar outros componentes, em outras partes do framework, em virtude da inclusão de um determinado *pattern*. É bastante razoável de se admitir que os projetistas do framework sejam as pessoas mais aptas a realizar este trabalho, uma vez que são eles os responsáveis pelo projeto e pela construção do framework. Logo, é durante o design do framework que deverão ser

tomadas as principais decisões de design objetivando facilitar as tarefas dos usuários de um framework. Entre estas decisões de design certamente estará a aplicação de determinados *Design Patterns* em pontos específicos de um framework.

Esta observação é ainda mais pertinente quando se leva em conta que o principal objetivo deste trabalho (seção 1.5) é propor uma abordagem que dê suporte a um processo de instanciação de alto nível, sem que os usuários de um framework tenham que recorrer ao estudo minucioso do design do mesmo. Para tal, é necessário que sejam aplicados os maiores esforços possíveis durante o design e a construção do framework, para que a sua adaptação seja executada do modo mais simples possível.

A partir do que foi exposto acima, foram introduzidos dois *patterns* no DTFrame para facilitar o processo de inclusão de novas figuras nas ferramentas de desenho que serão instanciadas. O primeiro *pattern*, *Abstract Factory*, irá fornecer uma interface para a criação de novas subclasses relativas aos novos tipos de figuras. Isto será feito em virtude da decisão de colocar os dados e as operações de uma figura em classes separadas (Figura 4.9). Tal decisão se deve à diferença existente na representação de figuras tais como, por exemplo, retângulos e círculos.

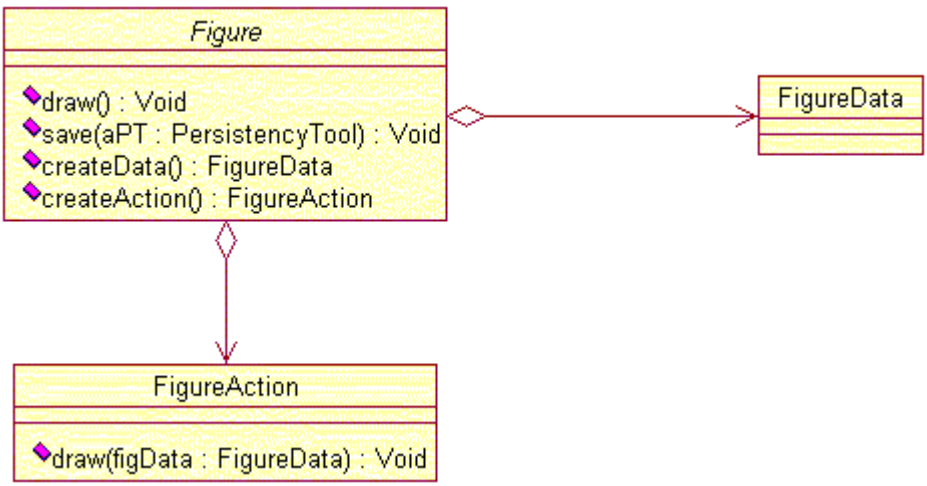


Figura 4.9 – Separando os Dados e as Operações das Figuras

O segundo *pattern*, *Strategy*, permitirá a configuração dinâmica do algoritmo usado para desenhar uma figura. A sua aplicação no DTFrame está ligada à necessidade de se flexibilizar ao máximo a redefinição do método **draw** da classe **FigureAction** (Figura 4.10).

Figura 4.10 – O Uso do *Pattern Strategy* no Framework DTFrame.

A uso dos dois *patterns* anteriores introduz novos Pontos de Adaptação no framework DTFrame. O primeiro é na classe abstrata **FigureData**, que passa a ter uma extensão do tipo *classExtension*. Esta extensão irá permitir a inclusão de subclasses que irão armazenar os dados de um novo tipo de figura (Figura 4.11). O segundo Ponto de Adaptação está localizado na classe abstrata **FigureAction**. De maneira semelhante à classe **FigureData**, esta classe irá armazenar as operações de um novo tipo de figura.

Os dois últimos Pontos de Adaptação estão localizados na classe **DrawStrategy**. O primeiro, ligado à própria classe, é uma extensão do tipo *classExtension*. Ela irá permitir a inclusão de diferentes implementações do método **draw**. O segundo, do tipo *methodExtension*, está ligado ao próprio método **draw**, que deverá ser redefinido nas subclasses de **DrawStrategy**.

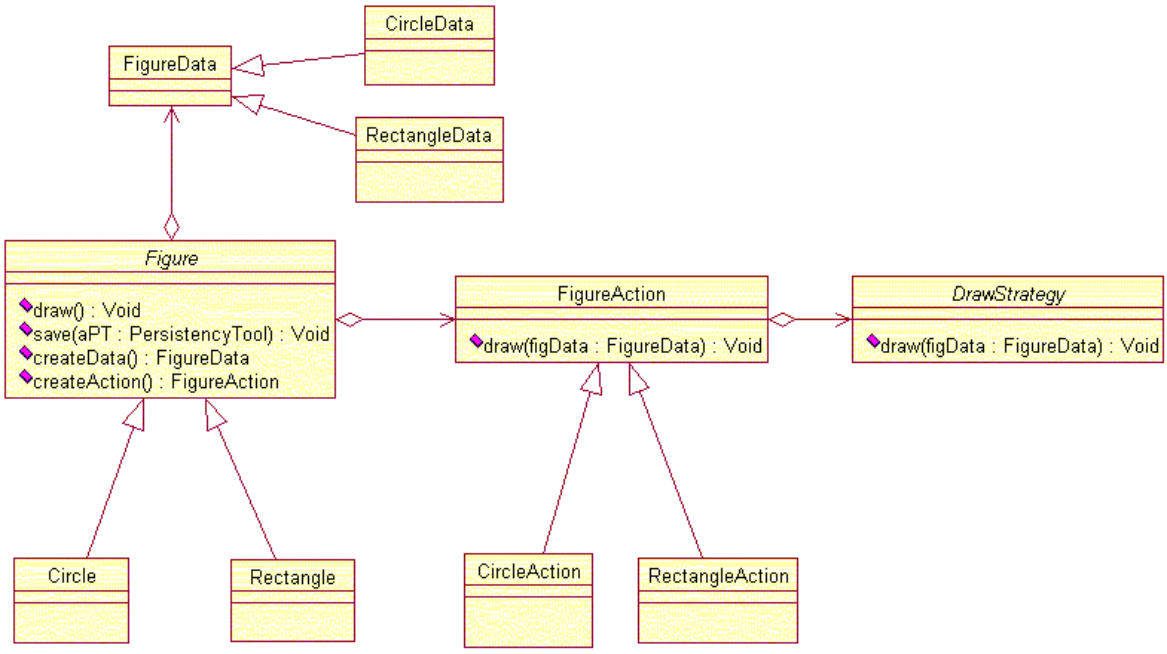


Figura 4.11 – Inclusão das Figuras *Circle* e *Rectangle* no Framework DTFrame.

A partir do que foi exposto até o momento podemos elaborar uma lista com os Pontos de Adaptação existentes no framework DTFrame. A Tabela 4.1 a seguir exibe o nome do elemento de modelagem, o tipo do elemento, a obrigatoriedade ou não do Ponto de Adaptação e o tipo de extensão.

Pontos de Adaptação do Framework DTFrame

Nome	Tipo de Elemento	Obrigatório	Tipo de Extensão
Figure	Classe	Sim	classExtension
Figure.save	Método	Não	methodExtension
Figure.createData	Método	Sim	methodExtension
Figure.createAction	Método	Sim	methodExtension
FigureData	Classe	Sim	classExtension
FigureAction	Classe	Sim	classExtension
DrawStrategy	Classe	Sim	classExtension
DrawStrategy.draw	Método	Sim	methodExtension
ExportManager	Classe	Não	
ExportTool	Classe	Não	selectClassExtension
ExportTool.write	Método	Não	methodExtension

PersistencyManager	Classe	Não	
PersistencyTool	Classe	Não	selectClassExtension
DrawingTool.thePM	Atributo	Não	
DrawingTool.theEM	Atributo	Não	
DrawingTool.autoPersistencyRate	Atributo	Sim	valueAssignment
GIFFormat	Classe	Não	
TIFFormat	Classe	Não	
BMPFormat	Classe	Não	
OODB	Classe	Não	classExtension
RecDB	Classe	Não	classExtension
XMIDB	Classe	Não	classExtension
RelDB	Classe	Não	selectClassExtension
OracleDB	Classe	Não	
MSSQLDB	Classe	Não	

Tabela 4.1 - Pontos de Adaptação do Framework DTFrame

4.3 O Modelo de *Features* do Framework DTFrame

Uma vez examinadas as classes que compõem o framework DTFrame, e listados os Pontos de Adaptação existentes, é necessário agora construir um Modelo de *Features* que descreva as características obrigatórias e opcionais deste framework. A tarefa de construção deste modelo é de responsabilidade dos projetistas do framework. Não serão discutidas aqui questões relativas à elaboração do Modelo de *Features* no contexto do processo de desenvolvimento utilizado na construção do framework. A utilização de métodos *top-down* ou *bottom-up* na construção de frameworks, discutida na seção 1.2 desta dissertação, é irrelevante do ponto de vista do processo de instanciação. Isto é, não há nenhuma diferença se o Modelo de *Features* foi elaborado durante uma etapa explícita de Análise de Domínio ou se a sua construção foi realizada a partir de um framework já existente.

A construção do Modelo de *Features* será dividida em três partes:

a. A construção de um diagrama com as *Features*, as composições de *Features* e os relacionamentos de dependência entre as *Features*.

b. A associação das *Features* com os Pontos de Adaptação que as implementam.

c. A implementação de um mecanismo que permita representar as dependências existentes entre os Pontos de Adaptação.

Nesta seção serão abordadas algumas das questões que envolvem a primeira etapa (item a) da elaboração de um Modelo de *Featrures*. As etapas relativas aos demais itens serão tratadas nas seções 4.4 e 4.5, respectivamente.

A Figura 4.12 mostra o diagrama obtido após a execução da primeira etapa de construção do Modelo de *Features* para o framework DTFrame. Neste diagrama podemos observar as características do DTFrame descritas na seção 4.2. Em primeiro lugar, podemos ver que a configuração de uma figura (*Feature* **Figure**) envolve obrigatoriamente a configuração dos tipos de figuras que serão necessários (*Feature* **Figure Shapes**) e de dois aspectos opcionais: a exportação (*Feature* **Figure Exportation**) e a peristência (*Feature* **Figure Persistency**).

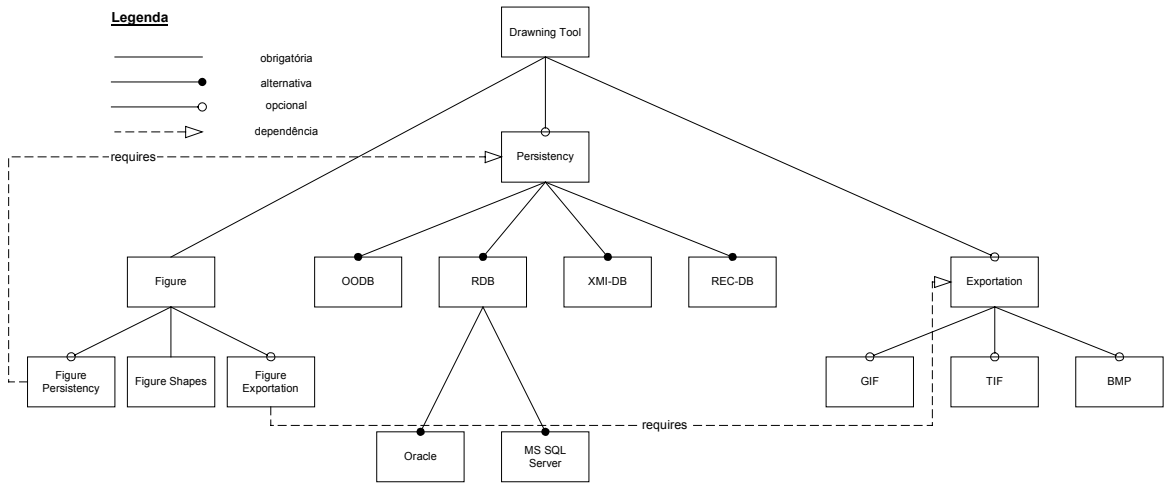


Figura 4.12 – O Diagrama de *Features* do Framework DTFrame.

Em segundo lugar, podemos ver que as características de persistência (*Feature* **Persistency**) e de exportação (*Feature* **Exportation**) são opcionais. No caso do aspecto de persistência são fornecidas várias alternativas, embora no máximo uma delas possa ser selecionada. Já em relação à exportação, pode-se selecionar muitas alternativas (incluindo zero).

Por último, o diagrama da Figura 4.12 mostra que a escolha do aspecto de persistência de uma figura (*Feature* **Figure Persistency**) requer a seleção de um mecanismo de persistência (*Feature* **Persistency**). O mesmo acontece entre o aspecto de exportação de uma figura (*Feature* **Figure Exportation**) e o mecanismo de exportação (*Feature* **Exportation**).

É importante destacar, no que tange à representação das características opcionais e obrigatórias de um framework, a clareza e objetividade do Diagrama de *Features* (Figura 4.12) quando comparado ao diagrama de classes do framework DTFrame (Figura 4.4). Mesmo que usássemos recursos visuais, tais como múltiplas cores, para representar os Pontos de Adaptação diretamente no diagrama de classes, ainda sim o diagrama de *Features* se mostraria mais simples e objetivo. A razão pela qual isto acontece é que quando pensamos em uma família de aplicações para um dado domínio, estamos interessados nas principais características funcionais e tecnológicas presentes neste domínio, sejam elas obrigatórias ou não. Isto é capturado de maneira simples e objetiva por um Diagrama de *Features*, que é capaz de agrupar diversos Pontos de Adaptação relacionados, e espalhados por vários componentes de um design, em uma única *Feature*. Desta forma, é possível representar as características de um domínio de aplicação sem entrar nos detalhes de design do framework.

4.4 A Relação Entre os Modelos de *Features* e de Design

A segunda etapa da construção do Modelo de *Features* envolve o estabelecimento de relacionamentos de dependência entre as *Features* e os elementos do design de um framework envolvidos na implementação de tais *Features*. A execução desta tarefa permitirá localizar no design de um framework os Pontos de Adaptação envolvidos na inclusão de uma determinada característica na aplicação a ser instanciada. Este mecanismo é que irá permitir aos usuários do framework criar aplicações a partir de uma visão de alto nível das diversas opções disponibilizadas pelo framework, sem que haja necessidade de um estudo muito detalhado do seu design.

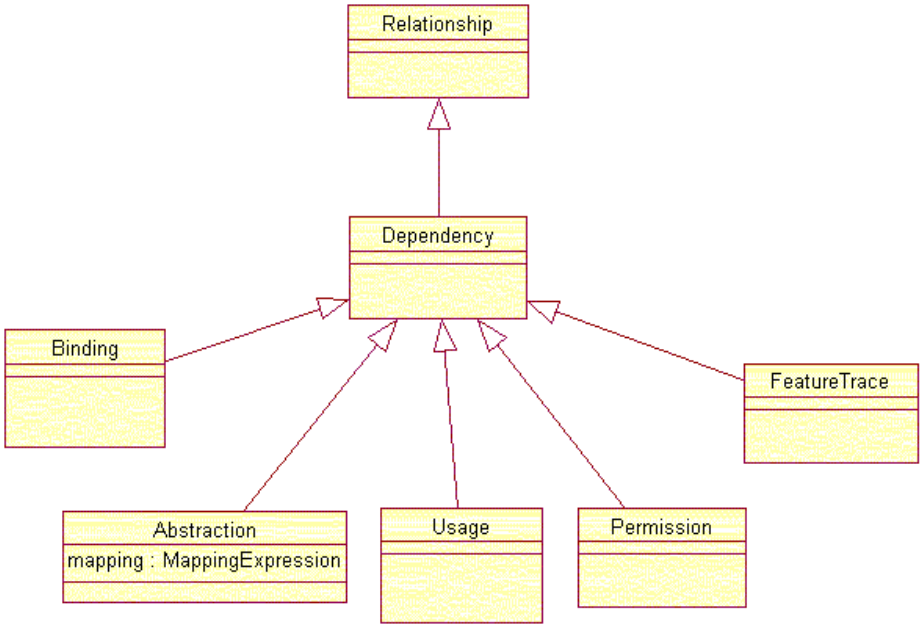


Figura 4.13 – Inclusão de Um Novo Tipo de Dependência no Meta-modelo da UML.

O relacionamento entre os elementos de design e as *Features* será feito através da criação de uma nova meta-classe no meta-modelo da UML. Apesar de a versão atual do meta-modelo da UML fornecer um mecanismo para relacionar elementos localizados em modelos distintos, uma dependência rotulada com o estereótipo <<trace>>, a inclusão de um novo tipo de dependência permite o estabelecimento de regras mais rígidas para o estabelecimento de relacionamentos entre os elementos dos dois

modelos em questão. Logo, foi criada uma nova subclasse da meta-classe **Dependency**, chamada **FeatureTrace** (Figura 4.13), com o objetivo de estabelecer um vínculo entre uma *Feature* e os Pontos de Adaptação responsáveis pela implementação das características associadas com tal *Feature*. Desta maneira será possível associarmos à meta-classe **FeatureTrace** uma restrição que determine que os elementos relacionados com este tipo de dependência têm que ser obrigatoriamente instâncias das meta-classes **DomainFeature** e **AdaptableElement** (Figura 4.14).

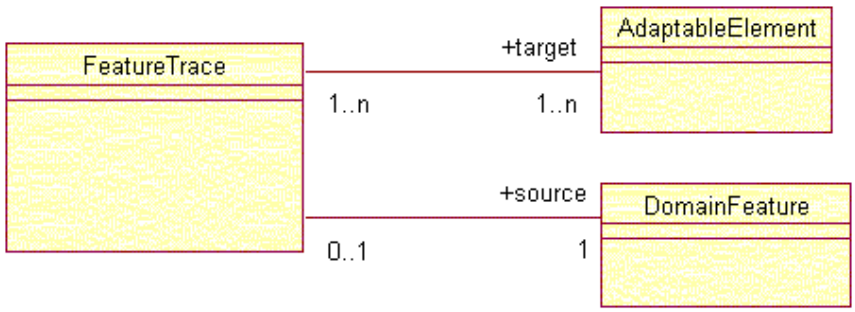


Figura 4.14 – A Meta-classe *FeatureTrace* e os Seus Relacionamentos.

A Tabela 4.2 abaixo mostra a relação das *Features* existentes no Modelo de *Features* do framework DTFrame e os Pontos de Adaptação a elas associados através do relacionamento **FeatureTrace**. É importante lembrar que esta tarefa exige uma grande dose de atenção por parte dos projetistas do framework, já que será necessário avaliar com muito cuidado o critério que será adotado para associar determinados Pontos de Adaptação a uma dada *Feature*. Em última instância será esta atividade que irá determinar a eficácia ou não de todo o processo de instanciação de um framework.

Relação das Dependências Entre as *Features* e os Pontos de Adaptação do Framework DTFrame	
Features	Ponto de Adaptação
Figure	
Figure Persistency	Figure.save
Figure Shapes	Figure
	Figure.createData
	Figure.createAction
	FigureData
	FigureAction
	DrawStrategy
	DrawStrategy.draw
Figure Exportation	
Exportation	ExportManager
	ExportTool
	DrawingTool.theEM
	ExportTool.write
GIF	GIFFormat
TIF	TIFFormat
BMP	BMPFormat
Persistency	PersistencyManager
	PersistencyTool
	DrawingTool.thePM
	DrawingTool.autoPersistencyRate
OODB	OODB
REL-DB	RelDB
XMI-DB	XMIDB
REC-DB	RecDB
Oracle	OracleDB
MS SQL Server	MSSQLDB

Tabela 4.2 - Relação das Dependências Entre as *Features* e os Pontos de Adaptação do Framework DTFrame

Neste momento é necessário estabelecer algumas regras que orientem a etapa de construção dos relacionamentos entre os Modelos de *Features* e de Design de um

framework. Isto será muito útil também na especificação de uma ferramenta que forneça suporte às idéias aqui apresentadas.

Nesta seção, as regras serão definidas informalmente. Isto é, elas serão definidas e explicadas usando apenas a linguagem natural. No Apêndice B, entretanto, serão usados os elementos do meta-modelo da UML, juntamente com expressões em OCL, para tornar mais precisas as regras aqui estabelecidas.

Regras Para a Construção do Modelo de *Features*

1) Uma *Feature* terá uma lista, possivelmente vazia, de Pontos de Adaptação a ela relacionados através de dependências do tipo **FeatureTrace**. Tais Pontos de Adaptação deverão ser considerados como uma unidade lógica de instanciação; isto é, a seleção de uma *Feature* implica na instanciação de todos os Pontos de Adaptação presentes na lista.

2) O nível zero do Diagrama de *Features* irá conter um único nó (o nó raiz da árvore), que irá representar o framework como um todo. Este nó terá caráter meramente ilustrativo, não podendo ter nenhum Ponto de Adaptação a ele associado.

3) Como os *frozen-spots* representam características obrigatórias de uma família de aplicações, não será permitido relacioná-los com nenhuma *Feature*.

A terceira regra acima pode parecer, em uma primeira análise, algo restritivo demais. Entretanto, como a instanciação de um framework envolve fundamentalmente a instanciação dos Pontos de Adaptação, não há sentido em incluir neste processo componentes que não precisam ser adaptados.

4.5 As Dependências Entre os Pontos de Adaptação

Para que o processo de instanciação de um framework seja bem documentado é necessário criar um mecanismo que represente adequadamente as dependências existentes entre os Pontos de Adaptação; ou seja, as dependências que envolvem as classes, os métodos e os atributos.

Analisando-se os tipos de Pontos de Adaptação (opcional e extensão) e os tipos de extensão propostos, podemos dividir as dependências entre os Pontos de Adaptação em duas grandes categorias:

a. Dependência de Existência (DE).

b. Dependência de Seleção Prévia (DSP).

A Dependência de Existência surge quando a seleção de um elemento adaptável para compor uma instância de um framework depender da existência ou não de um outro elemento adaptável no design final desta instância. Por exemplo, o método opcional **Figure.save** (Figura 4.8) possui um parâmetro, **aPT**, da classe **PersistencyTool**. A seleção deste método está subordinada à inclusão da classe **PersistencyTool** no design final da aplicação Como esta classe é também opcional, será necessário estabelecer uma dependência do tipo **requires** entre o método **Figure.save** e a classe **PersistencyTool**.

Em termos de design, as Dependências de Existências mais comuns são:

- A seleção de um método opcional depende da existência de uma classe.

- A seleção de um atributo opcional depende da existência de uma classe.

Embora a ocorrência das outras formas de Dependência de Existência (por exemplo, entre um atributo e um método) seja bastante improvável, não foi imposto à priori nenhum tipo de restrição no estabelecimento de Dependências de Existência entre quaisquer elementos adaptáveis.

O segundo tipo de dependência, Dependência de Seleção Prévia, se aplica apenas à dependência entre um método e uma subclasse da classe onde ele está localizado. Isto é, para instanciar uma extensão do tipo **methodExtension** é necessário apontar antecipadamente a(s) subclasse(s) onde o método em questão será redefinido.

A Tabela 4.3 abaixo mostra as dependências existentes entre os Pontos de Adaptação do framework DTFrame.

Tabela de Dependências Entre os Pontos de Adaptação

Ponto de Adaptação	Dependência	Tipo	Ação
Figure.save	Figure	DSP	Criar subclasse de Figure
Figure.save	PersistencyTool	DE	Selecionar PersistencyTool
Figure.createData	Figure	DSP	Criar subclasse de Figure
Figure.createAction	Figure	DSP	Criar subclasse de Figure
DrawStrategy.draw	DrawStrategy	DSP	Criar subclasse de DrawStrategy
ExportTool	ExportManager	DE	Selecionar ExportManager
ExportTool.write	ExportTool	DSP	Criar subclasse de ExportTool
PersistencyTool	PersistencyManager	DE	Selecionar PersistencyManager
DrawingTool.thePM	PersistencyManager	DE	Selecionar PersistencyManager
DrawingTool.theEM	ExportManager	DE	Selecionar ExportManager

Tabela 4.3 - Tabela de Dependências Entre os Pontos de Adaptação

O que será mostrado a seguir é que os relacionamentos de dependências entre *Features*, **requires** e **mutexWith**, juntamente com o estabelecimento de algumas regras de instanciação de um Modelo de *Features*, serão suficientes para representar adequadamente as dependências entre os Pontos de Adaptação.

Regras Para a Instanciação de um Modelo de *Features*

As regras abaixo estão formalmente definidas, usando a linguagem OCL, no Apêndice B. O número listado entre parênteses facilita a localização da mesma.

1) Quando uma *Feature* for selecionada todos os Pontos de Adaptação a ela relacionados terão o valor do atributo **selected** alterados para a constante Booleana **true** (regra B.7).

2) A seleção (**selected = true**) de um Ponto de Adaptação contendo uma extensão do tipo **methodExtension** implica na seleção da classe na qual ele está inserido (regra B.6).

3) A seleção (**selected = true**) de um Ponto de Adaptação contendo uma extensão do tipo **valueAssignment**, ou **valueSelection**, implica na seleção da classe na qual ele está inserido (regra B.2).

4) A seleção (**selected = true**) de um Ponto de Adaptação contendo uma extensão do tipo **classExtension**, ou **selectClassExtention**, implica na seleção de todas as classes ascendentes da classe em questão (regra B.4).

As quatro regras acima resolvem a questão da Dependência de Existência entre uma classe e as suas classes ascendentes, entre um método e a classe onde ele está inserido e entre um atributo e a classe na qual ele está inserido.

Os demais casos podem ser resolvidos associando-se os elementos dependentes a uma única *Feature*, ou à *Features* distintas relacionadas pela dependência **requires**.

Voltemos agora à questão da Dependência de Existência entre o método **Figure.save** e a classe **PersistencyTool**. Para resolvermos esta questão poderíamos relacionar ambos os Pontos de Adaptação a uma mesma *Feature*; por exemplo, a *Feature* **Persistency**. Desta maneira, a seleção desta característica implicaria na seleção simultânea do dois Pontos de Adaptação.

A outra solução, que foi a solução adotada, seria associar o método **Figure.save** a uma *Feature*, digamos **Figure Persistency**, e a classe **PersistencyTool** a uma outra *Feature*, digamos **Persistency**. Por último estabeleceríamos uma dependência do tipo **requires** entre a *Feature* (**Figure Persistency**) na qual estivesse localizado o elemento dependente, e a *Feature* (**Persistency**) na qual estivesse localizado o elemento independente. Logo, a seleção da *Feature* onde estivesse o elemento dependente implicaria na seleção da *Feature* onde estivesse localizado o elemento independente e, por conseguinte, a inclusão de ambos os Pontos de Adaptação no design final.

Em relação à Dependência de Seleção Prévia existem duas situações possíveis. Na primeira delas a subclasse na qual o método será inserido já existe no design do framework e foi previamente selecionada. Neste caso a dependência seria resolvida propagando-se o método em questão pelas subclasses selecionadas. Na segunda situação a subclasse não existe ainda no design do framework e precisa ser inserida pelo responsável pela instanciação do framework. Neste caso a instanciação do método dar-se-á após a classe ter sido inserida no design pelo usuário do framework.

Um último comentário em relação ao estabelecimento de dependências entre *Features* faz-se necessário. Apesar de a atividade de relacionar elementos do Modelo de

Features com os elementos de design de um framework ser uma tarefa criativa, algum suporte ferramental é necessário para auxiliar os projetistas do framework na execução desta tarefa. Em primeiro lugar a ferramenta deverá garantir o cumprimento de todas as regras de construção e instanciação de um Modelo de *Features* descritas anteriormente. Além disso, será necessária a utilização de algoritmos que detectem a existência de ciclos nos grafos [Aho95] de dependências de um Modelo de *Features*. Alguns tipos de ciclos são especialmente indesejáveis, entre eles está aquele que permitiria criar uma situação onde a escolha de uma *Feature* dependesse da escolha de duas outras *Features* que fossem mutuamente exclusivas. A existência de um ciclo desta natureza levaria a uma situação na qual nenhuma instância válida poderia ser obtida a partir de tal Modelo de *Features*.

4.6 O Design Final da Aplicação Instanciada

De acordo com as idéias aqui apresentadas, o processo de instanciação de um framework será realizado selecionando-se as *Features* desejadas e, em decorrência disto, alterando-se o design original do framework para produzir o design final da aplicação sendo instanciada.

As etapas que cobrem a construção e a instanciação de um Modelo de *Features* foram devidamente tratadas nas seções anteriores deste capítulo. A última etapa que falta para ser discutida é a maneira pela qual o design do framework será alterado a partir das escolhas realizadas no Modelo *Features*.

Em termos gerais, esta etapa irá gerar um novo Modelo de Design do qual irão constar todos os elementos obrigatórios do design original do framework, mais os elementos opcionais que foram selecionados a partir do Modelo de *Features* e mais as classes que serão inseridas pelos *desenvolvedores* de aplicação durante o processo de

instanciação. Estas alterações poderão ser realizadas diretamente por uma ferramenta com o mínimo de interferência humana. As únicas informações adicionais necessárias, externas à ferramenta, serão os nomes das subclasses a serem inseridas durante o processo de adaptação de extensões do tipo *methodExtension*, e os valores que serão atribuídos aos atributos durante a adaptação de extensões dos tipos *valueAssignment* e *valueSelection*.

A estratégia que foi adotada será gerar um *script* de instanciação na linguagem RDL (*Reuse Description Language*) [Oliveira01a] para que a **RDLVirtualMachine**, componente da ferramenta xFIT [Oliveira01a] responsável pela execução do *script*, possa realizar as alterações necessárias no design do framework e gerar o design final da aplicação sendo instanciada. Esta opção deve-se à existência de um protótipo da máquina virtual de RDL e às afinidades existentes entre os conceitos de instanciação de frameworks aqui propostos e àqueles apresentados em [Oliveira01a].

Neste momento faz-se necessária uma breve descrição da linguagem RDL para que o processo de geração do design final de uma aplicação a partir de um Modelo de *Features* possa ser bem compreendido.

A Linguagem RDL é uma linguagem textual capaz de representar as atividades relativas ao domínio de instanciação de frameworks. Um programa RDL é constituído por uma seqüência de atividades de instanciação que pode ser vista como um *Cookbook* para a produção de aplicações a partir de um framework.

As atividades de instanciação da RDL são divididas em quatro categorias:

1) **Básicas** – estas atividades objetivam representar os mecanismos básicos de extensão de uma aplicação orientada a objetos (seção 4.1). Elas representam o núcleo do processo de instanciação.

2) **Estendidas** – estas atividades permitem o aprimoramento da qualidade do design final da aplicação. Para tal elas implementam, por exemplo, primitivas para a inclusão de *Design Patterns*.

3) **Restritivas** – estas atividades permitem a especificação de dependências na execução das atividades de instanciação.

4) **Organizacionais** – estas atividades permitem organizar o processo de instanciação em blocos de atividades que podem ser designados para diferentes equipes de desenvolvimento. É possível, inclusive, determinar que alguns blocos de atividades possam ser executados paralelamente.

O processo de instanciação aqui proposto utilizará apenas as atividades básicas e algumas das atividades estendidas definidas na RDL. Tais atividades são, de acordo com [Oliveira01a], suficientes para produzir aplicações a partir de um framework. Não serão utilizadas, por exemplo, as primitivas para a aplicação de *Design Patterns*, presentes nas atividades estendidas. As razões para tal decisão foram amplamente discutidas nas seções 4.1 e 4.2 desta dissertação. Além disso, as atividades restritivas também não serão levadas em consideração, uma vez que as dependências entre os Pontos de Adaptação serão tratadas diretamente no Modelo de *Features*.

A não utilização das atividades organizacionais tem origem no próprio paradigma do processo de instanciação aqui proposto. Ao criarmos uma camada *translúcida*, representada pelo Modelo de *Features*, entre os *desenvolvedores* de aplicação e o design do framework a ser instanciado, estaremos levantando algumas barreiras que irão dificultar, por exemplo, a organização do processo de instanciação em atividades paralelas. Neste caso seria necessário um estudo detalhado do design do framework para que as tarefas que pudessem ser realizadas em paralelo fossem alocadas a grupos de *desenvolvedores* distintos. Além disso, a versão do Modelo de *Features* aqui

proposto não carrega consigo informações suficientes para que as atividades de instanciação possam ser organizadas de acordo com o processo de desenvolvimento de software adotado pela equipe responsável pela instanciação do framework.

Entretanto, seria possível incorporar, a uma ferramenta de auxílio ao processo de instanciação, mecanismos que permitissem à equipe de desenvolvimento organizar as *Features* em grupos correlatos e independentes. Isto é, em grupos que implementem funcionalidades correlatas e que apresentem apenas dependências internas. No caso do DTFrame, poderíamos dividir as *Features* em três grupos independentes. O primeiro iria conter apenas a *Feature* **Figure Shapes**. O segundo a *Feature* **Figure Exportation**, a *Feature* **Exportation** e todas as *Features* que representem os diferentes mecanismos de exportação. Por último, o terceiro grupo seria composto pela *Feature* **Figure Persistency**, a *Feature* **Persistency** e todos as *Features* que representem os diferentes mecanismos de persistência. Desta maneira seria possível alocar cada um dos três grupos à equipes de desenvolvimento independentes, que poderiam realizar as suas respectivas tarefas de instanciação paralelamente.

Para a geração do design final da aplicação a ser instanciada serão usados os seguintes comandos RDL:

Loop <comandos> **end_loop;**

Permite a especificação de atividades repetitivas. Na versão atual da RDL a parada da repetição é determinada pelo responsável pela execução do *script* de instanciação; ou seja, os <comandos> são executados enquanto for necessário.

class_extension (CLASS_EXP**);**

Será inserida no modelo uma nova classe que especializa a classe CLASS_EXP. O nome desta nova classe deverá ser fornecido pelo responsável pela execução do *script* de instanciação.

selection_class_extension (CLASS_EXP**);**

Será selecionada uma subclasse da classe CLASS_EXP. A escolha desta subclasse deverá ser feita pelo responsável pela execução do *script* de instanciação.

method_extension (CLASS_EXP, CLASS_EXP, METHOD_EXP**);**

Redefine o método METHOD_EXP, presente na classe definida pelo primeiro parâmetro, e o insere na classe definida pelo segundo parâmetro. A classe definida pelo segundo parâmetro deverá ser subclasse da classe definida pelo primeiro parâmetro.

value_assignment (CLASS_EXP, ATTRIB_EXP**);***Inicializa* o atributo ATTRIB_EXP, presente na classe CLASS_EXP, com um valor fornecido pelo responsável pela execução do *script* de instanciação.

value_selection (CLASS_EXP, ATTRIB_EXP, LIST)*Inicializa* o atributo ATTRIB_EXP, presente na classe CLASS_EXP, com um valor selecionado a partir da lista de valores disponibilizadas em LIST. A seleção do elemento da lista de valores é feita pelo responsável pela execução do *script* de instanciação.

A geração de um *script* RDL será realizada percorrendo-se o Diagrama de *Features* em duas etapas. A primeira etapa irá modificar o valor do atributo **selected** (**selected=true**) de todos os elementos adaptáveis opcionais que foram selecionados. A execução desta etapa irá resolver todas as Dependências de Existência presentes no Modelo de *Features*.

A segunda etapa irá percorrer novamente o Diagrama de *Features* e, a partir de consultas feitas aos Modelos de Design do framework, irá proceder a geração do *script* RDL. A geração de tal *script* dar-se-á mediante à observação das seguintes regras:

Regras Para a Geração de um *Script* de Instanciação em RDL

1) A seleção de uma *Feature* irá gerar, para cada Ponto de Adaptação do tipo *classExtension* e *selectClassExtention* associado à *Feature*, a seguinte seqüência de comandos RDL:

   1.1) Um **Loop** com um comando **class_extension**, seguido de tantos comandos **method_extension** quantos forem os Pontos de Adaptação do tipo *methodExtension* existentes na classe. Note que se já existirem subclasses no modelo a propagação será automaticamente realizada pelo mecanismo de herança.

2) Cada Ponto de Adaptação do tipo *valueAssignment*, ou *valueSeletion*, irá gerar um comando RDL **value_assignment**, ou **value_selection**.

Tomemos agora como exemplo uma instância do Diagrama de *Features* do framework DTFrame, mostrada pela Figura 4.15. Neste diagrama, as Features escolhidas para compor a aplicação a ser instanciada estão marcadas com a cor cinza.

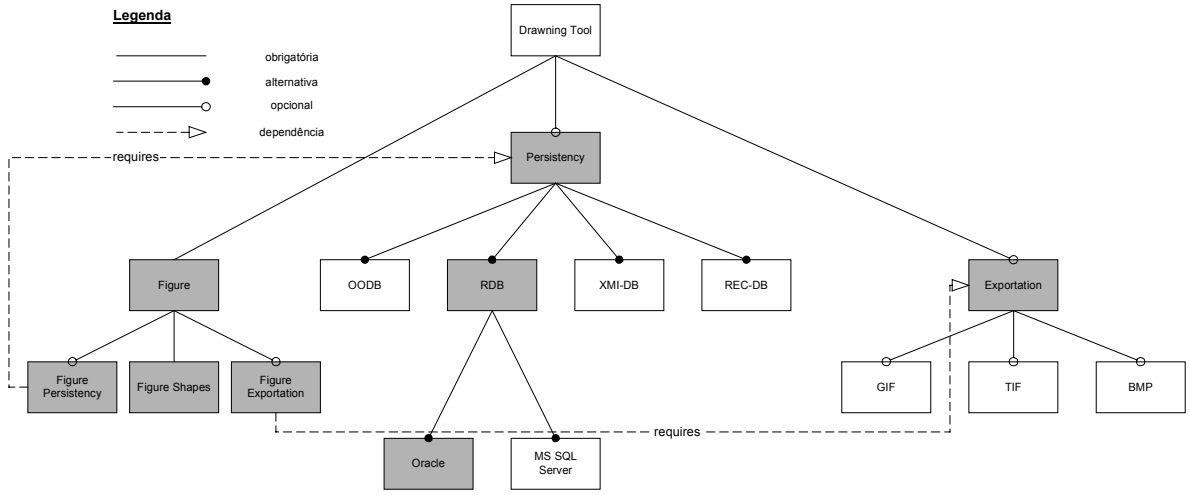


Figura 4.15 – Uma Instância do Diagrama de *Features* do Framework DTFrame.

Baseado nas escolhas mostradas na Figura 4.15, nas informações contidas na Tabela 4.2 e nas regras descritas acima, o seguinte *script* RDL será gerado:

```
COOKBOOK DTFrame
  RECIPE MAIN;

    LOOP
      V1=CLASS_EXTENSION(Figure);
      METHOD_EXTENSION(Figure,V1,save);
      METHOD_EXTENSION(Figure,V1,createAction);
      METHOD_EXTENSION(Figure,V1,createData);
    END_LOOP;

    LOOP
      V1=CLASS_EXTENSION(FigureData);
    END_LOOP;

    LOOP
      V1= CLASS_EXTENSION(FigureAction);
    END_LOOP;

    LOOP
      V1=CLASS_EXTENSION(DrawStrategy);
      METHOD_EXTENSION(DrawStrategy,V1,draw);
    END_LOOP;

    LOOP
      V1=CLASS_EXTENSION(ExportTool);
      METHOD_EXTENSION(ExportTool,V1,write);
    END_LOOP;

    VALUE_ASSIGNMENT(DrawingTool, autoPersistencyRate);

  END_RECIPE;
END_COOKBOOK;
```

4.7 Considerações Finais

O presente capítulo apresentou uma minuciosa descrição de todo o processo de instanciação de frameworks proposto nesta dissertação. Inicialmente foram definidos os dois tipos de Pontos de Adaptação (opcional e extensão) que podem ser encontrados no design de um framework. Tal definição foi baseada na premissa de que um framework possui aspectos flexíveis que devem ser adaptados para produzir aplicações para um determinado domínio. Por se tratar de um design orientado a objetos, a adaptação será executada através da aplicação dos mecanismos clássicos de extensão de aplicações orientadas a objetos. Além disso, um Ponto de Adaptação pode representar uma característica que pode ou não ser incluída no design final das aplicações instanciadas.

Em seguida foram descritas as regras para a construção de um Diagrama de *Features* que represente de maneira objetiva as características opcionais e obrigatórias de um domínio de aplicação. Características estas que estão, de uma maneira ou de outra, refletidas no design de um framework para tal domínio.

O passo seguinte foi discutir e apresentar as regras que definem todo o processo de associação das *Features* com os elementos do design de um framework que implementem tais *Features*. Neste contexto foram definidas também as regras para o tratamento da questão das dependências existentes entre os Pontos de Adaptação.

Por último foi abordada a estratégia usada para produzir o design final da aplicação a ser instanciada a partir da modificação do design original do framework. Esta estratégia está baseada na geração de um *script* de instanciação, composto por comandos da linguagem RDL, que reflita as escolhas das características que os

desenvolvedores de aplicação desejam ver incluídas no design final da aplicação a ser instanciada.

O próximo capítulo irá apresentar uma ferramenta capaz de dar suporte a todas a fases de um processo de instanciação de frameworks baseado nas idéias apresentadas anteriormente.

Capítulo 5

A Ferramenta Sagan *Tool*

O objetivo deste capítulo é apresentar uma visão geral da ferramenta Sagan[3] *Tool*. Esta ferramenta foi projetada para implementar as idéias vistas nos dois capítulos anteriores desta dissertação. Ou seja, ela permitirá gerar um *script* de instanciação RDL a partir de dois modelos fundamentais de um framework: um diagrama de classes que represente os principais aspectos do seu design e um Modelo de *Features* que descreva as suas características obrigatórias e opcionais. A Figura 5.1 mostra o resumo do processo de instanciação implementado por Sagan *Tool*.

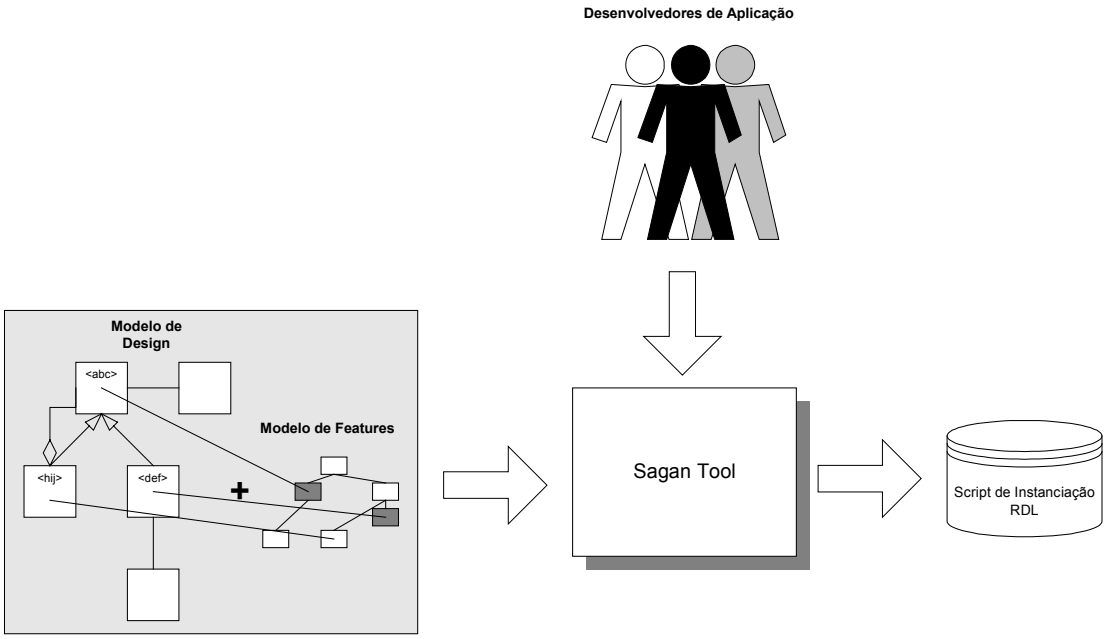


Figura 5.1 – A Geração de Um *Script* de Instanciação RDL.

[3] Homenagem ao físico e astrônomo americano Carl E. Sagan (1934-1996)

Além dos principais aspectos arquiteturais serão vistos também alguns detalhes da implementação de um protótipo desta ferramenta. Este protótipo servirá para comprovar a viabilidade das idéias aqui apresentadas. Entretanto, o detalhamento da modelagem do protótipo de Sagan *Tool* foi deixado para o Apêndice C desta dissertação.

5.1 Os Principais Subsistemas de Sagan *Tool*

Sagan *Tool* é composta por cinco subsistemas, como pode ser visto na Figura 5.2. O subsistema principal, *Main*, é responsável pelo recebimento de quaisquer solicitações oriundas da interface com os usuários, *User Interface*, e pelo conseqüente repasse das mesmas para os subsistemas responsáveis pela execução de tais solicitações. Além disso, este módulo é responsável pela coordenação de tarefas que envolvam a cooperação dos vários subsistemas. Isto permite o desacoplamento entre os componentes responsáveis pela implementação das principais funcionalidades da ferramenta, e a interface gráfica usada para a comunicação com os usuários.

O subsistema *Design Model Manager* é responsável pela gerência do Modelo de Design do framework. Idealmente este componente deverá importar os diagramas de classes diretamente da ferramenta *CASE* usada para modelar o framework. Esta *importação* poderá ser feita de duas formas distintas. Na primeira delas as informações seriam obtidas acessando-se diretamente os arquivos da ferramenta *CASE* responsável pelo armazenamento dos modelos. Contudo, esta opção dever ser evitada, uma vez que seria necessário implementar diferentes versões do subsistema para as diferentes ferramentas existentes no mercado. Além disso, o acesso à documentação sobre a organização interna de tais ferramentas *CASE* pode ser outro fator que dificulte a adoção desta modalidade de *importação* de modelos.

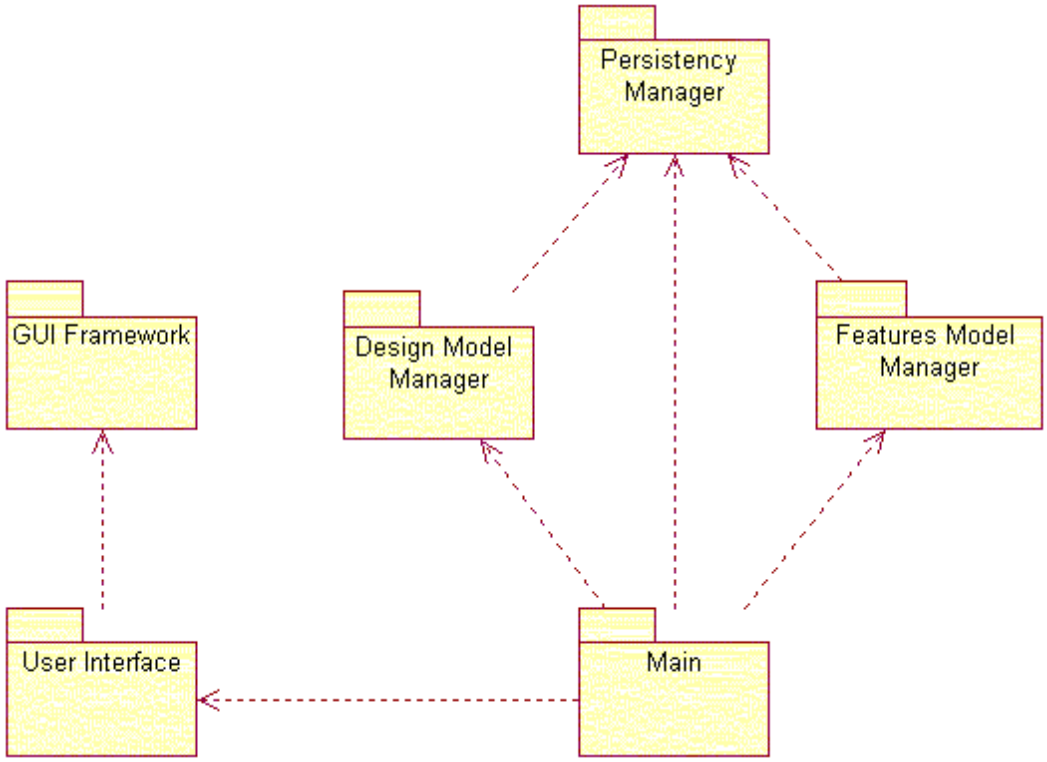


Figura 5.2 – Os Subsistemas da Ferramenta Sagan *Tool*.

A segunda opção seria utilizar uma notação padrão intermediária para a representação dos modelos de design do framework. Desta forma, a ferramenta *CASE* poderia *exportar* os modelos para o formato intermediário, que seriam então *importados* pelo subsistema *Design Model Manager*. Esta opção é certamente a mais vantajosa, principalmente devido à existência do padrão XMI [XMI], uma especificação da linguagem UML usando a linguagem XML [XML].

O subsistema *Features Model Manager* é responsável pelo gerenciamento do Modelo de *Features*. Este componente tem duas responsabilidades básicas. A primeira é permitir a criação de um Modelo de *Features* para um dado framework. Este modelo deverá conter um Diagrama de *Features*, deverá representar as dependências existentes em um Modelo de *Features* e, por último, associar as *Features* com os elementos de design que representem os Pontos de Adaptação de um framework.

A segunda responsabilidade deste subsistema é produzir uma instância do Modelo de *Features* com base na seleção de algumas das características que serão incorporadas na aplicação que será instanciada a partir do framework.

O último subsistema, *Persistency Manager*, é o componente responsável pelo armazenamento dos modelos manipulados pela ferramenta em uma base de dados persistente.

5.2 A Descrição do Protótipo Implementado

Passaremos agora para a descrição da implementação do protótipo da ferramenta Sagan *Tool*. Em primeiro lugar é importante listar algumas simplificações que foram feitas no protótipo em relação à proposta original, apresentada na seção anterior. As simplificações são as seguintes:

1) Tanto o Diagrama de *Features* como o Diagrama de Classes de um framework são apresentados de maneira textual, embora o ideal fosse representá-los graficamente. Esta simplificação deve-se à dificuldade de se implementar em tempo hábil uma interface gráfica semelhante às disponibilizadas pelas ferramentas *CASE* existentes no mercado. Além disso, o objetivo do protótipo é mostrar a viabilidade das idéias aqui apresentadas. Para tal, a interface disponibilizada pelo protótipo é suficiente.

2) O Diagrama de Classes representado o design de um framework não foi obtido a partir da importação da sua representação em uma ferramenta *CASE*. Eles foram implementados diretamente na base de dados da ferramenta. Esta simplificação deve-se ao fato de que as idéias aqui apresentadas implicam na modificação do meta-modelo da UML. Logo, não seria possível obter uma representação XMI, a partir de uma ferramenta *CASE* existente, que implementasse as alterações no

meta-modelo da UML aqui propostas. Entretanto, as implementações das tabelas que armazenam as classes, os métodos e os atributos seguem rigorosamente as especificações apresentadas em [UML01], acrescidas das modificações no meta-modelo da UML anteriormente apresentadas.

3) A versão implementada do subsistema *Features Model Manager* não permite a criação do Modelo de *Features* de maneira interativa. Assim como no caso do Diagrama de Classes, o Modelo de *Features* foi implementado diretamente na base de dados da ferramenta. Apenas a parte relativa à instanciação de um Modelo de *Features* é que foi implementada. Novamente isto não traz maiores inconvenientes para a validação das idéias aqui apresentadas, já que o processo de instanciação proposto concentra-se justamente na fase que começa com a instanciação de um Modelo de *Features*.

A implementação do protótipo da ferramenta Sagan *Tool* foi feita no ambiente de programação Borland Delphi 5.0 [Delphi] e usou o sistema de arquivos Microsoft Access 2000 [Access] para implementar a gerência de persistência. Embora o MS Access não seja um gerenciador de banco de dados relacional completo, ele apresenta um modelo de dados baseado no Modelo Relacional [Date99] e uma interface SQL para a manipulação dos dados.

O uso da ferramenta começa pela escolha de um projeto que já exista previamente na base de dados de Sagan *Tool*. A criação de novos projetos, bem como a *importação* de Diagramas de Classes, não está implementada na versão atual do protótipo. A Figura 5.3 mostra a tela inicial com os projetos existentes.

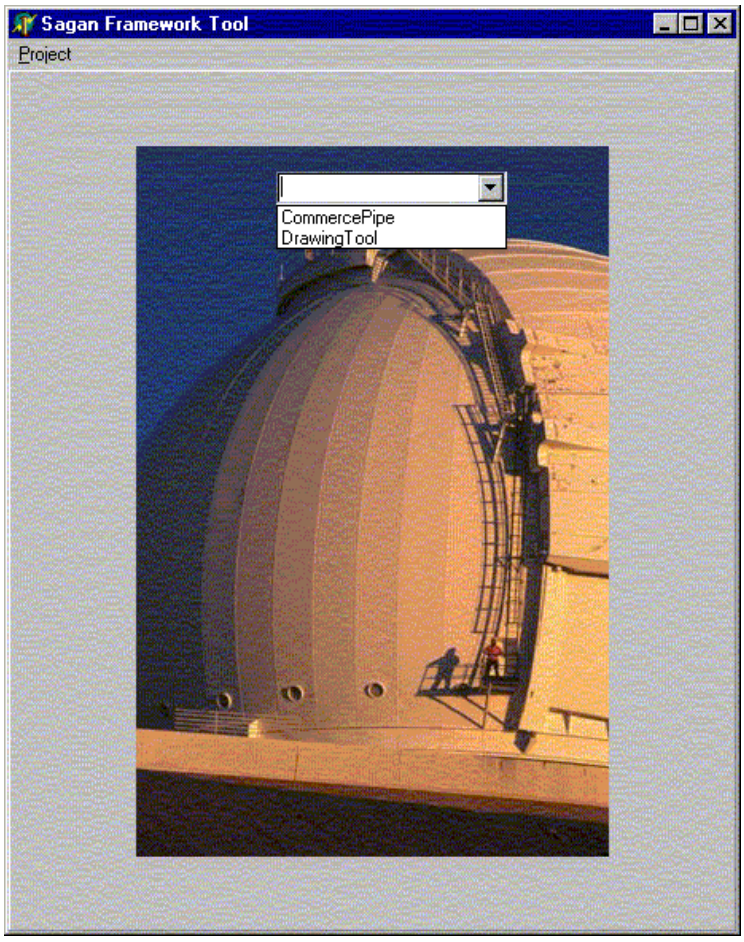


Figura 5.3 – Tela de Abertura do Protótipo de Sagan *Tool*.

A segunda etapa consiste na carga dos Modelos de *Features* e de design do framework escolhido. A carga dos modelos implicará na leitura da base de dados e na implementação, em memória, das estruturas necessárias para manter as referências entre os elementos que constituem os dois modelos (*features*, classes, métodos, atributos e etc).

Como o Modelo de *Features* será apresentado de maneira textual foi necessário criar uma nova notação para representar os diversos tipos de *Features* existentes. Desta maneira, uma *Feature* obrigatória será representada apenas pelo nome da mesma, uma *Feature* opcional será representada apresentado-se o nome da mesma entre colchetes e uma composição alternativa será apresentada com os nomes das *Features* mostrados entre chaves angulares duplas (<< >>).

A Figura 5.4 mostra a versão do protótipo de Sagan *Tool* para o Diagrama de *Features*.

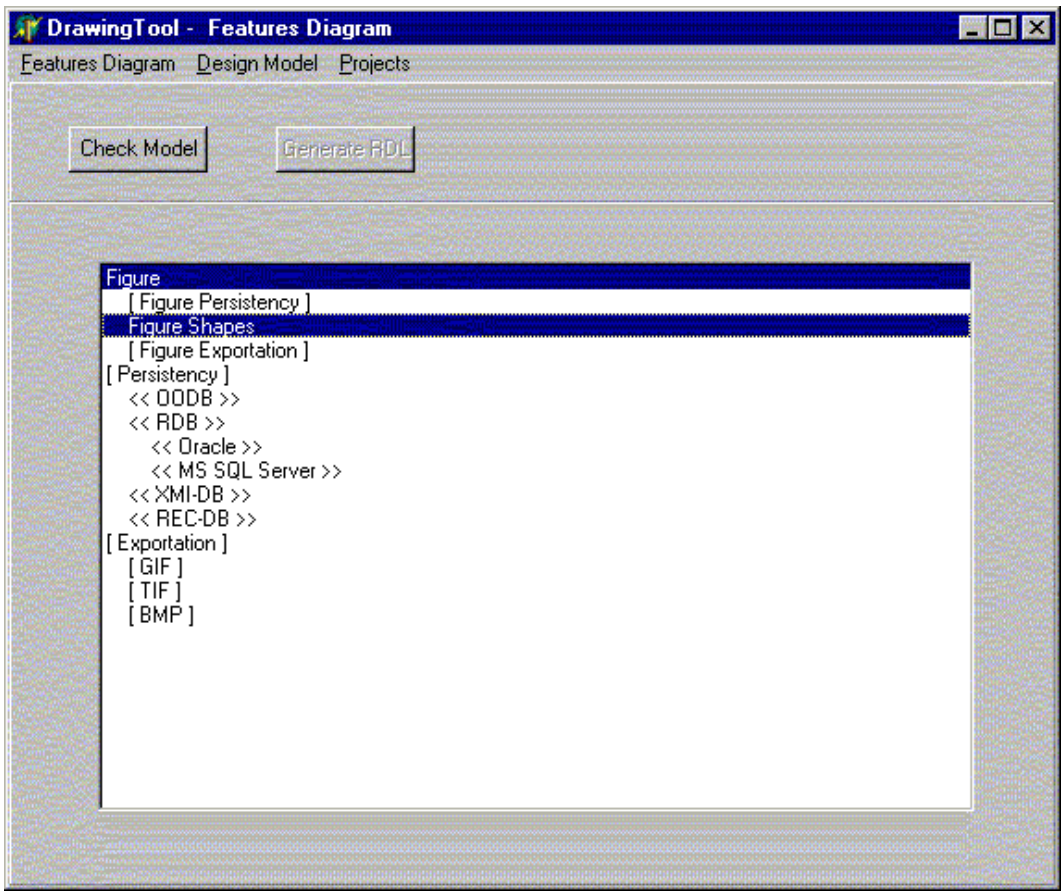


Figura 5.4 – O Diagrama de *Features* do Protótipo de Sagan *Tool*.

Na figura acima é possível identificar o Diagrama de *Features* do framework DTFrame (Figura 4.18), usado como exemplo no Capítulo 4 desta dissertação. As *Features* **Figure** e **Figure Shapes** aparecem selecionadas por serem *Features* obrigatórias. É possível também observar a ocorrência de *Features* opcionais, tal como **Exportation**, e de *Features* que participam de composições alternativas, tais como **OODB**, **Oracle** e **XMI-DB**.

O caráter textual do Diagrama de *Features* torna inconveniente a exibição de todas as informações sobre o modelo em um único componente visual. Por exemplo, não é possível saber, a partir do diagrama da Figura 5.4, se a escolha da *Feature* **Figure Persistency** requer ou não a escolha da *Feature* **Persistency**. Outra informação não

disponível é a relação de Elementos Adaptáveis associados a uma determinada *Feature*. Esta deficiência foi contornada com a inclusão de um Inspetor de *Features* na ferramenta Sagan *Tool* (Figura 5.5).

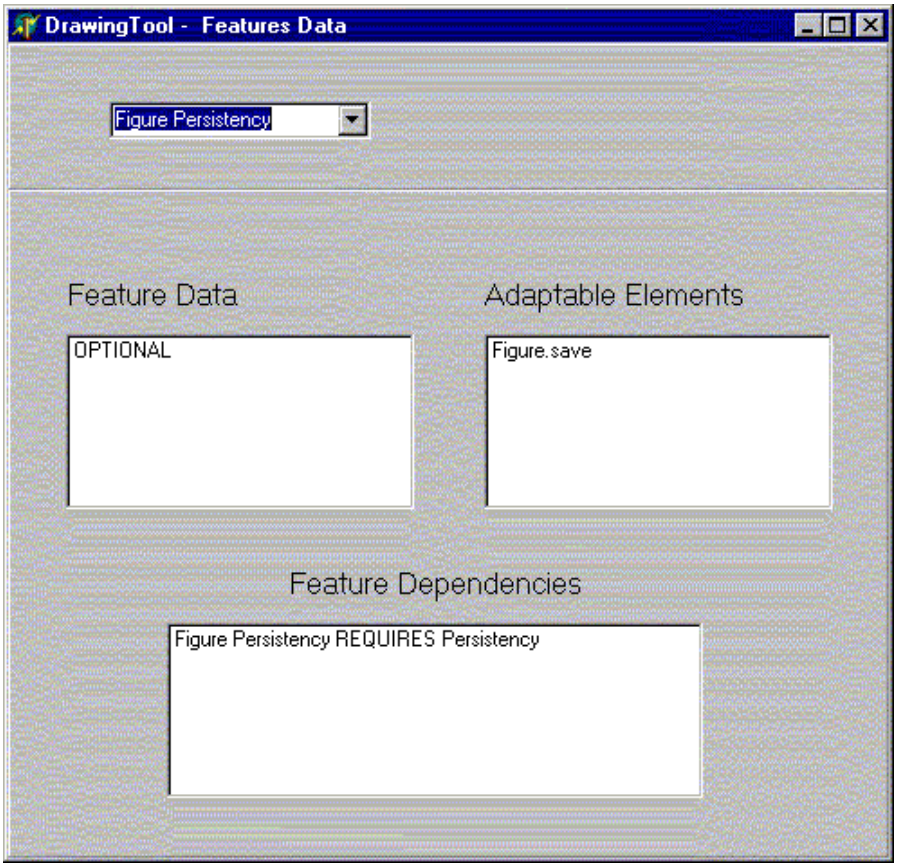


Figura 5.5 – O Inspetor de *Features* do Protótipo de Sagan *Tool*.

A partir do Inspetor de *Features* acima é possível saber que a escolha da *Feature* **Figure Persistency** requer a escolha da *Feature* **Persistency** para que seja produzida uma instância válida do Modelo de *Features*. Além disso, podemos ver que **Figure Persistency** é uma *Feature* opcional e que a sua escolha implica na adaptação do método **save** da classe **Figure**.

Em virtude das mesmas limitações apresentadas acima, foi incorporado ao protótipo de Sagan *Tool* um Inspetor de Classes (Figura 5.6). Desta maneira é possível obter informações sobre os Elementos Adaptáveis que compõem o design de um dado framework.

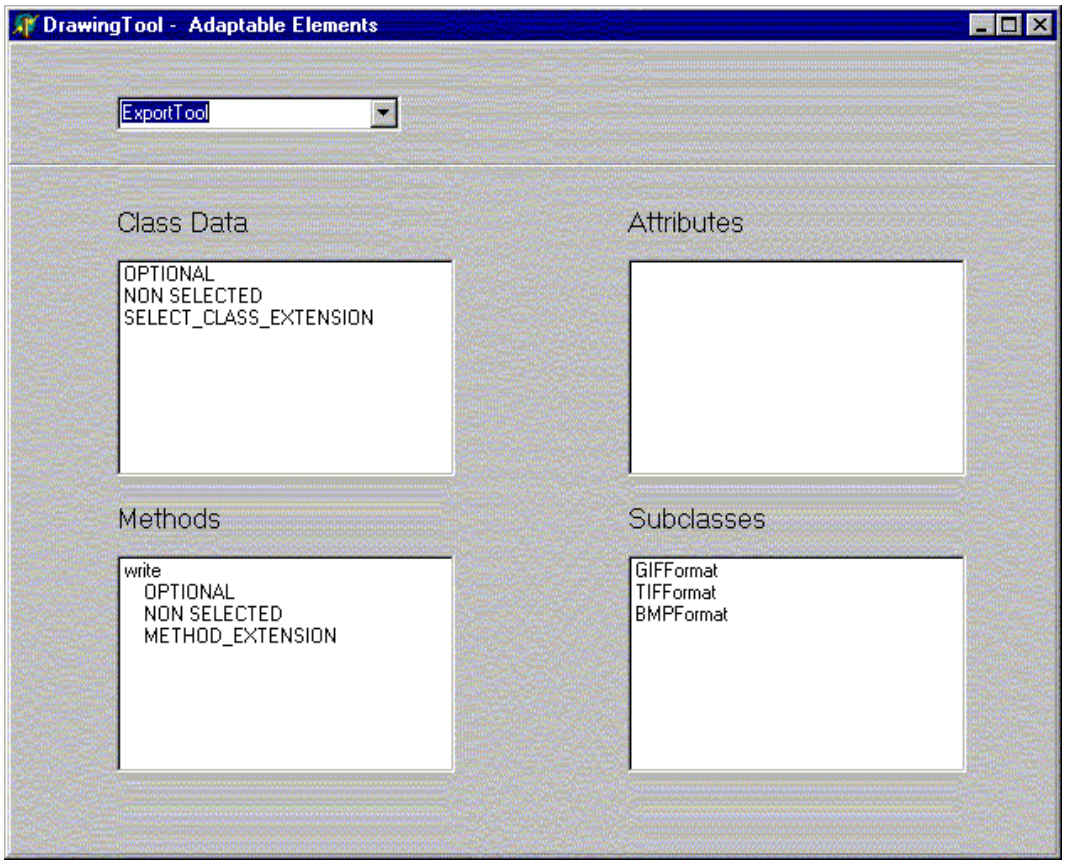


Figura 5.6 – O Inspetor de Classes do Protótipo de Sagan *Tool*.

5.3 A Geração de um *Script* de Instanciação em RDL

Nesta seção iremos descrever os passos necessários para gerar um *script* de instanciação em RDL a partir das escolhas feitas sobre o Modelo de *Features* de um framework.

O primeiro procedimento é verificar se as escolhas feitas irão produzir uma instância válida do Modelo de *Features*. Isto é executado ao se selecionar o botão **Check Model** na interface de apresentação do Diagrama de *Features* (Figura 5.4). Caso existam inconsistências nas escolhas, mensagens adequadas serão exibidas em um outro elemento da interface (Figura 5.7). Por outro lado, se a instância produzida for válida o botão **Generate RDL** (Figura 5.4) será disponibilizado, permitindo assim que um *script* RDL possa ser produzido a partir de uma varredura nos modelos de *Features*

e de Design do framework. O último passo necessário para a produção do *script* RDL é selecionar o botão **Generate RDL**.

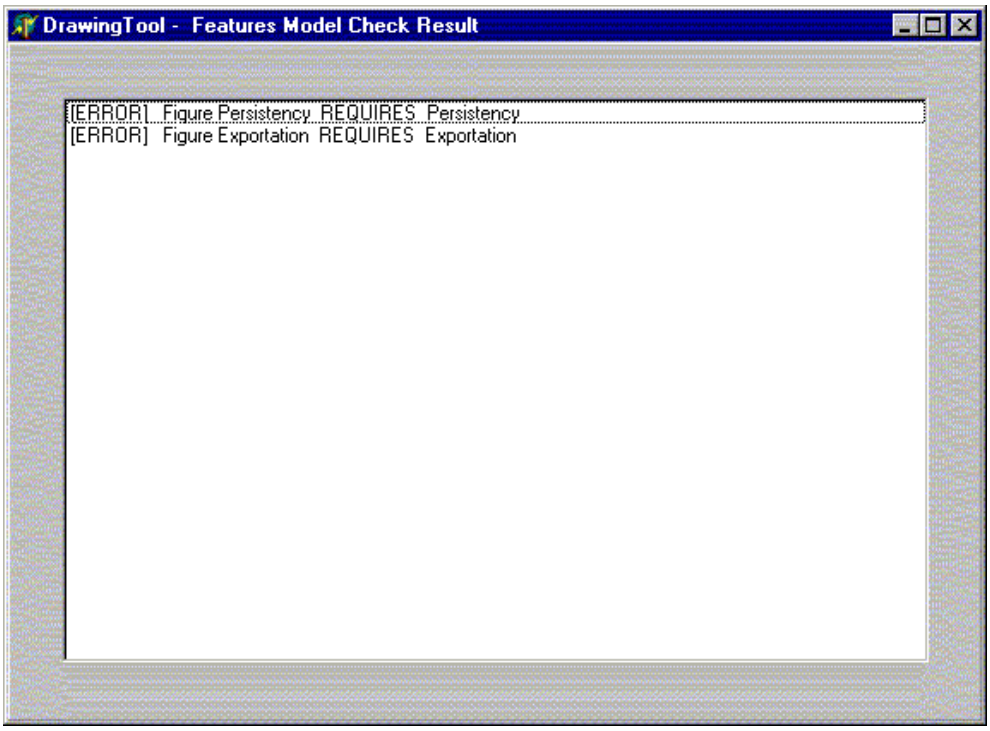


Figura 5.7 – Resultado da Validação de uma Instância de um Modelo de *Features*.

A geração do *script* de instanciação é controlada pelo subsistema principal de Sagan *Tool* (*Main*). Ele envia a mensagem **selectElements** para todas as *Features* selecionadas, que por sua vez enviam a mensagem **select** para cada um dos Elementos Adaptáveis (classes, métodos e atributos) associados a cada uma das *Features* selecionadas. Após a execução desta varredura todos os elementos de design envolvidos terão sido selecionados, eliminando-se assim a questão da Dependência de Existência (vide seção 4.5).

Por último, o subsistema principal envia a mensagem **generateRDL** para cada uma das classes selecionadas, e estas enviam a mesma mensagem para cada uma dos seus métodos selecionados.

A geração dos comandos RDL para a adaptação de atributos (extensões dos tipos *valueAssignment* e *valueSelection*) é um pouco diferente. Como todos os atributos serão adaptados conjuntamente, eles são colocados em uma lista durante a execução do método **generateRDL**. Isto é, cada uma das classes que recebe esta mensagem insere em uma lista, passada como parâmetro, os seus atributos que deverão ser adaptados. Após a geração do código relativo às classes e aos métodos a lista de atributos será percorrida, sendo então gerados os comandos RDL necessários à adaptação dos atributos.

A Figura 5.8 mostra o código gerado para a instância do Modelo de *Features* mostrada na Figura 4.15.

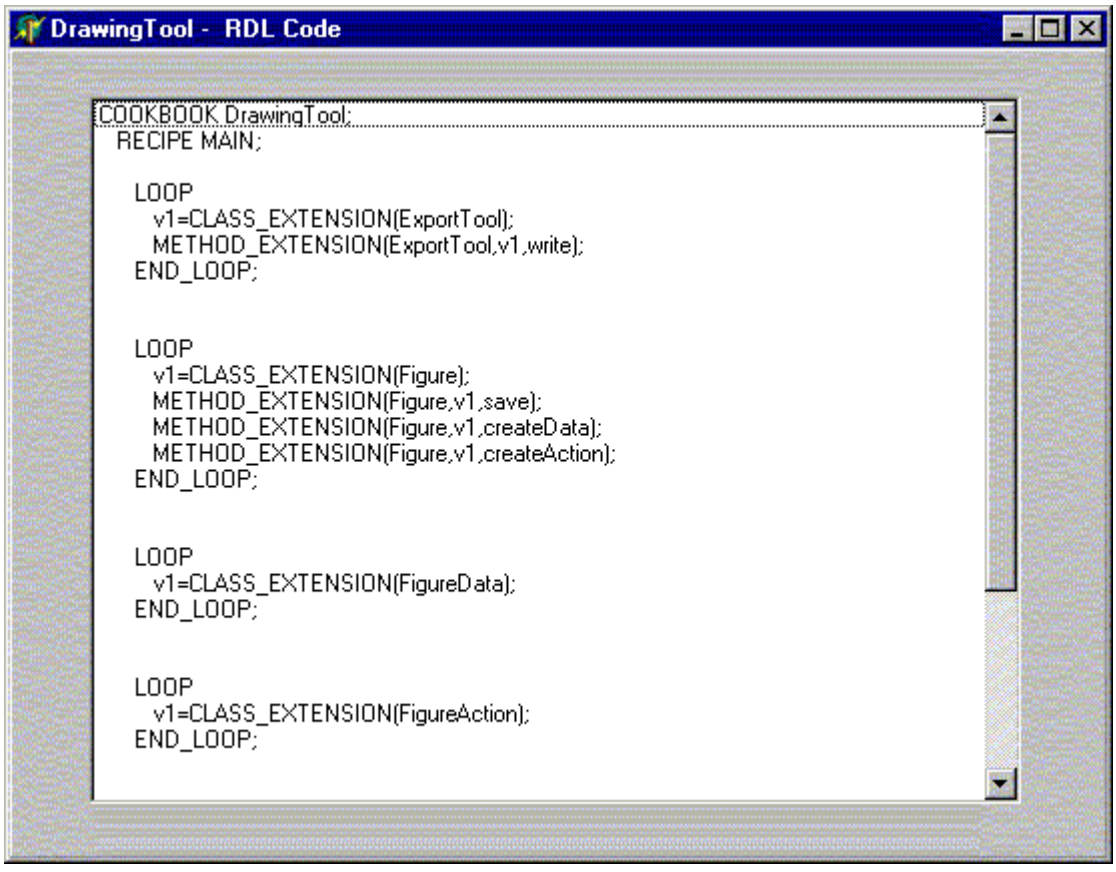


Figura 5.8 – *Script* RDL Gerado a Partir de uma Instância de um Modelo de *Features*.

5.4 Considerações Finais

Este capítulo descreveu a arquitetura de uma ferramenta, chamada Sagan *Tool*, capaz de implementar os conceitos apresentados nesta dissertação sobre um processo de alto nível para a instanciação de frameworks. Esta ferramenta cobre todas as etapas do processo de instanciação aqui descrito; desde a elaboração de um Modelo de *Features* até a geração de um *script* que permita a instanciação do design de um framework a partir da seleção de algumas características que se deseja ver incluídas no design final da aplicação sendo instanciada.

O capítulo descreve também um protótipo da ferramenta Sagan *Tool*. Esta protótipo, apesar de algumas limitações, comprova a coerência das idéias aqui apresentadas. Neste ponto é importante destacar que a implementação do protótipo seguiu à risca as especificações apresentadas nos Capítulos 3 e 4 desta dissertação, desde as alterações feitas no meta-modelo da UML para atender às necessidades geradas pela incorporação do Modelo de *Features* (Capítulo 3), até a observação das regras para a integração do referido modelo com os elementos de design de um framework (Capítulo 4).

Capítulo 6

Estudos de Casos

Neste capítulo serão apresentados os resultados da aplicação do processo proposto nesta dissertação na instanciação de três frameworks distintos: o framework *CommercePipe* [Neto00], um framework para o domínio de Promoções de Venda, apresentado em [Dalebout99], e um framework para o planejamento de redes de telecomunicações, apresentado em [Messmer00]. Entretanto, é importante esclarecer que estes estudos de casos não têm por objetivo estabelecer comparações, qualitativas ou quantitativas, entre os métodos de instanciação de frameworks tradicionais, alguns deles vistos no Capítulo 2 desta dissertação, e o método de instanciação baseado no Modelo de *Features*, discutido nos dois capítulos anteriores. Dessa maneira, estes estudos de casos servirão apenas como provas de conceito das idéias apresentadas neste trabalho.

6.1 O Framework *CommercePipe*

O *CommercePipe* é um framework que se apresenta como uma solução tecnológica para a instanciação de aplicações para mercados virtuais C2B (*Consumer to Business*) na Internet. Ele se enquadra na categoria de leilões reversos, permitindo assim que os consumidores estabeleçam parâmetros de negociação, tais como o preço e a forma de pagamento, que devem ser atendidos pelos vendedores para que uma negociação seja consumada. Além disso, o *CommercePipe* apresenta duas importantes características:

a. Permite a realização de transações comerciais através de dispositivos móveis que utilizem o protocolo de aplicação WAP (*Wireless Application Protocol*) [Arehart00].

b. Utiliza processos de negociação automáticos e semi-automáticos baseados na utilização de agentes de software [Bradshaw97] como mediadores do processo de negociação.

Em relação à arquitetura (Figura 6.1), o *CommercePipe* pode ser subdividido em dois grandes subsistemas: o *frontend* e o *backend*.

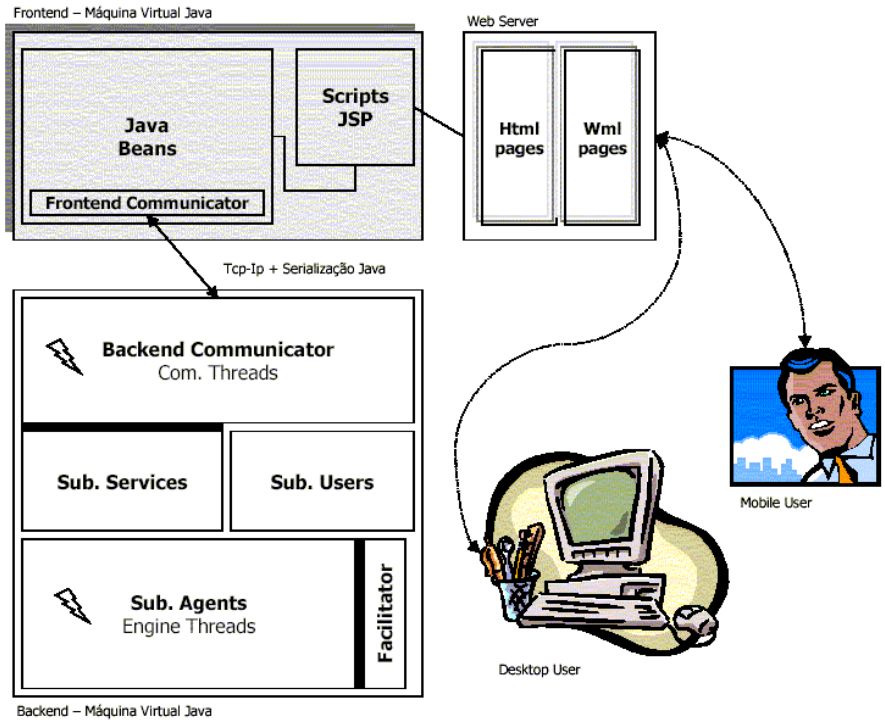


Figura 6.1 – A Arquitetura do Framework *CommercePipe*.

O *frontend* é o componente responsável pelo acesso dos usuários às principais funcionalidades disponibilizadas por um mercado virtual. Para tal, o *CommercePipe* utiliza *scripts* JAVA (JSP) [Ayers99] para a geração dinâmica de interfaces baseadas em *Web Sites* (HTML) e *Wireless Sites* (WML) [Arehart00].

O *backend* implementa todos os subsistemas que compõem o mercado virtual no qual as transações são realizadas. Este componente, desenvolvido completamente na

linguagem Java [Horstmann99], é executado em uma máquina virtual distinta da utilizada pelo *frontend*, permitindo assim a distribuição de ambos os componentes por diferentes nós de uma rede de computadores. A comunicação entre o *frontend* e o *backend* é realizada por componentes JavaBeans, que enviam e recebem objetos *serializados*.

O *backend* é composto por cinco subsistemas, cujas arquiteturas são de grande importância para este estudo de caso, já que é lá que se encontram os principais Pontos de Adaptação do framework *CommercePipe*. Desta forma, passaremos a seguir à descrição das classes e dos Pontos de Adaptação destes subsistemas.

6.1.1 O Subsistema de Usuários

Neste subsistema são definidos os principais usuários de um mercado virtual C2B. Este tipo de mercado possui basicamente três tipos de usuários: consumidores, vendedores e empresas (Figura 6.2).

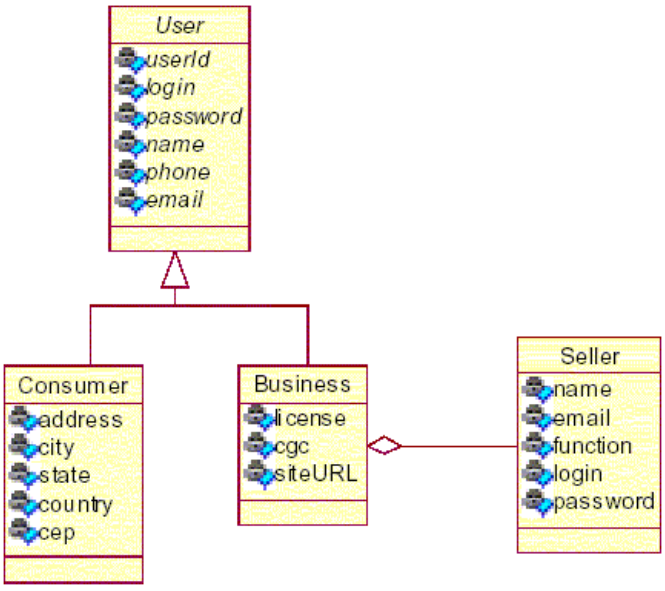


Figura 6.2 – Os Tipos de Usuários de Um Mercado Virtual C2B.

Embora uma instância do *CommercePipe* tenha que incluir obrigatoriamente estes três tipos de usuários, é possível acrescentar novas subclasses à classe *User*. Para tal é necessário que as novas subclasses implementem a interface *Trader* (Figura 6.3). Logo, a classe *User* define uma extensão do tipo *classExtension*, e o método *trade* define uma extensão do tipo *methodExtension*.

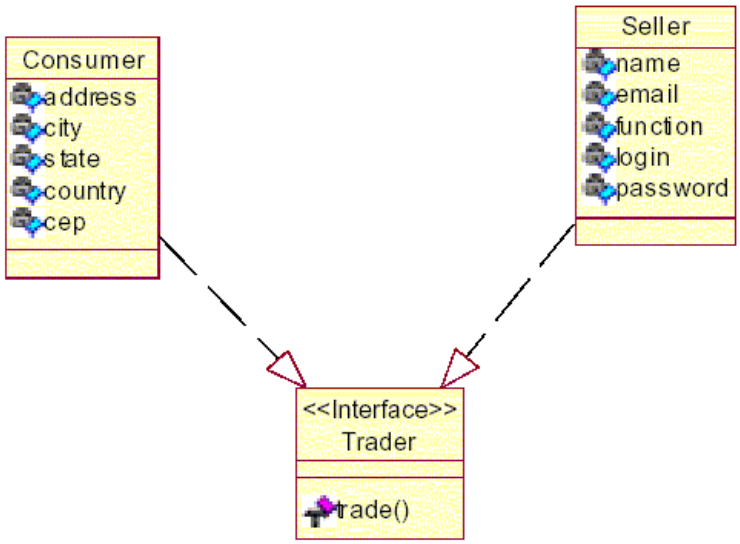


Figura 6.3 – A Interface *Trader*.

6.1.2 O Subsistema de Agentes

Este subsistema define os agentes responsáveis pela realização das negociações entre compradores e vendedores. Além disso, neste subsistema encontram-se as classes responsáveis pela definição do protocolo de negociação e dos tipos de itens (livros, carros, cd´s e etc) negociados em um mercado virtual. A Figura 6.4 apresenta uma visão panorâmica do Modelo de Classes do Subsistema de Agentes.

Um dos Pontos de Adaptação encontrados neste subsistema se refere aos estados dos agentes. Cada tipo de agente possui uma máquina de estados implementada através do *design pattern State* [Gamma95]. Desta forma, a adição de um novo estado dar-se-á através da inclusão de novas subclasses na hierarquia de generalização definida pela

classe abstrata *AgentState*. Logo, a classe *AgentState* define uma extensão do tipo *classExtension*.

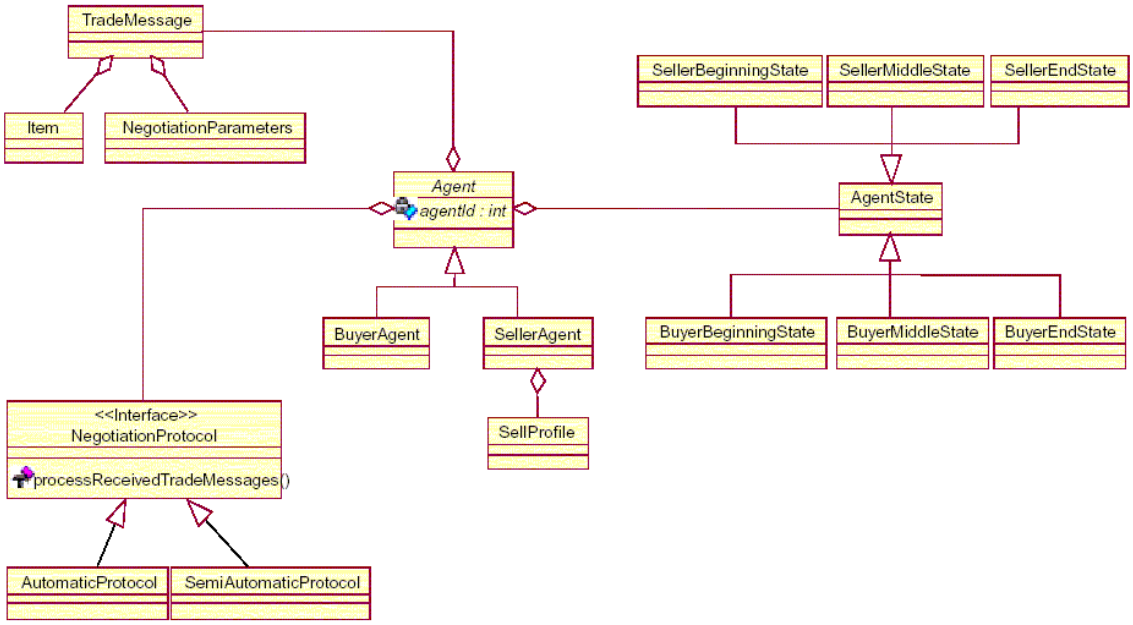


Figura 6.4 – O Modelo de Classes do Subsistema de Agentes.

A própria definição dos agentes é em si um Ponto de Adaptação; isto é, podemos adicionar novos tipos de agentes a um mercado virtual através da inserção de novas subclasses da classe abstrata *Agent* (*classExtension*). Além disso, é possível verificar, examinando-se a Figura 6.4, a existência de uma dependência entre os tipos de agentes existentes em um mercado virtual e os estados modelados pela classe *AgentState*. Esta dependência será capturada no Modelo de *Features* através da associação deste dois Pontos de Adaptação a uma mesma *Feature*.

O último Ponto de Adaptação existente no subsistema de agentes é o protocolo de negociação. O framework *CommercePipe* fornece dois tipos de protocolos pré-definidos: o protocolo automático, modelado pela classe *AutomaticProtocol*, e o protocolo semi-automático, modelado pela classe *SemiAutomaticProtocol*. Como apenas um dos dois protocolos pode ser incluído em uma instância do *CommercePipe*, ambas as classes definem Pontos de Adaptação opcionais.

6.1.3 O Subsistema de Facilitação

O Subsistema de Facilitação é responsável por alguns serviços de gerência de um mercado virtual. Ele é composto por três classes abstratas que definem três Pontos de Adaptação distintos.

O primeiro Ponto de Adaptação é definido pela classe *AgentsManager* (Figura 6.5), que é responsável pela identificação, pela remoção e pela criação de agentes em um mercado virtual. Para cada novo tipo de agente que for inserido em uma instância do *CommercePipe* será necessário criar uma nova subclasse de *AgentsManager* e redefinir o método *executeAgents*. Logo, a classe em questão define uma extensão do tipo *classExtension*, enquanto o método *executeAgents* define uma extensão do tipo *methodExtension*.

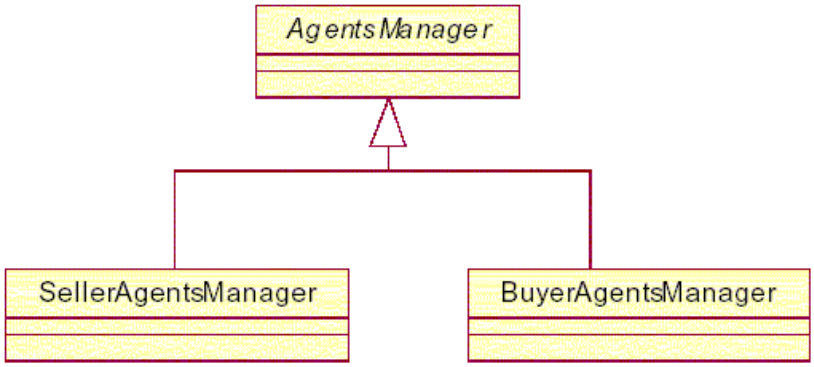


Figura 6.5 – O Ponto de Adaptação de Gerência dos Agentes.

Outro Ponto de Adaptação existente neste subsistema é o mecanismo de aproximação entre os compradores e os vendedores que potencialmente podem atender às necessidades definidas por um certo grupo de compradores. O mecanismo de aproximação é definido pela classe abstrata *FilterTechnique* (Figura 6.6). O *CommercePipe* fornece duas técnicas de filtragem pré-codificadas: *SimpleFilter* e *AdvancedFilter*. Embora apenas uma técnica de filtragem possa ser instanciada, é possível definir novas subclasses para que se implemente novas técnicas de filtragem.

Além disso, é necessário redefinir o método *filter*, que desta maneira constitui-se em uma extensão do tipo *methodExtension*.

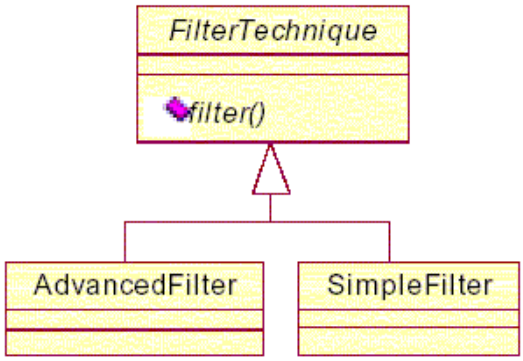


Figura 6.6 – As Técnicas de Filtragem do *CommercePipe*.

O terceiro Ponto de Adaptação deste subsistema é responsável pela implementação de diferentes técnicas de gerenciamento de transações. Através dele será possível evitar a ocorrência de estados inconsistentes pela ausência de controle na execução das *threads* (Figura 6.7).

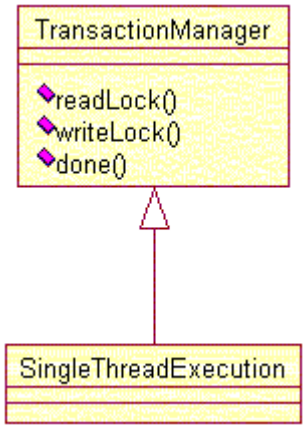


Figura 6.7 – As Técnicas de Gerência de Transações.

O Subsistema de Facilitação fornece um controle de transações *default*, implementado pela classe *SingleThreadExecution*. Novas técnicas de gerência de transações podem ser incluídas em um mercado virtual através da criação de novas subclasses da classe *TransactionManager*, e pela redefinição dos métodos *readLock*, *writeLock* e *done*. Um

outro detalhe importante é que apenas um método de gerência de transações poderá ser implementado em uma instância do *CommercePipe*.

6.1.4 O Subsistema de Serviços

O Subsistema de Serviços é responsável pela implementação dos diversos serviços que podem ser disponibilizados em um mercado virtual. Ele é composto por uma única classe abstrata (Figura 6.8) e por várias classes concretas, cada uma implementando um serviço específico.

Novos serviços podem ser incorporados ao mercado virtual inserindo-se novas subclasses de *Service*, e redefinindo-se o método *doService*. Outro detalhe importante é que a adaptação deste *hot-spot* irá repercutir fortemente nas interfaces com os usuários finais, já que é através destas que os atores de um mercado virtual selecionam os serviços desejados.

Figura 6.8 – A Classe Abstrata *Service*.

6.1.5 O Subsistema de Comunicação

O último subsistema do *CommercePipe*, o Subsistema de Comunicação, é responsável pela comunicação entre o *frontend* e o *backend* de um mercado virtual. Praticamente não existem Pontos de Adaptação neste subsistema. Apenas os aspectos relativos ao número máximo de *threads* em execução e ao número máximo de conexões ativas podem ser configurados. Tais parâmetros são mantidos pelos atributos *maxThreadNumber* e *maxConnectionNumber*, ambos definidos na classe

BackEndCommunicator. Desta maneira, tais atributos definem extensões do tipo *valueAssignment*.

6.1.6 O Modelo de *Features* do *CommercePipe*

A partir da descrição dos subsistemas do *CommercePipe*, feita nas seções anteriores, é possível agora construir um Modelo de *Features* que descreva as características presentes neste framework. A primeira etapa da elaboração de tal modelo é a construção de um Diagrama de *Features*, que pode ser visto na Figura 6.9.

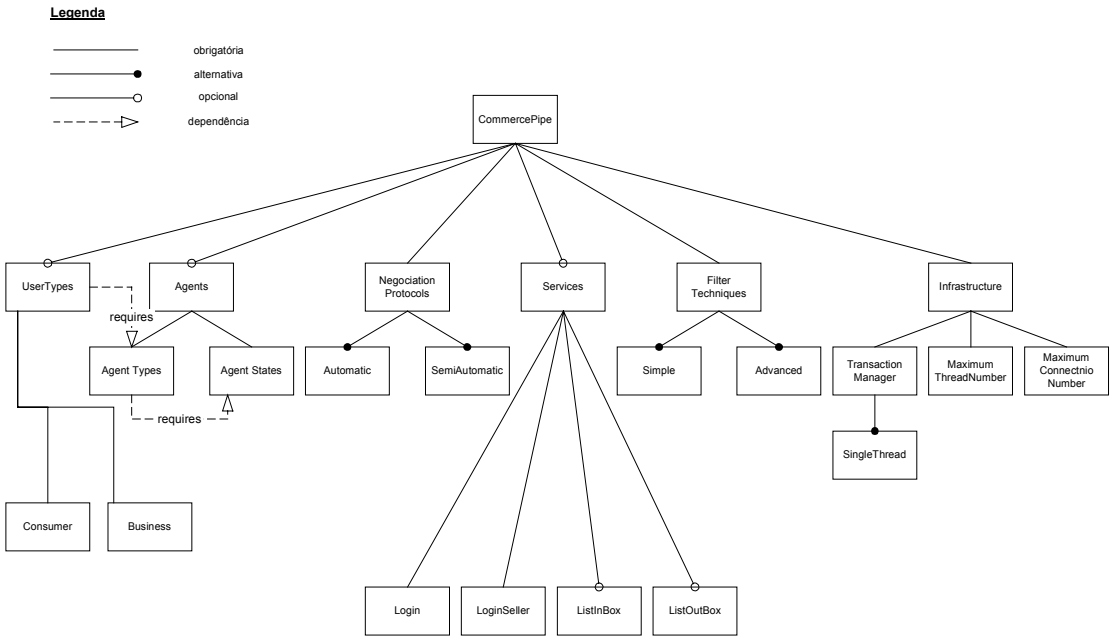


Figura 6.9 – O Diagrama de *Features* do Framework *CommercePipe*.

A próxima etapa da construção do Modelo de *Features* é estabelecer os relacionamentos entre as *Features* e os elementos de design do *CommercePipe* que são responsáveis pela implementação de tais *Features* (Tabela 6.1).

Relação das Dependências Entre as *Features* e os Pontos de Adaptação do Framework *CommercePipe*	
Features	Ponto de Adaptação
User Types	User
	User.trade
Consumer	
Business	
Agents	
Agent Type	Agent
	AgentsManager
	AgentsManager.executeAgents
Agent States	AgentState
	AgentState.execute
Negotiation Protocols	NegotiationProtocol
Automatic	AutomaticProtocol
SemiAutomatic	SemiAutomaticProtocol
Services	Service
	Service.doService
Login	
Login Seller	
List In-Box	ListInBox
List Out-Box	LIstOutBox
Filter Techniques	FilterTechnique
	FilterTechnique.filter
Simple	SimpleFilter
Advanced	AdvancedFilter
Infrastructure	
Transaction Manager	TransactionManager
	TransactionManager.readLock
	TransactionManager.writeLock
	TransactionManager.done
Single Thread	SingleThreadExecution
Maximum Thread Number	BackEndCommunicator.maxThreadNumber
Maximum Connection Number	BackEndCommunicator.maxConnectionNumber

Tabela 6.1 - Relação das Dependências Entre as *Features* e os Pontos de Adaptação do Framework *CommercePipe*

É possível notar, na Tabela 6.1, que algumas *Features* não possuem Pontos de Adaptação a elas relacionados. No caso das *Features* **Agents** e **Infrastructure** isto se deve ao fato de elas estarem ali apenas para agregar funcionalidades. Por exemplo, a *Feature* **Agents** foi inserida no modelo para agregar as *Features* **Agent Types** e **Agent States**. Desta forma fica claro quais características devem ser adaptadas para que os agentes possam ser devidamente configurados.

Em relação às *Features* **Consumer**, **Business**, **Login** e **Login Seller** a razão pela qual não existem Pontos de Adaptação a elas relacionados é que estas *Features* representam aspectos obrigatórios (*frozen-spots*) do *CommercePipe*. Desta maneira, não é necessário associá-las com os elementos de design que as implementam. Tais *Features* só foram incluídas no diagrama para que ficasse documentado quais os tipos de usuários que são obrigatórios em um mercado virtual instanciado a partir do *CommercePipe*.

6.1.7 A Instanciação de Um Mercado Virtual Para a Compra e Venda de Automóveis

O nosso objetivo agora é instanciar, a partir do *CommercePipe*, um mercado virtual onde se possa comprar e vender automóveis. Para tal, é necessário que o processo de negociação seja semi-automático, uma vez que os valores dos bens negociados podem ser elevados. Além disso é provável que ocorra uma intensa negociação em torno do preço de compra e venda, que certamente irá variar de acordo com o estado do automóvel, no caso de ele ser usado, e com as condições de pagamento e financiamento.

Outro aspecto deste mercado virtual é que o critério de filtragem tem que ser mais específico. Isto ocorre porque devemos levar em consideração vários critérios na aproximação de vendedores e compradores. Entre eles podemos citar o ano, a marca, a cor e o preço do automóvel a ser negociado.

A Figura 6.10 mostra as seleções realizadas sobre o Diagrama de *Features* para a construção de um mercado virtual com estas características.

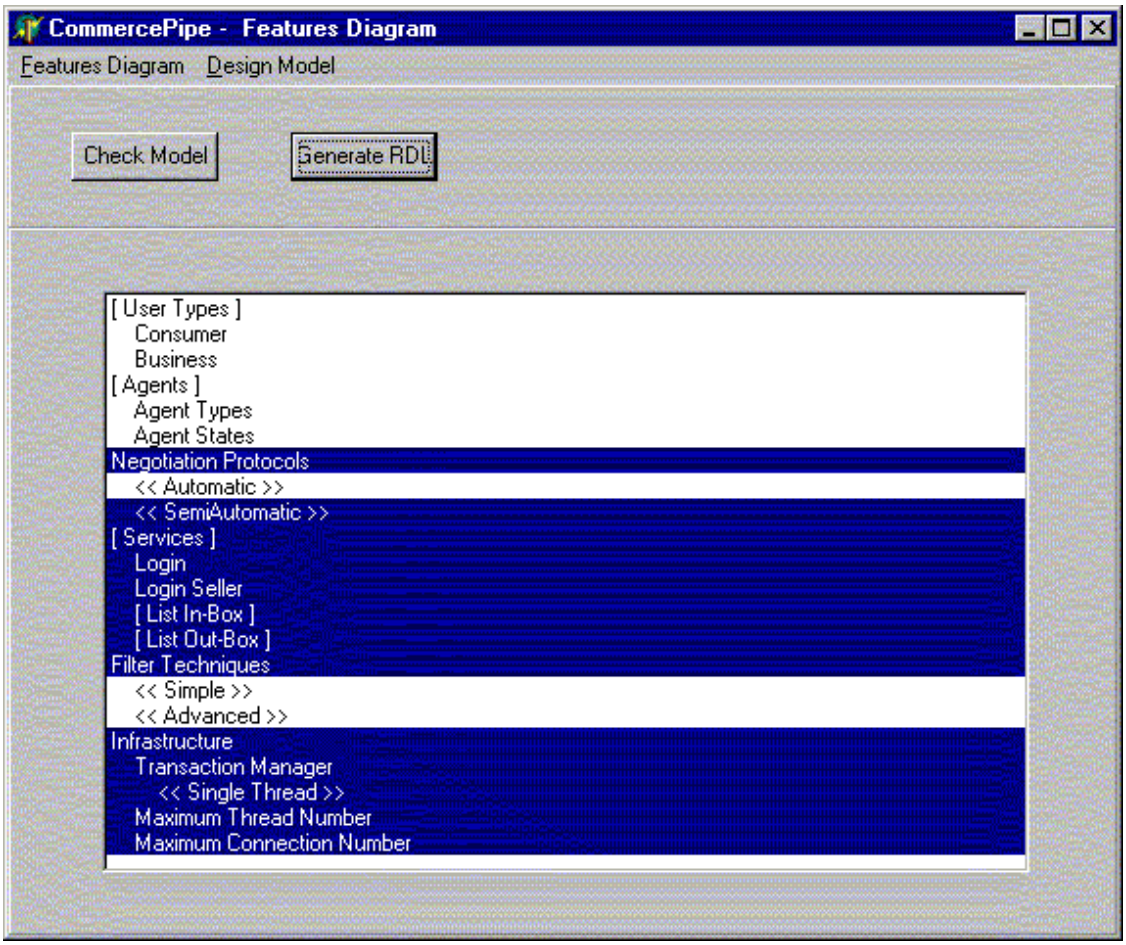


Figura 6.10 – As *Features* de um Mercado Virtual para Comercializar Automóveis.

A partir das escolhas mostradas na Figura 6.10 a ferramenta Sagan *Tool* irá produzir um script RDL para que o design do *CommercePipe* possa ser adaptado de acordo com as necessidades de um mercado virtual para a compra e venda de automóveis. A Figura 6.11 mostra o script RDL gerado.

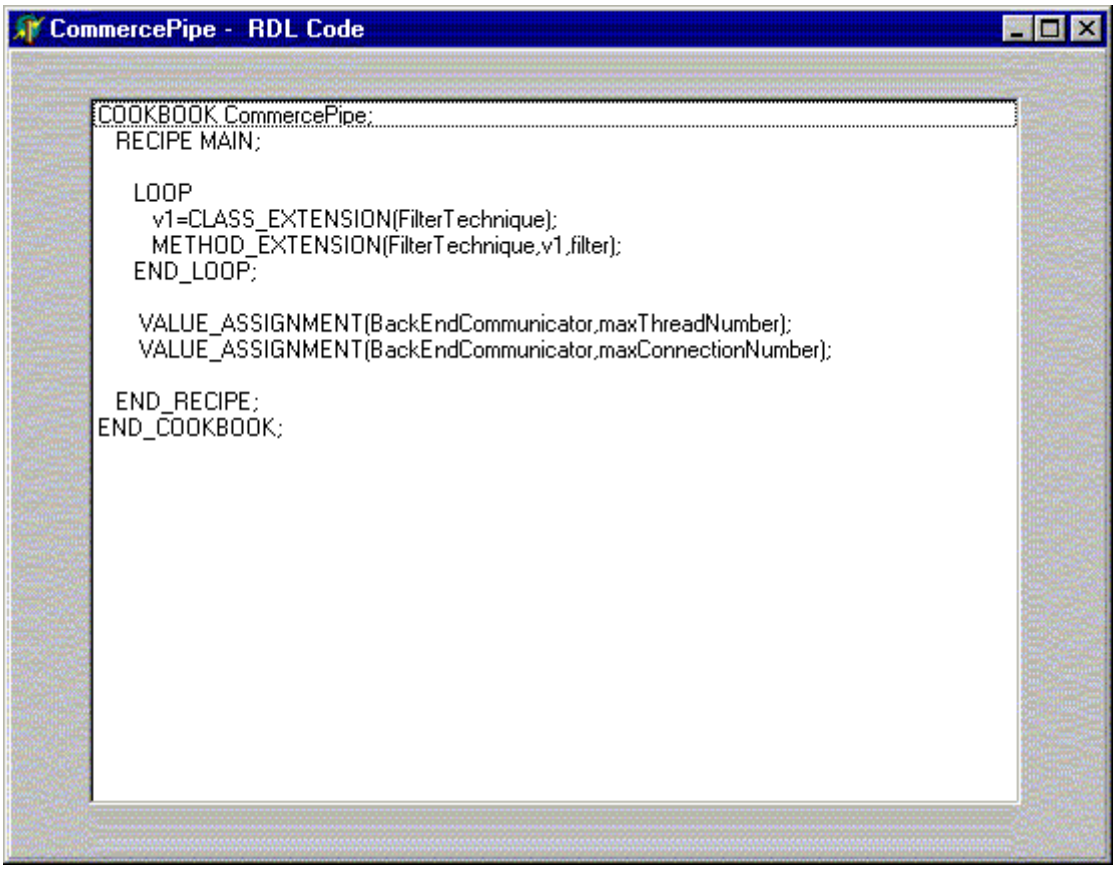


Figura 6.11 – O Script RDL Gerado a Partir das *Features* Selecionadas.

6.1.8 Considerações Finais

De um modo geral não houve maiores dificuldades em construir mercados virtuais C2B a partir da aplicação, sobre o framework *CommercePipe*, do método de instanciação de frameworks aqui proposto. Entretanto, dois importantes aspectos da adaptação ficaram de fora devido às limitações do processo em questão.

Em primeiro lugar, as interfaces do mercado virtual com os usuários finais não puderam ser diretamente adaptadas pelo fato de estarem escritas em HTML e WML. Como estas linguagens não são orientadas a objetos, não foi possível aplicar o processo de instanciação em questão ao Subsistema de *Frontend* do *CommercePipe*.

O segundo aspecto que ficou de fora foi a inclusão de um novo item a ser comercializado; neste caso, um automóvel. A adaptação deste *hot-spot* não é feita,

como se poderia esperar, a partir da criação de novas subclasses da classe **Item** (Figura 6.12). Para tornar o *CommercePipe* o mais flexível possível optou-se por armazenar os atributos de um item em uma *Hash Table* definida como um atributo (**attributes**) da classe **Item**. Desta forma, a adaptação deste *hot-spot* é feita diretamente sobre o código fonte da aplicação, sem que haja necessidade de alteração no design do framework.

Figura 6.12 – A Classe Item.

6.2 Um Framework para o Domínio de Promoção de Vendas

Como a maioria das áreas ligadas aos negócios, o setor de marketing é cada vez mais dependente da disponibilidade de informações precisas no momento adequado. Tais informações são fundamentais para que a tomada de decisões no dia-a-dia dos negócios esteja calcada em uma sólida base de informações. Embora exista uma grande quantidade de produtos de software no mercado para o auxílio à tomada de decisões, a maioria deles apresenta limitações quanto à capacidade de serem adaptados às necessidades específicas dos profissionais responsáveis pelas decisões a serem tomadas no desenvolvimento dos seus negócios.

Um destes profissionais, o Gerente de Produtos, é responsável pela fixação de preços, distribuição, propaganda e promoção de vendas de um ou mais produtos. Para tal, um Gerente de Produtos precisa de aplicativos que se encaixem nas suas concepções do

domínio de Promoções de Vendas. Tais concepções consistem de elementos típicos da área de marketing, como os conceitos de marcas, preço, vendas, propaganda e segmentos de mercado, além dos atores típicos, como os clientes, os competidores, os produtores e os revendedores.

O que veremos a seguir é a descrição [Dalebout99] de um framework projetado para auxiliar a tomada de decisões no domínio de Promoções de Vendas. Em seguida será mostrada a aplicação do método proposto nesta dissertação na instanciação de aplicações para este domínio.

6.2.1 A Arquitetura do Framework

Devido à complexidade do domínio de Promoções de Vendas o framework em questão foi organizado em vários subsistemas, cada um dos quais responsável pela implementação de um conceito típico do domínio. A Figura 6.13 mostra os subsistemas do framework em questão e as relações de dependências existentes entre eles.

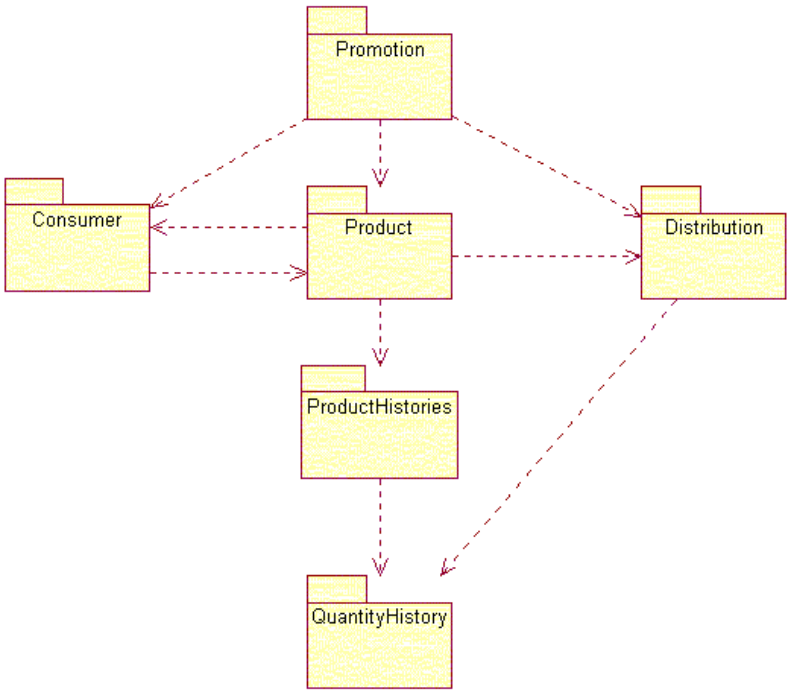


Figura 6.13 – Os Subsistemas do Framework de Promoção de Vendas.

Apenas alguns dos subsistemas do framework de Promoção de Vendas serão abordados nesta seção. Isto se deve ao fato de os principais Pontos de Adaptação estarem localizados nestes subsistemas.

O subsistema **Consumer** (Figura 6.14) é responsável pela implementação de um dos principais conceitos no domínio de Promoção de Vendas. É nele que serão representadas as informações pertinentes aos consumidores e aos seus respectivos históricos de compras.

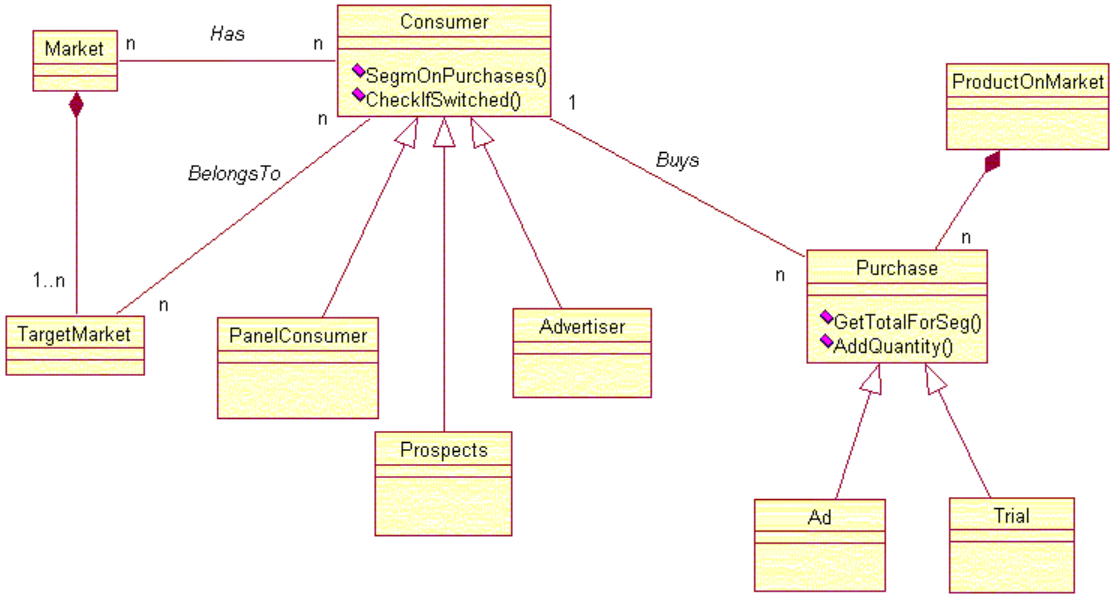


Figura 6.14 – O Subsistema *Consumer*.

Neste subsistema encontramos dois Pontos de Adaptação. O primeiro deles, definido pela classe abstrata **Consumer**, permite a inclusão de novos tipos de consumidores, enquanto o segundo, definido pela classe abstrata **Purchase**, permite representar os diversos tipos de compras que um consumidor poderá realizar. Em ambos os casos são fornecidas subclasses concretas que representam componentes pré-definidos.

O subsistema **Promotion** (Figura 6.15) mostra os dois tipos básicos de promoção encontrados no framework em questão: a promoção baseada no preço e a promoção baseada na fidelidade do consumidor. Ambos os tipos de promoção definem Pontos de

Adaptação através do mecanismo de classes abstratas. Tais Pontos de Adaptação deverão ser especializadas para que as necessidades específicas de um determinado negócio possam ser atendidas. Em ambos os casos são fornecidos componentes pré-definidos através de subclasses concretas de ambas as classes.

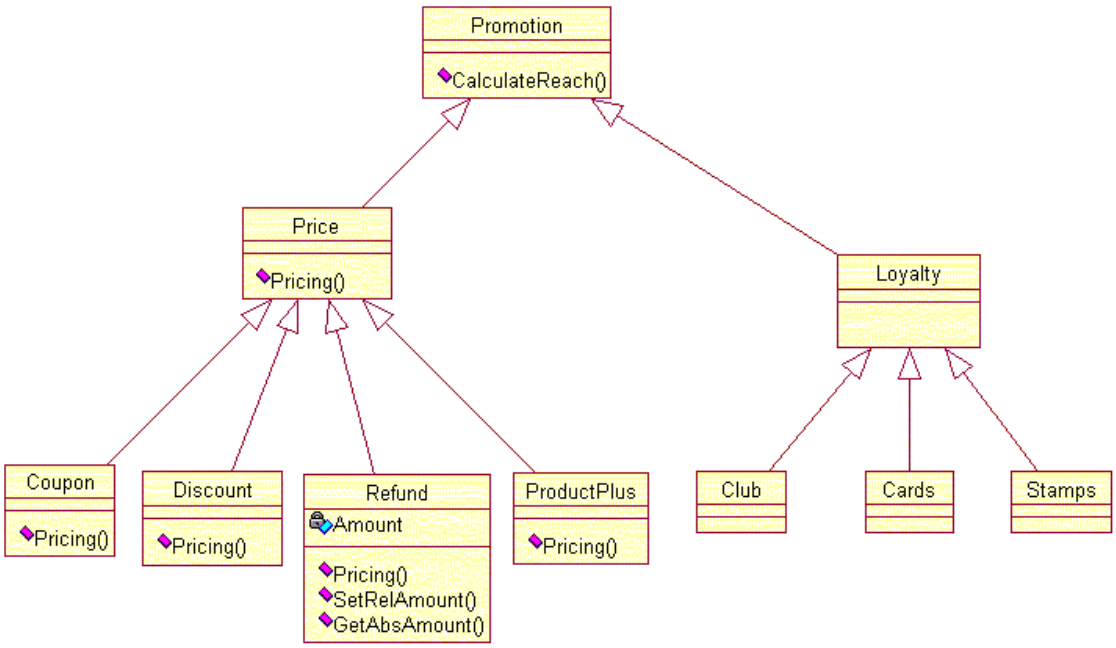


Figura 6.15 – O Subsistema *Promotion*.

O último subsistema a ser abordado, o **QuantityHistory**, permite a representação de uma variedade de dados que devem ser armazenados através do tempo para que possam ser usados nas projeções de promoções futuras. Os dois Pontos de Adaptação existentes neste subsistema tratam da representação de diferentes quantidades (classe **Quantity**) e dos mecanismos de conversão entre tais quantidades (classe **UnitConverter**). Por exemplo, é possível converter dados históricos de vendas realizadas em dólares americanos para os seus respectivos valores em Real.

A Figura 6.16 mostra as classes que compõem o subsistema **QuantityHistory** juntamente com as subclasses que fornecem implementações pré-definidas para os Pontos de Adaptação supracitados.

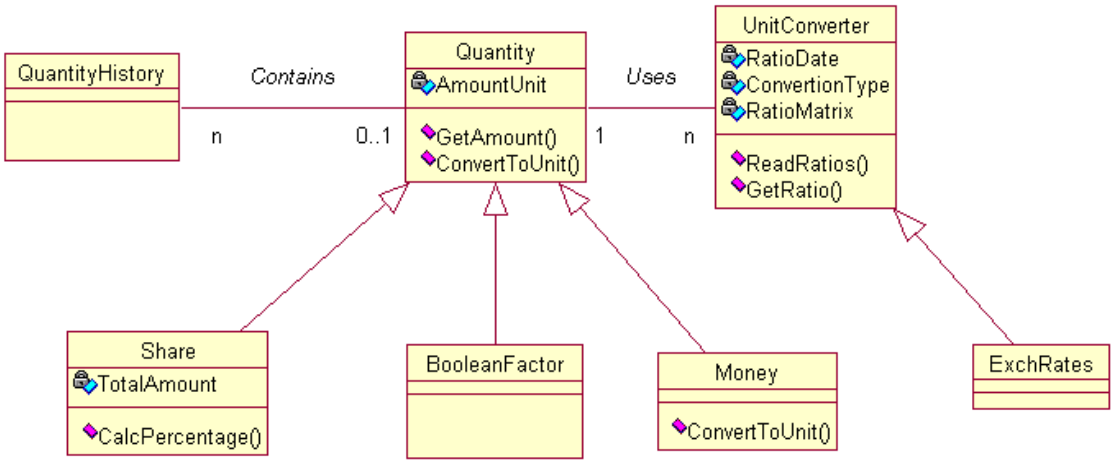


Figura 6.16 – O Subsistema *QuantityHistory*.

6.2.2 O Modelo de *Features* do Framework de Promoção de Vendas

A partir da descrição dos subsistemas do framework de Promoção de Venda é possível construir um Modelo de *Features* que descreva as características presentes neste domínio de aplicação. A primeira etapa da elaboração de tal modelo é a construção de um Diagrama de *Features*, que pode ser visto na Figura 6.17.

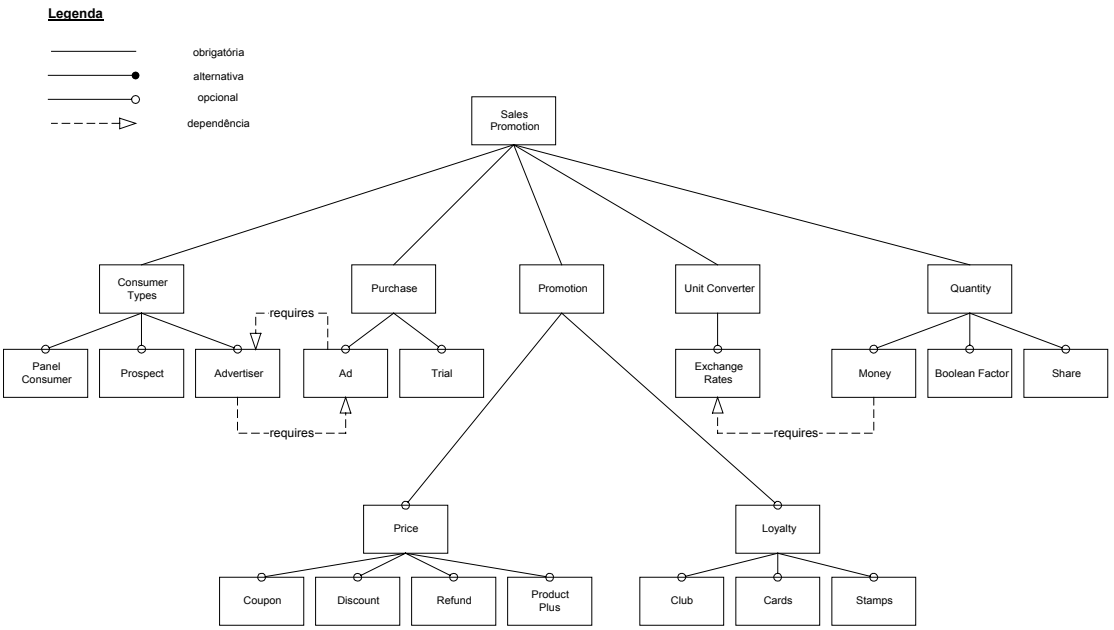


Figura 6.17 – O Diagrama de *Features* do Framework de Promoção de Vendas.

A próxima etapa da construção do Modelo de *Features* é estabelecer os relacionamentos entre as *Features* e os elementos de design que são responsáveis pela implementação de tais *Features* (Tabela 6.2).

Relação das Dependências Entre as *Features* e os Pontos de Adaptação do Framework de Promoção de Vendas

Features	Ponto de Adaptação
Consumer Types	Consumer
	Consumer.SegmOnPurchases
	Consumer.CheckIfSwitched
Panel Consumer	PanelConsumer
Prospect	Prospects
Advertiser	Advertiser
Purchase	Purchase
	Purchase.GetTotalForSeg
	Purchase.AddQuantity
Ad	Ad
Trial	Trial
Promotion	Promotion
	Promotion.CalculateReach
Price	Price
	Price.Pricing
Coupon	Coupon
Discount	Discount
Refund	Refund
Product Plus	ProductPlus
Loyalty	Loyalty
Club	Club
Cards	Cards
Stamps	Stamps
Unit Converter	UnitConverter
	UnitConverter.ReadRatios
	UnitConverter.GetRatio
Exchange Rates	ExchRates

Quantity	Quantity
	Quantity.GetAmount
	Quantity.ConvertToUnit
Share	Share
Boolean Factor	BooleanFactor
Money	Money

Tabela 6.2 - Relação das Dependências Entre as *Features* e os Pontos de Adaptação do Framework de Promoção de Vendas.

6.2.3 A Instanciação do Framework de Promoção de Vendas

A Figura 6.18 mostra as seleções realizadas sobre o Diagrama de *Features* para a construção de uma aplicação específica de Promoção de Vendas.

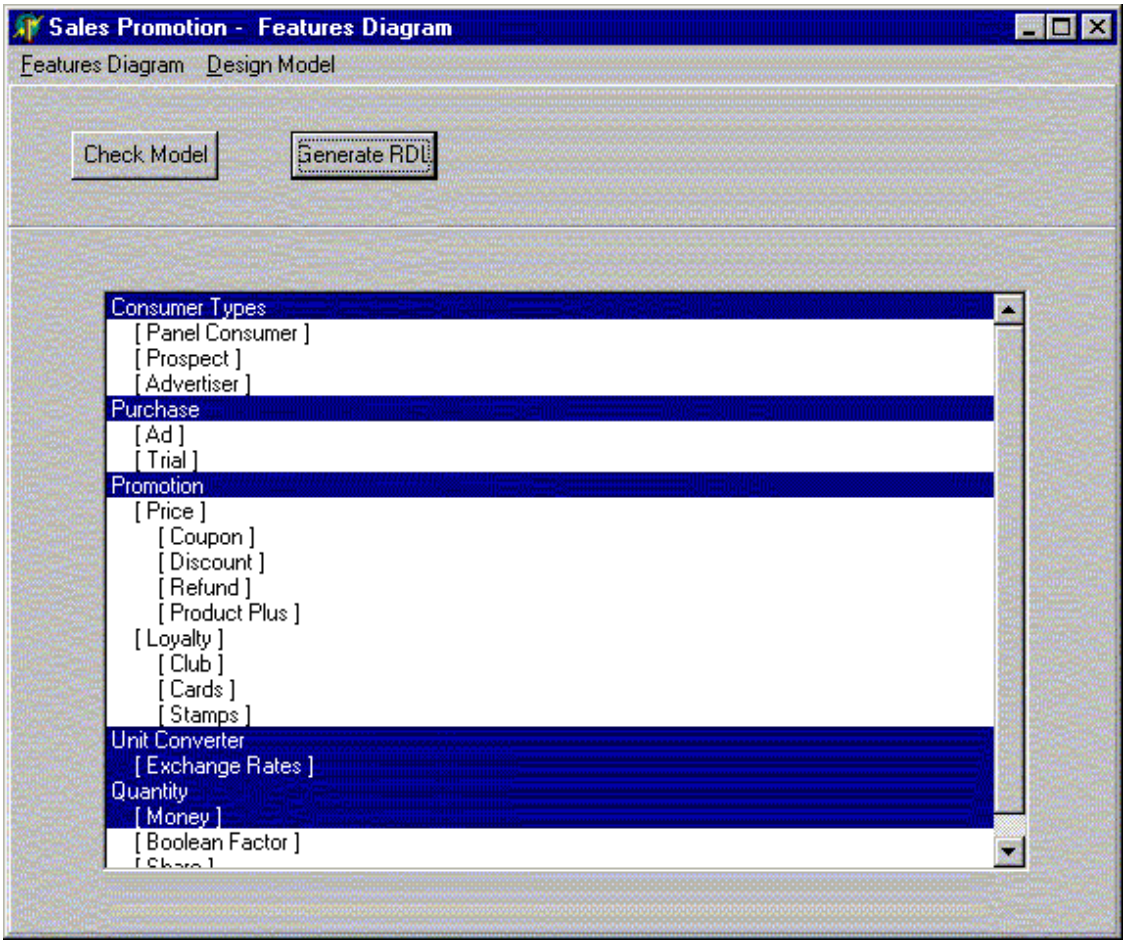


Figura 6.18 – As *Features* de uma Instância do Framework de Promoção de Vendas.

A partir das escolhas mostradas na Figura 6.18 a ferramenta Sagan *Tool* irá produzir um script RDL para que o design do framework de Promoção de Vendas possa ser adaptado. A Figura 6.19 mostra o script RDL gerado.

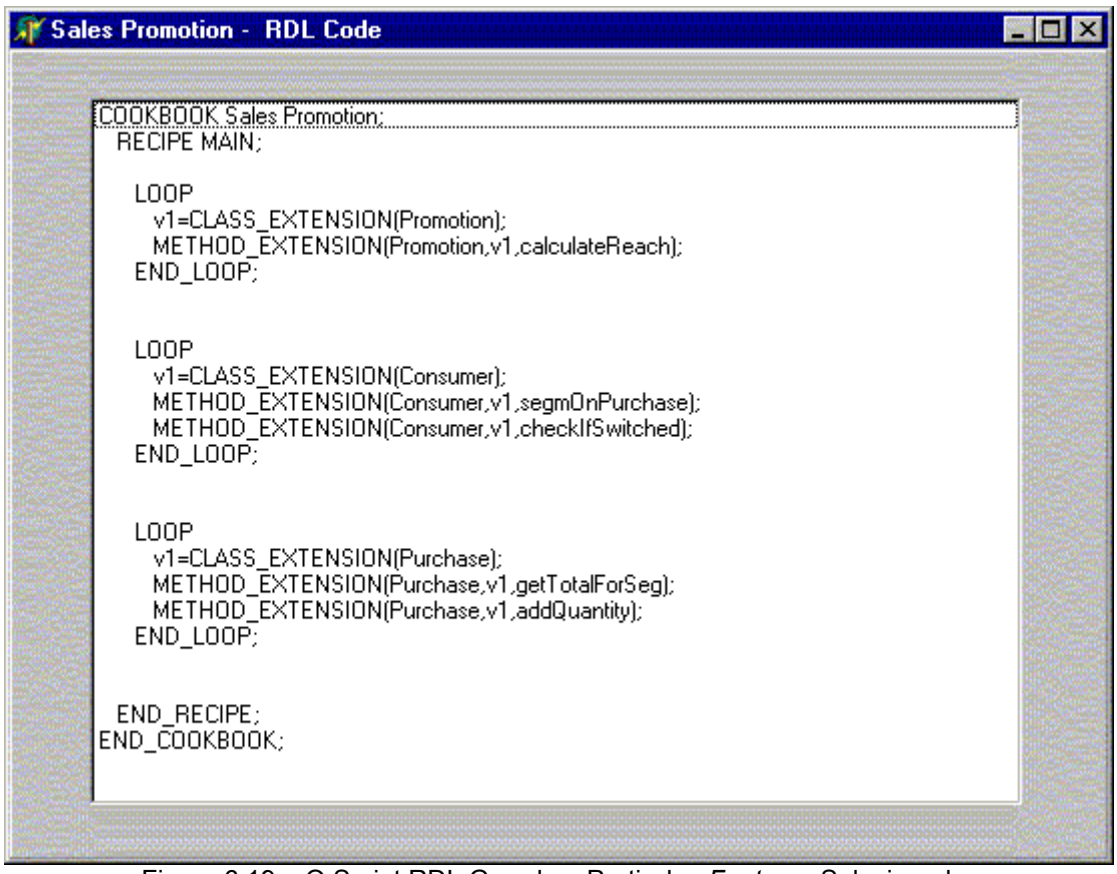


Figura 6.19 – O Script RDL Gerado a Partir das *Features* Selecionadas.

Analisando-se a Figura 6.19 é possível notar que será necessário acrescentar algumas classes, e redefinir alguns métodos, no design do framework em questão. Isto se deve ao fato de não terem sido escolhidas classes pré-definidas para implementar as *Features* **Promotion**, **Consumer Types** e **Purchase**.

6.2.4 Considerações Finais

Por se tratar de um framework eminentemente *white-box*, o processo de instanciação do framework de Promoção de Vendas, descrito nas seções anteriores, deverá ser complementado com a inclusão de código específico nos Pontos de Adaptação associados aos *métodos de extensão*. Isto ocorre devido à natureza pouco estruturada

do processo de tomada de decisão na área de Promoção de Vendas, causada principalmente pela relativa imaturidade do domínio em questão. Em [Dalebout99] é sugerido inclusive o uso de algumas ferramentas CASE que permitem a inclusão de código diretamente através da própria ferramenta, dispensando assim a integração com ambientes de desenvolvimento específicos. Outra sugestão apresentada na mesma referência é a utilização de pseudocódigo na implementação do código de uma aplicação específica. Desta forma seria possível produzir aplicações escritas em diferentes linguagens de programação a partir de um único conjunto de modelos que descreva um dado framework.

6.3 O Framework NETPLAN

Devido à crescente competição na área de telecomunicações, resultante do processo de liberalização dos negócios em escala mundial neste ramo de atividades, as redes de telecomunicações têm que ser projetadas levando-se em consideração tanto os custos de instalação como também a capacidade requerida. Desta forma, é fundamental que o desempenho dos novos protocolos sejam aferidos. Isto permitirá fornecer aos usuários de tais redes, serviços que correspondam às suas necessidades.

Embora existam no mercado numerosas ferramentas para o planejamento, o projeto e a otimização de redes de telecomunicações, a grande maioria não permite que novos algoritmos de otimização e diferentes métodos de análise possam ser facilmente incorporados às aplicações já existentes. Diante desta situação, a Swisscom, operadora de telecomunicação com base na Suíça, decidiu construir o framework NETPLAN com o objetivo de agilizar a construção de novas aplicações para o planejamento, construção e otimização de redes de telecomunicações.

Nas seções seguintes serão apresentadas as principais características do framework em questão [Messmer00], além de uma descrição detalhada dos seus principais componentes. Por último, será apresentado um exemplo de instanciação deste framework baseado no processo proposto por esta tese.

6.3.1 A Organização do Framework NETPLAN

A Figura 6.20 mostra a organização geral do framework NETPLAN. Ele é composto por um framework global e por três sub-frameworks: o **Application Framework**, o **Interface Framework** e o **Algorithm Framework**. Nesta figura, os componentes do framework estão ilustrados na cor branca, enquanto os componentes de uma aplicação específica aparecem na cor cinza escuro.

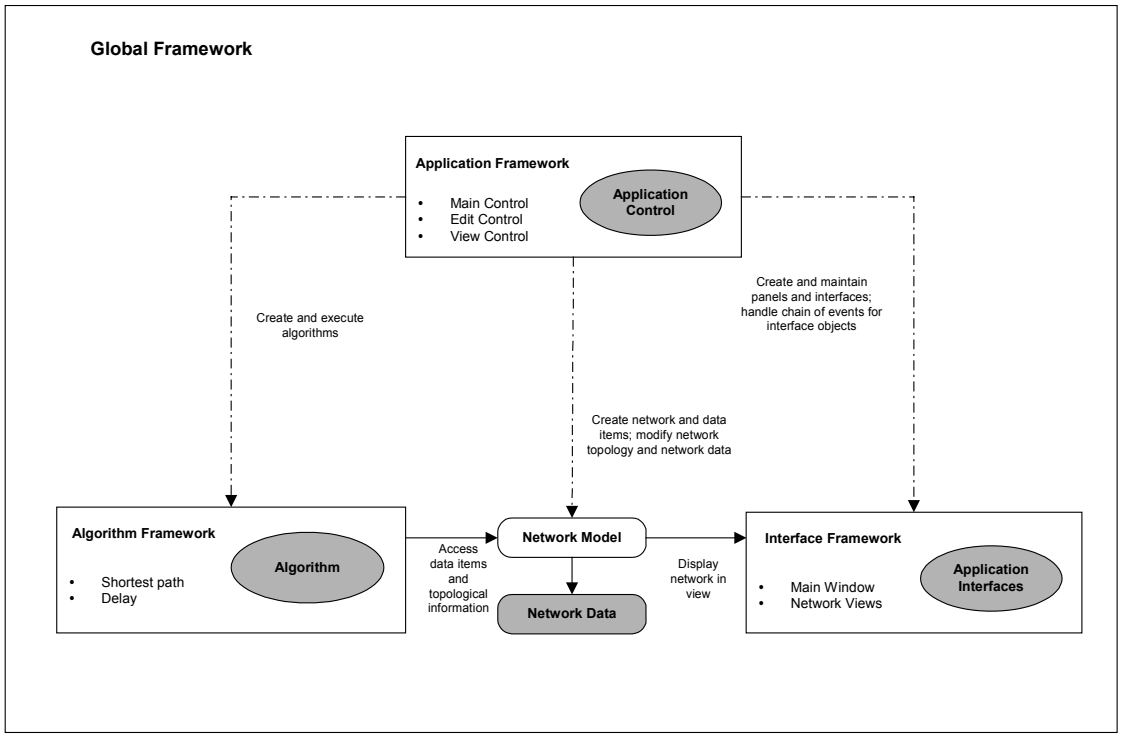


Figura 6.20 – A Organização do Framework NETPLAN.

Cada um dos componentes mostrados na figura acima será visto de maneira mais detalhada nas seções seguintes.

6.3.2 O Componente *Network Model*

O **Network Model** é um grafo que representa a topologia da rede em termos de nós e *links*. No framework NETPLAN, os componentes de uma rede de telecomunicações, tais como *roteadores*, pontes e terminais ISDN, são modelados através de um nó genérico no grafo da rede. Além disso, itens de dados são conectados diretamente aos nós do grafo com o objetivo de descrever as propriedades e os parâmetros de um componente em particular. Através da separação dos componentes de uma rede das suas respectivas descrições é possível gerenciar de maneira flexível, e em tempo de execução, mudanças nas informações de configuração. A Figura 6.21 mostra o diagrama de classes do componente **Network Model**.

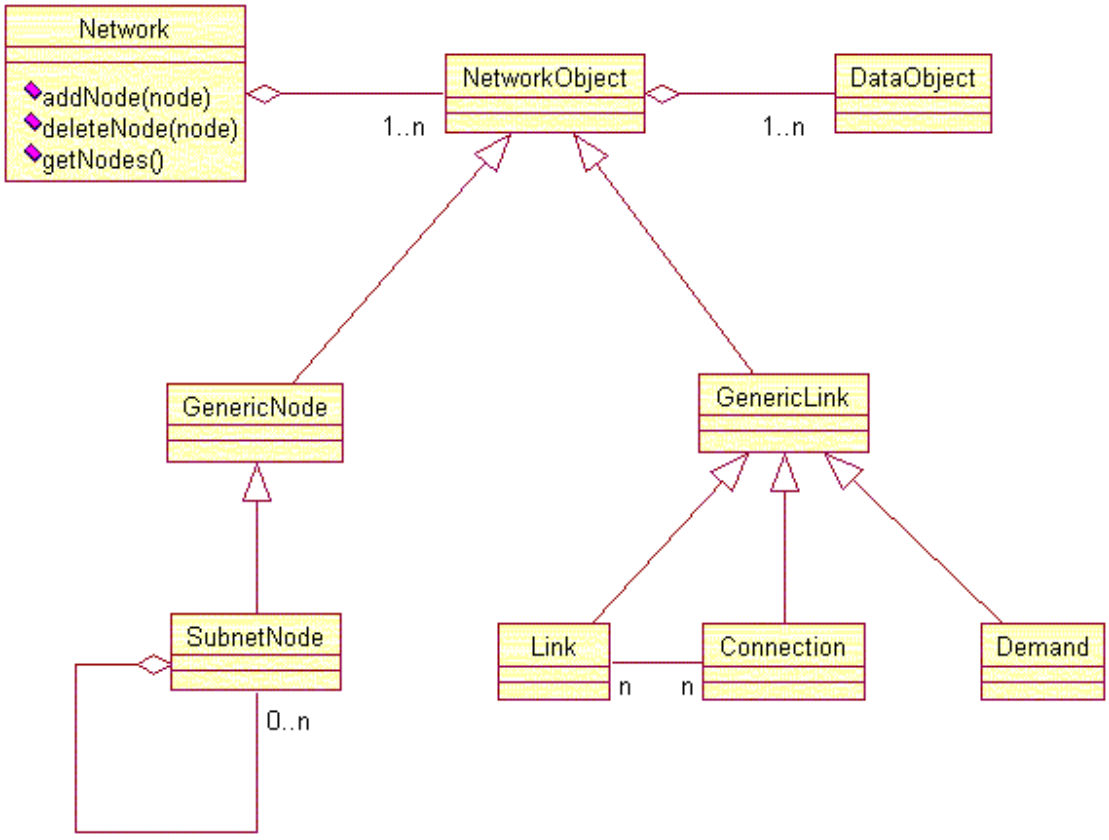


Figura 6.21 – O Diagrama de Classes do *Network Model*.

É importante destacar que o único Ponto de Adaptação existente neste componente é relativo à representação das propriedades de um componente de uma rede de telecomunicações. A instanciação deste Ponto de Adaptação deverá ser realizada através da inclusão de subclasses da classe **DataObject** no design final de uma aplicação produzida a partir do NETPLAN.

6.3.3 O Sub-Framework *Controller*

Toda aplicação requer alguma forma de mecanismo de controle que organize as interações com os usuários, mantenha a interface gráfica e execute os algoritmos responsáveis pela implementação das funcionalidades de uma dada aplicação. No caso do NETPLAN, o componente responsável pelo controle global de uma aplicação tem sua arquitetura baseada no *pattern Model-View-Controller*, descrito em [Gamma95]. A Figura 6.22 mostra o diagrama de classes do sub-framework responsável por esta tarefa.

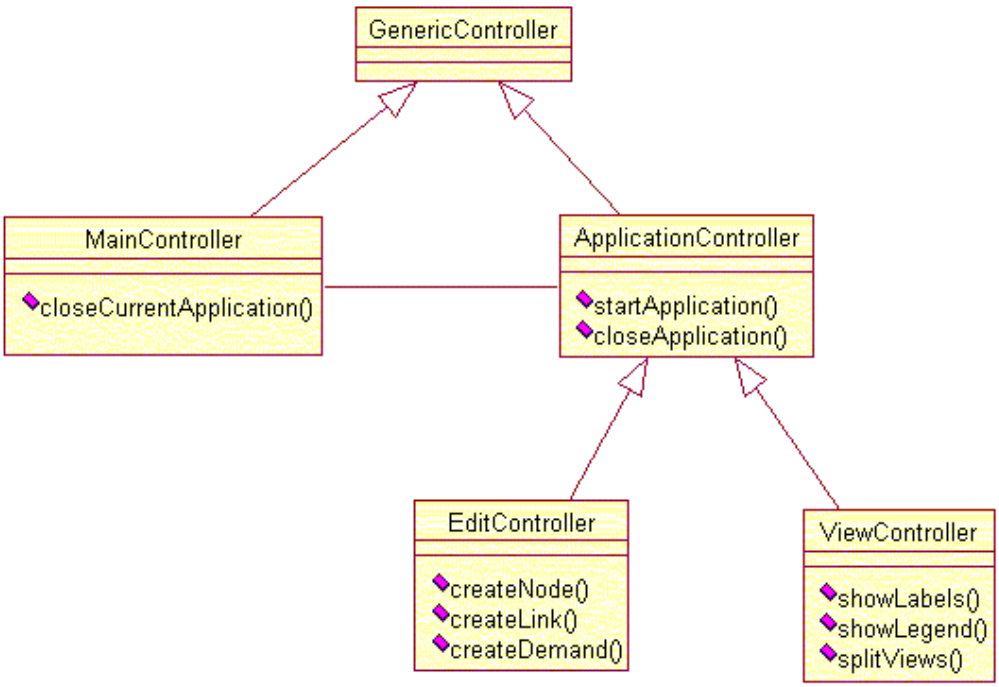


Figura 6.22 – O Diagrama de Classes do Sub-Framework *Controller*.

A partir do diagrama acima verificamos que todos os controladores são derivados de uma classe abstrata chamada **GenericController**. Uma das suas subclasses, a classe **MainController**, é responsável por manter uma referência para o controlador específico de uma aplicação (subclasse de **ApplicationController**). Esta referência é obtida após a criação do controlador da aplicação, que então se registra no controlador principal (instância da classe **MainController**). Graças a este mecanismo os eventos gerados na interface de uma aplicação são redirecionados, pelo controlador principal, para o controlador da aplicação corrente, onde tais eventos são trados de forma adequada.

O controlador da aplicação (instância da classe **ApplicationController**) é responsável pelo início (método **startApplication**) e pelo término da mesma (método **closeApplication**). Além disso, objetos da classe **EditController**, responsável pelo gerenciamento dos botões e dos itens de menu da janela principal, e da classe **ViewController**, responsável pela exibição dos rótulos e das legendas dos componentes de uma rede, são referenciados pelo controlador principal.

O único Ponto de Adaptação existente neste sub-framework tem por objetivo permitir a criação de um controlador específico para uma aplicação. Isto é feito através da inclusão de uma subclasse da classe **ApplicationController**.

6.3.4 O Sub-Framework *Interface*

O sub-framework de interface não apresenta nada de interessante com relação aos objetivos deste estudo de caso. Isto se deve ao fato dele estar baseado em um *toolkit* para a construção de interfaces gráficas capazes de serem facilmente adaptadas para diferentes plataformas, tais como *Motif* e *Microsoft Windows*. Desta forma é possível delegar às classes do *toolkit* toda a responsabilidade para implementar a flexibilidade necessária no que tange à interface gráfica.

A Figura 6.23 mostra alguns detalhes do sub-framework **Interface**. Apenas as classes responsáveis pela interface entre o *toolkit* e o resto da aplicação são mostradas. Entre elas é encontrado o único Ponto de Adaptação existente neste sub-framework. O seu objetivo é permitir o registro do objeto controlador da interface pelo controlador da aplicação (instância de uma subclasse de **ApplicationController**). A instanciação deste Ponto de Adaptação é feita criando-se uma subclasse de **GenericInterface**.

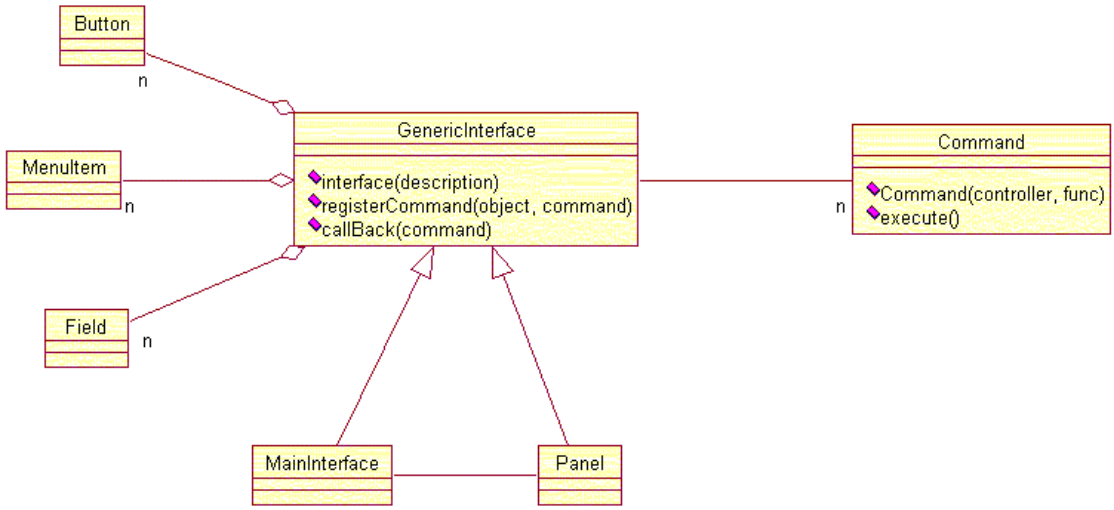


Figura 6.23 – O Diagrama de Classes do Sub-Framework *Interface*.

6.3.5 O Sub-Framework *Algorithm*

Para computar o atraso no tempo de resposta entre dois nós de uma rede de telecomunicação é necessário levar em conta os diversos protocolos usados nesta comunicação. Por exemplo, para computar o atraso causado pelo uso do protocolo HTTP entre dois nós de uma rede é necessário antes calcular o atraso devido ao protocolo TCP. O atraso do protocolo TCP, por sua vez, depende do atraso causado pelo protocolo IP. Entretanto, o atraso causado pelo protocolo IP difere de uma tecnologia de rede para a outra. Se uma determinada rede for baseada na tecnologia ATM, o atraso causado pelo protocolo IP deverá ser computado de maneira diferente

da situação onde a tecnologia da rede for ISDN. A Figura 6.24 resume a dependência existente no cálculo dos atrasos.

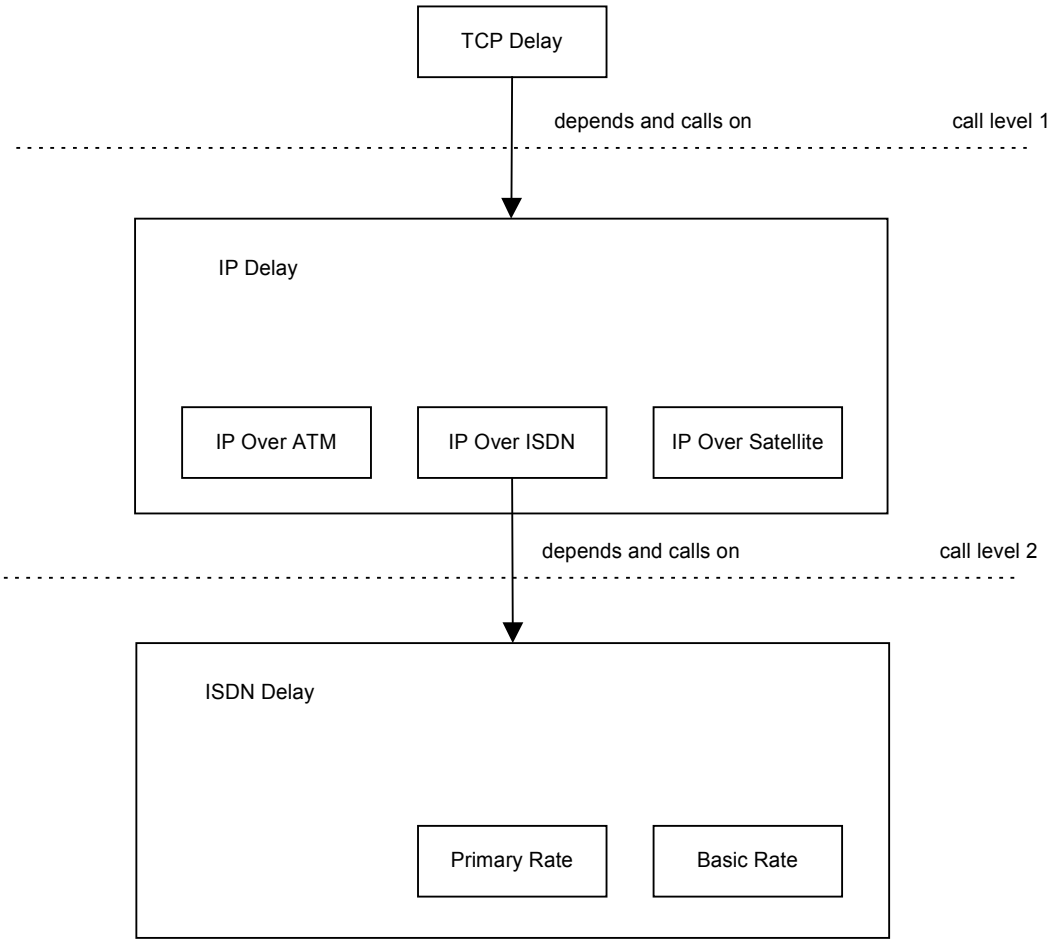


Figura 6.24 – Estrutura de Chamada dos Algoritmos para o Cálculo dos Atrasos.

A solução para este problema está baseada no conceito de família de algoritmos [Messmer00], nas quais os fragmentos de um algoritmo, ou modelos de algoritmos, são organizados. A Figura 6.25 mostra o modelo de classes do sub-framework **Algorithm**. Para modelar a família de algoritmos foi introduzida a classe **AlgorithmFamily**, cujo objetivo é fornecer uma interface comum para a seleção e execução dos algoritmos existentes na família. Para a implementação dos algoritmos em si é fornecida uma classe abstrata chamada **Algorithm**. Desta forma, a implementação de um determinado algoritmo é feita através de uma subclasse

concreta da classe **Algorithm**. Esta classe contém o único Ponto de Adaptação existente neste sub-framework.

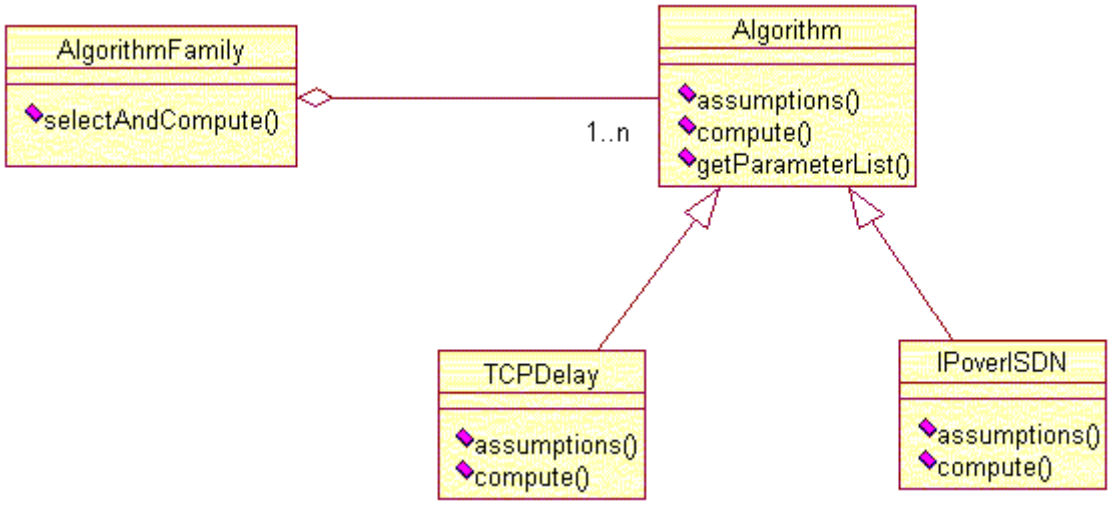


Figura 6.25 – O Diagrama de Classes do Sub-Framework *Algorithm.*

6.3.6 O Modelo de *Features* do Framework NETPLAN

A partir da descrição dos subsistemas do framework NETPLAN é possível construir um Modelo de *Features* que descreva as suas principais características. A primeira etapa da elaboração de tal modelo é a construção de um Diagrama de *Features*, que pode ser visto na Figura 6.26.

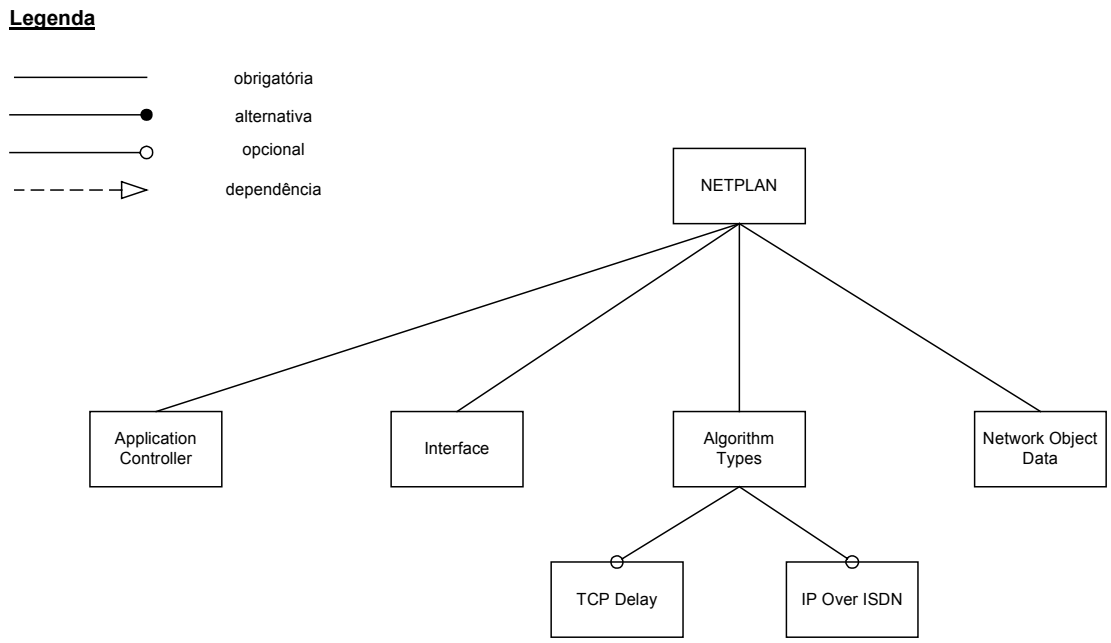


Figura 6.26 – O Diagrama de *Features* do Framework NETPLAN.

A próxima etapa da construção do Modelo de *Features* é estabelecer os relacionamentos entre as *Features* e os elementos de design que são responsáveis pela implementação de tais *Features* (Tabela 6.3).

Relação das Dependências Entre as *Features* e os Pontos de Adaptação do Framework NETPLAN	
Features	Ponto de Adaptação
Algorithms	Algorithm
	Algorithm.assumptions
	Algorithm.compute
TCP Delay	TCPDelay
IP Over ISDN	IPoverISDN
Application Controller	ApplicationController
	ApplicationController.startApplication
	ApplicationController.closeApplication
Interface	GenericInterface
	GenericInterface.interface
	GenericInterface.registerCommand

	GenericInterface.callBack
Network Object Data	DataObject

Tabela 6.3 - Relação das Dependências Entre as *Features* e os Pontos de Adaptação do Framework NETPLAN.

6.3.7 A Instanciação do Framework NETPLAN

A Figura 6.27 mostra as seleções realizadas sobre o Diagrama de *Features* para a construção de uma aplicação específica a partir do framework NETPLAN.

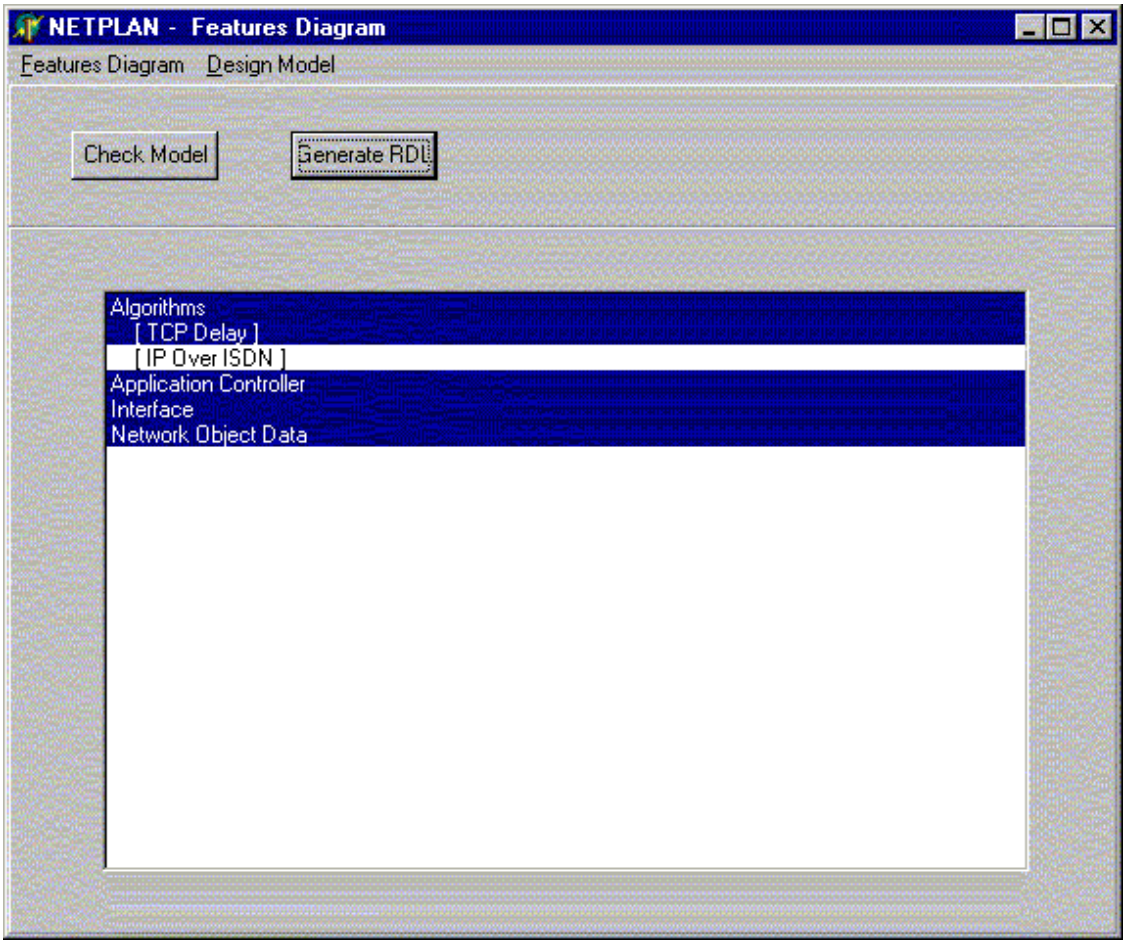


Figura 6.27 – As *Features* de uma Instância do Framework NETPLAN.

A partir das escolhas mostradas na Figura 6.27 a ferramenta Sagan *Tool* irá produzir um script RDL para que o design do framework NETPLAN possa ser adaptado. A Figura 6.28 mostra o script RDL gerado.

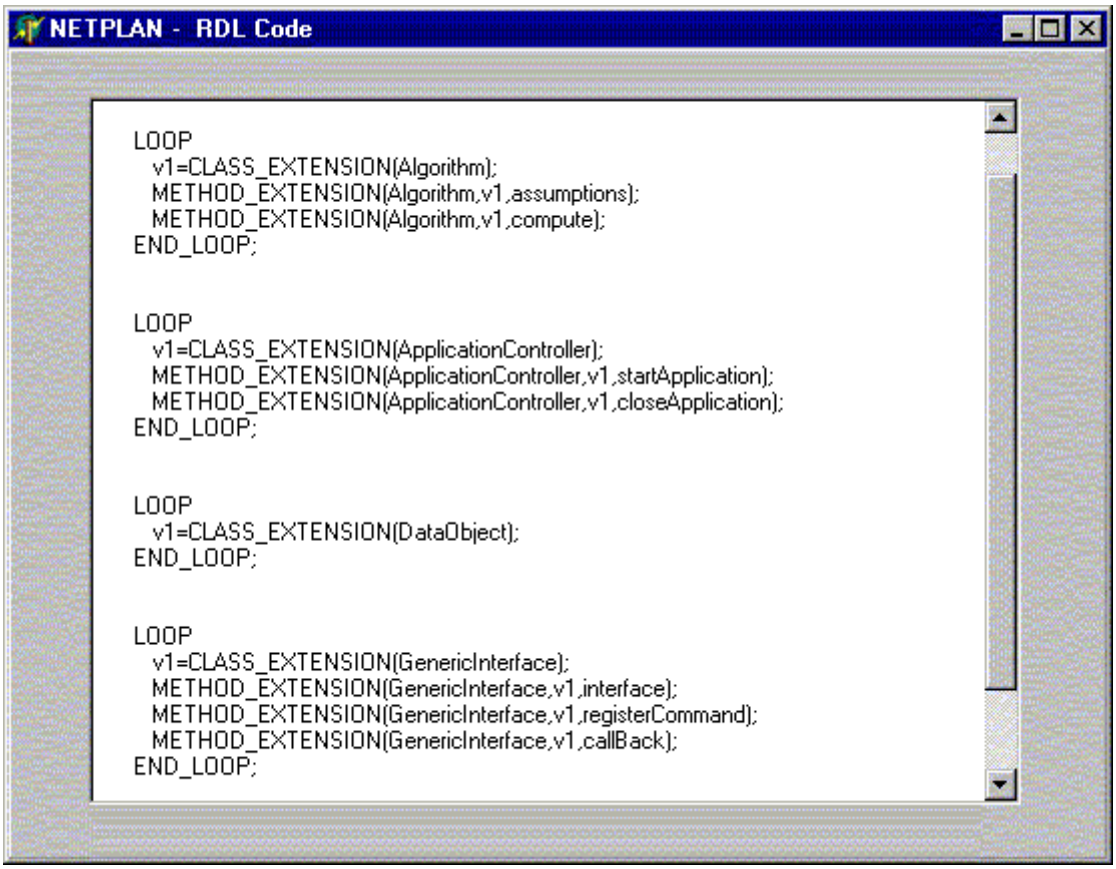


Figura 6.28 – O Script RDL Gerado a Partir das *Features* Selecionadas.

6.3.8 Considerações Finais

De um modo geral não existem grandes diferenças entre os processos de instanciação dos dois estudos de casos anteriores e o do NETPLAN, que foi apresentado nesta seção. Uma observação que deve ser feita é que à medida que novos algoritmos de cálculos de atrasos forem sendo acrescentados ao framework, ele tornar-se-á cada vez mais um framework *black-box*. Isto ocorre pelo fato de a *Feature* **Algorithm** ser o principal, e mais complexo, Ponto de Adaptação do framework NETPLAN.

Capítulo 7

Conclusão

A presente dissertação apresentou um novo enfoque para atacar o problema da instanciação de aplicações a partir de frameworks orientados a objetos. Este novo enfoque partiu da premissa de que um dos grandes entraves ao uso em larga escala da tecnologia de frameworks é o longo tempo necessário para que um *desenvolvedor* de aplicações se torne proficiente no uso de um determinado framework. Isto se deve, em grande parte, à inerente complexidade do design de um framework, que é projetado para atender aos requisitos de toda uma família de aplicações.

Buscando solucionar esta questão foi apresentada uma descrição detalhada de um processo de instanciação de alto nível baseado no Modelo de *Features*. A função deste modelo é apresentar aos *desenvolvedores* de aplicação uma visão simplificada das características funcionais e tecnológicas disponibilizadas por um determinado framework. Neste contexto, o processo de instanciação aqui proposto irá alterar os modelos de design originais de um framework, a partir das escolhas feitas no Modelo de *Features*, e criar modelos de design para uma aplicação específica.

Entretanto, para que o processo de instanciação como um todo pudesse ser viabilizado algumas importantes questões tiveram que ser resolvidas. Em primeiro lugar foi necessário definir precisamente as regras que estabelecem os relacionamentos entre o Modelo de *Features* e os modelos de classes que descrevem o design de um

framework. Isto foi realizado através da incorporação do Modelo de *Features* à linguagem UML. O uso do mesmo meta-modelo para descrever os dois modelos em questão, o de *Features* e o de classes, tornou mais fácil a elaboração das regras que governam todo o processo de documentação e instanciação aqui proposto. Além disso, foi necessário aperfeiçoar as idéias apresentadas em [Fontoura01], [Fontoura99] e [Oliveira01a], substituindo o uso de estereótipos e *tagged values* por alterações mais profundas no meta-modelo da UML. De modo semelhante ao que aconteceu com Modelo de *Features*, esta decisão permitiu o estabelecimento de regras que tornaram a descrição do processo com um todo mais precisa e objetiva.

Em relação aos requisitos de documentação definidos em [Johnson92] é importante destacar que a documentação apresentada nesta dissertação atende a todos de maneira satisfatória. Em primeiro lugar, o Modelo de *Features* descreve de maneira sucinta o objetivo e o domínio de aplicação de um framework, permitido assim a avaliação prévia da sua adequação a um determinado problema. Em segundo lugar, o processo como um todo, desde a escolha das *Features* mais adequadas a um problema até a geração de um script RDL, atende ao segundo requisito de documentação proposto em [Johnson92], uma vez que ensina aos *desenvolvedores* de aplicação a usar um determinado framework. Por último, os modelos de design de um framework, descritos na linguagem UML, poderão ser amplamente consultados, atendendo assim ao terceiro requisito de documentação.

No que tange ao processo de instanciação em si esta tese fez também algumas importantes contribuições. Em primeiro lugar ela corrige uma falha muito comum encontrada nos métodos de instanciação baseados em *cookbooks* ao inverter o processo de escolha das receitas para a instanciação de um framework. Ao invés de fornecer um conjunto de receitas previamente elaboradas e voltadas para o uso típico

de um determinado framework, ele expõe aos *desenvolvedores* de aplicação todo o leque de possibilidades existentes; para depois, então, definir as receitas que devem ser utilizadas para adaptar o framework aos requisitos de uma dada aplicação. Isto permite gerar planos específicos para diferentes situações, e evita que os *desenvolvedores* tenham que estudar todo um *cookbook* para selecionar as receitas mais adequadas as suas necessidades.

Outro importante avanço em relação aos métodos baseados em *cookbooks* em geral, e às propostas de [Johnson92] e [Froehlich97] em particular, é a integração do Modelo de *Features* com os modelos de design de um framework. Isto permite localizar precisamente os componentes de um framework que devem ser adaptados para atender às necessidades de uma certa aplicação. Mais ainda, o uso de marcadores de *hot-spots*, combinados com as regras de dependência entre as *Features*, *requires* e *mutexWith*, permite estabelecer um mapa das dependências existentes entre os Pontos de Adaptação de um framework. Em última análise são estes dois aspectos, a localização dos Pontos de Adaptação e a ocorrência de dependências entre eles, que irão permitir a geração automática de planos de instanciação a partir das necessidades definidas pelos usuários de um framework.

Em relação ao processo de instanciação de alto nível proposto em [Ortigosa00a], foram apresentadas também algumas importantes melhorias. Em primeiro lugar, o uso do Modelo de *Feature* se mostrou uma ferramenta mais adequada para descrever as características de um framework do que as interfaces baseadas em hipertexto mostradas em [Ortigosa00a]. Isto se deve basicamente ao fato de o Modelo de *Features* permitir uma rápida leitura das características obrigatórias, opcionais e alternativas existentes em um framework.

Em segundo lugar, o processo aqui descrito utiliza técnicas muito mais integradas para associar a descrição de alto nível de um framework com os seus Pontos de Adaptação. Enquanto em [Ortigosa00a] não são apresentadas com clareza como as regras de instanciação definidas em *TOON* são *mapeadas* no design de um framework, a abordagem aqui descrita integrou a descrição de alto nível, o Modelo de *Features*, com a descrição do design. Em seguida utilizou então o mesmo meta-modelo para definir regras bem claras para realizar o *mapeamento* necessário.

Por último, este trabalho apresentou uma descrição detalhada de uma ferramenta para dar suporte a todo o processo de instanciação de frameworks aqui proposto. Diferentemente de [Ortigosa00a], que propôs uma ferramenta baseada em agentes de software para conduzir todo o processo de instanciação de frameworks sem, contudo, apresentar nenhuma descrição objetiva de como isso seria feito, a presente dissertação apresentou um protótipo em funcionamento de uma ferramenta com esta finalidade. Esta ferramenta, chamada Sagan *Tool*, foi aplicada em dois casos de estudo e, em ambos, apresentou resultados condizentes com as expectativas existentes no início dos trabalhos de pesquisa; comprovando, desta maneira, a viabilidade das idéias aqui apresentadas.

Capítulo 8

Trabalhos Futuros

Algumas importantes questões referentes ao uso do Modelo de *Features* no processo de desenvolvimento e instanciação de frameworks ficaram de fora do escopo desta tese. Entretanto, devido à importância de algumas delas, iremos incluí-las como sugestões para trabalhos futuros.

Uma questão que ficou bem estabelecida durante todo o desenvolvimento deste trabalho é que o método de instanciação proposto trataria apenas das modificações no design de um framework necessárias para atender aos requisitos de uma aplicação específica. Não foram discutidas questões que envolvem modificações no código de um framework para acomodar as mudanças feitas no nível do design. Entretanto, algumas destas mudanças no código podem ser realizadas durante a execução do script de instanciação RDL, permitido assim que a abordagem aqui proposta seja ainda mais efetiva.

Para uma melhor ilustração da questão descrita acima iremos recorrer ao exemplo da Figura 8.1, baseado no framework DTFrame. Esta figura apresenta uma solução largamente utilizada para flexibilizar o design de uma aplicação orientada a objetos. Ela mostra uma classe abstrata, **Figure**, que define uma interface que deve ser implementada por outras classes que tenham um comportamento compatível com o seu. O mecanismo usado para realizar esta interface pode ser tanto o uso de herança,

como mostrado na Figura 8.1, como também o conceito de interface existente na linguagem Java.

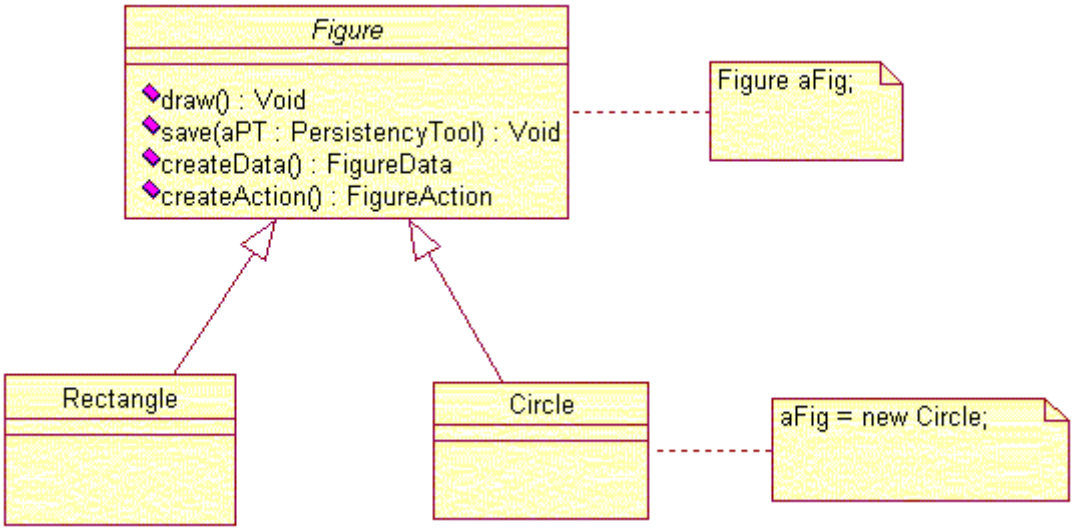


Figura 8.1 – Mecanismo de Flexibilização do Design.

Em ambos os casos, a necessidade de flexibilização do design irá se refletir no código do framework através da existência de inúmeros trechos onde serão feitas referências para um objeto *polimórfico* da classe **Figure**. Contudo, em algum momento será necessário lidar diretamente com um objeto de uma subclasse concreta da classe abstrata **Figure**. Por exemplo, para permitir a criação de novas figuras, a interface de uma aplicação baseada no DTFrame irá provavelmente apresentar uma barra de ferramentas com a lista de figuras disponíveis. Se pararmos agora para imaginar esta situação, chegaremos à conclusão de que em algum lugar do método que irá tratar o evento de criação de uma nova figura existirá uma definição de variável semelhante à existente na nota ligada à classe **Figure** (Figura 8.1). Entretanto, no momento da criação de uma figura qualquer, será necessário especificar a classe do objeto sendo criado. O código para realizar esta tarefa é semelhante ao existente na nota ligada à classe **Circle** (Figura 8.1).

Uma solução bastante viável para esta questão seria inserir marcadores no código do framework e associá-los à instanciação de um Ponto de Adaptação. No momento da execução do script RDL, quando as subclasses concretas estiverem sendo inseridas no design da aplicação final, o código marcado seria substituído pelo nome das subclasses apropriadas. Um tema interessante para trabalhos futuros seria investigar outras situações que poderiam ser tratadas de maneira semelhante, além de desenvolver ferramentas que permitissem alterar diretamente o código do framework para refletir algumas mudanças ocorridas no design.

Uma outra sugestão para trabalhos futuros é estudar a utilização do Modelo de *Features* no desenvolvimento de frameworks [Mathias01]. Mais precisamente, é preciso definir como o Modelo de *Features* poderá ser empregado nas fases iniciais do desenvolvimento de frameworks (Análise de Domínio) de tal maneira que a geração da documentação necessária para o processo de instanciação aqui proposto seja totalmente integrada ao processo de desenvolvimento. Alguns trabalhos anteriores, notadamente [Griss98] e [Vici98], procuraram integrar o Modelo de *Features* com os Modelos de Casos de Uso no âmbito de diferentes projetos para a produção de famílias de aplicações para o domínio de telecomunicações. Entretanto, tais trabalhos não deram ao Modelo de *Features* o mesmo papel ativo dado pelo processo de instanciação de frameworks proposto nesta dissertação. Em ambos os casos o papel do Modelo de *Features* fica praticamente restrito às etapas iniciais do desenvolvimento de uma solução genérica para uma família de problemas relacionados.

A última sugestão para trabalhos futuros envolve a questão da integração da ferramenta Sagan *Tool* com as ferramentas CASE usadas no desenvolvimento de sistemas orientados a objetos. Como foi visto no Capítulo 5 desta dissertação, o protótipo desenvolvido usou uma base de dados relacional para armazenar os Modelos

de Features e de Design de um framework, que foram carregados diretamente na base de dados da ferramenta. Todavia, a melhor abordagem a ser empregada seria obter tais informações diretamente das bases de dados da ferramenta CASE usada no desenvolvimento de um framework. Para tal, a melhor solução seria estudar a introdução do Modelo de *Features* na especificação XMI e produzir ferramentas que possam trabalhar diretamente sobre a representação em XMI dos Modelos de *Features* e de Design de um framework.

Apêndice A

A.1 As Regras de Boa Formação

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 3.3 com algumas regras de boa formação. Para tal, usaremos um *mix* de linguagem natural e expressões escritas na linguagem OCL [Warmer98]. 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 à sintaxe 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(x|x.role=#none)->size<=1
```

[2] Uma *DomainFeature* só poderá fazer o papel de parte (*part*) em uma única composição.

```
self.featureCompositionEnd->select(x|x.role<>#none)->size<=1
```

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

```
((self.directParts->size>0)
 and
 (self.directParts.featureCompositionEnd->
  select(x|x.role=#optional)->size=self.directParts->size))
implies
(self.inclusion=#optional)
```

[4] Uma *DomainFeature* que participa de uma composição no papel de uma parte opcional ou alternativa, tem que ser uma *Domain Feature* opcional.

```
(self.featureCompositionEnd->select(x|x.role=#optional or
 x.role=#alternative)->size>0)
implies
(self.inclusion=#optional)
```

[5] Uma *DomainFeature* obrigatória só pode participar de uma composição no papel de uma parte se esta parte for obrigatória.

```
(self.inclusion=#mandatory)
implies
((self.featureCompositionEnd->select(x|x.role<>#none)->size=0)
 or
 (self.featureCompositionEnd->select(x|x.role<>#none)->
  forAll(x|x.role=#mandatory))
```

[6] Se uma *DomainFeature* for opcional, todas as suas partes (*DomainFeature*) diretas ou indiretas (fecho transitivo) têm que ser também opcionais.

```
(self.inclusion=#optional)
implies
(self.allParts->forAll(x|x.inclusion=#optional)
```

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

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

[8] 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

[9] Se, em uma composição, uma das partes (*DomainFeature)* for uma alternativa todas as demais também o serão.

```
(self.part->select(x|x.role=#alternative)->size>0)
implies
(self.part->size=self.part->select(x|x.role=#alternative)->size)
```

[10] Em uma composição, um *FeatureCompositionEnd* só poderá estar no papel de *whole* se, e somente se, o seu atributo *role* for igual a *none*.

```
(self.whole.role=#none) and
(self.part->forAll(x|x.role<>#none)
```

DomainFeatureDependency

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

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

[12] Uma *DomainFeature* não pode estar envolvida em uma Dependência, seja ela *requires* ou *mutexWith*, com *DomainFeatures* que sejam seus *parents* ou suas *parts*.

```
(self.source.allParts->intersection(self.target)->isEmpty)
and
(self.source.allParents->intersection(self.target)->isEmpty)
```

[13] Uma Dependência, seja ela *requires* ou *mutexWith*, só pode relacionar *DomainFeatures* opcionais.

```
self.source.inclusion=#optional and
self.target.inclusion=#optional
```

Operações Adicionais

[14] Esta operação retorna um conjunto com todas as partes (*DomainFeatures*) relacionadas diretamente com uma dada *DomainFeature* (*self*) através de uma composição.

```
directParts : Set(DomainFeature)

directParts=self.featureCompositionEnd->
select(x|x.role=#none).domainFeatureComposition.part.
domainFeature
```

[15] Esta operação retorna um conjunto com todas as partes (fecho transitivo) relacionadas com uma dada *DomainFeature* (*self*) através de relacionamentos de composição.

```
allParts : Set(DomainFeature)

allParts=if(self.directParts->isEmpty)then
          Set{  }    --returns an empty set
```

```
          else
            self.directParts->union(self.directParts.allParts)
          endif
```

[16] Esta operação retorna a *DomainFeature* que faz o papel do todo (*whole*) em uma composição onde uma dada *DomainFeature* (*self*) faz o papel de uma parte (*part*).

```
directParent : DomainFeature

directParent=self.featureCompositionEnd->
select(x|x.role<>#none).domainFeatureComposition.whole.
domainFeature
```

[17] Esta operação retorna um conjunto contendo todas as *DomainFeatures* (fecho transitivo) que estão no caminho de uma dada *DomainFeature* (*self*) até o nó raiz de uma árvore de composição do Diagrama de *Features*.

```
allParents : Set(DomainFeature)

allParents=if(self.directParent->isEmpty)then

            Set{  }  --returns an empty set
          else
            self.directParent->
            union(self.directParent.directParent)
          endif
```

A.2 As 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] Todas as *DomainFeatures* obrigatórias têm que ser selecionadas.

```
(self.inclusion=#mandatory) implies (self.isSelected=true)
```

[2] Se uma *DomainFeature* for selecionada todos os seus parentes também o serão.

```
(self.isSelected=true)
implies
(self.allParents->forAll(x|x.isSelected=true))
```

[3] Se uma *DomainFeature* não for selecionada todas as suas partes, diretas ou indiretas, também não poderão ser selecionadas.

```
(self.isSelected=false)
implies
(self.allParts->forAll(x|x.isSelected=false))
```

DomainFeatureComposition

[4] Em uma composição alternativa no máximo uma *DomainFeature* (*part*) poderá ser selecionada.

```
(self.part->select(x|x.role=#alternative)->size>0)
implies
(self.part.domainFeature->select(y|y.isSelected=true)->size<=1)
```

[5] Dada uma composição, se a *DomainFeature* no papel de todo (*whole*) for selecionada, todas as *DomainFeatures* ligadas às partes obrigatórias serão também selecionadas.

```
(self.whole.domainFeature.isSelected=true)
implies
(self.part->select(x|x.role=#mandatory)->
 forAll(y|y.domainFeature.isSelected=true))
```

DomainFeatureDependency

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

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

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

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

Apêndice B

A seguir serão apresentadas as regras que devem ser observadas para que se possa integrar um Modelo de *Features* com os diagramas de classes que descrevem o design de um framework. Tais regras já foram apresentadas de maneira informal no Capítulo 4 desta dissertação.

Os métodos utilizados aqui serão os mesmos apresentados no Apêndice A. Ou seja, será usado um *mix* de linguagem natural com expressões escritas na linguagem OCL.

Attribute

[1] Define os tipos de extensões válidos para um atributo adaptável.

```
(self.extension=#valueAssignment) or
(self.extension=#valueSelection) or
(self.extension=#none)
```

[2] Se um atributo for selecionado então a classe na qual ele está inserido deverá ser também selecionada.

```
(self.isSelected=true) implies (self.owner.isSelected=true)
```

Class

[3] Define os tipos de extensões válidos para uma classe adaptável.

```
(self.extension=#classExtension) or
(self.extension=#SelectClassExtension) or
(self.extension=#none)
```

[4] Se uma classe for selecionada então todas as suas classes ascendentes deverão ser também selecionadas.

```
(self.isSelected=true) implies
(self.allParents->collect(x|x.oclAsType(Class))->
 forAll(y|y.isSelected=true))
```

Method

[5] Define os tipos de extensões válidos para um método adaptável.

```
(self.extension=#methodExtension) or
(self.extension=#none)
```

[6] Se um método for selecionado então a classe na qual ele está inserido deverá ser também selecionada.

```
(self.isSelected=true) implies (self.owner.isSelected=true)
```

DomainFeature

[7] Se uma *DomainFeature* for selecionada então todos os elementos adaptáveis a ela relacionados têm que ser selecionados.

```
(self.isSelected=true) implies
(self.featureTrace.target->forAll(x|x.isSelected=true))
```

[8] Se uma *DomainFeature* estiver relacionada com algum elemento adaptável então este elemento tem que ser um *hot-spot*.

```
(self.featureTrace.target->forAll(x|x.isHotSpot()=true))
```

Observação

O papel *owner* e a operação *allParents* estão definidas em [UML01].

Apêndice C

O objetivo deste Apêndice é apresentar alguns detalhes da arquitetura do protótipo da ferramenta Sagan *Tool*. Serão mostrados alguns diagramas contendo as classes mais importantes e os seus relacionamentos. Os detalhes, como os atributos e os métodos de cada classe, serão exibidos apenas quando forem fundamentais para o entendimento de um determinado modelo. Serão mostrados também diagramas de seqüência contendo os detalhes da troca de mensagens entre os objetos quando da carga dos Modelos de *Features* e de Classes, e da geração do código RDL.

C.1 Os Diagramas de Classes

A Figura C.1 mostra a classe **TSaganTool** e algumas das principais classes que compõem o protótipo em questão. A classe **TSaganTool** é o controlador de toda a ferramenta. Todas as operações realizadas nas interfaces com os usuários serão traduzidas em mensagens que serão enviadas para ela.

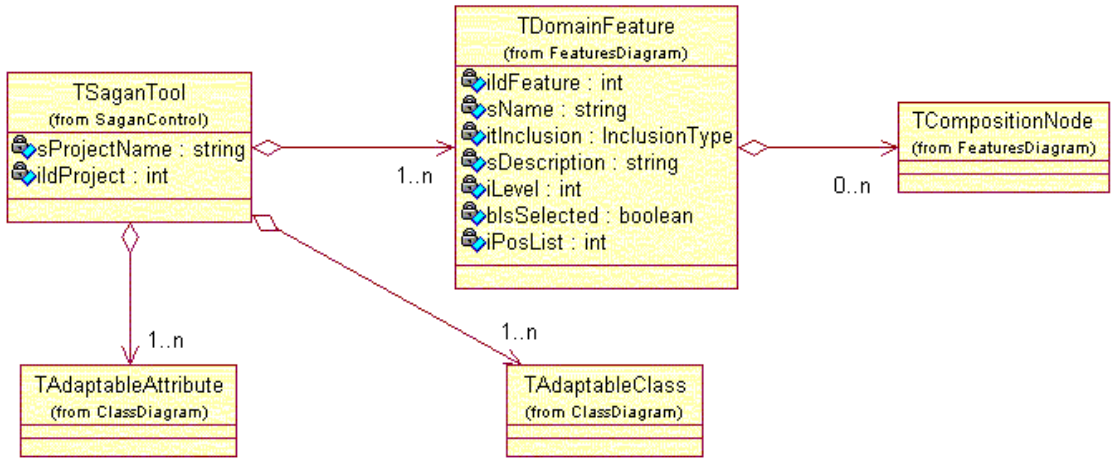


Figura C.1 – As Principais Classes da Ferramenta Sagan *Tool*.

Outro importante detalhe é a implementação dos Elementos Adaptáveis. A Figura C.2 mostra que o design utilizado é muito semelhante ao que foi apresentado no Capítulo 4, com o objetivo de inserir os referidos elementos no meta-modelo da UML.

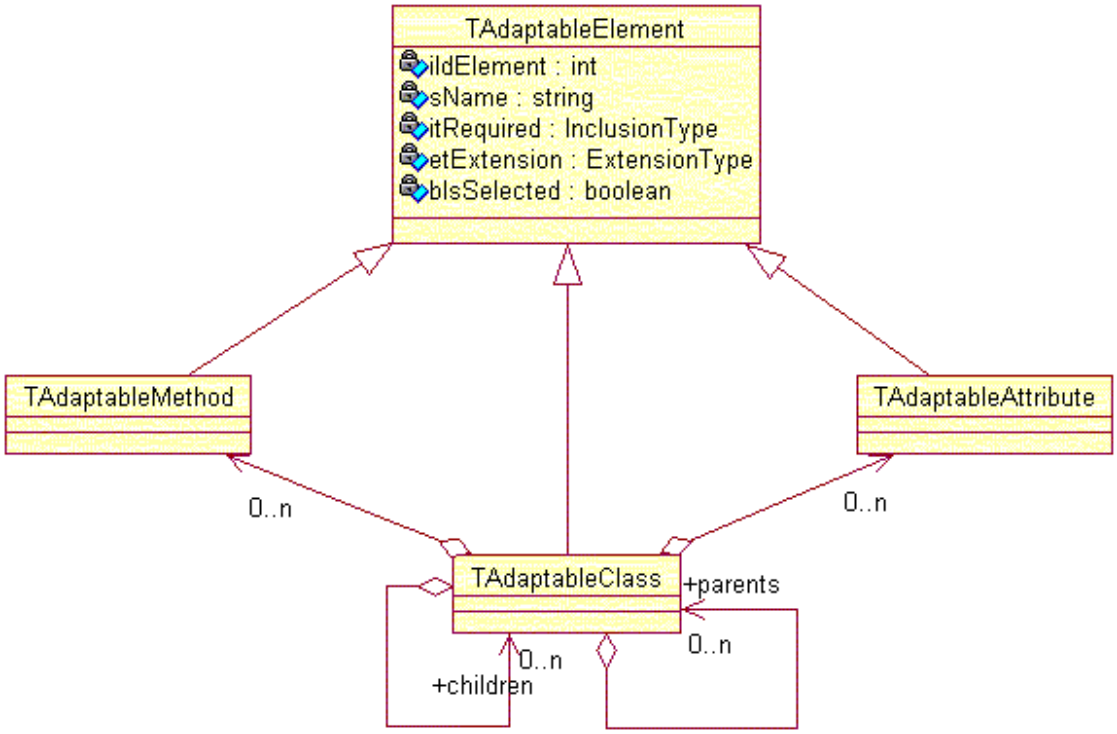


Figura C.2 – A Implementação dos Elementos Adaptáveis.

A partir de uma rápida análise da Figura C.2 é possível observar que uma Classe Adaptável (**TAdaptableClass**) mantém referências para todos os seus métodos e todos os seus atributos. É possível observar também que são mantidas referências para as suas classes ascendentes (*parents*) e descendentes (*children*).

As Figuras C.3 e C.4 mostram alguns detalhes da arquitetura da interface gráfica e do mecanismo de persistência do protótipo de Sagan *Tool*. Como a implementação do protótipo foi feita com a ferramenta *Borland Delphi*, as classes utilizadas na construção destes dois subsistemas foram baseadas em componentes existentes na Biblioteca de Componentes Visuais (VCL). Esta biblioteca é fornecida juntamente com a ferramenta *Borland Delphi*.

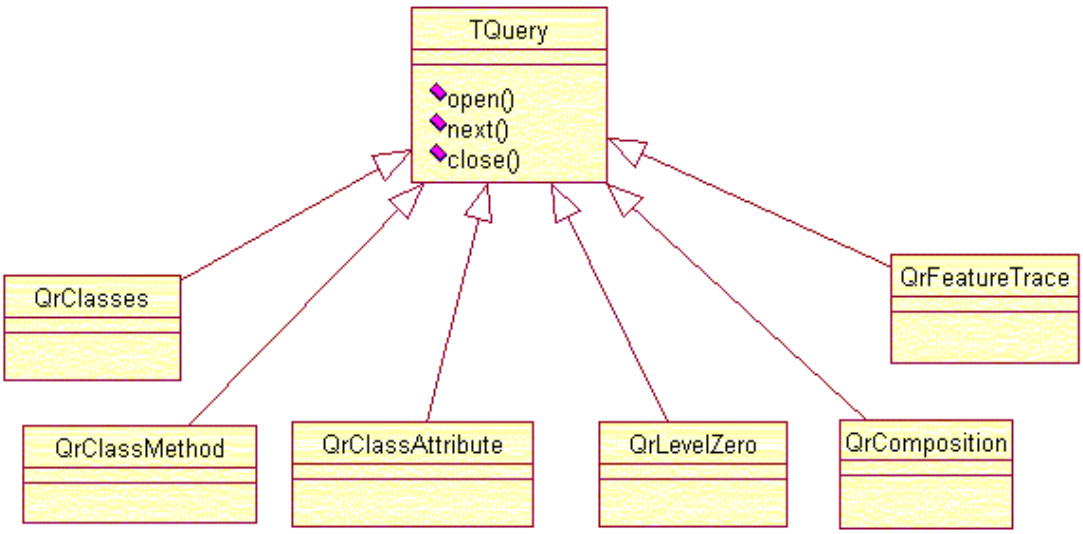


Figura C.3 – O Subsistema de Persistência.

A Figura C.3 mostra algumas das classes envolvidas no acesso à base de dados de Sagan *Tool*. Todas elas herdam do componente **TQuery**. Este componente permite o envio de consultas para um gerenciador de banco de dados, ou para um sistema de arquivos, capaz de processar comandos SQL. Já a Figura C.4 mostra algumas das classes utilizadas na construção da interface com os usuários finais. Todas elas herdam do componente **TForm**, que é a classe da VCL que implementa os formulários utilizados na construção de interfaces gráficas.

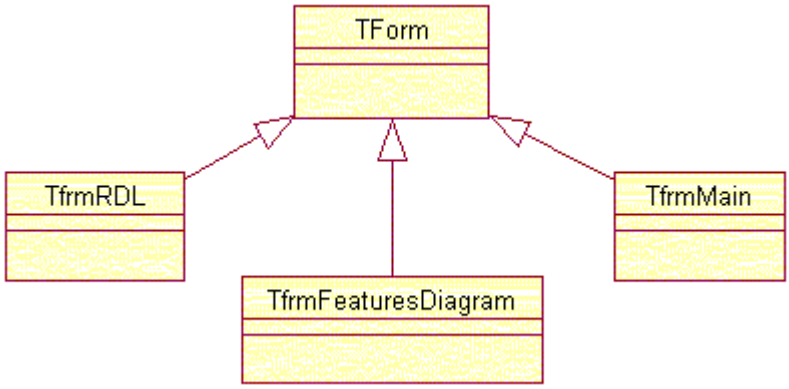


Figura C.4 – O Subsistema de Interface com o Usuário.

C.2 Os Diagramas de Seqüência

A seguir serão mostrados alguns diagramas de seqüência que irão ilustrar as trocas de mensagens entre os objetos da ferramenta durante a execução de duas importantes operações. A primeira delas é a carga inicial dos Modelos de *Features* e de Design. A segunda é a etapa de geração de código RDL para uma dada instância de um Modelo de *Features*.

A Figura C.5 mostra a carga inicial dos Modelos de *Features* e de Design de um determinado framework. Esta operação é disparada quando o usuário da ferramenta selecionar a opção **Load**, apresentada no menu principal do Diagrama de *Features*.

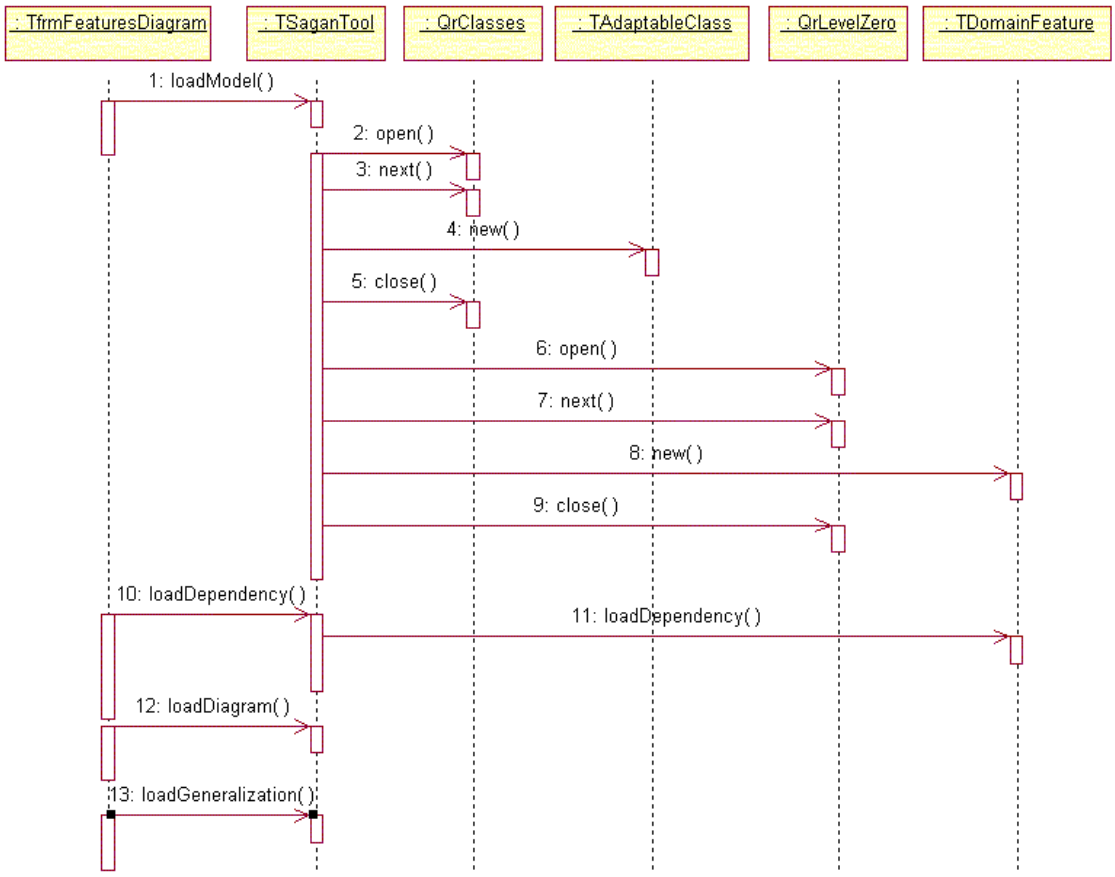


Figura C.5 – Uma Visão Geral da Carga dos Modelos de *Features* e de Design.

Inicialmente a classe **TSaganTool** recebe uma mensagem da interface do sistema solicitando a carga dos modelos. Em resposta a esta mensagem será executado o método **loadModel**. Entre as responsabilidades deste método está a carga de todas as classes e de todas as *Features* que compõem um dado framework. Além disso, será realizada a carga das dependências existentes entre as *Features*, e a carga dos relacionamentos de generalização existentes entre as classes. Por último, será executada a carga do Diagrama de *Fatures* em si. O referido diagrama será então exibido pela interface.

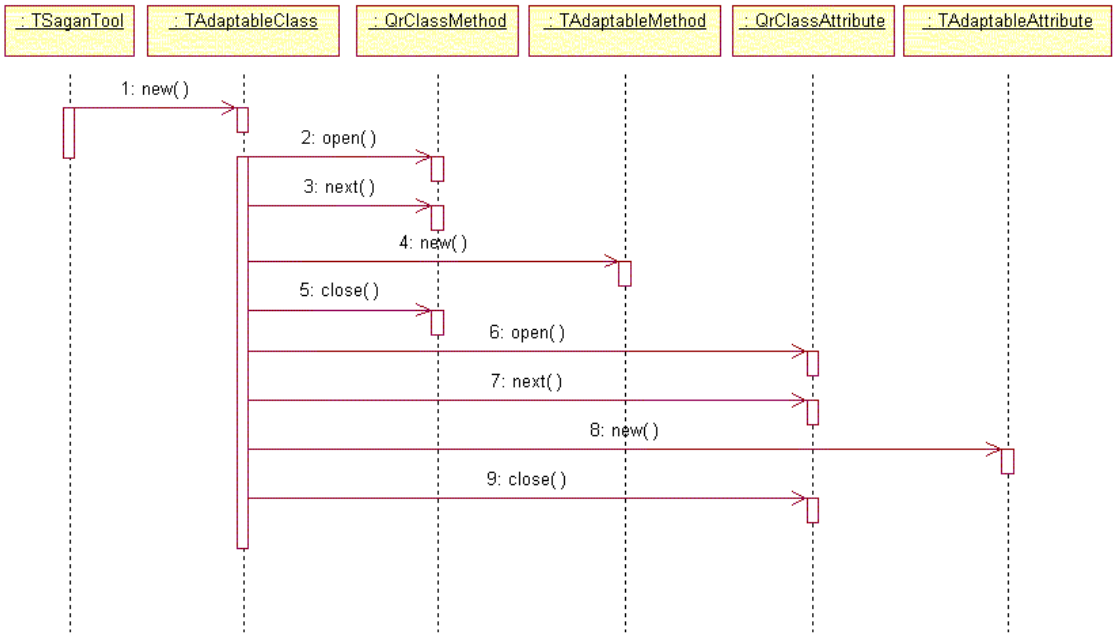


Figura C.6 – As Operações Realizadas pelo Construtor da Classe **TAdaptableClass**.

Durante a criação de uma determinada classe, algumas operações são realizadas pelo *construtor* (**new**) da classe **TAdaptableClass**. A Figura C.6 acima mostra a carga de todos os métodos e de todos os atributos da classe em questão.

Um procedimento semelhante é realizado durantes a criação de uma determinada *Feature* (Figura C.7). Neste caso, o construtor da classe **TDomainFeature** é responsável pela carga de uma eventual composição que tenha a *Feature* em questão no papel de todo (*whole*). Além disso, são criadas as referências para todos os

Elementos Adaptáveis que representam os Pontos de Adaptação associados à *Feature* em questão.

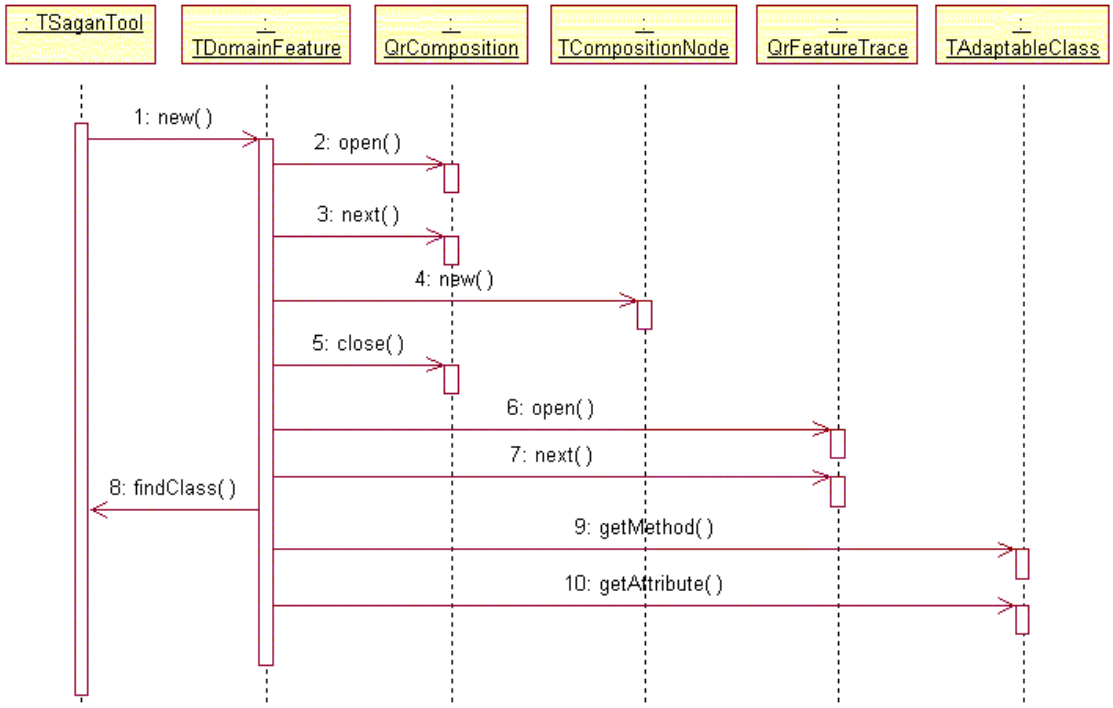


Figura C.7 – As Operações Realizadas pelo Construtor da Classe **TDomainFeature**.

A última operação que será mostrada é a geração de código RDL para uma dada instância de um Modelo de *Features*. Esta operação é disparada quando um usuário selecionar o botão **Generate RDL**, existente na interface do Diagrama de *Features*. Em resposta a esta solicitação é enviada uma mensagem para a classe **TSaganTool**, que começa então a coordenar as ações necessárias (Figura C.8).

Em primeiro lugar são enviadas mensagens para as classes que compõem o framework em questão. Cada classe é responsável pela geração do seu próprio código RDL. Em resposta ao envio de tal mensagem o método **generateRDL** é executado. Este método irá gerar o código relativo à classe em si. Ele também irá enviar a mesma mensagem para os métodos e para os atributos que compõem a classe cujo código está sendo gerado. Ao receber uma mensagem **generateRDL**, um método irá gerar o seu próprio código RDL. Já um atributo irá apenas inserir a si próprio em uma lista de

atributos passada como parâmetro. No final do processo de geração de código RDL, a classe **TSaganTool** irá proceder a geração do código referente aos atributos que estiverem na lista em questão.

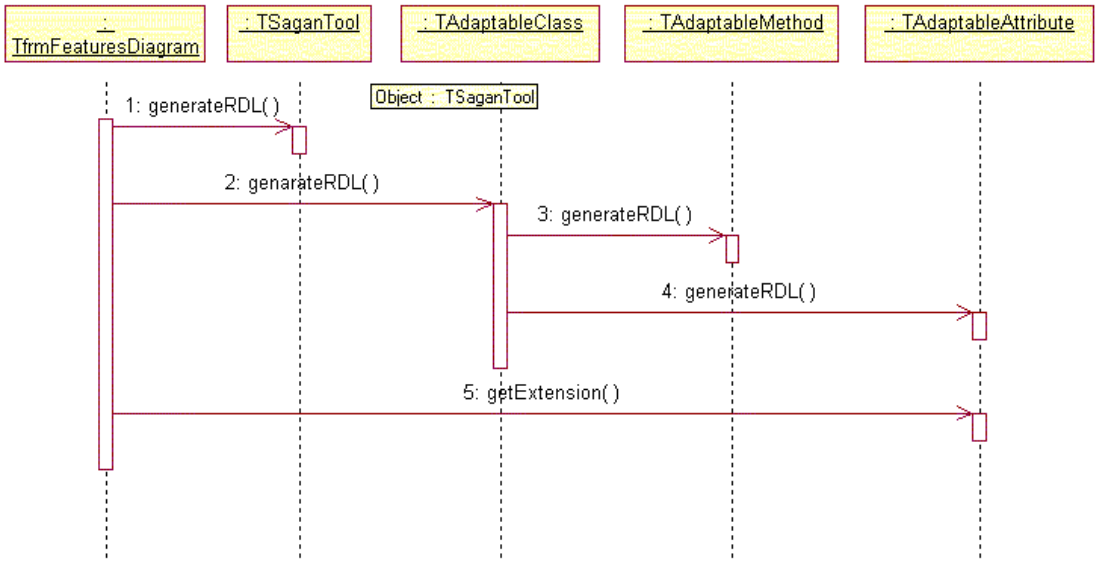


Figura C.8 – A Geração de Código RDL.

Apêndice D

Durante o desenvolvimento desta tese vários artigos foram publicados em congressos internacionais de Engenharia de Software. Estas publicações mostram o processo evolutivo pelo qual passaram as idéias aqui apresentadas. O conteúdo desta dissertação reflete o amadurecimento dos conceitos e dos processos que começaram a ser elaborados há mais de dois anos atrás. Isto permitiu que alguns dos capítulos que compõem esta dissertação fossem, em grande parte, baseados no conteúdo apresentado por estes artigos.

A relação das publicações segue uma ordem cronológica. Isto permitirá uma melhor compreensão da evolução das idéias aqui apresentadas.

1. A Framework Based Approach for Workflow Software Development [Oliveira00]. Novembro de 2000.

Este artigo propõe uma nova abordagem para a construção de frameworks para sistemas de *Workflow*. Nele estão registrados os primeiros passos na tentativa de se propor um método de desenvolvimento de frameworks baseado em artefatos produzidos pela análise do domínio de aplicação do framework. Deste estudo surgiu a idéia de se adaptar o Modelo de *Features* para aplicá-lo na construção de frameworks.

2. Domain Oriented Framework Construction [Mathias01]. Julho de 2001.

Neste trabalho foram apresentadas algumas melhorias em relação ao papel do Modelo de *Features* dentro do contexto do desenvolvimento de frameworks. Um dos aspectos mais explorados foi o estabelecimento de relacionamentos entre o Modelo de *Features* e os Modelos de Casos de Uso [Jacobson97], empregados na descrição abstrata das principais funções disponibilizadas por uma família de aplicações. Embora esta questão

não tenha sido abordada nesta tese, foi daí que começaram os primeiros esforços no sentido de incluir o Modelo de *Features* na linguagem UML. Desta maneira seria possível corrigir algumas das deficiências existentes nos Modelos de Casos de Uso, notadamente seu caráter fundamentalmente textual e pouco estruturado.

A outra novidade apresentada por este trabalho foi a mudança do papel exercido pelo Modelo de *Features*. Até então este modelo era usado apenas como uma ferramenta para definir as características obrigatórias, opcionais e alternativas de um domínio de aplicação. A partir de então o Modelo de *Features* passou a atuar efetivamente na etapa de instanciação de frameworks, tornando-se o elemento catalisador de todo o processo de adaptação de um framework às necessidades de uma aplicação específica.

Este artigo foi selecionado para ser publicado em um livro [Mathias02a] contendo os melhores trabalhos apresentados no congresso.

3. Using XML and Frameworks to Develop Information Systems [Oliveira01b]. Julho de 2001.

Este artigo é um complemento das idéias apresentadas no trabalho anterior. Enquanto em [Mathias01] é dado um tratamento mais metodológico às questões envolvendo a construção e a instanciação de frameworks a partir de artefatos gerados durante a etapa de Análise de Domínio, em [Oliveira01b] são apresentados maiores detalhes sobre a integração do Modelo de *Features* com os modelos de design de um framework. Entre eles está a utilização de uma notação mais precisa para representar os *hot-spots* de um framework, a extensão UML-F, e a utilização do padrão XML/XMI na representação dos modelos usados no processo de desenvolvimento de frameworks.

4. A Proposal for the Incorporation of the Features Model into the UML Language [Mathias02b]. Abril de 2002.

Este artigo contém o detalhamento da proposta de incorporação do Modelo de *Features* à linguagem UML. O Capítulo 3 desta dissertação é uma revisão do conteúdo apresentado neste trabalho.

5. Frameworks – A High Level Instantiation Approach [Oliveira02]. Abril de 2002.

Este artigo é o resumo das idéias apresentas nesta dissertação. Ele mostra todo o processo de instanciação de alto nível aqui proposto, desde a construção do Modelo de *Features*, com as necessárias ligações com os *hot-spots* de um framework, até a geração de um script de instanciação RDL baseado nas características selecionadas.

Bibliografia

[Access] A descrição do Microsoft Access pode ser encontrada no seguinte endereço eletrônico: http://www.microsoft.com/office/access/.

[Aho95] Aho, A.V. e Ullman, J.D. *Foundations of Computer Science – C Edition*. W.H. Freeman and Company, New York, EUA, 1995.

[Arango93] Arango, G. *Domain Analysis Methods*. Em: Prieto-Díaz, R. e Frakes, W.B. eds. Advances in Software Reuse: Selected Papers from the Second International Workshop on Software Reusability. IEEE Computer Society Press, Los Alimos, EUA, pp.17-49, 1993.

[Arehart00] Arehart, C. *et all* . *Professional WAP*. Wrox Press, Birmingham, Reino Unido, 2000.

[Argo] A descrição do Argo/UML pode ser encontrada no seguinte endereço eletrônico: http://www.argouml.org.

[Ayers99] Ayers, D. *et all* . *Professional Java Server Programming*. Wrox Press, Birmingham, Reino Unido, 1999.

[Batini92] Batini, C.; Ceri, S. e Navathe, S.B. *Conceptual Database Design – An Entity-Relationship Approach*. The Benjaming/Cummings Publishing Company, Redwood City, EUA, 1992.

[Booch94] Booch, G. *Designing an Application Framework*. Dr. Dobb´s Journal, Vol. 19, Nº. 2, Fevereiro, pp. 24-32, 1994.

[Booch99] Booch, G.; Rumbaugh, J. e Jacobson, I. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, EUA, 1999.

[Bosch] Bosch, J. e Dittrich, Y. *Domain-Specific Languages for a Changing World*. Encontrado em http://www.cs.rug.nl/~bosch.

[Bradshaw97] Bradshaw, J.M. (ed.). *Software Agents*. AAAI Press/MIT Press, Cambridge, EUA, 1997.

[Braun92] Braun, C.; Hatch, W.; Ruegsegger, T.; Balzer, B.; Feather, M.; Goldman, N. e Wile, D. *Domain-Specific Software Architectures: Command and Control*. Relatório Técnico CMU/SEI-92-SR-9. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, EUA, 1992.

[Budd91] Budd, T. *An Introduction to Object-Oriented Programming*. Addison-Wesley, Reading, EUA, 1991.

[Butler97] Butler, G.; Denommée, P. *Documenting Frameworks*. Em: Proceedings of the 8th Annual Workshop on Institutionalizing Software Reuse (WISR'8), Ohio, EUA, 1997.

[Calder95] Calder, P. *InterViews: A Framework for X-Windows*. Em: Lewis, T. ed. Object-Oriented Application Frameworks. Manning Publications, Greenwich, EUA, pp. 195-220, 1995.

[Campbell92] Campbell, R.H. e Islam, N. *A Technique for Documenting the Framework of an Object-Oriented System*. Em: Proceedings of the 2nd International Workshop on Object-Oriented Operating Systems, Paris, França. Pp. 288-300, 1992.

[Carvalho98] Carvalho, S.E.R.; Cruz, S.O. e Oliveira, T.C. *Second Generation Object-Oriented Development*. Second CNPq/NFS Workshop on Formal Aspects on Computation, New Orleans, EUA, 1998.

[Casanova87] Casanova, M.A.; Giorno, F.A.C. e Furtado, A.L. *Programação em Lógica e a Linguagem Prolog*. Edgar Blücher, São Paulo, Brasil, 1987.

[Cheatham89] Cheatham, T.E. *Reusability Through Program Transformations*. Em: Biggerstaff, T.S. e Perlis, A. eds. Software Reusability. ACM Press, New York, EUA, Vol. 1, pp. 321-335, 1999.

[Codenie97] Codenie, W.; Hondt, K.D.; Steyaert, P. e Vercammen, A. *From Custom Applications to Domain-Specific Frameworks*. Communication of the ACM, Vol. 40, Nº. 10, Outubro, pp. 71-77, 1997.

[Cohen92] Cohen, S.G.; Stanley, J.L.; Peterson, A.S. e Krut, R.W. *Application of Feature-Oriented Domain Analysis to the Army Movement Control Domain*. Relatório Técnico CMU/SEI-91-TR-28. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, EUA, 1992.

[Dalebout99] Dalebout, A.; Hillegersberg, J.V. e Wierenga, B. *Domain Framework for Sales Promotions*. Em: Fayad, M.E. Schmidt, D.C. e Johnson, R. eds. Implementing Application Frameworks – Object-Oriented Frameworks at Work. John Wiley,New York, EUA, pp. 7-24, 1999.

[Date99] Date, C.J. *An Introduction to Database Systems*. 7ª. ed., Addison-Wesley, Reading, EUA, 1999.

[Delphi] A descrição do Borland Delphi pode ser encontrada no seguinte endereço eletrônico: http://www.borland.com/delphi/.

[Duke94] Duke, R.; Rose, G. e Smith, G. *Object-Z: A Specification Language Advocated for the Description of Standards*. Relatório Técnico SVRC 94-45. Software Verification Research Center, The University of Queensland, Austrália, 1994.

[Fayad99] Fayad, M.E.; Schmidt, D.C. e Johnson, R. *Application Frameworks*. Em: Fayad, M.E. Schmidt, D.C. e Johnson, R. eds. Building Application Frameworks – Object-Oriented Foundations of Framework Design. John Wiley,New York, EUA, pp. 3-27, 1999.

[Fontoura99] Fontoura, M.F.M.C. *Uma Abordagem Sistemátca para o Desenvolvimento de Frameworks.* Tese de Doutorado, Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio), 1999.

[Fontoura01] Fontoura, M.F.M.C. e Lucena, C.J.P. *Extendind UML to Improve the Representation of Design Patterns*. Jornal of Object-Oriented Programming, Vol. 13, Nº. 11, Março, pp. 12-19, 2001.

[Frakes94] Frakes, W.; Isoda, S. *Success Factors of Systematic Reuse*. IEEE Software, Vol. 11, Nº. 5, Setembro, pp.15-19, 1994.

[Froehlich97] Froehlich, G.; Hoover, H.J.; Liu, L. e Sorenson, P. *Hooking into Object-Oriented Application Frameworks*. Em: Proceedings of the 19th International Congress on Software Engineering, Boston, EUA. ACM Press, New York, EUA, pp. 491-501, 1997.

[Gamma93] Gamma, E.; Helm, R.; Johnson, R. e Vlissides, J. *Design Patterns: Abstractions and Reuse of Object-Oriented Design*. Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP'93), Kaiserlauten, Alemanha, 1993.

[Gamma95] Gamma, E.; Helm, R.; Johnson, R. e Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, EUA, 1995.

[Gangopadhyay95] Gangopadhyay, D. e Mitra, S. *Understanding Frameworks by Exploration of Exemplars*. Em: Proceedings of the 7th International Workshop on Computer Aided Software Engineering (CASE95), Toraonto, Canadá, pp. 90-99, 1995.

[Garlan95] Garlan, D.; Allen, R.; Ockerbloom, J. *Architectural Mismatch: Why Reuse Is So Hard*. IEEE Software, Vol. 12, Nº. 6, Novembro, pp.17-26, 1995.

[Gersting82] Gersting, J.L. *Mathematical Structures for Computer Science*. W.H. Freeman and Company, San Francisco, EUA, 1982.

[Griss98] Griss, M.L.; Favaro, J. e d'Alessandro, M. *Integrating Feature Modeling with the RSEB*. Em: Proceedings of the 5th International Conference on Software Reuse, Victoria, Canada. IEEE Computer Society Press, Los Alimos, EUA, pp.76-85, 1998.

[Helm90] Helm, R.; Holland, I.M. e Gangopadhyay, D. *Contracts: Specifying Behavioral Composition in Object-Oriented Systems*. ACM SIGPLAN Notices, Vol. 25, Nº. 10, Setembro, pp.169-180, 1990.

[Horstmann99] Horstmann, C.S. e Cornell, G. *Core Java 2*. Vol. 1, Sun Microsystems Press-Prentice Hall, Upper Saddle River, EUA, 1999.

[Hudak96] Hudak, P. *Building Domain-Specific Embedded Languages*. ACM Computing Surveyes, Vol. 28, Nº. 14, Dezembro, 1996.

[Jacobson97] Jacobson, I.; Griss, M. e Jonsson, P. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley, Reading, EUA, 1997.

[Jacobson99] Jacobson, I; Booch, G. e Rumbaugh, J. *The Unified Software Development Process*. Addison-Wesley, Reading, EUA, 1999.

[Johnson88] Johnson, R.E. e Foote, B. *Designing Reusable Classes*. Jornal of Object-Oriented Programming, Vol. 1, Nº. 2, Junho/Julho, pp. 22--35, 1988.

[Johnson92] Johnson, R.E. *Documenting Frameworks Using Patterns*. ACM SIGPLAN Notices, Vol. 27, Nº. 10, Setembro, pp.63-76, 1992.

[Johnson97] Johnson, R.E. *Frameworks = (Components + Patterns)*. Communication of the ACM, Vol. 40, Nº. 10, Outubro, pp. 39-42, 1997.

[Kang93] Kang, K.C.; Cohen, S.G.; Hess, J.A.; Novak, W.E. e Peterson, A.S. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Relatório Técnico CMU/SEI-90-TR-21. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, EUA, 1993.

[Kang98] Kang, K.C.; Kim, S.; Lee, J.; Kim, K.; Shin, E. e Huh, M. *FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architecture*. Annals of Software Engineering, Vol. 5. Kluwer Academic Publishers, Dordrecht, Holanda, pp. 143-168, 1998.

[Krasner88] Krasner, G.E. e Pope, S.T. *A Cookbook for Using the Model-View-Controller User Interface Paradigm in the Smalltalk-80*. Journal of Object-Oriented Programming, Vol. 1, Nº. 3, Agosto/Setembro, pp.26-49, 1988.

[Kruglinski98] Kruglinski, D.J.; Shepherd, G. e Wingo, S. *Programming Microsoft Visual C++*. 5ª. ed., Microsoft Press, Remond, EUA, 1998.

[Kwanwoo00] Kwanwoo, L.; Kang, K.C.; Koh, E.; Chae, W.; Kim, B. e Choi, B.W. *Domain-Oriented Engineering of Elevator Control Software: A Product Line Practice*. Em: Donohoe, P. ed. Software Product Lines: Experience and Research Directions. Kluwer Academic Publishers, Norwell, EUA, pp. 3-22, 2000.

[Lajoie95] Lajoie, R. e Keller, R.K. *Design and Reuse in Object-Oriented Frameworks: Patterns, Contracts, and Motifs in Concert*. Em: Alagar, V.S. e Missaoui, R. eds. Object-Oriented Technology for Database and Software Systems. World Scientific Publishing, Singapura, pp.295-312, 1995.

[Lange95] Lange, D.B. e Nakamura, Y. *Interactive Visualization of Design Patterns Can Help in Framework Understanding*. Em: Proceedings of OOPSLA' 95, Outubro de 1995, Austin, EUA. ACM Press, New York, EUA, pp. 342-357, 1995.

[Lano94] Lano, K. *Object-Oriented Specification in VDM++*. Em: Lano, K. e Haughton, H. eds. Object Oriented Specification Case Studies. Prentice Hall, Upper Saddle River, EUA, 1994.

[Larman98] Larman, C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*. Prentice Hall, Upper Saddle River, EUA, 1998.

[Lewis95] Lewis, T. *Origins of the Species*: Lewis, T. ed. Object-Oriented Application Frameworks. Manning Publications, Greenwich, EUA, pp. 3-26, 1995.

[Markiewicz01] Markiewicz, M.E. e Lucena, C.J.P. *Object-Oriented Framework Development*. ACM Crossroads, Vol. 7, N$^{o.}$ 4, Summer, pp. 3-9, 2001.

[Mathias01] Mathias, I., Oliveira, T.C. e Lucena, C.J.P. *Domain Oriented Framework Construction*. Proceedings of the 3rd International Conference on Enterprise Information Systems (ICEIS2001), Julho de 2001, Setúbal, Portugal, pp. 599-607, 2001.

[Mathias02a] Mathias, I., Oliveira, T.C. e Lucena, C.J.P. *Domain Oriented Framework Construction*. Em: Filipe, J., Sharp, B. e Miranda, P. eds. Enterprise Information Systems III. Kluwer Academic Publishers, Dordrecht, Holanda, pp. 171-179, 2002.

[Mathias02b] Mathias, I., Oliveira, T.C. e Lucena, C.J.P. *A Proposal for the Incorporation of the Features Model into the UML Language*. Proceedings of the 4th International Conference on Enterprise Information Systems (ICEIS2002), Abril de 2002, Ciudad Real, Espanha, 2002.

[Messmer00] Messmer, B.; Vijayananda, K. e Liver, B. *Telecomunication Network Planning Framework*. Em: Fayad, M.E. e Johnson, R. eds. Domain-Specific Application Frameworks – Frameworks Experience by Industry. John Wiley, New York, EUA, pp. 397-417, 2000.

[Meyer92] Meyer, B. *Applying "Design by Contact"*. IEEE Computer, Vol. 25, Nº. 10, Outubro, pp.40-51, 1992.

[Meyer97] Meyer, B. *Object-Oriented Software Construction.* 2ª. ed., Prentice Hall, Upper Saddle River, EUA, 1997.

[Miller99] Miller, G.G.; McGregor, J. e Major, M.L. *Capturing Frameworks Requirements*. Em: Fayad, M.E. Schmidt, D.C. e Johnson, R. eds. Building Application Frameworks – Object-Oriented Foundations of Framework Design. John Wiley,New York, EUA, pp. 309-322, 1999.

[Neto00] Neto, A. M. *CommercePipe – Um Framework para Criação de Canais Comerciais Consumer to Business na Internet.* Dissertação de Mestrado, Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio), 2000.

[Oliveira00] Oliveira, T.C., Mathias, I. e Lucena, C.J.P. *A Framework Based Approach for Workflow Software Development*. Proceedings of the 4th IASTED International Conference on Software Engineering and Applications (SEA2000), Novembro de 2000, Las Vegas, EUA, pp. 330-335, 2000.

[Oliveira01a] Oliveira, T.C. *Uma Abordagem Sistemática para a Instanciação de Frameworks Orientados a Objetos.* Tese de Doutorado, Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio), 2001.

[Oliveira01b] Oliveira, T.C., Mathias, I. e Lucena, C.J.P. *Using XML and Frameworks to Develop Information Systems*. Proceedings of the 3rd International Conference on Enterprise Information Systems (ICEIS2001), Julho de 2001, Setúbal, Portugal, pp. 571-577, 2001.

[Oliveira02] Oliveira, T.C., Mathias, I. e Lucena, C.J.P. *Frameworks – A High Level Instantiation Approach*. Proceedings of the 4th International Conference on Enterprise Information Systems (ICEIS2002), Abril de 2002, Ciudad Real, Espanha, 2002.

[OMG00] *OMG Meta-Object Facility*. Versão 1.3, 2000. Pode ser encontrada em http://www.omg.org/technology/documents/formal/mof.htm.

[Ortigosa00a] Ortigosa, A.; Campo, M.; Moriyón, R. *Towards Agent-Oriented Assistence for Framework Instantiation*. Em: Proceedings of OOPSLA' 2000, Outubro de 2000, Minneapolis, EUA. ACM Press, New York, EUA, pp. 253-264, 2000.

[Ortigosa00b] Ortigosa, A. *Um Método para la Aplicación de Documentación Inteligente em la Instanciación de Frameworks Orientados a Objetos*. Tese de Doutorado (em espanhol), Universidad Autónoma de Madri, Espanha, 2000.

[Parnas76] Parnas, D.L. *On the Design and Development on Program Families*. IEEE Transaction on Software Engineering, Vol. SE-2, Nº. 1, Março, pp.1-9, 1976.

[Pree94] Pree, W. *Design Patterns for Object-Oriented Software Development.* Addison-Wesley, Reading, EUA, 1994.

[Pree95] Pree, W.; Pomberger, G; Schappert, A. e Sommerlad, P. *Active Guidance of Framework Development.* Journal Software—Concepts and Tools, Vol. 16, Nº. 3. Springer Verlag, Heidelberg, Alemanha, pp.94-103, 1995.

[Pree97] Pree, W. *Object-Oriented Design Patterns and Hot-Spot Cards*. IEEE International Conference on the Engineering of Complex Computer Systems (ICECCS'97), Como, Itália, Setembro, 1997.

[Pree99] Pree, W. *Hot-Spot-Driven Development*. Em: Fayad, M.E. Schmidt, D.C. e Johnson, R. eds. Building Application Frameworks – Object-Oriented Foundations of Framework Design. John Wiley,New York, EUA, pp. 379-393, 1999.

[Pressman92] Pressman, R.S. *Software Egineering: A Practitioner´s Approach*. 3ª. ed., McGraw-Hill, New York, EUA, 1992.

[Prieto-Díaz93] Prieto-Díaz, R. *Historical Overview*. . Em: Prieto-Díaz, R. e Frakes, W.B. eds. Advances in Software Reuse: Selected Papers from the Second International Workshop on Software Reusability. IEEE Computer Society Press, Los Alimos, EUA, pp.1-16, 1993.

[Riehle98] Riehle, D. e Gross, T. *Role Model Based Framework Design and Integration*. Em: Proceedings of OOPSLA' 98, Outubro de 1998, Vancouver, Canadá. ACM Press, New York, EUA, pp. 117-133, 1998.

[Rose] A descrição do Rational Rose pode ser encontrada no seguinte endereço eletrônico: http://www.rational.com/products/rose/index.jsp.

[Schmid97] Schmid, H.A. *Systematic Framework Design by Generalization*. Communication of the ACM, Vol. 40, Nº. 10, Outubro, pp. 48-51, 1997.

[Sommerlad95] Sommerlad, P; Schappert, A. e Pree, W. *Automated Support for Development with Frameworks.* Em: Proceedings of the 17th International Conference on Software Engineering on Symposium on Software Reusability. ACM Press, New York, EUA, pp. 123-127, 1995.

[Together] A descrição do Together pode ser encontrada no seguinte endereço eletrônico: http://www.togethersoft.com.

[UML01] *OMG Unified Modeling Language Specification*. Versão 1.4, 2001. Pode ser encontrada em http://www.omg.org/uml/.

[Vici98] Vici, A. D.; Argentieri, N.; Mansour, A.; d'Alessandro, M. e Favaro, J. *FODAcom: An Experience with Domain Analysis in the Italian Telecom Industry*. . Em: Proceedings of the 5th International Conference on Software Reuse, Victoria, Canada. IEEE Computer Society Press, Los Alimos, EUA, pp.166-175, 1998.

[XMI] A especificação do padrão XMI pode ser encontrada no seguinte endereço eletrônico: http://www.omg.org/technology/xml/index.htm.

[XML] A especificação da linguagem XML pode ser encontrada no seguinte endereço eletrônico: http://www.w3.org/XML/.

[Warmer98] Warmer, J. e Kleppe, A. *The Object Constraint Language – Precise Modeling with UML*. Addison-Wesley, Reading, EUA, 1998.

[Wirfs-Brock89] Wirfs-Brock, R. e Wilkerson, B. *Object-Oriented Design: A Resposibility-Driven Approach*. Em: Proceedings of OOPSLA' 89, Outubro de 1989. ACM Press, New York, EUA, pp. 71-75, 1989.

[Wirfs-Brock90] Wirfs-Brock, R. e Johnson, R.E. *Surveying Current Research in Object-Oriented Design*. Communication of the ACM, Vol. 33, Nº. 9, Setembro, pp.104-124, 1990.